NashDB: Fragmentation, Replication, and Provisioning using Economic Methods

[Demonstration]

Ryan Marcus
Brandeis University
ryan@cs.brandeis.edu

Chi Zhang
Brandeis University
chi@cs.brandeis.edu

Shuai Yu
Brandeis University
shuaiyu@cs.brandeis.edu

Geoffrey Kao
Brandeis University
geoffrey@cs.brandeis.edu

Olga Papaemmanouil
Brandeis University
olga@cs.brandeis.edu

ABSTRACT

Modern elastic computing systems allow applications to scale up and down automatically, increasing capacity for workload spikes and ensuring cost savings during lulls in activity. Adapting database management systems to work on top of such elastic infrastructure is not a trivial task, and requires a deep understanding of the sophisticated interplay between data fragmentation, replica allocation, and cluster provisioning. This demonstration showcases NashDB, an end-to-end method for addressing these concerns in an automatic way. NashDB relies on economic models to maximize query performance while staying within a user's budget. This demonstration will (1) allow audience members to see how NashDB handles shifting workloads in an adaptive way, and (2) allow audience members to test NashDB themselves by constructing synthetic workloads and seeing how NashDB adapts a cluster to them in real time.

PVLDB Reference Format:
Ryan Marcus, Chi Zhang, Shuai Yu, Geoffrey Kao, Olga Papaemmanouil. NashDB: Fragmentation, Replication, and Provisioning using Economic Methods. *PVLDB*, 12(12): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Elastic data management systems, i.e. systems which frequently change the number of machines in a database cluster, are critical for both (1) scaling up to handle workload spikes and (2) scaling down during lulls in activity to reduce costs. To achieve this, such elastic systems must make decisions about data fragmentation, replication, and cluster sizing simultaneously, while balancing query performance and resource utilization costs.

This brings about a number of complications for system administrators. First, administrators must decide how many cluster nodes to *provision*, as under-provisioning leads to diminished performance and over-provisioning leads to undue server provisioning cost. This is especially critical for cloud-deployed applications, as provisioning cost directly translates to monetary cost. Second, administrators must decide how to *distribute* data across the cluster. Here, the goal is to identify a data partitioning and data replication scheme. Since workloads often change over time, static data distribution strategies can decrease query performance (e.g., by under-replicating "hot" data fragments). Third, supporting *query prioritization*, the expectation that queries with a higher priority should experience relatively higher performance than ones with a lower priority, is often expected by users with complex and diverse query workloads.

To address the above, administrators must simultaneously navigate cluster sizing, data replication, and data placement decisions, all while taking query priorities and dynamic workloads into account. Unfortunately, administrators typically approach these complex decisions by manually adjusting the cluster size, re-partitioning, and re-replicating the data to simply avoid hot spots (an approach automated in [18]). These decisions often rely on rule-of-thumb estimations and gut instincts; even when administrators know exactly what performance levels are required, it is difficult to translate performance goals and query prioritization policies into an actualized distributed deployment (e.g., cluster size, data fragments, replicas).

Previous works rarely addressed all of these issues in an end-to-end manner. Several workload-driven fragmentation and replication strategies (e.g., [3, 5, 6, 16, 17]) assume a fixed cluster size or do not support query priorities. Many cluster sizing techniques (e.g., [7, 8, 10, 12–14]) rely on the underlying database to handle fragmentation and replication. Existing work in elastic databases (e.g., [4, 15, 18]) handles workload spikes by incrementally scaling up/down the cluster and re-distributing data on the new cluster configuration, but does not take query prioritization into account.

This demonstration will showcase *NashDB*, a data distribution and cluster sizing framework for making priority-aware data distribution and node provisioning decisions for read-only OLAP systems. NashDB uses methods from economics and game theory to make decisions about fragmentation, replication, and cluster sizing in an end-to-end manner while respecting query priority. NashDB directly translates the monetary value of a query to the user (i.e., the price the user is willing to pay to process that query) into a query priority, and identifies the cluster size and data distribution scheme that balances data (supply) to the value of incoming queries (demand). If all queries are assigned the same value, NashDB will balance the data distribution with respect to data access patterns, adding more replicas for more popular tuples, scaling up the cluster during workload spikes, and scaling down during lulls in activ-

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

ity. When users adjust the price of queries, indicating their priority, NashDB will reconfigure the database cluster to favor high-valued queries, which will thus experience better performance relative to low-valued ones. NashDB was introduced in [9].

Conference attendees will be able to observe NashDB's automatic data fragmentation, replication, placement, and cluster sizing strategies on a live data management system. Additionally, they will be able to modify the priority and volume of each query in a workload, and observe how NashDB is able to adaptively and automatically custom-tailor a cluster to the workload.

2. SYSTEM OVERVIEW

NashDB is a data distribution framework for read-only OLAP analytic applications. NashDB serves a distributed DBMS running on an elastic cluster (e.g., a cluster built on an IaaS provider or a private cloud). We assume a shared-nothing cluster where nodes have access to a fixed amount of non-shared storage, e.g. local SSD or attached Amazon EBS volumes [1].

2.1 Economic model

The primary intuition behind NashDB is an *economic model* of nodes, data, and queries. NashDB models queries as customers ("patrons") who purchase data ("goods") from nodes ("firms"). The priority of a query is modeled as a price that the user is willing to pay to acquire the data needed to process the query – a higher price represents a higher priority. As in a free market, NashDB seeks to balance the supply of data with the demand for data. This entails identifying a data distribution scheme that is in *Nash equilibrium*. In order to achieve this, we depend on economic theory and the efficiency of market systems.

Let us assume a distributed DBMS running on an elastic cluster. Each cluster node has a cost per unit time that the node is running (e.g., rent cost) and a certain amount of disk space for storing data. We also assume that DBMS tables are stored in some physical ordering (e.g., arbitrary or clustered), and that tables are horizontally fragmented into a set of disjoint fragments.

Each incoming query has an associated price indicating its priority. In our economic model, a query's price is equally divided among the tuples accessed by that query (see [9] for a formalization). NashDB continuously monitors the tuples accessed by incoming queries and the price paid for each tuple in the database. This allows us to define the *value of a tuple*, i.e., the *total expected income* earned from a particular tuple. The value of a tuple is affected by (1) the price of the queries accessing the tuple (higher-priced queries provide more value), and (2) the number of queries accessing the data fragments that include the particular tuple (a higher number of queries provide more value).

We model each tuple as a "good" that can be provided by a node. A node is paid by queries for access to tuples, and thus each node has an incentive to provide tuples. The higher the price of a query, the more income the node will receive from that query. However, nodes must pay costs for each provided tuple (e.g. storage fees). Therefore, nodes wishing to maximize their income will choose to provide profitable tuples.

Furthermore, tuples are replicated across the cluster nodes. As in a market system, an increase in the supply of a good results in a decrease in the price of that good. Specifically, as the number of nodes providing a replica of a tuple increases (an increase in quantity supplied), the income a node expects to receive from a replica decreases. Eventually, we aim to replicate each tuple such that storing a replica of that tuple is *minimally* profitable: all current replicas are profitable, but the cost of storing a single additional replica exceeds the diminished expected income from that replica. In this setting, NashDB strives to *balance supply against demand*: it seeks to replicate each tuple such that each replica is expected to be profitable, but offering an additional replica does not increase the expected profit for any of the cluster nodes. This condition represents a *Nash equilibrium* [11].

2.2 NashDB functionality

NashDB generates fragmentation, replication, and cluster sizing strategies that are aware of query priorities and adapt to workload shifts. We conceptualize the priority of each query as the price the user is willing to pay to process that query. The higher the query price (a.k.a. the query's value) the more resources (i.e., replicas, cluster nodes) will be allocated to serve that query, relatively to lower-priced queries. Hence, higher-priced queries will enjoy improved performance related to low-priced ones. Under no query prioritization (i.e., all queries are assigned the same price), NashDB adapts the number of replicas and the cluster size to data access patterns, scaling up the cluster during workload spikes, and scaling the cluster down during lulls in activity.

NashDB operates underneath a traditional DBMS, providing access to the underlying data stored on disk. A high-level overview of NashDB is depicted in Figure 1. First, a user-submitted query is transformed into a query execution plan by a query optimizer. The leaf nodes of this query execution plan represent operations which access data. Instead of requesting a specific range of bytes from a hard disk, each of these leaf operators sends those requests to NashDB in the form of a *range scan*: a starting and an ending tuple index (based on the on-disk order) to read. NashDB keeps track of how often each tuple is read and and maintains each tuple's value. At a user-defined interval, NashDB uses the tuple value information to reconfigure the cluster. During cluster reconfiguration, NashDB fragments the database by grouping together adjacent tuples that have similar value. Second, NashDB replicates fragments proportionally to their aggregated tuple value. NashDB then allocates replicas onto "just the right number" of cluster nodes, and routes data access requests to nodes aiming to minimize data access latency. Collectively, these techniques can produce a system that offer low query execution times. We next describe each of these components in more detail.

Tracking tuple values NashDB examines both query prices and query plan information from incoming queries to analyze tuples access frequency and to estimate the "importance" of a tuple (a.k.a. *tuple value*). Intuitively, tuples that are accessed frequently by high-priority queries should have a higher value than queries that are accessed rarely, or by low-priority queries. NashDB continually updates its estimation of each tuple's value, using an efficient data structure called a tuple value estimation tree [9].

Fragmentation During cluster reconfiguration, NashDB customizes the underlying cluster to the user's workload. To do this, NashDB first splits up the data into consecutive groups of tuples called *fragments* (e.g., the blue, green, and orange shaded regions in Figure 1). NashDB picks fragment boundaries to minimize the variance of tuple value within a fragment, i.e., to group tuples with similar tuple values together. In [9], we give an optimal quadratic time algorithm for finding such fragment boundaries, as well as a linear time greedy approximation.

Replication After splitting up the database into a set of fragments, NashDB next decides how many replicas of each fragment to create. To do this, NashDB first computes the cost of storing a fragment (e.g., cloud provider storage fees) and the value of each fragment (the sum of the value of each tuple within the fragment). For example, the costs and values of three example fragments are

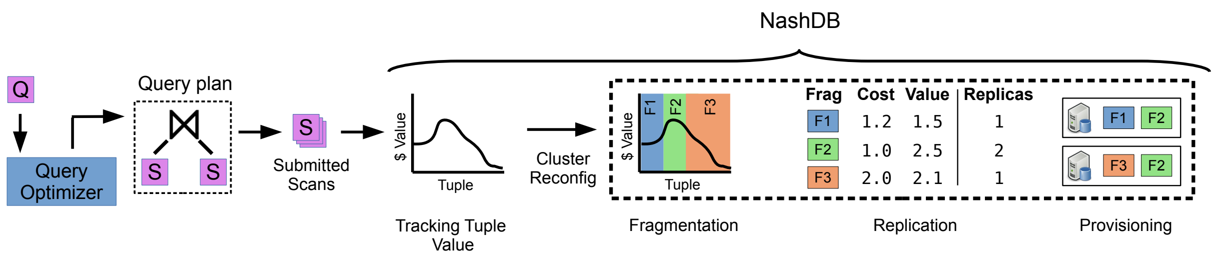


Figure 1: Workflow of NashDB's functionality

shown in Figure 1: F1 (blue) has a cost of 1.2 and a value of 1.5. NashDB chooses to replicate each fragment the same number of times as an ideal free market would choose to "produce" each fragment as a product. For example, fragment F1 is replicated once, creating a single replica of F1 which will have a cost of 1.2 and an income of 1.5. If a second replica of F1 had been created, each replica would have to pay a cost of 1.2, but would only receive an income of 0.75 each, since queries ("patrons") would split their accesses ("purchases") between both replicas ("firms"). On the other hand, fragment F2 also has a cost of 1, but since the tuples in F2 have a higher value than those in F1, fragment F2 has a higher value of 2.5. NashDB thus creates two replicas of F2.

We present a formalization of this replication strategy in [9], and prove that it necessarily results in a Nash equilibrium: adding any additional replica, or removing any replica, cannot lead to an increase in profit. While the Nash equilibrium does not necessarily represent a *global* optimal, it does represent a "steady state" in which no single change can improve performance.

Server provisioning After choosing how many replicas of each fragment to produce, NashDB next decides how to allocate them into a cluster. To do this, NashDB attempts to find the smallest number of server nodes (each node incurs a fixed cost) that can (1) hold all of the replicas (each machine has only finite disk space) and (2) such that no two replicas of the same fragment are on the same machine (as there is no benefit to redundantly storing the same information on the same machine). This problem, known as the class-constrained bin packing problem, is NP-Hard. We apply a greedy heuristic, called "Best First Fit Decreasing" [9,19], in order to come up with an approximation of the optimal solution.

Cluster Transitioning Once a new cluster configuration is computed, NashDB determines the most efficient way to transition the cluster from its previous data distribution state to the newly computed state. This involves changes to (1) fragment boundaries, (2) the number of replicas, (3) the number of cluster nodes, and (4) the allocation of replicas to nodes. Finding a transition strategy that minimizes data transfer is critical to quickly transitioning between schemes. NashDB deals with this challenge by using a bipartite graph matching algorithm [9] which minimizes data transfer costs.

This process – fragmentation, replication, and provisioning – is repeated periodically to keep the underlying cluster in-sync with the demands of the user's workload. This allows NashDB automatically respond to changes in query volume and query prioritization.

3. DEMONSTRATION

Our demonstration is split into two scenarios. In the first, users can compare NashDB's behavior on a real-world workload, comparing NashDB's cluster configuration to a naive, value-based partitioning scheme. In the second, users will be able to modify the volume and budget of queries in a synthetic workload, and observe the changes to query latency and cluster configuration in real time.

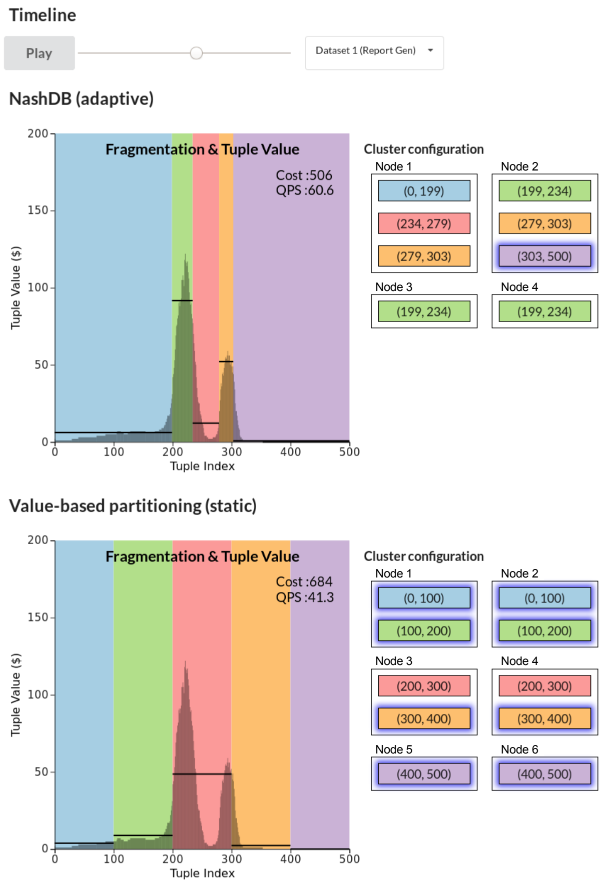


Figure 2: Scenario 1

3.1 Scenario 1: Adaptive Management

In the first scenario, we will compare NashDB's effectiveness with a naive, value-based partitioning scheme running on AWS [2] `t2.large` instances (the cluster size ranges between 4 and 25 nodes). Both techniques are applied to two datasets provided by a large corporation on the condition of anonymity. The user can observe how the NashDB's fragmentation changes over time, and how NashDB adapts the cluster configuration to match the shifts in the workload. The value-based scheme offers a static fragmentation and uses NashDB's replication and server provisioning approach.

The interface for this scenario is shown in Figure 2. The "Play" button at the top runs the real-world workload from the corporation. Users will be able to see the tuple value graph (left) and the cluster configuration (right) for both NashDB (top) and a naive value-based partitioning scheme (bottom). The tuple value graph shows the tuple index on the x-axis and the tuple's monetary value on the y-axis. Each colored range of tuples corresponds to a different fragment indicating where the fragmentation boundaries are defined. For example, in the top tuple value graph, one can observe that four fragments are defined (e.g., the blue fragment includes tuples 1-200). The graph also indicates that this fragmentation creates a high variance in tuple values, with the green fragment having the "hottest" tuples followed by the orange fragment.

The right hand side of the plot demonstrates the cluster configuration. Each outer box represents a cluster node (the top right shows 4 nodes, the bottom right shows 6 nodes), and each colored box, e.g., the blue box labeled (0, 199), corresponds to a fragment on that node. Fragments that are over-replicated (i.e., it would be more profitable to have fewer of them) glow in blue, whereas fragments that are under-replicated (i.e., it would be more profitable to have more of them) glow in red. In Figure 2, the purple fragment appears over-replicated since there are no queries requesting it (tuples values are zero for that fragment). NashDB must create at least one replica of each fragment, otherwise the data would be lost.

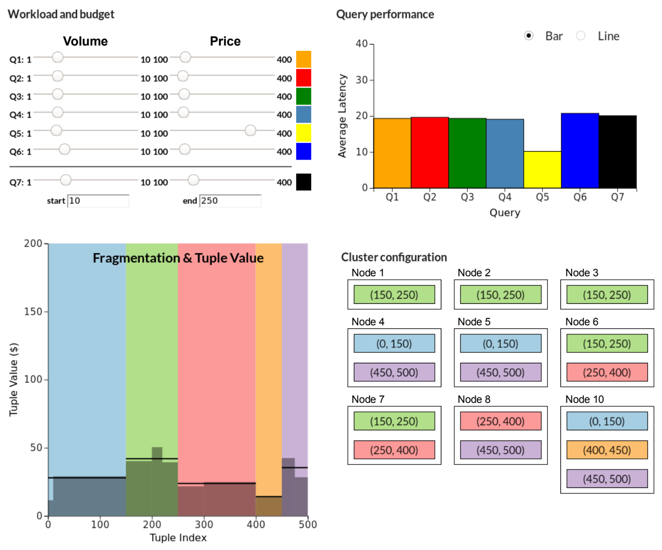
Additionally, the user can compare the current throughput of each system (i.e., queries per second (QPS)), as well as the monetary cost incurred by each system so far (the numbers appear on the top right corner of the tuple value graph). At this point in the simulation, NashDB is operating at 60 QPS and has incurred a cost of \$4.98, whereas the naive approach is operating at a lower throughput (41 QPS) and has incurred a higher cost (\$6.72).

The audience member may press the "Play" button at the top, or use the slider to manually select a time, to progress or rewind the workload (best viewed in the accompanying video). This allows the audience member to (1) understand how NashDB reconfigures a cluster to match a shifting workload, and, via comparison with the naive method, (2) the importance (in terms of cost and performance) of adaptive systems in general.

3.2 Scenario 2: Query Prioritization

In the second scenario, we will show a live NashDB system executing queries from a synthetic workload on a cloud provider (AWS [2]). Figure 3 depicts the user interface for the second scenario. In the top left, the audience can use the sliders to adjust the volume and price (priority) of a number of (synthetic) simple range queries. The audience member may also insert their own values for a simple range query (Q7). The tuple value graph in the bottom left, and the cluster configuration in the bottom right, will update in real-time as the audience member moves any slider.

In the top right, the audience will see the average latency of each range query, computed using a simple sliding window. The predicate values of each range query can be viewed by hovering over them. The audience member can switch between a bar plot or a line graph of query latency using the toggle button in the upper right. In Figure 3, the latency of Q5 (yellow) is especially low because the user was willing to pay a high price for Q5. If the audience member were to increase the budget for Q1 and decrease the budget for Q5, NashDB would reconfigure the cluster in real-time (the new fragments and provisioning scheme would be shown in the bottom right), and the user would be able to see the affect on query latency in the upper right plot: the orange bar would fall and the blue bar would rise. This again stresses the importance of adaptive elastic systems. By changing the underlying physical layout of the data based on the workload, we can create systems that adapt to user needs. Additionally, the second scenario allows the audience member to gain an intuition for the relationship between query priority and replication: when a particular tuple receives significant traffic, it makes sense to replicate it many times, and, on the other hand, if a tuple is hardly read at all, a smart DBMS will maintain as few copies of it as possible.



start 10 end 250

[✓ Bar] Line

Node	Fragments
Node 1	(150, 250)
Node 2	(150, 250)
Node 3	(150, 250)
Node 4	(0, 150); (450, 500)
Node 5	(0, 150); (450, 500)
Node 6	(150, 250); (250, 400)
Node 7	(150, 250); (250, 400)
Node 8	(250, 400); (450, 500)
Node 10	(0, 150); (400, 450); (450, 500)

Figure 3: Scenario 2

4. CONCLUSION

NashDB provides an economics-driven method for automatically adapting an elastic database to the user's workload needs. Our demonstration will allow the audience to explore NashDB's behavior on a real-world workload, emphasizing the importance of adaptive fragmentation systems. Additionally, audience members can change the query volume and prioritization in a real-time workload, directly observing how workload requirements can translate into a cluster configuration to benefit cost and query latency.

5. ACKNOWLEDGMENTS

This research was funded by NSF III-1815701.

6. REFERENCES

[1] Amazon EBS, https://aws.amazon.com/ebs/.
[2] Amazon Web Services, http://aws.amazon.com/.
[3] C. Curino et al. Schism: A workload-driven approach to database replication and partitioning. *VLDB '14*.
[4] S. Das et al. ElasTraS: An Elastic, Scalable, and Self-managing Transactional Database for the Cloud. *TODS '13*.
[5] J. O. Hauglid et al. DYFRAM: Dynamic fragmentation and replica management in distributed database systems. *Distrib Parallel DB '10*.
[6] K. A. Kumar et al. SWORD: Workload-aware Data Placement and Replica Selection for Cloud Data Management Systems. *VLDB Journal '14*.
[7] K. Lolos et al. Elastic management of cloud applications using adaptive reinforcement learning. In *Big Data '17*.
[8] R. Marcus et al. A Learning-Based Service for Cost and Performance Management of Cloud Databases. In *ICDE '17*.
[9] R. Marcus et al. NashDB: An Economic Approach to Fragmentation, Replication and Provisioning for Elastic Databases. In *SIGMOD '18*.
[10] R. Marcus et al. WiSeDB: A Learning-based Workload Management Advisor for Cloud Databases. *VLDB '16*.
[11] J. F. Nash. Equilibrium points in n-person games. *PNAS '50*.
[12] J. Ortiz et al. PerfEnforce Demonstration: Data Analytics with Performance Guarantees. In *SIGMOD '16*.
[13] J. Ortiz et al. SLAOrchestrator: Reducing the Cost of Performance SLAs for Cloud Data Analytics. In *USENIX ATX'18*.
[14] J. Rogers et al. A generic auto-provisioning framework for cloud databases. In *ICDEW '10*.
[15] M. Serafini et al. Accordion: Elastic Scalability for Database Systems Supporting Distributed Transactions. *VLDB '14*.
[16] M. Serafini et al. Clay: Fine-grained Adaptive Partitioning for General Database Schemas. *VLDB '16*.
[17] J. Sidell et al. Data replication in Mariposa. In *ICDE '96*.
[18] R. Taft et al. E-Store: Fine-grained Elastic Partitioning for Distributed Transaction Processing Systems. *VLDB '15*.
[19] E. C. Xavier et al. The class constrained bin packing problem with applications to video-on-demand. *Theoretical Computer Science '08*.