NashDB: An End-to-End Economic Method for Elastic Database Fragmentation, Replication, and Provisioning

Ryan Marcus, Olga Papaemmanouil, Sofiya Semenova, Solomon Garber
Brandeis University
[ryan,olga,sofiya,solomon]@cs.brandeis.edu

ABSTRACT

Distributed data management systems often operate on "elastic" clusters that can scale up or down on demand. These systems face numerous challenges, including data fragmentation, replication, and cluster sizing. Unfortunately, these challenges have traditionally been treated independently, leaving administrators with little insight on how the interplay of these decisions affects query performance. This paper introduces *NashDB*, an adaptive data distribution framework that relies on an economic model to automatically balance the supply and demand of data fragments, replicas, and cluster nodes. NashDB adapts its decisions to query priorities and shifting workloads, while avoiding underutilized cluster nodes and redundant replicas. This paper introduces and evaluates NashDB's model, as well as a suite of optimization techniques designed to efficiently identify data distribution schemes that match workload demands and transition the system to this new scheme with minimum data transfer overhead. Experimentally, we show that NashDB is often *Pareto dominant* compared to other solutions.

KEYWORDS

Database management systems; partitioning; fragmentation

ACM Reference Format:
Ryan Marcus, Olga Papaemmanouil, Sofiya Semenova, Solomon Garber. 2018. NashDB: An End-to-End Economic Method for Elastic Database Fragmentation, Replication, and Provisioning. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3183713.3196935

1 INTRODUCTION

Large-scale elastic data management systems are becoming popular as applications face increasing data sizes and workload demands. These large-scale systems offer high availability and query performance by fragmenting and replicating data across multiple cluster nodes. The elasticity of these systems (growing and shrinking the number of fragments, replicas, and nodes as needed) is critical for handling workload spikes and reducing cluster maintenance costs.

Elastic distributed data management systems bring about many 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 resource usage cost. Second, administrators must decide how to *distribute* data across the cluster. Since workloads often change over time, static data distribution decisions can decrease query performance. 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. Thus, administrators must simultaneously navigate cluster sizing, data replication, and data placement decisions, all while taking into account query priorities and dynamic workloads.

Administrators typically approach these complex decisions by manually adjusting the cluster size, re-partitioning, and re-replicating the data to simply avoid hot spots. 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).

In this paper, we present *NashDB*, a data distribution and cluster sizing framework for making priority-aware data distribution and node provisioning decisions for read-only OLAP systems. NashDB provides automatic data fragmentation, replication, placement, transitioning, and cluster sizing strategies, all while taking into account user-defined query priorities. Ultimately, NashDB aims to balance fragment and replica *supply* to workload *demand*, as captured by query priority and data access patterns.

Previous works rarely address all of these issues in an end-to-end manner. Several workload-driven fragmentation and replication strategies (e.g., [12, 19, 24, 39, 41]) assume a fixed cluster size or do not support query priorities. Many cluster sizing techniques (e.g., [9, 21, 26, 28, 30, 34, 37, 46]) rely on the underlying database to handle fragmentation and replication. Existing work in elastic databases (e.g., [13, 38, 42]) handle workload spikes by incrementally scaling up/down the cluster, and re-distributing data on the new cluster configuration. However, it is not straightforward to predict the impact of adding/removing nodes on query performance.

NashDB relies instead on a more user-friendly approach: its takes query prioritization – the monetary value of each query to the user (i.e., the price the user is willing to pay for that query) – as an input, and identifies the cluster size and data distribution scheme that balances data supply to the value of incoming queries. NashDB uses economics-inspired methods to make decisions about fragmentation, replication, and cluster sizing in an end-to-end manner while respecting query priority. If all queries are assigned the same value, NashDB will balance the data distribution to data access patterns, adding more replicas for more popular tuples, scaling up the cluster during workload spikes, and scaling down during lulls in activity. When users adjust the value of queries, indicating their priority,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'18, June 10–15, 2018, Houston, TX, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196935

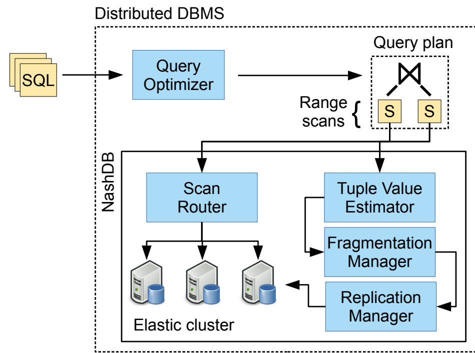


Figure 1: NashDB system architecture

high-valued queries will experience better performance relative to low-valued ones.

To accomplish this, NashDB first fragments the data by grouping together adjacent tuples that have similar *value*, a metric defined based on the frequency and price of the queries accessing a tuple. NashDB relies on a novel tuple value estimation tree structure for efficiently storing and retrieving tuple values. Second, NashDB replicates fragments proportionally to their aggregated tuple value. In this stage, we model each fragment as a product that can be offered by the cluster. We replicate each fragment the same number of times that the an ideal free market would choose to "offer" it (i.e., until it is no longer profitable to add one more replica), and we show that this replication strategy results in a *Nash equilibrium*. 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. Finally, it includes a mechanism for transitioning the cluster from an old data distribution scheme to a new scheme with minimal data transfer overhead. Collectively, these techniques can produce systems that offer low query execution times.

The contributions of this paper are:

- a data fragmentation algorithm that is aware of data access patterns and query priorities,
- a replication algorithm that balances replica supply to workload demands,
- an algorithm for transitioning between two different data distribution schemes with optimal data transfer overhead,
- and a latency-aware strategy for routing data access requests to cluster nodes.

The rest of this paper is organized as follows. Section 2 describes the NashDB system model and Section 3 presents the economic intuition behind NashDB. Section 4 describes how NashDB maintains tuple value information. Section 5 describes our fragmentation approach and Section 6 introduces our replication algorithm. Section 7 describes the mechanism for transitioning between distribution schemes and Section 8 introduces our data access routing approach. Related work is presented in Section 9. Experimental results are presented in Section 10, and concluding remarks are in Section 11.

2 SYSTEM MODEL

NashDB is a data distribution framework for read-only OLAP analytic applications. Figure 1 depicts our system model. NashDB serves a distributed DBMS running on an elastic cluster (e.g., a cluster built on a IaaS provider [2, 3, 5] or a private cloud). We assume a shared-nothing cluster where each node has access to a fixed amount of non-shared storage, e.g. local SSD or attached Amazon EBS volumes [1].

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 still 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 examines both query prices and query plan information from incoming queries to analyze tuples access frequency and to estimate the "importance" of a tuple (aka *tuple value*). As shown in Figure 1, incoming queries are analyzed by the DBMS optimizer. NashDB receives the scans on the plan's input relations. In OLAP applications, these data access scans are typically performed on ordered relations (i.e., a clustered table is ordered on a primary key). Since these scan operators retrieve contiguous range of tuples, we refer to them as "range scans" or "scans." Since NashDB's goal is to understand data *access* patterns, we consider all fetched tuples to be accessed by a query, even if a range scan fetches a block of tuples that is irrelevant to the query, or that are later filtered out.

Data access scans are sent to the *tuple value estimator*, which maintains a value estimate of each tuple in the database. These estimates are used periodically by the *fragmentation manager*, which responds to changes in data access patterns by computing updated fragmentation schemes. New schemes are sent to the *replication manager*, which determines (a) how many times to replicate each fragment, and (b) how to allocate these replicas across a cluster of "just the right size". NashDB also facilitates the transitioning from one distribution scheme to another with minimal data transfer overhead. Finally, for each incoming query, the *scan router* dispatches data access requests to data replicas, accounting for both the span (the number of nodes serving a request) and data access latency.

3 ECONOMIC MODELING

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. We next describe our model.

Problem Statement Let us assume a distributed DBMS running on an elastic cluster. Each cluster node has a cost per unit time it is used (e.g., rent cost) and a certain amount of disk space for storing data. We also assume the DBMS operates on a horizontally fragmented data set. Here, tables are stored in some physical ordering

(e.g. arbitrary or clustered), and tables are horizontally fragmented into a set of disjoint fragments. The first task of NashDB is to continuously evaluate the fragmentation of each table in the database, and identify new fragmentation schemes that match the data access patterns of incoming queries.

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 (formalized in Section 4.1). NashDB continuously monitors the tuples accessed by incoming queries and the price paid for each tuple in the database. Since tuples are organized into fragments, this allows us to define the *value of a fragment*, i.e., the *total expected income* earned from all the tuples in a fragment. The value of a fragment is affected by (1) the price of the queries accessing the fragment (higher-priced queries provide more value), and (2) the number of queries accessing a fragment (a higher number of queries provide more value).

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

Furthermore, fragments 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 fragment increases (an increase in supply), the income a node expects to receive from a replica decreases. Eventually, we aim to replicate each fragment such that storing that a replica of that fragment 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 fragment 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* [33] (formalized in Section 6).

4 TUPLE VALUE ESTIMATION

To measure demand (and thus balance it against supply), NashDB maintains an estimate of the monetary value of each tuple in the database. Next, we provide the formal definition of tuple and fragment value and we introduce an augmented binary search tree structure that enables efficiently storing and accessing these values.

4.1 Tuple & Fragment Value Definitions

Here, we formally define the notation of *tuple value*. For each incoming query q, let the associated query price (or priority) be $Price(q)$, and let S_q be the scans issued by the corresponding query execution plan (see Figure 1). Since the input relations of our queries have some physical ordering (e.g. arbitrary or clustered), we denote the starting (inclusive) and ending (exclusive) tuples of a scan $s_i \in S_q$ as $Start(s_i)$ and $End(s_i)$, respectively.[1] We define the size of a scan to be equal to the number of tuples it accesses, i.e., $Size(s_i) = End(s_i) - Start(s_i)$. Next, we define the price of a range scan operation of a query q, $Price(s_i)$, to be proportional to its size as follows:

$$Price(s_i) = \frac{Size(s_i)}{\sum_{s_j \in S_q} Size(s_j)} \times Price(q) \quad (1)$$

Intuitively, the monetary value of each tuple is the amount of income that tuple generates from scans. Given the price of a range scan s_i, $Price(s_i)$, the value from each tuple retrieved by the scan s_i is $Price(s_i)/Size(s_i)$. Since tuples are accessed by multiple scans from different queries with potentially diverse prices, NashDB maintains an average value estimate $V(x)$ for all tuples over a window W of the most recent range scans. This value represents the income a node can expect to receive per scan by holding a copy of a tuple, assuming there are no other copies of that tuple:

$$V(x) = \frac{1}{|W|} \sum_{s_i \in W} \begin{cases} \frac{Price(s_i)}{Size(s_i)} & \text{if } x \text{ was read by } s_i \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

Given the value of tuple, we can now define the value of a fragment. Similarly to a range scan, a fragment f_i of a table t is an ordered set of tuples (according to the physical ordering of the table t) starting at the tuple with index $Start(f_i)$ and ending at the tuple with index $End(f_i)$. We define the size of fragment to be equal to number of tuples it holds, i.e., $Size(f_i) = End(f_i) - Start(f_i)$. Next, we define the value of a fragment f_i, $Value(f_i)$, as the sum of the average value of each tuple within that fragment. Formally:

$$Value(f_i) = \sum_{x=Start(f_i)}^{End(f_i)} V(x) \quad (3)$$

We use the value of each fragment to decide how many times a particular fragment should be replicated (see Section 6). Because storing values for each tuple would be costly, we next introduce a tree structure to allow for fast computation of fragment values while using relatively little storage.

4.2 Value Estimation Tree

NashDB must track the average monetary value of each tuple. However, storing the value of each tuple directly in the database would introduce significant overhead. Thus, we introduce a *tuple value estimation tree*, an augmented binary search tree, similar to an interval tree [15], which enables efficient storage and retrieval of tuples values based on a window of processed range scans.

Intuitively, the tree works by tracking the *change* in the monetary value of each tuple compared to the value of the previous tuple based on the relation ordering. The monetary value of a tuple, relative to the previous tuple, can only change if a scan stops or starts at that tuple. Thus, a node is added to the tree for each unique starting point and stopping point of the processed data access scans within the window of scans W.

Let us denote as n_i a node of the tree, with key $K(n_i)$ being the tuple index represented by the tree node (i.e., a unique starting/ending point). Each tree node n_i also stores the price of the scans starting or ending at $K(n_i)$, $S(n_i)$ and $E(n_i)$, respectively. Formally, with W representing the current window of recent range

[1] The $Start(s_i)$ and $End(s_i)$ values refer to the index of a tuple relatively to the physical ordering of the original table.

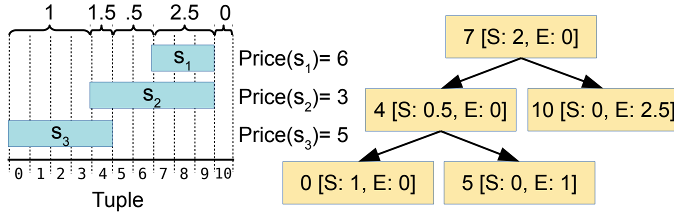


Figure 2: Example value estimation tree. Each node represents a starting or ending point for a scan, and the S and E fields correspond to the value of the range scans "starting" and "ending", respectively.

scans, these prices can be defined as:

$$S(n_i) = \sum \left\{ \frac{Price(s_j)}{Size(s_j)} \;\middle|\; s_j \in W \land Start(s_j) = K(n_i) \right\}$$

$$E(n_i) = \sum \left\{ \frac{Price(s_j)}{Size(s_j)} \;\middle|\; s_j \in W \land End(s_j) = K(n_i) \right\}$$

Example Figure 2 shows three example range scans and the corresponding tree. Each tree node represents a starting or ending point of one or more scans, e.g. the root node r has a key of $K(r) = 7$, which corresponds to $Start(s_1) = 7$. Similarly the tree has nodes with keys equal to 4, 5, 0, and 10 because there are scans that start or end at these tuples. For the root r with $K(r) = 7$, the value $S(r)$ is equal to the sum of the normalized price for all scans starting at tuple 7. Since s_1 is the only scan that starts at tuple 7, its price is 6, and it touches 3 tuples, we have $S(r) = 6/3 = 2$. The value $E(r)$ is zero, because no scans end at tuple 7. Similarly, for the rightmost node n with a key of $K(n) = 10$, the associated scans are s_1 and s_2, which end at tuple 10. Since scans s_1 and s_2 have prices of 6 and 3 respectively, the node has $E(n) = 6/3 + 3/6 = 2.5$.

Algorithm 1 Value estimation tree iteration

```
function IterateValues(tree)
    α ← 0
    for n_i ∈ tree (in-order) do
        α ← α + S(n_i) − E(n_i)
        NoteValue(K(n_i), K(n_{i+1}), α/|W|)
    end for
end function
```

Tree Lookups Since the tuple value at a key $K(n_i)$ increases by $S(n_i)$ and decreases by $E(n_i)$, the value of any tuple x, $V(x)$, can be determined by summing the $S(n_i) - E(n_i)$ values for all nodes n_i such that $K(n_i) \leq x$. Thus, the value of every tuple can be computed using an in-order traversal of the tree (Algorithm 1). We initialize an accumulator $\alpha = 0$, and then begin an in-order traversal of the tree. For each node n_i in the traversal, we add the value of $S(n_i)$ and subtract the value of $E(n_i)$ from α. We then note that the value of all tuples from $K(n_i)$ to $K(n_{i+1})$ have a value of α. Since this process requires only an in-order traversal of the tree, it requires $O(|W|)$ time and constant space.

Example We next show how we iterate through the tree in Figure 2, which is created over a window of $|W| = 3$ scans. We first set $\alpha = 0$ and then begin an in-order traversal of the tree. The first node is 0, so we add its S value to α, yielding $\alpha = 1$. This means that all the tuples from the current node (0) to the next node (4) have a value of $\frac{\alpha}{|W|} = \frac{1}{3}$. The next node is 4, and we add its S value of 0.5 to α, yielding $\alpha = 1.5$. Therefore, all the tuples from the current node (4) to the next node (5) have a value of $\frac{1.5}{3}$. The next node is 5, which has an E value of 1, so we subtract 1 from α, yielding $\alpha = 0.5$. We note that all tuples between 5 and 7 have a value of $\frac{0.5}{3}$. Processing the next node, 7, gives $\alpha = 2.5$ and tells us that the tuples between 7 and 10 have a value of $\frac{2.5}{3}$. Finally, processing node 10 causes us to subtract 2.5 from α, yielding $\alpha = 0$. This signifies that all tuples 10 and above have a value of zero.

Tree Updates When a query q arrives, the starting and stopping points of its scans $s_i \in q$ can be inserted into the tree by searching the tree for a node n_1 with $K(n_1) = Start(s_i)$ and a node n_2 with $K(n_2) = End(s_i)$, and then incrementing $S(n_1)$ and $E(n_2)$ by $Price(s_i)/Size(s_i)$. If either n_1 or n_2 do not exist, they are created. The balance of the tree can be maintained using standard techniques [8]. A scan is removed by finding the appropriate n_1 and n_2 nodes, decrementing the $S(n_1)$ and $E(n_2)$ values, and then removing either node if both $S(n_i)$ and $E(n_i)$ are zero.

To remove scans that fall outside the scan window size, we additionally store a circular buffer of $(Start(s_i), End(s_i), Price(s_i))$ values for each scan s_i in the scan window. When a new query q arrives, the scans $s_i \in q$ are added to the buffer, and if the size of the buffer exceeds the scan window size, the oldest scans are removed from the buffer and the tree. Since adding and removing an element from the buffer takes $O(1)$ time and inserting/retrieving values from a binary search tree requires $O(\log n)$ time, inserting a new scan into the tree can thus be done in $O(\log |W|)$ time, and the tree itself requires $O(|W|)$ space. Additional optimizations are given in Appendix A.

Scan Window Size The size of the window of scans controls how responsive the value estimation tree is to changes in a workload. Small scan window sizes cause the value estimation tree to respond quickly to the most recent queries, as the scans of old queries will be quickly evicted from the buffer. Large scan window sizes enable the value estimation tree to capture more complex workload trends (e.g. when queries become small and disjoint, or simply more complex). The scan window size must be tuned by the administrator, aiming for a value that is sufficient to capture access patterns and respond to changes at an acceptable rate.

5 FRAGMENTATION

In traditional distributed DBMSes (e.g., [6, 25]), fragmentation schemes are chosen by skilled administrators, who normally use a value-based or hash-based approach. However, such fragmentation approaches are difficult to tune and require frequent intervention as workloads shift. Furthermore, this approach, as well as previous techniques (e.g., [12, 24, 39, 42]) are agnostic to query priorities and often act independently from replication decisions.

NashDB introduces an *automatic* and *workload-driven* approach to fragmentation that is also tightly coupled to replication. NashDB fragments data and replicates fragments across multiple machines. As described previously, NashDB maintains an up-to-date estimate of the value of each fragment. Given this value, NashDB aims to replicate fragments with higher value because they are required by more scans (or by higher priced/priority scans). Specifically, we want to replicate fragments proportionally to each fragment's

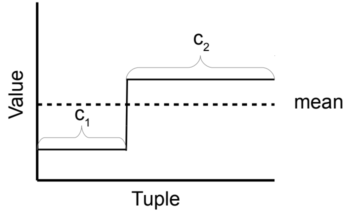


Figure 3: An example of an imbalanced fragment: the tuples in c_1 will be under-replicated, and the tuples in c_2 will be over-replicated.

value, which is the sum of the values of the tuples in the fragment. Since fragments will be replicated based on their value, we strive to create fragments where the monetary value of each tuple is as uniform as possible. This avoids under or over replicating tuples.

To illustrate this, consider Figure 3, which shows an example of a fragment with non-uniform value. The dotted line indicates the average value of the fragment. The tuples labeled c_1 have a value less than the mean, and the tuples labeled c_2 have a value greater than the mean. If the entire fragment is replicated based on its mean value, then the tuples labeled c_1 will be over-replicated, and the tuples labeled c_2 will be under-replicated. To create fragments that are as uniform as possible, the tuples in c_1 and c_2 should be placed into different fragments.

We next explain how fragment uniformity is measured, and give the optimization problem to be solved. Then, we present two automatic fragmentation algorithms. The first uses a dynamic programming scheme [29] to find optimally uniform fragments for small databases. The second uses a greedy heuristic to create good, but not necessarily optimal, fragmentation schemes for large databases.

5.1 Fragment Uniformity

One intuitive measure of the inefficiency caused by variation from the mean value within a fragment is the unnormalized variance, which we refer to as the error. If the fragment f_i stores the tuples from index $Start(f_i)$ to $End(f_i)$ and the number of tuples it includes is $Size(f_i) = End(f_i) - Start(f_i)$, then this error can be defined as:

$$Err(f_i) = \sum_{x=Start(f_i)}^{End(f_i)} \left(V(x) - \frac{Value(f_i)}{Size(f_i)} \right)^2 \tag{4}$$

Maximally uniform fragments could be created by placing each tuple into its own fragment, but this level of granularity could not be effectively stored in disk blocks. Since a block represents the minimum level of granularity readable by a disk, we set a cap on the maximum number of fragments, $maxFrags$, such that, on average, each fragment fits in a disk block.

We choose to make the *average* fragment size equal to the size of a disk block instead of requiring each fragment to be approximately the size of a block. Doing so enables our fragmentation algorithm to pack even small groups of high-valued tuples into fragments that can be highly replicated. If a small set of high-valued tuples were forcibly grouped with neighboring low-value tuples, the resulting fragment would not be sufficiently replicated. By requiring that only the average fragment size be equal to the disk block size, we balance disk-read efficiency with the total fragment error. We thus seek a set of horizontal disjoint fragments $F = \{f_1, f_2, \ldots\}$ of each table that minimizes the sum of the errors:

$$\min_F \sum_{f_i \in F} Err(f_i), \text{ subject to } |F| = maxFrags \tag{5}$$

5.2 Computing Optimal Fragments

Prior work [20, 22, 29] has shown how to find optimal fragmentations using dynamic programming in $O(kn^2)$ time and $O(kn)$ space, where n is the number of tuples in the table to be fragmented and k is the number of fragments. This solution works with arbitrary error functions, but requires the error function for a potential fragment starting at $Start(f_i)$ and ending at $End(f_i)$ to be computable in constant time. We next explain how our error function (Equation 4) can be computed in constant time.

Since our error function is the unnormalized variance of a fragment, it can be expressed in terms of the squared sum and sum of squares of each fragment (see Appendix B for a derivation):

$$Err(f_i) = \sum_{x=Start(f_i)}^{End(f_i)} V(x)^2 - \left(\sum_{x=Start(f_i)}^{End(f_i)} V(x) \right)^2 \tag{6}$$

We can thus compute the error of our potential fragment, $Err(f_i)$, using only the squared sum of values and the sum of squared values between $Start(f_i)$ and $End(f_i)$. Therefore, we precompute two arrays of size n, s and s_2, to store the cumulative sum and sum of squared $V(x)$ values for all tuples up to y, $0 \leq y \leq n$:

$$s[y] = \sum_{x=0}^{y} V(x) \qquad s_2[y] = \sum_{x=0}^{y} V(x)^2$$

Both sums can be computed trivially in linear time. The error function $Err(f_i)$ can thus be computed in constant time:

$$Err(f_i) = (s_2[End(f_i)] - s_2[Start(f_i)]) - (s[End(f_i)] - s[Start(f_i)])^2$$

Using these precomputed values, the dynamic programming scheme of [29] can be used to find an optimal fragmentation. However, the time complexity of $O(kn^2)$ and the space complexity of $O(nk)$ may be prohibitive for very large databases. In the next section, we present a greedy approximation for optimizing Equation 5.

5.3 Greedy Fragmentation

For very large databases, computing the optimal fragmentation could be time and space prohibitive. For this case, we propose a greedy strategy based on successively splitting and merging together fragments in a way that greedily minimizes the fragment error (Equation 4) at each step.

Given a cap on the maximum number of fragments to create, $maxFrags$, our greedy fragmentation consist of two procedures:

- When the number of fragments is less than $maxFrags$, we *split* one fragment into two in a way that maximizes uniformity within the resulting fragments.
- When the number of fragments is equal to $maxFrags$, we *merge* three adjacent fragments into two fragments in a way that maximizes uniformity within the resulting fragments.

The merging and splitting procedures are executed at user-specified time intervals. Both procedures make greedy decisions that minimize error and help adapt the fragmentation scheme to dynamic workloads. Intuitively, the splitting procedure splits the

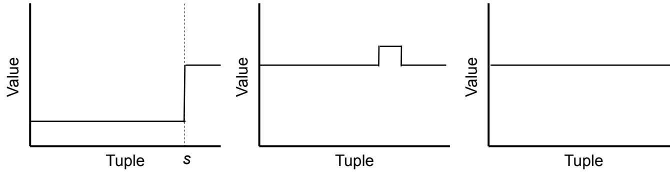


Figure 4: When considered, these three adjacent fragments will be joined into two fragments, split at the point marked by S.

most advantageous fragment, increasing the number of fragments by one and decreasing the error. However, once the splitting procedure has created *maxFrags* fragments, no new fragments can be created, but the workload may still shift over time. The merging procedure is then used to decrease the number of fragments by one – it selects the three fragments that, when combined into two, increase the error the least. This allows the splitting procedure to find new splits that are beneficial to the shifted workload.

5.3.1 Fragment Splitting. Intuitively, splitting a fragment at the right point can reduce the unnormalized variance by creating two new fragments where the values of each tuple in each new fragment are closer to the fragment's mean. For example, the fragment in Figure 3 could be split at the point between c_1 and c_2 to create two new fragments with significantly lower unnormalized variance.

In order to split one fragment into two, we first determine the optimal splitting point, $Split(f_i)$, for each fragment $f_i \in F$. Then, we select the fragment for which the optimal splitting point produces the greatest reduction in error. Formally, we define $Split(f_i)$, the optimal splitting point for a fragment f_i, as the split point that would result in the minimum sum of error:

$$Split(f_i) = \min_p \left[Err(f_j) + Err(f_k)\right] \text{ s.t.} \quad (7)$$
$$Start(f_j) = Start(f_i), End(f_j) = p$$
$$Start(f_k) = p, End(f_k) = End(f_i)$$

To find this split point, one can track the sum and the sum of the squared value $V(x)$ for all tuples with an index (position within the fragment) less than the currently considered point, and for all tuples with an index greater than the currently considered point. These values can be used to compute the quality of a potential split using Equation 6. This split point can be efficiently computed in constant space and $O(n)$ time, where n is the number of tuples in the fragment. We give the precise algorithm in Appendix C.

After computing the optimal split points, $Split(f_i)$, for each $f_i \in F$, we select the fragment for which the split produced the largest reduction in error.[2] That is, we split the fragment f_i into the fragments f_j and f_k for which $Err(f_i) - (Err(f_j) + Err(f_k))$ is maximized. While greedy, each split operation is guaranteed to reduce the sum of unnormalized variance [10] of the fragmentation. If the fragmentation scheme is already optimal, the split will leave the sum of the unnormalized variance unchanged.

5.3.2 Fragment Merging. Simply using the splitting procedure until the maximum number of fragments have been created is not enough to enable NashDB to adapt to changes in workload. Thus, when the maximum number of fragments have been created, NashDB recombines fragments with similar means so that more advantageous splits can be found. Since adjacent tuples are likely to have similar value, NashDB seeks to combine adjacent fragments where the mean value of both fragments are similar.

A simple strategy of combining pairs of adjacent fragments leaves room for improvement. Consider the three adjacent fragments in Figure 4. The second and third fragments have similar mean value, so a simple strategy would combine them. However, the combined second and third fragments have a mean value significantly different from the first fragment's mean. Even though the tuples on the right-hand side of the first fragment could be advantageously combined with the second fragment, the simple fragment joining strategy may never combine them.

Thus, we consider merging three adjacent fragments into two. Our merging procedure is motivated by the fact that the best single split point (Equation 7) can be found in linear time, whereas finding the optimal split point for turning $k \geq 4$ adjacent fragments into $k - 1$ fragments would require quadratic time.

Intuitively, NashDB's fragment merging procedure works by first measuring the unnormalized variance of joining together each window of three adjacent fragments into two hypothetical fragments, and then selecting the triplet of adjacent fragments that led to the best observed unnormalized variance. This results in one fewer total fragments. Formally, we check each adjacent set of three fragments, f_i, f_j and f_k, and find the point that optimally divides the three fragments into two new fragments, f_α and f_β.

$$Merge(f_i, f_j, f_k) = \min_p \left[Err(f_\alpha) + Err(f_\beta)\right] \text{ s.t.}$$
$$Start(f_\alpha) = Start(f_i), End(f_\alpha) = p$$
$$Start(f_\beta) = p, End(f_\beta) = End(f_k)$$
$$Start(f_i) < Start(f_j) < Start(f_k)$$

We then select the triplet of fragments which, when joined into two fragments, maximizes the decrease (or, when this is not possible, minimizes the increase) of the total sum of the errors. Specifically, we select the triplet for which $Err(f_i) + Err(f_j) + Err(f_k) - \left[Err(f_\alpha) + Err(f_\beta)\right]$ is minimized. This will not always decrease the sum of unnormalized variance, but this will always decrease the number of fragments by one. If the number of fragments before joining was *maxFrags*, this allows the splitting procedure (Section 5.3.1) to be performed again. As workloads shift, this ensures that NashDB can continuously adapt its fragmentation scheme.

6 REPLICATION

Given any fixed fragmentation scheme (optimally or greedily generated), NashDB decides (1) how many replicas of each fragment to create, (2) how many cluster nodes to provision, and (3) how to allocate replicas onto those nodes. We refer to these decisions collectively as a *cluster configuration.* Intuitively, fragments with higher value (fragments which are used by high-value queries, or a large number of low-value queries) should be replicated more than fragments with lower value. Our goal is to identify a configuration such that the demand for a fragment (the fragment values) is balanced against the cost of storing these fragments (the number of replicas of these fragments and their associated storage costs).

[2] In practice, to avoid unnecessary splitting, one might wish only to split a fragment if the reduction in unnormalized variance is sufficiently large.

Many different configurations may balance the number of replicas and the cost of storing each replica, but we aim for a configuration that is in *Nash equilibrium*. Intuitively, a solution is in Nash equilibrium if and only if no single node can unilaterally increase its profit by dropping, adding, or swapping replicas. While previous work [41] used economic models to offer similarly balanced replication schemes, their approach relies on market simulation and hence *slowly* reaches Nash equilibrium with high probability given sufficient time. A major advantage of NashDB lies in its ability to compute a converged solution directly, without requiring a costly market simulation. Our algorithm assumes a fixed fragmentation scheme as an input, and will find a Nash equilibrium for any fragmentation scheme.

Nash equilibrium Let us assume a cluster where each node has some usage cost $Cost$ per unit time (e.g., rent cost), as well as some fixed disk capacity $Disk$. Then the average cost of disk storage per time period is $\frac{Cost}{Disk}$. For simplicity, we assume that the cost and disk space of all nodes are equal, but our techniques can be easily extended to work with non-uniform costs and disk sizes.

Given a set of fragments G to replicate on an elastic cluster, the *expected cost* of storing a replica of fragment $f_i \in G$ with size $Size(f_i)$ on a cluster node is:

$$C(f_i) = Size(f_i) \times \frac{Cost}{Disk}$$

The *expected income* per replica of a fragment f_i, $I(f_i)$, is then the expected income of the fragment f_i over a window of scans W, divided by the number of replicas for that fragment, $Replicas(f_i)$. Hence, the more replicas are available, the lower their value is to the nodes. This expected income is defined as:

$$I(f_i) = |W| \times \frac{Value(f_i)}{Replicas(f_i)}$$

Finally, ignoring the cost of unused space, we define the profit of a node $m_i \in M$ storing replicas of a set of fragments $G_i \subset G$ as:

$$Profit(m_i, G_i) = \sum_{f_i \in G_i} I(f_i) - C(f_i) \tag{8}$$

Definition 6.1. We say that a cluster configuration is in Nash equilibrium if and only if all of the following hold: (1) no node can remove a fragment to gain a profit, (2) no node can add a fragment to gain a profit, (3) no node can swap one fragment for another and gain a profit, and (4) no new node can find any set of fragments such that it produces a profit if added in the cluster. This definition is formalized in Appendix D.

We next express NashDB's algorithm for finding a cluster configuration that meets Definition 6.1, followed by a proof of the algorithm's correctness. Our algorithm works in two parts. First, given a set of fragments G to replicate, we determine the number of replicas, $Replicas(f_i)$, for each fragment $f_i \in G$. We replicate each fragment f_i such that *any* node owning one of the replicas of f_i will make a profit, but if a single extra replica of f_i is created, *no* node owning a replica of f_i will make a profit. Second, we find an assignment of these replicas to cluster nodes.

Number of Replicas We define $Ideal(f_i)$ as the largest value of $Replicas(f_i)$ such that the profit from owning a replica of f_i is at least zero, but adding an extra replica would cause the profit to go below zero. Thus, $Ideal(f_i)$ is equal to:

$$\max_{Replicas(f_i)} |W| \times \frac{Value(f_i)}{Replicas(f_i)} - Size(f_i) \times \frac{Cost}{Disk} \geq 0$$

It is trivial to show that the ideal number of replicas to create for a particular fragment is equal to the total revenue earned by the fragment divided by the cost of the fragment:

$$Ideal(f_i) = \left\lfloor \frac{|W| \times Value(f_i)}{Size(f_i) \times \frac{Cost}{Disk}} \right\rfloor = \left\lfloor \frac{|W| \times Value(f_i) \times Disk}{Size(f_i) \times Cost} \right\rfloor \tag{9}$$

Intuitively, this formula states that when any of (1) the number of scans per unit time, (2) the average value of the tuples in a fragment, or (3) the amount of disk space available increases (*ceteris paribus*), the number of replicas should increase. Additionally, if the size of the fragment or the cost of a node increases (*ceteris paribus*), the number of replicas should decrease.

Replica Allocation The formulation of profit given in Equation 8 does not penalize nodes for unused disk space (it assumes each node only pays for used space). However, costs are normally incurred regardless of how much local disk space is used. In order to minimize wasted space, we will seek the tightest possible packing of the selected number of replicas into the fewest number of nodes.

We constrain ourselves to assignments where no node gets a duplicate of a fragment (since storing the same data twice on the same machine would not be useful). Finding an assignment of each replica to one of a minimal number of cluster nodes, thus achieving the minimum wasted free space, is equivalent to the class-constrained bin packing problem [40], which is NP-Hard. However, there are greedy heuristics with known error bounds [17, 45].

We employ the Best First Fit Decreasing (BFFD) algorithm of [45], which has an approximation factor of 2. BFFD works by maintaining a list of nodes (bins) M, initially with a single empty machine. BFFD first places the replicas of the fragment f_i for which $Replicas(f_i)$ is the highest. To place a replica, BFFD scans the current list of machines M and places the replica on the first machine on which the replica fits. If no such machine exists, a new one is created and added to the end of the list. Once all of the replicas of f_i are placed, BFFD moves on to the fragment with the next highest number of replicas. This process repeats until all replicas are placed.

This procedure may still leave unused space on some nodes. One could take advantage of this small amount of extra space to store additional replicas. This would ignore the economic model (the additional replicas are not profitable), and might produce additional overhead when transitioning the cluster.

Theorem 6.1. *The set of cluster nodes produced and the associated replication scheme is in Nash equilibrium.*

Proof. The intuition is that Equation 9 replicates each fragment such that each replica's profit is greater than or equal to zero, but creating one additional replica of any fragment will cause the profit to go below zero. Therefore, deleting or adding any fragment (including in a swap) cannot increase profit. We prove that each of four conditions given in Definition 6.1 are satisfied in Appendix E. □

7 CLUSTER TRANSITIONING

At user-specified time intervals, NashDB will use the up-to-date tuple value estimator to compute a new fragmentation and replication scheme. Once this new scheme is selected, NashDB transitions the cluster from the old scheme to the new scheme, which might include 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 the transition strategy that minimizes data transfer is critical to quickly transitioning between schemes.

The problem of finding the optimal transition strategy can be expressed as a bipartite graph matching problem. Each node in the old scheme is represented as a vertex in the first partition of the graph, and each node in the new scheme is represented as a vertex in the second partition. The edges between the vertices represent the cost of transitioning one node into another. The minimum-weight perfect matching, which can be found using the Kuhn-Munkres algorithm [23], represents the optimal transitioning strategy.

Formally, given initial nodes $\{m_1, m_2, \dots, m_k\} \in V$ and a new configuration $\{m'_1, m'_2, \dots, m'_j\} \in V'$, we create a bipartite graph $G = \{V \cup V', E\}$. G is complete, i.e. $E = \{(m, m') \mid m \in V, m' \in V'\}$. If the number of nodes in each partition are not equal (i.e, $|V| \neq |V'|$), we add "dummy vertices" to whichever partition (V or V') has fewer vertices so that $|V| = |V'|$. A dummy vertex in the first partition V represents a node that will be added (the new scheme has more nodes than the previous scheme), and a dummy vertex in the second partition V' represents a node that will be removed (the previous scheme has more nodes than the new scheme).

The edge weights, $w(m_i, m'_j)$, between two vertices $m_i \in V$ and $m'_j \in V'$ represent the total amount of data that would have to be transferred to transition m_i into m'_j. Intuitively, turning a node in the old scheme into a node in the new scheme with similar fragments should have a low cost, and turning a node in the old scheme into a node in the new scheme with dissimilar fragments should have a high cost. Let $Data(m_i) = \{t_a, t_b, \dots\}$ represent the tuples assigned to fragments on machine m_i, and let $Data(m'_j) - Data(m_i)$ be the tuples that are on m'_j but not on m_i. Then, the edge weights $w(m_i, m'_j)$ can be defined as:

- When $m_i \in V$ is not a dummy vertex and $m'_j \in V'$ is a dummy vertex, the edge (m_i, m'_j) represents entirely removing the node represented by m_i, so the weight of the edge is zero, i.e. $w(m_i, m'_j) = 0$.
- When $m_i \in V$ is a dummy vertex and $m'_j \in V'$ is not a dummy vertex, the edge (m_i, m'_j) represents provisioning a fresh node, so all the data required, $Data(m'_j)$, must be copied. Therefore, $w(m_i, m'_j) = |Data(m'_j)|$.
- When $m_i \in V$ and $m'_j \in V'$ are both non-dummy vertices, the edge (m_i, m'_j) represents turning m_i into m'_j. The new data that must be copied is any data that is on m'_j but not on m_i, so $w(m_i, m'_j) = |Data(m'_j) - Data(m_i)|$.

We define a *transition strategy*, T, to be a perfect matching of G: a set of edges such that each vertex appears in exactly one edge. The bold edges in Figure 5 shows one such perfect matching. An edge $(m_i, m'_j) \in T$ means that the machine represented by m_i is turned into the new machine represented by m'_j. Since T is a perfect

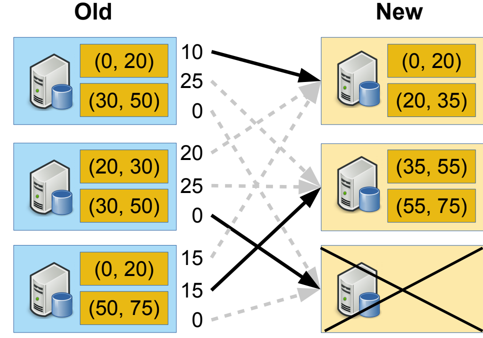


Figure 5: A perfect minimal weight bipartite matching between the old scheme (left) and the new scheme (right). The original top and bottom nodes transition, and the middle node is destroyed.

matching, every machine in the old scheme is turned into some machine in the new scheme (or deleted / created, if matched with a dummy vertex). Note that it is impossible for both $m_i \in V$ and $m'_j \in V'$ to be a dummy vertex, since dummy vertices are only added to the partition with fewer members (i.e., a dummy vertex can be added to either V or V', but not both).

The cost (number of tuples to be copied) for a strategy T is the sum of the edge weights, and we seek the optimal T. Letting $\mathbb{T}$ be the set of all possible perfect matchings, the optimal $T \in \mathbb{T}$ is:

$$\min_{T \in \mathbb{T}} \sum_{(m_i, m'_j) \in T} w(m_i, m'_j) \quad (10)$$

While there are $O(|V|!)$ possible perfect matchings, the Kuhn-Munkres algorithm [23] can find the minimal perfect matching in $O(|V|^3)$ time [43]. In our experiments, we found standard implementations [4] to be sufficiently fast even for thousands of nodes.

Example Figure 5 shows an example graph. The left-hand side represents the old configuration (S), containing three nodes. The right-hand side represents the new configuration (S'). Each edge weight represents the cost of turning the left-hand side node into the connected node on the right-hand side. For example, the edge weight between the two top-most nodes is 10, because 10 new tuples would need to be transferred. The edge weight between the middle two nodes is 25, because tuples 50-75 would have to be transferred onto the machine. The edge from all the machines on the left-hand side to the bottommost dummy vertex on the right-hand side has weight zero, since removing a node does not require any data transfer. The bold edges show a minimal perfect matching, and is thus a solution to Equation 10.

NashDB computes a new fragmentation and replication scheme and transitions to it based on a user-defined time interval. This interval must be tuned by the user. If the interval is too short, NashDB may initiate a large number of small but unnecessary data transfers. If the interval is too long, workload drift may cause significant performance degradation. We suggest setting the interval based on how frequently the tuple value estimation buffer cycles through data: for example, if the maximum number of scans held in the buffer represents an hour of data, we suggest transitioning the cluster every hour. We leave to future work the task automatically detecting when the cluster should be transitioned.

8 ROUTING DATA ACCESS REQUESTS

NashDB includes a range scan router module that strives to identify the best replicas to read when processing an incoming query. Previous approaches aimed to either minimize query span [24] (i.e., the total number of nodes used to fetch the input of a query) or data access time [42] (by load balancing the access of popular tuples across many nodes). Here, we present the **Max of mins** algorithm, which balances these two goals: **Max of mins** seeks to take advantage of highly-replicated, popular tuples to improve access latency, while increasing the data access span only when it is beneficial to the overall performance of the query.

When routing a range scan s of a query, the DBMS breaks down the range scan operation into the set of fragments to be fetched, $F(s)$, and then selects a replica of each required fragment. Let us denote as $E(s)$ the nodes that store at least one fragment in $F(s)$. NashDB makes routing decisions based on the wait time on these nodes (due to the bottleneck of disk access). Specifically, we assume that fragment access requests are queued on nodes and the time to read a fragment is proportional to the number of tuples in the fragment. Hence, the wait time to read a fragment from a node m_i, $Wait(m_i)$, is equal to the total number of tuples in the node's already-queued requests. Like previous work [9, 11, 26, 39], we note that, for OLAP workloads, these queues can be tracked with relatively little overhead, as scans tend to take a long time to process and thus dominate communication costs.[3]

To model the query span overhead, **Max of mins** adds an estimate of the penalty from increasing the span by one, ϕ, to the estimated wait time of any node not currently included in the span. A new node m_i is used for serving a scan only if doing so is beneficial despite the penalty ϕ.

Max of mins schedules fragment requests on the node with the shortest queue, in order based on the largest minimum possible wait time for access to a fragment. The maximum of the minimum wait times is selected because we assume queries can finish only after *all* required fragments have been fetched. Therefore, since the request whose minimum possible processing time is maximal is a bottleneck, we schedule it first.

Let $U(s, m_j)$ be a function that indicates if the node m_j has already been selected to process the range scan s. Formally, **Max of mins** schedules the fragment request that satisfies the following:

$$\max_{f_i \in F(s)} \left(\min_{m_j \in E(s)} Wait(m_j) + U(s, m_j) \times \phi \right) \quad (11)$$

The fragment request selected is scheduled on the node for which the wait time was minimal. Intuitively, **Max of mins** only increases the span when doing so is better than using any of the nodes currently selected to process the fragment read.

9 RELATED WORK

Previous work on fragmentation and replication have been diverse. Mariposa [41] assumes a fixed number of nodes which bid on queries and bargain in an open marketplace. Nodes split, sell, and replicate fragments. Mariposa adapts to changes and converges to a good fragmentation strategy over time. Unlike NashDB, Mariposa directly simulates a marketplace, creating overhead while slowly driving the system towards equilibrium. NashDB computes this equilibrium directly, avoiding the overhead of a market simulation.

DYFRAM [19] maintains access frequency histograms at each of a fixed number of nodes. Each node can choose to split or combine fragments based on a cost function, or replicate fragments from their neighbors. DYFRAM's primary goal is *decentralization*, making it more resilient to failures at the cost of increased overhead.

SWORD [24] and Schism [12] represents tuples and queries as vertices and edges of a hypergraph. Given a fixed number of nodes, a hypergraph is cut into n disjoint partitions, trying to break as few edges as possible. Each partition is assigned to a node. To fill excess space, tuples are replicated based on a heuristic to further decrease the number of broken edges. Both systems try to decrease network costs for distributed query processing, and thus consider replication only as a tool to decrease potential communication overhead, and not as a way to increase performance in general.[4]

E-Store [42] uses a thresholding approach in which tuples are either "hot" (accessed frequently) or "cold" (accessed infrequently). The hot and cold tuples are assigned to one of a fixed number of nodes. "Hot" tuples are replicated aggressively. E-Store adjusts the size of the cluster by adding or removing single nodes when the CPU usage of the cluster moves outside a threshold, after which fragmentation and replication strategies are recomputed. This approach is easy to implement, but responds only to CPU usage and raw access frequency, as opposed to dollar-cost and query priority.

Clay [39] combines the threshold approach with the hypergraph approach, actively migrating "hot" tuples, and tuples frequently accessed along with them, away from overloaded nodes. Clay does not handle data replication or cluster sizing.

Unlike NashDB, all of these solutions are agnostic to query prioritization. Modern DBMSes generally allow for some form of query priority, but existing solutions are limited to allocating compute resources (CPU, RAM, etc.) at query runtime [31].

Squall [16] is a system for performing pre-defined live migrations on main-memory partitioned databases, concerned with safety and efficiency for OLTP databases. In principle, Squall could be used to execute the transition plans created by NashDB (Section 7).

10 EXPERIMENTS

In this section, we evaluate NashDB experimentally. We built a prototypical NashDB on the AWS [2] cloud with `m2.large` EC2 instances. Our prototype executed SQL queries on PostgreSQL [6] and we replaced the physical access operators of PostgreSQL to route data requests through NashDB's scan router.

Workloads In our experiments we used two different types of workloads from several datasets (details in Appendix F):

(1) *Static workloads*: these workloads represent batch jobs in which a large number of queries are sent simultaneously. Here, the *TPC-H* workload consists of all query templates of the TPC-H [7] benchmark with a size of 1TB. The *Bernoulli* workload is a variation of the TPC-H workload that consists of simple range queries over the 1TB TPC-H fact table. It simulates a time-series analysis in which more recent tuples are accessed

[3] If scans were to be smaller, one could adapt "Power of 2" techniques [32, 35] to schedule scans (e.g., consider only two random nodes from the eligible set, and choose the node from that pair which satisfies our goal as captured by Equation 11).

[4] As noted in [12, 24], this applies strongly to OLTP workloads.

more frequently than older tuples: each range query ends at the last tuple, and the starting points are drawn from a binomal distribution, such that 95% of the queries access the last 1GB of data, 90% of the queries access the second to last 1GB of data, and $100 \times \frac{19}{20}^n$ percent of the queries access the nth from the last GB of data. Finally, we used a real-world workload (*Real data 1*) that represents a batch workload of 1000 queries over an 800GB database that is periodically executed to update analytic dashboards at a large corporation, which provided data on the condition of anonymity. We provide more information about this dataset in Appendix F.

(2) *Dynamic workloads*: these workloads represent online jobs, spanning 72-hour periods. The *Random* workload is synthetic, and represents a sequence of aggregated range queries with uniformly distributed start and end points over a TPC-H fact table. *Real data 1* and *Real data 2* represent real-world queries issued by analysts and other applications on a production database server at two large corporations that provided data on the condition of anonymity. The databases are 300GB and 3TB in size, and the workload includes 1,220 and 2,500 queries respectively. We provide more information about this dataset in Appendix F.

System Parameters Unless otherwise noted, we collect our value estimation statistics over a the scan window size of 50 scan requests. When applicable, the cluster transitioning algorithm was ran every hour. We include the overhead of cluster transitioning and scheduling in our cost and latency measurements.

10.1 Fragmentation

First, we evaluate NashDB's ability to find low-variance fragments. In Figure 6a and 6b, we compare the inner-fragment variance of several algorithms, which we describe next.

Fragmentation algorithms The *Optimal* algorithm computes optimal fragmentations using a dynamic programming scheme as described in Section 5.2. The *NashDB* algorithm uses the greedy splitting/joining approach we introduced in Section 5.3. The *DT* algorithm greedily searches for the best split point of the data, then recursively splits the resulting two halves until the maximum number of partitions have been created. This is equivalent to only running the "split" procedure of NashDB, and is similar to the CART [10] decision tree induction algorithm. The *Hypergraph* algorithm uses a hypergraph partitioning approach to create partitions that minimize query span, similar to SWORD [24]. This approach treats each tuple and each query as the vertices and edges, respectively, of a hypergraph. Standard hypergraph partitioning techniques are applied to give a fixed number of partitions with few edges spanning multiple partitions. Tuples are selected for replication to further decrease the number of broken edges ("Improved LMBR" in [24]). While this approach aims to minimize query span, and not to minimize the error (Equation 4) as NashDB does, we compare against it to highlight the difference in the fragmentation strategies selected by the two approaches. The *Naive* algorithm partitions the database into fragments of equal size.

Static workloads Figure 6a shows the performance of various partitioning algorithms on static workloads. To evaluate the performance of each algorithm, we run the static workload first and then measure the error (Equation 4) of each partitioning algorithm after the whole workload has been seen. For both synthetic workloads, the *DT* algorithm outperformed the *Hypergraph* and *Naive* methods. The *Bernoulli* dataset represents an adversarial input for the hypergraph approach, since the best k-cut of the graph will place the first $k-1$ tuples in their own partitions, and the remaining tuples will be grouped together in one partition. On the real-world workload, the *Hypergraph* approach outperformed both the *Naive* and *DT* approaches, implying that the good min-cut partitionings of the query span hypergraph can correspond with low-variance fragmentation strategies in real world data. In each case, the *NashDB* algorithm is within 50% of the *Optimal* partitioning, and, in each case, the *NashDB* algorithm matches or outperforms all the other heuristics.

Dynamic workloads Figure 6b shows the performance of the fragmentation algorithms on dynamic workloads. The fragmentation scheme is recalculated after each query, and the sum of the total error over each fragmentation scheme is computed. The significantly higher gap (when compared to the static case) between the *Optimal* and the *NashDB* algorithm shows how the suboptimal nature of the split/join heuristics can become significant over time. The difference between the partitions created by *NashDB* and *DT* (a factor of approximately two) shows the importance of being able to split *and* join fragments as workloads progress and change. By employing both a splitting and joining heuristic, *NashDB outperforms other heuristics on dynamic data by approximately a factor of two.*

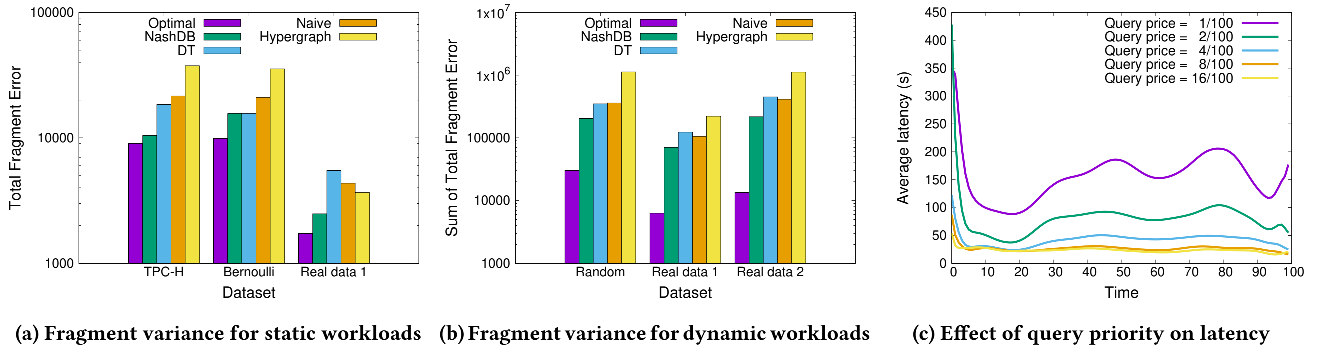
Value estimation overhead We also measured the overhead of the tuple value estimation tree (Section 4.2). With a window size of $|W| = 50$ scan requests, the size of the value estimation tree and buffer was always under 1 kilobyte, and the access time was always under 5ms. Increasing the scan window size to 1000 scans (a value significantly higher than necessary), the size was always under 4 kilobytes, again with access times less than 5ms. *We conclude that the tuple value estimation tree is able to maintain its online estimates with sufficiently low overhead.*

10.2 Query Prioritization

NashDB allows users to specify a price for each query sent, with higher prices resulting in lower query latencies. Figure 6c shows the average latency over time of TPC-H [7] workloads running on NashDB. Here, all queries are given the same price, and we vary this price. When the price of each query in the batch is set to 1/100 of a cent, the average latency has high variance and mean. As the price per query increases, both the mean and variance decrease, as this increase in query value causes NashDB to generate additional replicas and to provision more nodes. A higher resource usage cost is incurred, but the query execution times are improved.

NashDB also supports workloads with mixed query priorities. Using the same TPC-H workload, we varied the price of all instances of TPC-H template #7 (the template with latency closest to the average latency for the whole workload) from 1/100 of a cent to 16/100 of a cent, while keeping the priority of all other queries fixed at 1/100 of a cent. As a result, the average latency of instances of template #7 fell significantly (by a factor of four), while the queries with fixed priority saw only a modest (10%) improvement (Figure 9a in Appendix G).[5] This modest improvement is due to the fact that

[5] All other templates exhibit similar behavior. We plot only #7 due to space constraints.



(a) Fragment variance for static workloads **(b) Fragment variance for dynamic workloads** **(c) Effect of query priority on latency**

Figure 6: Variance and query priority

the other queries can often still take advantage of the extra replicas created for higher priority queries. *NashDB's responsiveness to query priorities provides a single knob to adjust performance and cost, at both the workload and query level.*

10.3 Cost and Performance Evaluation

Here, we evaluate various cost and latency properties of NashDB and compare it to other heuristics. First, we compare NashDB with the *Hypergraph* approach we described in Section 10.1. This approach is tuned by adjusting the number of partitions created by the hypergraph, which correspond to the number of nodes used. Increasing the number of partitions increases the cluster size, which increases cost while decreasing latency. We also compare against a thresholding algorithm, labeled *Threshold*. Like in E-Store [42], this algorithm first partitions the data into "hot" and "cold" sets of tuples, and then distributes those tuples over a number of nodes.[6] Since the E-Store system was designed for OLTP databases without replicated tuples, we additionally replicate each tuple in linear proportion to the tuple's access frequency. This approach is tuned by adjusting the size of the cluster, i.e. the number of nodes. More nodes incur a higher cost, but leads to better query performance.

Pareto Analysis In this section, we compare the end-to-end performance of NashDB against the *Hypergraph* and *Threshold* heuristics on our static workloads. To compare these systems, we vary their parameters through a wide range of reasonable values and plot the resulting monetary cost and average query latency in Figure 7. For NashDB, we varied the query priority between 0 and 128, by steps of 0.005. For *Hypergraph*, we vary the number of partitions between 4 (the minimum, given our dataset size) and 400. For *Threshold*, we vary the number of nodes between 4 (the minimum) and 400. Figures 7a, 7b, 7c show the latency and cost achieved by each of these algorithms for static each workload (i.e. the *production possibilities* for each algorithm). A point is *Pareto optimal* if there is no other point that has both an equal or lower latency and an equal or lower cost. The set of all Pareto optimal points is the *Pareto front*.

For both synthetic workloads, shown in Figure 7a and 7b, all the points on the Pareto front are from NashDB. In other words, no configurations of the *Threshold* or *Hypergraph* approaches were found for which NashDB does not provide a equally good or better performance for equally good or lower cost. For the real-world workload (Figure 7c), a single configuration of the hypergraph algorithm is on the Pareto front. Setting this point aside, we conclude that for these workloads, there exists an NashDB configuration that *strictly dominates* any configuration of the other two systems.

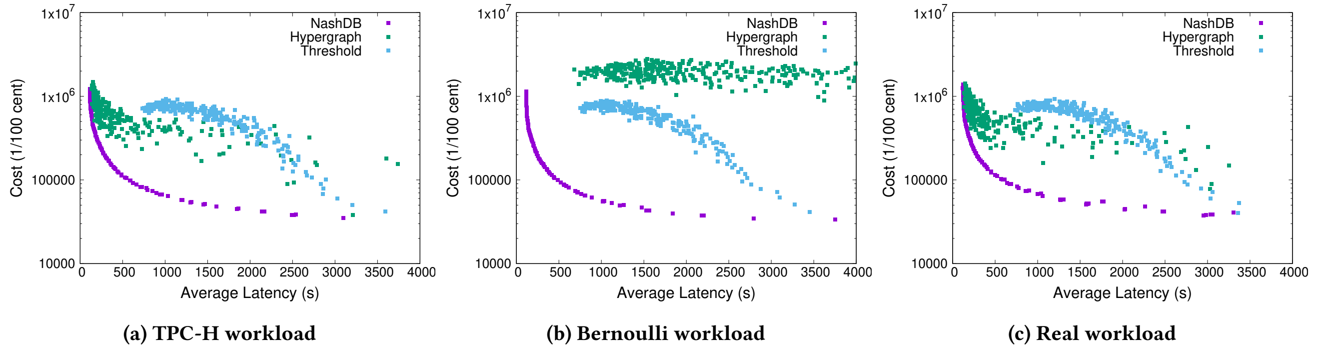
Fixed Latency/Cost We compare NashDB, *Hypergraph*, and *Threshold* in terms of monetary cost and data transfer overhead after adjusting each algorithm to achieve identical average latency on our dynamic datasets. During this experiment, the transitioning techniques for all three system ran hourly. The monetary cost and latency overheads of these transitions, as well as overhead included from query routing, are included in the results.

Figure 8a shows that *NashDB can achieve the same average query latency at significantly lower cost than* Threshold *or* Hypergraph. For the *Real data 2* dataset, the cost of NashDB is approximately 15% lower that *Hypergraph*. Furthermore, when keeping the monetary cost fixed, NashDB allows for lower data access latency as shown in Figure 8b: with a fixed total cost of $20, NashDB provides an average query latency that is 20% – 50% lower than other approaches. The 95% and 99% tail latencies (Figure 10 in Appendix G.1) also confirm that a high percentage of queries benefit from NashDB, i.e., have lower latency compared to the other techniques.

Transitioning Overhead NashDB incurs significantly higher data transfer overhead when transitioning between two different configurations (Figure 9b in Appendix G). *Hypergraph* has the lowest data transfer cost as it is designed to optimize for that overhead. *Threshold* comes second in terms of transition overhead. However, both *Threshold* and *Hypergraph* still result in higher total cost and latency than NashDB, as shown in Figure 8a and 8b (note that the latency measurements include transition overhead). Hence, *despite the fact that NashDB transfers more data during transitions, it is still Pareto optimal with respect to monetary cost and query latency.*

We note that the average data transferred per hour by NashDB is very small (< 200MB) and the average time to perform a transition was under two seconds. To put this number in context, for the *Real data 2* dataset (the dataset with the largest transfer overheads), the per-minute data throughput of NashDB varied between 255GB/m and 265GB/m, representing a variance of less than 5%. Similar observations can be made from the other two datasets (Figure 11 in Appendix G.2). Hence, *the transitioning overhead has a minimum impact on the throughput of the system.*

[6] We use the "Greedy extended" algorithm proposed in [42].

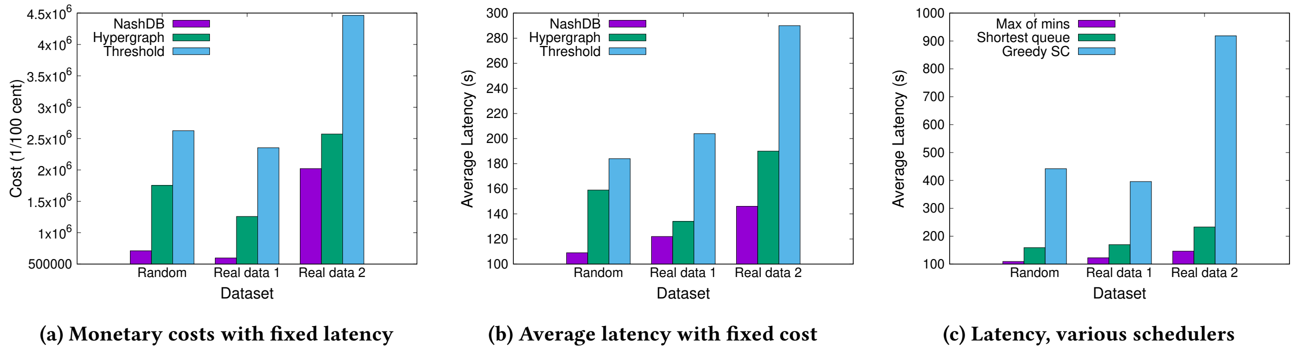


(a) TPC-H workload

(b) Bernoulli workload

(c) Real workload

Figure 7: Cost and latency tradeoffs for static workloads.



(a) Monetary costs with fixed latency

(b) Average latency with fixed cost

(c) Latency, various schedulers

Figure 8: Experiments on dynamic workloads

10.4 Routing Evaluation

Next we evaluate **Max of mins**, the algorithm for routing fragment read requests to replicas, by comparing to two other approaches: (a) *Greedy SC*, which minimizes query span of each query by repeatedly selecting the node with highest number of remaining tuples [24], and (b) *Shortest queue*, which minimizes fragment request latency by scheduling each fragment request on the node with the shortest queue. We compare these algorithms for dynamic query workloads.

For **Max of mins** we set ϕ, the cost of increasing the query span by one, to $\phi = 350ms$ after performing a simple experiment on AWS [2]. Each node reports its queue length whenever a fragment read completes. This overhead is included in all of our experiments.

Figure 8c shows that, at approximately the same cost, **Max of mins** delivers substantially shorter latency than either *Shortest queue* or *Greedy SC*. This advantage in latency comes from taking into account *both* the queue size and the query span. The significantly higher latency incurred by the *Greedy SC* is due to nodes with particularly popular *sets* of fragments becoming performance bottlenecks – other nodes sit idly while the few nodes with popular fragment sets process their queues.

We measured the average span (number of nodes used per query) of the produced schedules (Figure 9c in Appendix G). As expected, *Greedy SC* has the lowest average query span (1.1 on "Real Data 2"). *Shortest queue* has the highest query span (3.3), because the algorithm does not take query span into account at all. **Max of mins**, which increases query span only when there is a latency benefit, has an average query span of 1.5, which is significantly lower than *Shortest queue*, but slightly higher than *Greedy SC*. **Max of mins** *strikes a balance between span and wait time, which provides both low-latency and low-cost schedules.*

11 CONCLUSIONS

This work introduced NashDB, a data distribution framework for OLAP workloads that relies on economic theory to automatically fragment, replicate, and allocate replica on an elastic cluster. NashDB takes an input query priorities (expressed as prices) and data access requests, and strives to optimally balance replica supply to workload demands. Specifically, it tightly couples an economic model to (a) an automatic fragmentation algorithm that adapts to changes in "importance" (value) of tuples and (b) replication and replica allocation mechanisms that are guaranteed to produce a data distribution scheme that is in Nash equilibrium. We have shown that a prototypical implementation of NashDB displays *Pareto dominate* performance (in terms of cluster usage cost and query execution latency) on a number of synthetic and real-world workloads.

Moving forward, we plan to investigate applications of NashDB's model to OLTP workloads, as well as integrating more advanced query scheduling algorithms into the system. We are also considering how economic models could provide performance guarantees.

ACKNOWLEDGMENTS

This research was funded by NSF IIS 1253196.

REFERENCES

[1] Amazon EBS, https://aws.amazon.com/ebs/.
[2] Amazon Web Services, http://aws.amazon.com/.
[3] Google Cloud Platform, https://cloud.google.com/.
[4] JGraphT, http://jgrapht.org/.
[5] Microsoft Azure Services, http://www.microsoft.com/azure/.
[6] PostgreSQL database, http://www.postgresql.org/.
[7] The TPC-H benchmark, http://www.tpc.org/tpch/.
[8] Adelson-Velsky, G., et al. An algorithm for the organization of information. *Soviet Mathematics '62*.
[9] Azar, Y., et al. Cloud scheduling with setup cost. In *SPAA '13*.
[10] Breiman, L., et al. *Classification and Regression Trees*.
[11] Chi, Y., et al. iCBS: Incremental Cost-based Scheduling Under Piecewise Linear SLAs. *PVLDB '11*.
[12] Curino, C., et al. Schism: A workload-driven approach to database replication and partitioning. *VLDB '14*.
[13] Das, S., et al. ElasTraS: An Elastic, Scalable, and Self-managing Transactional Database for the Cloud. *TODS '13*.
[14] Dean, J., et al. The Tail at Scale. *Comm. ACM '13*.
[15] Edelsbrunner, H. A New Approach to Rectangle Intersections. *Journal of Computer Math '83*.
[16] Elmore, A. J., et al. Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. In *SIGMOD '15*.
[17] Epstein, L., et al. Class constrained bin packing revisited. *Theoretical Computer Science '10*.
[18] Hall, M., et al. The WEKA Data Mining Software: An Update. *SIGKDD '09*.
[19] Hauglid, J. O., et al. DYFRAM: Dynamic fragmentation and replica management in distributed database systems. *Distrib Parallel DB '10*.
[20] Jagadish, H., et al. Optimal Histograms with Quality Guarantees. *VLDB '98*.
[21] Jalaparti, V., et al. Bridging the Tenant-provider Gap in Cloud Services. In *SoCC '12*.
[22] Konno, H., et al. Best piecewise constant approximation of a function of single variable.
[23] Kuhn, H. W. The Hungarian method for the assignment problem. *NRLQ '55*.
[24] Kumar, K. A., et al. SWORD: Workload-aware Data Placement and Replica Selection for Cloud Data Management Systems. *VLDB Journal '14*.
[25] Lamb, A., et al. The Vertica Analytic Database: C-store 7 Years Later. *VLDB '12*.
[26] Leitner, P., et al. Cost-Efficient and Application SLA-Aware Client Side Request Scheduling in an Infrastructure-as-a-Service Cloud. In *CLOUD '12*.
[27] Li, J., et al. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *SOCC '14*.
[28] Lolos, K., et al. Elastic management of cloud applications using adaptive reinforcement learning. In *Big Data '17*.
[29] Mahlknecht, G., et al. A scalable dynamic programming scheme for the computation of optimal k -segments for ordered data. *Information Systems '17*.
[30] Marcus, R., et al. WiSeDB: A Learning-based Workload Management Advisor for Cloud Databases. *VLDB '16*.
[31] McWherter, D., et al. Priority mechanisms for OLTP and transactional Web applications. In *ICDE '04*.
[32] Mitzenmacher, M. The Power of Two Choices in Randomized Load Balancing. *IEEE Parallel Distrib. Sys. '01*.
[33] Nash, J. F. Equilibrium points in n-person games. *PNAS '50*.
[34] Ortiz, J., et al. PerfEnforce Demonstration: Data Analytics with Performance Guarantees. In *SIGMOD '16*.
[35] Ousterhout, K., et al. Sparrow: Distributed, Low Latency Scheduling. In *SOSP '13*.
[36] Pedregosa, F., et al. Scikit-learn: Machine Learning in Python. *JMLR '11*.
[37] Rogers, J., et al. A generic auto-provisioning framework for cloud databases. In *ICDEW '10*.
[38] Serafini, M., et al. Accordion: Elastic Scalability for Database Systems Supporting Distributed Transactions. *VLDB '14*.
[39] Serafini, M., et al. Clay: Fine-grained Adaptive Partitioning for General Database Schemas. *VLDB '16*.
[40] Shachnai, H., et al. Polynomial time approximation schemes for class-constrained packing problems. *J. of Scheduling '01*.
[41] Sidell, J., et al. Data replication in Mariposa. In *ICDE '96*.
[42] Taft, R., et al. E-Store: Fine-grained Elastic Partitioning for Distributed Transaction Processing Systems. *VLDB '15*.
[43] Tomizawa, N. On some techniques useful for solution of transportation network problems. *Networks '71*.
[44] Welford, B. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics '62*.
[45] Xavier, E. C., et al. The class constrained bin packing problem with applications to video-on-demand. *Theoretical Computer Science '08*.
[46] Xiong, P., et al. Intelligent management of virtualized resources for database systems in cloud environment. In *ICDE '11*.

A VALUE ESTIMATION TREE OPTIMIZATION

It is not necessary to store both the $S(n_i)$ and $E(n_i)$ values for each node. Since α is always updated by adding the quantity $S(n_i) - E(n_i)$, we can store this value, $\Delta(n_i)$, instead. When inserting or removing a scan s_i, two nodes are retrieved or created, n_1 and n_2, corresponding to $Start(q_i)$ and $End(q_i)$. When inserting, $\Delta(n_1)$ is updated by adding $Price(q_i)$ and $\Delta(n_2)$ is updated by subtracting $Price(q_i)$ from $\Delta(n_2)$. The addition and subtraction are swapped for removing a scan.

B ERROR FUNCTION

Here, we show how Equation 4 can be computed using only the sum of squares and square sum, as given in Equation 6. Since our error function is the unnormalized variance, it can be computed by taking the variance of a fragment, $Var(f_i)$, and multiplying $Var(f_i)$ by the size of the fragment. Thus, letting $a = Start(f_i)$, $b = End(f_i)$, and $E[f(x)]_\alpha^\beta$ be the expected value of $f(x)$ over the interval (α, β):

$$
\begin{aligned}
Err(f_i) &= \sum_{x=a}^{b} \left(V(x) - \frac{\sum_{i=a}^{b} V(i)}{b-a} \right)^2 = (b-a) \times Var(f_i) \\
&= (b-a) * \left(E[V(x)^2]_a^b - \left(E[V(x)]_a^b \right)^2 \right) \\
&= \sum_{x=a}^{b} V(x)^2 - \left(\sum_{x=a}^{b} V(x) \right)^2
\end{aligned}
$$

C FINDING OPTIMAL SPLIT POINTS

Algorithm 2 Finding the best split point

```
1:  function FindSplit(f_i)
2:      α ← V(Start(f_i))
3:      α_2 ← V(Start(f_i))^2
4:      β ← Σ_{x=Start(f_i)+1}^{End(f_i)} V(x)
5:      β_2 ← Σ_{x=Start(f_i)+1}^{End(f_i)} V(x)^2
6:      BestPoint ← 0
7:      BestPointVal ← ∞
8:      for PotSplit ← Start(f_i) + 1 to End(f_i) do
9:          CurrScore ← (α_2 − α^2) + (β_2 + β^2)
10:         if CurrScore < BestPointVal then
11:             BestPointVal ← CurrScore
12:             BestPoint ← PotSplit
13:         end if
14:         α ← α + V(PotSplit)
15:         α_2 ← α_2 + V(PotSplit)^2
16:         β ← β − V(PotSplit)
17:         β_2 ← β_2 − V(PotSplit)^2
18:     end for
19:     return BestPoint
20: end function
```

Algorithm 2, a modified version of Welford's algorithm [44] inspired by CART [10], computes optimal binary split point in a fragment in linear time and constant space. The variables α and α_2 are used to track of the sum and squared sum of $V(x)$ for the

Name	Use	DB Size	# Queries	Med. read	Min. read	Med. # results
Static "Real data 1"	Dashboard	800GB	1000	600GB	5G	10k
Dynamic "Real data 1"	Descriptive analytics	300GB	1220	50GB	< 1GB	20k
Dynamic "Real data 2"	Predictive analytics	3TB	2500	450GB	80KB	< 2k

Table 1: Statistics about datasets

values to the left of the current point, and the variables β and β_2 are used to track the respective values to the right of the current point. Lines 2-5 initialize these variables, and lines 14-17 update them at each iteration. Line 9 uses α, α_2, β, and β_2 to compute the sum of the error of the fragments that would result from splitting f_i at the potential splitting point $PotSplit$. At the end of the loop, the best splitting value is known and returned. Similar algorithms have been used in various implementations [18, 36] of regression trees [10].

While Algorithm 2 iterates over every tuple for simplicity, the optimal split point can only be at a point where the value ($V(x)$) changes [10, 29]. Thus, it can be optimized to only iterate over the "chunks" of tuples and tuple values generated by Algorithm 1 by checking only the beginning and end of each chunk, and adding the appropriate summed values between each step (the summed values can easily be computed by multiplying $V(x)$ by the size of the chunk).

D NASH EQUILIBRIUM DEFINITION

Definition 6.1 can be formalized as follows. A set of nodes M is in Nash equilibrium if all the following conditions hold. For each $m_i \in M$, let G_i be the set of fragments assigned to the machine m_i. Conditions 1-3 are formalized as:

$$\begin{aligned}
&\forall m_i \in M, \neg\exists f, g \in G \text{ s.t.} \\
&Profit(m_i, G_i) < Profit(m_i, G_i - \{f\}) \quad \vee \\
&Profit(m_i, G_i) < Profit(m_i, G_i \cup \{f\}) \quad \vee \\
&Profit(m_i, G_i) < Profit(m_i, G_i \cup \{f\} - \{g\})
\end{aligned}$$

Condition 4 is formalized as:

$$\neg\left(\exists m_i \notin M, \exists b \subset G \text{ s.t. } (Profit(m_i, b) > 0)\right)$$

E PROOF OF THEOREM 6.1

An extended proof of Theorem 6.1 is given below.

Proof. Equation 9 replicates each fragment such that each replica profit is greater than or equal to zero, but creating one additional replica of any fragment will cause the profit to go below zero. By replicating each fragment according to Equation 9, each of these conditions is satisfied.

- **Condition 1**: no node can remove a fragment and gain a profit. By Equation 9, each replica created will be profitable. Therefore, removing a fragment will decrease profit.
- **Condition 2**: no node can add a fragment and gain a profit. By Equation 9, each fragment is replicated so that a single additional replica would cause the profit for all replicas to become negative.
- **Condition 3**: no node can swap one fragment for another and gain a profit. Deleting a replica of f_i and then adding another replica of fragment f_j will not be profitable, since condition 2 shows that adding a replica of f_j will decrease profit, and removing a replica of f_i will (1) decrease profit as in condition 1, and (2) will not change the value of a replica of f_j.
- **Condition 4**: no *new* node can enter the market and make a profit. Any new node would have to add a replica of a fragment, which, by condition 2, would not be profitable.

□

F DATASET DESCRIPTIONS

The static "Real data 1" and dynamic "Real data 1" and "Real data 2" come from three different corporations that executed their queries against a Vertica [25] database, but all of our experimental results were produced using Postgres and the NashDB prototype. A few queries using Vertica-specific features were rewritten into equivalent Postgres-compatible SQL queries.

Since the static "Real data 1" and dynamic "Real data 1" and "Real data 2" datasets contain sensitive information, we can only provide summary statistics about them. Table 1 shows the database size, number of queries, median data read by a query, minimum data read by a query, and the median number of results returned by queries.

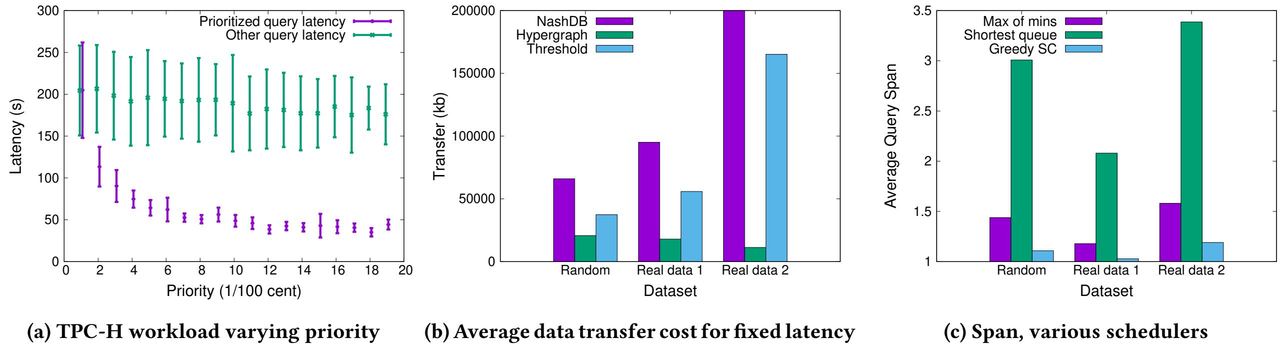
G ADDITIONAL EXPERIMENTS

G.1 Tail latency

While Section 10 investigated NashDB's performance in terms of average latency, here we additionally analyze NashDB's performance based on *tail latency* [14, 27], e.g. the 95th and 99th percentile of latency. Figure 10 shows the tail latency performance for NashDB, *Hypergraph*, and *Threshold* when each algorithm was adjusted to achieve an identical monetary cost (matching the parameters for the experiment described in Section 10.3). In addition to having superior average latency (Figure 8b), *NashDB exhibits superior tail latency on all three datasets.*

G.2 Throughput over time

In addition to the discussion of NashDB's overhead costs presented in Section 10.3, we additionally provide throughput-over-time graphs for the three real-world workloads in Figure 11. For the dynamic "real data 1" workload, throughput varied between 14 and 18GB/m. For the dynamic "real data 2" workload, throughput varied between 255 and 256GB/m. For the dynamic "random" workload, throughput varied between 112 and 118 GB/m, and exhibited relatively lower variation because each query is random (and thus there is no pattern over time, so the distribution stays constant



(a) TPC-H workload varying priority

(b) Average data transfer cost for fixed latency

(c) Span, various schedulers

Figure 9

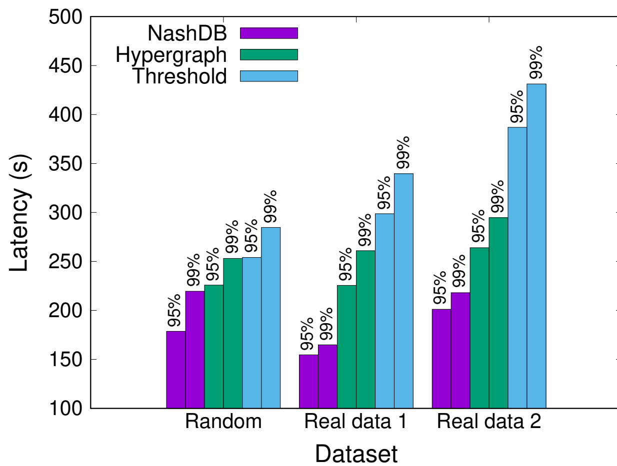
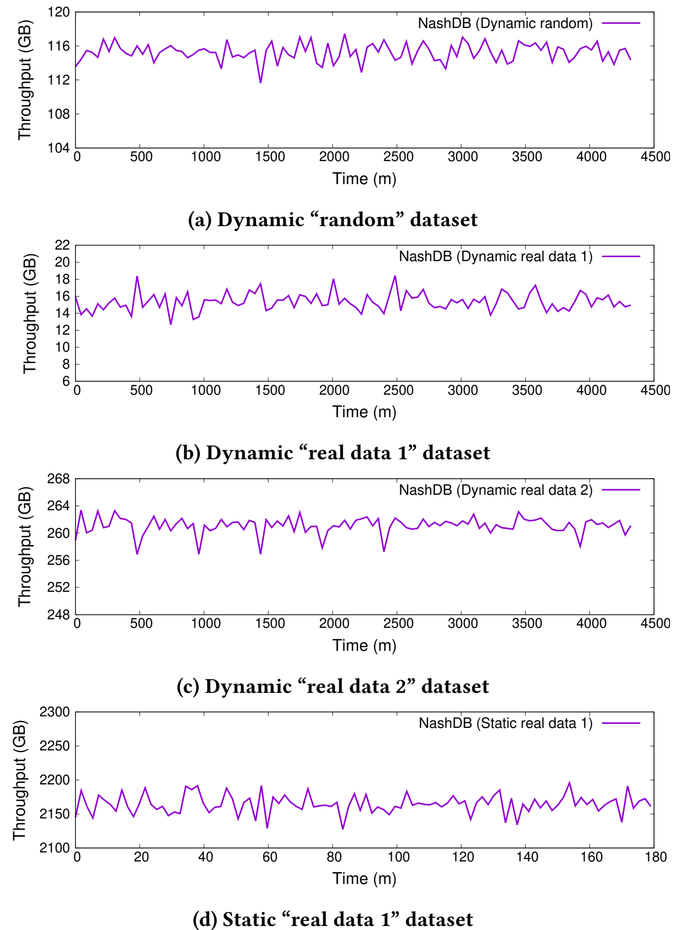


Figure 10: Percentile (tail) latency comparison



(a) Dynamic "random" dataset

(b) Dynamic "real data 1" dataset

(c) Dynamic "real data 2" dataset

(d) Static "real data 1" dataset

Figure 11: Throughput over time for various datasets

and the cluster transitioning algorithm moves less data). For the static "real data 1" workload, the throughput varies between 2148 to 2190GB/m. The lower relative variation of the static "real data 1" workload is due to the fact that the cluster transitioning technique was never applied (the workload represents a batch workload and so our approach identified a distribution schema and never had to transition to a new one). For all workload the transition overhead (which ranges between 5-200MB based on Figure 11) is significantly lower that the throughput (which ranges between 10s-1000s GB/m).

H ECONOMIC TERMINOLOGY

Throughout this paper, we use the colloquially understood phrases "increases in supply" or "decreases in demand." To readers deeply familiar with economic literature, we note that this increase in supply or demand is in reference to a change in *quantity* supplied or demanded, not a shift in the supply or demand curves themselves.