

ROBUS: Fair Cache Allocation for Multi-tenant Data-parallel Workloads

Mayuresh Kunjir, Brandon Fain, Kamesh Munagala, Shivnath Babu
Duke University
{mayuresh, btfain, kamesh, shivnath}@cs.duke.edu

ABSTRACT

Systems for processing big data—e.g., Hadoop, Spark, and massively parallel databases—need to run workloads on behalf of multiple tenants simultaneously. The abundant disk-based storage in these systems is usually complemented by a smaller, but much faster, *cache*. Cache is a precious resource: Tenants who get to use cache can see two orders of magnitude performance improvement. Cache is also a limited and hence shared resource: Unlike a resource like a CPU core which can be used by only one tenant at a time, a cached data item can be accessed by multiple tenants at the same time. Cache, therefore, has to be shared by a multi-tenancy-aware policy across tenants, each having a unique set of priorities and workload characteristics.

In this paper, we develop cache allocation strategies that speed up the overall workload while being *fair* to each tenant. We build a novel fairness model targeted at the shared resource setting that incorporates not only the more standard concepts of Pareto-efficiency and sharing incentive, but also define envy freeness via the notion of *core* from cooperative game theory. Our framework, ROBUS, uses randomization over small time batches, and we develop a proportionally fair allocation mechanism that satisfies the core property in expectation. We show that this algorithm and related fair algorithms can be approximated to arbitrary precision in polynomial time. We evaluate these algorithms on a ROBUS prototype implemented on Spark with RDD store used as cache. Our evaluation on a synthetically generated industry-standard workload shows that our algorithms provide a speedup close to performance optimal algorithms while guaranteeing fairness across tenants.

1. INTRODUCTION

Current data analytics systems process huge volumes of data collected on a daily basis through transactions, clickstream logs, social media, etc. As it is expensive to move data at such a large scale, the systems move computations to the data instead. Popular analytics systems, e.g., Hadoop and Spark, support system designs wherein multiple tenants share hardware resources as well as data, typically residing in a distributed file system such as HDFS. Such multi-tenant designs allow for a high resource utilization by removing a

need to provision for the peak load of each tenant.

Another development in the data analytics has been the hierarchical storage architectures wherein a disk-based storage is complemented by a smaller, but much faster, memory-based storage or cache. Examples of such architectures include Hadoop with Discardable Distributed Memory [1] and Spark with Tachyon [43]. Processing workload off cache can provide up to two orders of magnitude performance improvements by saving disk I/O costs [56]. Cache is shared across tenants and unlike resources like CPU, a cached data item can be accessed by all tenants at the same time. Since the cache is a limited resource, policies to decide *what* and *when* to cache are essential to make best use of the resource.

We present a small example to illustrate how different policies can impact workload performance and fairness across tenants. Consider three tenants A, B, C and three input tables *R*, *S*, and *P*, each of size equal to cache budget. For a given workload, the utilities, corresponding to savings in disk read costs, the tenants derive from each of the views are listed in Table 1 below.

Tenant	<i>R</i>	<i>S</i>	<i>P</i>
A	2	1	0
B	2	1	0
C	0	1	2

Table 1: Utilities of cached tables to tenants

Consider a policy that only tries to focus on overall performance of system by treating the entire workload as a set, e.g., any classical view materialization algorithm [30, 55, 8]. We brand such a policy as a *Multi-tenancy-unaware* policy (MTUP). In contrast, consider a policy that partitions cache in proportion to weights of the tenants. Let's brand this as a *Multi-tenancy-aware* policy (MTAP). Let's consider three scenarios.

1. The three tenants are equi-weighted and the cache can accommodate only one of the input tables. Here, MTUP would cache table *R* giving overall utility of 4. This policy is unfair to tenant C since it gets no utility at all. MTAP, on the other hand, is fair to everyone but provides no utility since no table can fit in the partitioned cache. A third alternative in this scenario could be to cache table *S*. It gives an overall utility of 3, 75% of the maximum utility, while being fair to each tenant.
2. Consider a scenario similar to the first; the only change being that the tenant C has a 50% higher weight than the other two; or in other words, the weights of the tenants are 1:1:1.5. MTUP would again cache *R* thus being unfair to C despite it being the highest priority tenant. MTAP does not provide

any utility once again. Caching S would be a better alternative once again.

3. In addition to increasing the weight of C , we double the cache in this scenario. MTUP would now pick tables R and S giving an overall utility of 7.5. This allocation, too, is unfair to C . Meanwhile, MTAP, despite having a larger cache budget, would still not be able to provide any utility. A better alternative here would be to cache R and P which provides a utility of 7 and is fair to all. It should be noted that table S is not picked by this policy despite being the only table shared by all tenants.

The above example shows non-trivial nature of the problem of cache allocation in multi-tenant setups. There is a need to make principled choices when it comes to picking data items to cache. This motivates the main challenge we address.

Develop a cache allocation policy that provides near-optimal performance speedups for tenants' workload while simultaneously achieving near-optimal fairness in terms of the tenants' performance.

1.1 Our Contributions

The main contributions of our work are listed next.

- In Section 2, we propose ROBUS, a system framework to optimize multi-tenant query workloads in an *online* manner using cache for speedup. This framework groups queries in small time-based batches and employs a randomized cache allocation policy on each batch.
- In Section 3, we consider the abstract setting of shared resource allocation within a batch, and enumerate properties that we desire from any allocation scheme. We show that the notion of *core* from cooperative game theory captures the fairness properties in a succinct fashion. We show that when restricted to randomized allocation policies within a batch, a simple algorithm termed *proportional fairness* generates an allocation which satisfies fairness properties in expectation for that batch.
- The policies we construct are based on convex programming formulations of exponential size. Nevertheless, in Section 4, we show that these policies admit to arbitrarily good approximations in polynomial time using the multiplicative weight update method. We present implementations of two fair policies: max-min fairness and proportional fairness. We also present faster and more practical heuristics for computing these solutions.
- We show a proof-of-concept implementation of ROBUS on a multi-tenant Spark cluster. Motivated by practical use cases, we develop a synthetic workload generator to create various scenarios. Implementation details and evaluation are provided in Section 5. Results show that our policies provide desirable throughput and fairness in a comprehensive set of setups.
- Finally, our policies are specified abstractly, and as such easily extend to other resource allocation settings. We discuss this in Section 3.4.

2. ROBUS FRAMEWORK

We present ROBUS(Random Optimized Batch Utility Sharing), our framework for cache management for multi-tenant data-parallel workloads, in Figure 1. Each tenant submits queries in an online fashion to a designated queue. Each queue corresponds to a user group within an organization and is characterized by a weight indicating its *fair* share of system resources. We next discuss our workflow and system specifics.

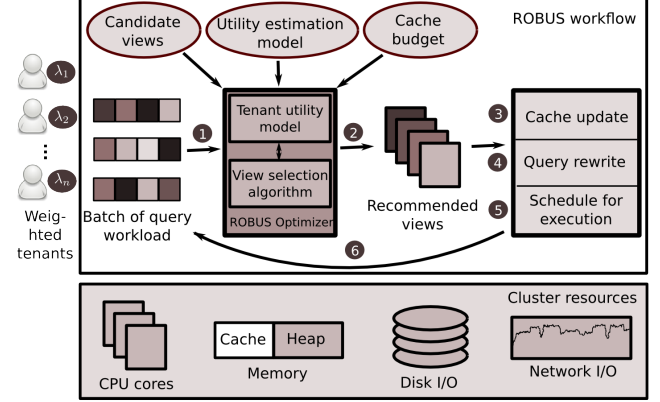


Figure 1: ROBUS framework

2.1 Workflow

ROBUS framework processes tenant queries in *batches* of fixed time interval. The workflow is pictured in Figure 1. Step 1 removes a batch of query workload from tenants' queues. Step 2 performs offline optimization over the batch to select a set of data items (views) to cache. This computation simultaneously optimizes performance and fairness; designing this subroutine is the main focus of this paper. A tenant utility model, detailed in Section 2.3, is critically used in the view selection procedure. The cache is updated with the recommended views in step 3. The queries are rewritten to use the cached views wherever possible before sending them for execution in steps 4 and 5 respectively. Finally, control is passed back to step 1 to consider next batch.

2.2 System Model

Every query runs data-parallel tasks. A task scheduler (e.g., [2, 25]) is responsible for allocating system resources to the tasks in order to simultaneously optimize performance and fairness across tenants. The resources include CPU, memory, disk bandwidth, and network bandwidth. Cluster memory is divided into two parts: one serving as a heap space for runtime objects and the other used as a cache to keep views critical for performance. While the heap is to be divided across tasks, and is allocated by the task scheduler, cache is shared by all queries simultaneously, and ROBUS framework provides a principled way to manage the resource.

Suppose a query accesses input tables d_1, d_2, \dots, d_n . There could be multiple views built over the input tables. Suppose v_1, v_2, \dots, v_m denotes the set of such candidate views. The query could then be rewritten to use $\{d_1, d_2, \dots, d_n\} + \{v_1, v_2, \dots, v_m\}$. ROBUS view selection algorithms pick a subset of views to cache, the choice of which is driven by performance and fairness measures. We do not consider any data access restrictions imposed on tenants by cluster admins.

2.3 Tenant Utility Model

As explained in Section 1, cache is a limited resource and is to be populated by a policy that is both multi-tenant-aware and near-

optimal in terms of query speedups. Towards this end, a tenant utility model plays an important role. The model helps our cache allocation algorithm pick a subset for each batch of workload from a set of candidate views on disk. We first list some of the options to build candidate views.

Disk-based materialized views: Any standard view materialization algorithm, e.g., [30, 55, 8, 40], can be used to build views by profiling tenants’ historical workload. Choices exist in terms of invoking the algorithm separately for each tenant or once on overall workload, or in terms of offline recommendations or adaptive changes to the set.

Tables: Input tables themselves could serve as candidate views.

Columns: Frequently accessed attributes of the input tables.

Partitions: Horizontal partitions of the input tables.

Hybrid: Any combination of the above could be used. e.g., daily partitions of the fact tables combined with a set of dimension tables can form candidate views.

Dynamic: Candidate views can change per batch of workload.

Any of the above choices could be plugged in ROBUS framework. We have used a static set of vertical projections over input datasets in the evaluation; details are included in Section 5.

The tenant utility model additionally requires a model to estimate utility provided to a query by a cached view. Several options exist here as well including:

1. **Disk I/O savings:** Most basic model would set the utility proportional to size of the view since the query can avoid reading from disk.
2. **View interaction consideration:** Multiple views can have complex interactions preventing the utility function from having convenient properties such as additivity or monotonicity [42].
3. **Dollar cost consideration:** The estimation model could profile savings in actual dollar costs in procuring system resources.

In our evaluation, we restrict each query to use only one view and the utility is set proportional to size of the view. A different estimation model could be plugged in for more tailored results.

3. FAIRNESS PROPERTIES AND POLICIES FOR SINGLE BATCH

In this section we study various notions of fairness when restricted to view selection for queries from a single batch. We consider policies that compute allocations that simultaneously provide large utility to many tenants, and enforce a rigorous notion of fairness between the tenants. Since this is very related to other resource allocation problems in economics [17, 6, 15], we draw heavily on that work for inspiration. However, the key difference from standard resource allocation problems is that in our setting, the resources (or views) are simultaneously shared by tenants. In contrast, the resource allocation settings in economics have typically considered *partitioning* resources between tenants. As we shall see below, this leads to interesting differences in the notions of fairness.

3.1 Fairness and Randomization

It is well-known in economics [18] that the combination of fairness and indivisible resources (in our case, the cache and views) necessitates randomization. To develop intuition, we present two examples.

First consider a simple fair allocation scheme that for N tenants simply allows each tenant to use $\frac{1}{N}$ of the total cache for her preferred view(s). It is plausible that some tenants prefer a large view that does not fit in this partition but does fit in the cache. Therefore, letting tenants have $\frac{1}{N}$ probability of using the whole cache can have arbitrarily larger expected utility than the scheme which with probability 1 lets them use $\frac{1}{N}$ fraction of the whole cache.

Next, consider a batch wherein two tenants each request a different large view such that only one can fit into the cache. In this case, there can be no deterministic allocation scheme that does not ignore one of the tenants. Using randomization, we can easily ensure that each tenant has the same utility in expectation. In fact, utility in expectation will be the per batch guarantee we seek, which over the long time horizon of a workload will lead to deterministic fairness.

Notation for Single Batch.

Since our view selection policy works on individual batches at a time, the notation and discussion below is specific to queries within a batch. Let N denote the total number of tenants. Define:

DEFINITION 1. A configuration S is the set of views feasible in that the sum of the view sizes $\sum_{S_i \in S} |S_i|$ is at most the cache size.

$U_i(S)$ denotes the utility to tenant i that would result from caching S , which is defined as the sum over all queries in i ’s queue of the utility for that query.

ROBUS generates a set Q of configurations which by definition can fit in the cache, and assigns a probability x_S to cache each configuration $S \in Q$. Define the vector of all such probabilities as:

DEFINITION 2. An Allocation \mathbf{x} is the vector corresponding to probabilities x_S of choosing configuration S normalized so $\|\mathbf{x}\| = \sum_{S \in Q} x_S = 1$.

We denote $U_i(\mathbf{x}) = \sum_{S \in Q} x_S U_i(S)$ as the expected utility of tenant i in allocation \mathbf{x} . ROBUS implements allocation \mathbf{x} by sampling a configuration from the probability distribution.

For each tenant i , let $U_i^* = \max_S U_i(S)$ denote the maximum possible utility tenant i can obtain if it were the only tenant in the system. For allocation \mathbf{x} , we define the *scaled utility* of i as $V_i(\mathbf{x}) = \frac{U_i(\mathbf{x})}{U_i^*}$. We will use this concept crucially in defining our fairness notions.

3.2 Basic Fairness Desiderata

The first question to ask when designing a fair allocation algorithm is what properties define fairness. There has been much recent work in economics and computer science on heterogeneous resource allocation problems and we begin by considering the properties that this related work examines [25, 49, 37]. Note that because we work within a randomized model, all of these properties are framed in terms of expected utility of tenants.

- **Pareto Efficiency (PE):** An allocation is Pareto-efficient if no other allocation simultaneously improves the expected utility of at least one tenant and does not decrease the expected utility of any tenant.
- **Sharing Incentive (SI):** This property is termed *individual rationality* in Economics. For N tenants, each tenant should expect higher utility in the shared allocation setting than she would expect from simply always having access to $\frac{1}{N}$ of the resources. Since our allocations are randomized, allocation \mathbf{x} satisfies SI if for all allocations \mathbf{y} with $\|\mathbf{y}\| \leq \frac{1}{N}$ and for tenants i , $U_i(\mathbf{x}) \geq U_i(\mathbf{y})$. In other words, $V_i(\mathbf{x}) \geq \frac{1}{N}$ for all tenants i , where $V_i(\mathbf{x})$ is the scaled utility function defined above.

One property that is widely studied in other resource allocation contexts is strategy-proofness on the part of the tenants (the notion that no tenant should benefit from lying). In our case, since the queries are seen by the query optimizer, strategy-proofness is not an issue. The above desiderata also omit envy-freeness (that no tenant should prefer the allocation to another tenant) which is something we revisit later.

We now consider a progression of view selection mechanisms on a single batch from very simple to more sophisticated. As a running example, suppose there is a cache of capacity 1. There are three views R , S , or P that are demanded by N tenants. Each view has unit size, so that we can cache only one view any time. Note that this is a drastically simplified example setup only intended to build intuition about why certain view selection algorithms might fail or are superior to others; our results and experiments do not only have unit views, are not limited to three tenants, and may have arbitrarily complex utilities compared to these examples.

We can summarize the input information our view selection might see in a given batch in a table (e.g., Table 2) where the numbers represent utilities tenants get from the views. An allocation here is a vector \mathbf{x} of three dimensions and $\|\mathbf{x}\| = 1$ that gives the probabilities in our randomized framework x_R, x_S, x_P for selecting the views.

Tenant	R	S	P
A	1	0	0
B	0	1	0
C	0	0	1

Table 2: Every tenant gets utility from a different view

Static Partitioning.

Recall that static partitioning is the algorithm that simply deterministically allows each of the N tenants to use $\frac{1}{N}$ of the shared resource. This algorithm does not take advantage of randomization. For the example in Table 2, this algorithm cannot cache anything because each user only gets to decide on the use of $\frac{1}{3}$ of the cache. The algorithm is sharing incentive in the standard deterministic setting, but is trivially not Pareto efficient and is not sharing incentive in expectation either. As mentioned previously, such examples motivate the randomization framework to start with.

Random Serial Dictatorship.

A natural progression from static partitioning is to consider random serial dictatorship (RSD), a mechanism that is widely considered [15, 6] for problems such as house allocation and school choice. We order the tenants in a random permutation. Each tenant sequentially computes the best set of views to cache (in the residual cache space) to maximize its own utility. In the example in Table 2, each tenant gets a $\frac{1}{3}$ chance of picking her preferred resource (since in a random permutation each tenant has a $\frac{1}{3}$ chance of appearing first) so the allocation is $\mathbf{x} = \langle x_R = \frac{1}{3}, x_S = \frac{1}{3}, x_P = \frac{1}{3} \rangle$, where each tenant has the same utility in expectation. In fact, it is easy to prove that RSD is always SI: Each tenant has $\frac{1}{N}$ chance of being first in the random ordering, so its scaled utility is at least $\frac{1}{N}$.

However, in contrast with resource partitioning problems, our problem has a shared aspect that RSD fails to capture. For example, consider the situation in Table 3; RSD computes the same allocation as in the example in Table: 2, $\mathbf{x} = \langle x_R = \frac{1}{3}, x_S = \frac{1}{3}, x_P = \frac{1}{3} \rangle$. However, on this example, though RSD is SI, it is not Pareto-efficient (PE). Tenants A and C have expected utility of 1 (a $\frac{1}{3}$ chance of getting 2 if they come first in the permutation and a $\frac{1}{3}$

chance of getting 1 if B does) and tenant B has expected utility of $\frac{1}{3}$ with this allocation. However, if we used allocation $\mathbf{x} = \langle x_R = 0, x_S = 1, x_P = 0 \rangle$ then tenants A, B, and C all have utility 1, which is strictly better for tenant B and as good for tenants A and C. RSD fails to capture the fact that while each tenant may have different top preferences, many tenants may share secondary preferences.

Tenant	R	S	P
A	2	1	0
B	0	1	0
C	0	1	2

Table 3: Every tenant gets utility from the same view

Utility Maximization Mechanism.

We next consider the mechanism which simply maximizes the total expected utility of an allocation, i.e., $\arg \max_{\mathbf{x}} \sum_i U_i(\mathbf{x})$. It is easy to check that this mechanism can ignore tenants who do not contribute enough to the overall utility. In other words, it cannot be SI.

Max-min Fairness (MMF).

In this algorithm we combine previous insights to optimize performance subject to fairness constraints to get a mechanism that is both SI and PE. For allocation \mathbf{x} , let $\mathbf{v}(\mathbf{x}) = (V_1(\mathbf{x}), V_2(\mathbf{x}), \dots, V_N(\mathbf{x}))$ denote the vector of scaled utilities of the tenants. We choose an allocation \mathbf{x} so that the vector $\mathbf{v}(\mathbf{x})$ is lexicographically max-min fair. This means the smallest value in $\mathbf{v}(\mathbf{x})$ is as large as possible; subject to this, the next smallest value is as large as possible, and so on. We present algorithms to compute these allocations in Section 4.

THEOREM 1. *The MMF mechanism is both PE and SI.*

PROOF. The RSD mechanism guarantees scaled utility of at least $\frac{1}{N}$ to each tenant. Since the MMF allocation is lexicographically max-min, the minimum scaled utility it obtains is at least the minimum scaled utility in RSD, which is at least $\frac{1}{N}$. To show PE, note that if there were an allocation that yielded at least as large utility for all tenants, and strictly higher utility for one tenant, the new allocation would be lexicographically larger, contradicting the definition of MMF. \square

Tenant	R	S
T_1	1	0
T_2	1	0
...
T_N	0	1

Table 4: All tenants except one get utility from the same view

Consider the example in Table 4. It is easy to see that the MMF value is $\frac{1}{2}$ and can be achieved with an allocation of $\langle x_R = \frac{1}{2}, x_S = \frac{1}{2} \rangle$. This allocation is both SI and PE.

3.3 Envy-freeness and the Core

The above discussion omits one important facet of fairness. A fair allocation has to be *envy free*, meaning no tenant has to envy how the allocation treats another tenant. In the case where resources are partitioned between tenants, such a notion is easy to define: No tenant must derive higher utility from the allocation to another tenant. However, in our setting, resources (views) are

shared between tenants, and the only common factor is the cache space. In any allocation \mathbf{x} , each tenant derives utility from certain views, and we can term the expected size of these views as the *cache share* of this user.

One could try to define envy-freeness in terms of cache space as follows: No tenant should be able to improve expected utility by obtaining the cache share of another tenant. But this simply means all tenants have the same expected cache share. Such an allocation need not be Sharing Incentive. Consider the example in Table 5, where each view R and S has size 1 and the cache has size 1. The only allocation that equalizes cache share caches S entirely. But this is not SI for tenant B .

Tenant	R	S
A	0	1
B	100	1

Table 5: Counterexample for perfect Envy-freeness

This motivates taking the utility of tenants into account in defining envy. However, this quickly gets tricky, since the utility can be a complex function of the entire configuration, and not of individual views. In order to develop more insight, we use an analogy to public projects. The tenants are members of a society, who contribute equal amount of tax. The total tax is the cache space. Each view is a public project whose cost is equal to its size. Users derive utility from the subset of projects built (or views cached). In a societal context, users are envious if they perceive an inordinate fraction of tax dollars being spent on making a small number of users happy. In other words, if they perceive a *bridge to nowhere* being built. Let us revisit the example in Table 4. Here, the MMF allocation sets $\mathbf{x} = \langle x_R = \frac{1}{2}, x_S = \frac{1}{2} \rangle$ and ignores the fact that an arbitrarily large number of tenants want R , compared to just one tenant who wants S . If we treat R as a school and S as a park, an arbitrarily large number of users want a school compared to a park, yet half the money is spent on the school, and half on the park. This will be perceived as unfair on a societal level.

Randomized Core.

In order to formalize this intuition, we borrow the notion of core from cooperative game theory and exchange market economics [27, 52, 12, 21]. We treat each user as bringing a *rate endowment* of $\frac{1}{N}$ to the system. If they were the only user in the system, we would produce an allocation \mathbf{x} with $\|\mathbf{x}\| = \frac{1}{N}$ and maximize their utility. An allocation \mathbf{x} over all tenants lies in the *core* if no subset of tenants can deviate and obtain better utilities for all participants by pooling together their rate endowments. More formally,

DEFINITION 3. *An allocation \mathbf{x} is said to lie in the (randomized) **core** if for any subset T of N tenants, there is no feasible allocation \mathbf{y} such that $\|\mathbf{y}\| = \frac{|T|}{N}$, for which $U_i(\mathbf{y}) \geq U_i(\mathbf{x}), \forall i \in T$ and $U_j(\mathbf{y}) > U_j(\mathbf{x})$ for at least one $j \in T$.*

It is easy to check that any allocation in the core is both SI and PE, by considering sets T of size 1 and N respectively. In the above example (Table 4), the allocation $\mathbf{x} = \langle x_R = \frac{N-1}{N}, x_S = \frac{1}{N} \rangle$ lies in the core. Tenant T_N gets its SI amount of utility and cache space. The more demanded view R is cached by a proportionally larger amount. In societal terms, each user perceives his tax dollars as being spent fairly. Similarly, in the example in Table 5, the allocation $\mathbf{x} = \langle x_R = \frac{1}{2}, x_S = \frac{1}{2} \rangle$ lies in the core.

In the context of provisioning public goods, there are two solution concepts that are known to lie in the core: The first, termed a Lindahl equilibrium [23, 44] attempts to find per-tenant prices that

implement a Walrasian equilibrium, while the second, termed ratio equilibrium [36] attempts to find per-tenant ratios of cache-shares. However, these concepts are shown to exist using fixed-point theorems, which don't lend themselves to efficient algorithmic implementations. We sidestep this difficulty by using randomization to our advantage, and show that a simple mechanism finds an allocation in the core.

Proportional Fairness.

DEFINITION 4. *An allocation \mathbf{x} is **proportionally fair (PF)** if it is a solution to:*

$$\text{Maximize } \sum_{i=1}^N \log(U_i(\mathbf{x})) \text{ subject to: } \|\mathbf{x}\| \leq 1 \quad (1)$$

We show the following theorem using the KKT (Karush-Kuhn-Tucker) conditions [41]. The proof also follows easily from the classic first order optimality condition of PF [47]; however, we present the entire proof for completeness. Subsequently, in Section 4, we show how to compute this allocation efficiently.

THEOREM 2. *Proportionally fair allocations satisfy the core property.*

PROOF. Let \mathbf{x} denote the optimal solution to (PF). Let d denote the dual variable for the constraint $\|\mathbf{x}\| \leq 1$. By the KKT conditions, we have:

$$x_S > 0 \implies \sum_i \frac{U_i(S)}{U_i(\mathbf{x})} = d$$

$$x_S = 0 \implies \sum_i \frac{U_i(S)}{U_i(\mathbf{x})} \leq d$$

Multiplying the first set of identities by x_S and summing them, we have

$$d = d \left(\sum_S x_S \right) = \sum_i \frac{\sum_S x_S U_i(S)}{U_i(\mathbf{x})} = \sum_i 1 = N$$

This fixes the value of d . Next, consider a subset T of users, with $|T| = K$, along with some allocation \mathbf{y} with $\|\mathbf{y}\| = \frac{K}{N}$. First note that the KKT conditions implied:

$$\sum_i \frac{U_i(S)}{U_i(\mathbf{x})} \leq N \quad \forall S$$

Multiplying by y_S and summing, we have:

$$\sum_i \frac{U_i(\mathbf{y})}{U_i(\mathbf{x})} \leq N \sum_S y_S = K$$

Therefore,

$$\sum_{i \in T} \frac{U_i(\mathbf{y})}{U_i(\mathbf{x})} \leq K$$

Therefore, if $U_i(\mathbf{y}) > U_i(\mathbf{x})$ for some $i \in T$, then there exists $j \in T$ for which $U_j(\mathbf{y}) < U_j(\mathbf{x})$. This shows that no subset T can deviate to improve their utility, so that the (PF) allocation lies in the core. \square

3.4 Discussion

Our notion of core easily extends to tenants having weights. Suppose tenant i has weight λ_i . Then an allocation \mathbf{x} belongs to the core if for all subsets T of tenants, there does not exist \mathbf{y} with

$\|\mathbf{y}\| = \frac{\sum_{i \in T} \lambda_i}{\sum_{i=1}^N \lambda_i}$ such that for all tenants $i \in T$, $U_i(\mathbf{x}) \leq U_i(\mathbf{y})$, and $U_j(\mathbf{x}) < U_j(\mathbf{y})$ for at least one $j \in T$. The proportional fairness algorithm is modified to maximize $\sum_i \lambda_i \log U_i(\mathbf{x})$ subject to $\|\mathbf{x}\| \leq 1$.

We note that the PF algorithm finds an allocation in the (randomized) core to any resource allocation game that can be specified as follows: The goal is to choose a randomization over feasible configurations of resources. Each configuration yields an arbitrary utility to each tenant. This model is fairly general. For instance, consider the setting in [25, 49], where resources can be partitioned fractionally between agents, and an agent's rate (utility) depends on the minimum resource requirement satisfied in any dimension. Suppose we treat each agent as being endowed with $\frac{1}{N}$ fraction of the supply of resources in all dimensions, the above result shows that the (PF) allocation satisfies the property that no subset of users can pool their endowments together to achieve higher rates for all participants.

Utilities under MMF and PF.

We now compare the total utility, $\sum_i V_i(\mathbf{x})$ for the optimal MMF and (PF) solutions. We present results showing that (PF) has larger utility than MMF in certain canonical scenarios. Our first scenario defines the following *grouped* instance: There are k views, $1, 2, \dots, k$ each of unit size. The cache also has size 1. There are k groups of tenants; group i has N_i tenants all of which want view i .

LEMMA 1. *The total utility of (PF) is at least the total utility of MMF for any grouped instance.*

PROOF. On grouped instances, MMF sets rate $1/k$ for each tenant, yielding a total utility of N/k for N tenants. The (PF) algorithm sets rate $x_i = N_i/N$ for all tenants in group i . This yields total utility of $\sum_i N_i^2/N$. Next note that

$$\frac{\sum_{i=1}^k N_i^2}{k} \geq \left(\frac{\sum_{i=1}^k N_i}{k} \right)^2$$

Noting that $\sum_i N_i = N$, it is now easy to verify that (PF) yields larger utility. \square

In fact, the ratio of the utilities of MMF and PF is precisely the Jain's index [35] of the vector $\langle N_1, N_2, \dots, N_k \rangle$. By setting $k = N/2 + 1$, and $N_2 = N_3 = \dots = N_k = 1$, this shows that (PF) can have $\Omega(N)$ times larger total utility than MMF. Our next scenario focuses on arbitrary instances with only two tenants.

LEMMA 2. *For two tenants, the total utility of (PF) is at least the total utility of MMF.*

PROOF. Let the utilities of the two tenants be a, b in (PF) and A, B in MMF. Assume $a \leq b$. Since MMF maximizes the minimum utility, we have $a \leq \min(A, B)$. Let $\alpha = A/a$ and $\beta = B/b$, so that $\alpha \geq 1$. Since $\log(a) + \log(b) = \log(ab)$ is maximized by definition of PF and \log is an increasing function, we have $ab \geq AB$, so $\alpha\beta \leq 1$. Since $\alpha \geq 1$, this implies $1/\beta \geq \alpha \geq 1$. Therefore

$$b - B = B(1/\beta - 1) \geq a(1/\beta - 1) \geq a(\alpha - 1) = A - a$$

This shows $a + b \geq A + B$ completing the proof. \square

Summary of Fairness Properties.

In summary, Table 6 shows the fairness properties that hold for all of our candidate algorithms. We abbreviate the properties SI for sharing incentive and PE for pareto efficiency. Based on this analysis, we suggest that proportional fairness is likely to be a preferable view selection algorithm for our ROBUS framework. The theoretical properties of proportional fairness suggest that it should perform fairly and efficiently.

Algorithm	SI	PE	CORE
Random Serial Dictatorship	✓		
Utility Maximization		✓	
Max-Min Fairness	✓	✓	
Proportional Fairness	✓	✓	✓

Table 6: Fairness properties of mechanisms

4. APPROXIMATELY COMPUTING PF AND MMF ALLOCATIONS

In this section, we show that the PF and MMF allocations can be computed to arbitrary precision. We then present fast heuristic algorithms for approximately computing PF and MMF allocations, which we implement in our prototype.

One key issue in computation is that the number of configurations is exponential in the number of views and tenants, so that the convex programming formulations have exponentially many variables. Nevertheless, since the programs have $O(N)$ constraints, we use the multiplicative weight method [10, 24] to solve them approximately in time polynomial in N and accuracy parameter $1/\epsilon$. These algorithms assume access to a *welfare maximization* subroutine that we term WELFARE.

DEFINITION 5. *Given weight vector \mathbf{w} , WELFARE(\mathbf{w}) computes a configuration S that maximizes weighted scaled utilities, i.e., solves $\arg \max_S \sum_{i=1}^N w_i V_i(S)$.*

The scaled utilities are computed using the tenant utility model described in Section 2.3. In our presentation, we assume WELFARE solves the welfare maximization problem exactly. Our algorithms will make polynomially many calls to WELFARE.

Multiplicative Weight Method.

We first detail the multiplicative weight method, which will serve as a common subroutine to all our provably good algorithms. This classical framework [10, 24] uses a Lagrangian update to decide feasibility of linear constraints to arbitrary precision.

We first define the generic problem of deciding the feasibility of a set linear constraints: Given a convex set $P \in \mathbf{R}^s$, and an $r \times s$ matrix A ,

$$\text{LP}(A, b, P): \exists x \in P \text{ such that } Ax \geq b?$$

Let $y \geq 0$ be an r dimensional dual vector for the constraints $Ax \geq b$. We assume the existence of an efficient ORACLE of the form:

$$\text{ORACLE } C(A, y) = \max\{y^T A z : z \in P\}.$$

The ORACLE can be interpreted as follows: Suppose we take a linear combination of the rows of Ax , multiplying row $a_i x$ by y_i . Suppose we maximize this as a function of $x \in P$, and it turns out to be smaller than $y^T b$. Then, there is no feasible way to satisfy all constraints in $Ax \geq b$, since the feasible solution x would make $y^T A x \geq y^T b$. On the other hand, suppose we find a feasible x . Then, we check which constraints are violated by this x , and increase the dual multipliers y_i for these constraints. On the other hand, if a constraint is too slack, we decrease the dual multipliers. We iterate this process until either we find a y which proves $Ax \geq b$ is infeasible, or the process roughly converges.

More formally, we present the Arora-Hazan-Kale (AHK) procedure [10] for deciding the feasibility of $\text{LP}(A, b, P)$. The running

time is quantified in terms of the WIDTH defined as:

$$\rho = \max_i \max_{x \in P} |a_i x - b_i|$$

Algorithm 1 AHK Algorithm

```

1: Let  $K \leftarrow \frac{4\rho^2 \log r}{\delta^2}$ ;  $y_1 = \vec{1}$ 
2: for  $t = 1$  to  $K$  do
3:   Find  $x_t$  using ORACLE  $C(A, y_t)$ .
4:   if  $C(A, y_t) < y_t^T b$  then
5:     Declare  $\text{LP}(A, b, P)$  infeasible and terminate.
6:   end if
7:   for  $i = 1$  to  $r$  do
8:      $M_{it} = a_i x_t - b_i$   $\triangleright$  Slack in constraint  $i$ .
9:      $y_{it+1} \leftarrow y_{it}(1 - \delta)^{M_{it}/\rho}$  if  $M_{it} \geq 0$ .
10:     $y_{it+1} \leftarrow y_{it}(1 + \delta)^{-M_{it}/\rho}$  if  $M_{it} < 0$ .
11:     $\triangleright$  Multiplicatively update  $y$ .
12:   end for
13:   Normalize  $y_{t+1}$  so that  $\|y_{t+1}\| = 1$ .
14: end for
15: Return  $x = \frac{1}{K} \sum_{t=1}^K x_t$ .
```

This procedure has the following guarantee [10]:

THEOREM 3. *If $\text{LP}(A, b, P)$ is feasible, the AHK procedure never declares infeasibility, and the final x satisfies:*

$$(a_i x - b_i) + \delta \geq 0 \quad \forall i$$

4.1 Proportional Fairness

Our algorithm uses the AHK algorithm as a subroutine and considers dual weights to find an additive ε approximation solution. The primary result is the following theorem:

THEOREM 4. *An approximation algorithm computes an additive ε approximation to (PF) with $O(\frac{4N^4 \log^2 N}{\varepsilon^2})$ calls to WELFARE, and polynomial additional running time.*

Proof.

For allocation \mathbf{x} , let $B(\mathbf{x}) = \sum_i \log V_i(\mathbf{x})$. Let $Q^* = \max_{\mathbf{x}} B(\mathbf{x})$ denote the optimal value of (PF), and let \mathbf{x}^* denote this optimal value. We first present a Lipschitz type condition, whose proof we omit from this version.

LEMMA 3. *Let \mathbf{y} satisfy $B(\mathbf{y}) \geq Q^* - \varepsilon$ for $\varepsilon \in (0, 1/6)$. Then, for all i , $V_i(\mathbf{y}) \geq V_i(\mathbf{x})/2$.*

The proof idea is to use the concavity of the log function to exhibit a convex combination of \mathbf{x} and \mathbf{y} whose value exceeds Q^* , which is a contradiction. It is therefore sufficient to find Q^* to an additive approximation in order to achieve at least half the welfare of (PF) for all tenants. Towards this end, for a parameter Q , we write (PF) as a feasibility problem $\text{PFFEAS}(Q)$ as follows:

DEFINITION 6. $\text{PFFEAS}(Q)$ *decides the feasibility of the constraints*

$$(F) = \left\{ \sum_S x_S V_i(S) - \gamma_i \geq 0 \quad \forall i \right\}$$

subject to the constraints:

$$(P1) = \left\{ \sum_S x_S \leq 1, x_S \geq 0 \quad \forall S \right\}$$

$$(P2) = \left\{ \sum_i \log \gamma_i \geq Q, \gamma_i \in [1/N, 1] \quad \forall i \right\}$$

The above formulation is not an obvious one, and is related to *virtual welfare* approaches recently proposed in Bayesian mechanism design [19, 14]. The key idea is to connect expected values (utility) to their realizations in each configurations via expected value variables, the γ_i . The constraints (P2) and (P1) are over expected values, and realizations respectively. The ORACLE computation in the multiplicative weight procedure will decouple into optimizing expected value variables over (P2), and optimizing WELFARE over (P1) respectively, and both these problems will be easily solvable.

We note that (P2) has additional constraints $\gamma_i \in [1/N, 1] \quad \forall i$. These are in order to reduce the width of the constraints (F). Note that otherwise, γ_i can take on unbounded values while still being feasible to (P2), and this makes the width of (F) unbounded. The lower bound of $1/N$ on γ_i is to control the approximation error introduced. We argue below that these constraints do not change our problem.

LEMMA 4. *Let Q^* denote the optimal value of the proportional fair allocation (PF). Then, $\text{PFFEAS}(Q)$ is feasible if and only if $Q \leq Q^*$.*

PROOF. In the formulation $\text{PFFEAS}(Q)$, the quantity γ_i is simply the scaled utility of tenant i . Consider the proportionally fair allocation \mathbf{x} . For this allocation, all scaled utilities lie in $[1/N, 1]$ since the allocation is SI. Therefore, \mathbf{x} is feasible for $\text{PFFEAS}(Q^*)$. On the other hand, if \mathbf{y} is feasible to $\text{PFFEAS}(Q)$ for $Q > Q^*$, then \mathbf{y} is also feasible for (PF), contradicting the optimality of \mathbf{x} . \square

We will therefore search for the largest Q for which $\text{PFFEAS}(Q)$ is feasible. Since each $\gamma_i \in [1/N, 1]$, we have $Q \in [-N \log N, 0]$. Therefore, obtaining an additive ε approximation to Q^* by binary search requires $O(\log N)$ evaluations of $\text{PFFEAS}(Q)$ for various Q , assuming constant $\varepsilon > 0$.

Solving $\text{PFFEAS}(Q)$. We now fix a value Q and apply the AHK procedure to decide the feasibility of $\text{PFFEAS}(Q)$. To map to the description in the AHK procedure, we have $b = 0$, and A is the LHS of the constraints (F). We have $r = N$. Since any $V_i(S) \leq 1$, and $\gamma_i \in [1/N, 1]$, the width ρ of (F) is at most 1. Finally, for small constant $\varepsilon > 0$, we will set $\delta = \frac{\varepsilon}{N^2}$. Therefore, $K = \frac{4N^4 \log N}{\varepsilon^2}$.

For dual weights \mathbf{w} , the oracle subproblem $C(A, \mathbf{w})$ is the following:

$$\text{Max}_{\mathbf{x}, \gamma} \sum_i (w_i V_i(\mathbf{x}) - \gamma_i)$$

subject to (P1) and (P2). This separates into two optimization problems.

The first sub-problem maximizes $\sum_i w_i V_i(\mathbf{x})$ subject to \mathbf{x} satisfying (P1). This is simply $\text{WELFARE}(\mathbf{w})$. The second sub-problem is the following:

$$\text{Minimize} \sum_i w_i \gamma_i$$

subject to \mathbf{w} satisfying (P2). Let L denote the dual multiplier to the constraint $\sum_i \log \gamma_i \geq Q$. Consider the Lagrangian problem:

$$\text{Minimize} \sum_i (w_i \gamma_i - L \log \gamma_i)$$

subject to $\gamma_i \in [1/N, 1]$ for all i . The optimal solution sets $\gamma_i(L) = \max(1/N, \min(1, L/w_i))$, which is a non-decreasing function of L .

We check if $\sum_i \gamma_i(L) < Q$. If so, we increase L till we satisfy the constraint with equality. This parametric search takes polynomial time, and solves the second sub-problem.

The AHK procedure now gives the following guarantee: Either we declare $\text{PF}(\text{FEAS}(Q))$ is infeasible, or we find (\mathbf{x}, γ) such that for all i , we have:

$$\sum_S x_S V_i(S) \geq \gamma_i - \varepsilon/N^2 \geq \gamma_i(1 - \varepsilon/N)$$

Since $\sum_i \log \gamma_i \geq Q$, the above implies:

$$B(\mathbf{x}) = \sum_i \log V_i(\mathbf{x}) \geq Q - \sum_i \log(1 - \varepsilon/N) \geq Q - \varepsilon$$

so that the value $Q - \varepsilon$ is achievable with the allocation \mathbf{x} .

Binary Search. To complete the analysis, since $\text{PF}(\text{FEAS}(Q^*))$ is feasible, the procedure will never declare infeasibility when run with $Q = Q^*$, and will find an \mathbf{x} with $B(\mathbf{x}) \geq Q^* - \varepsilon$, yielding an additive ε approximation. This binary search over Q takes $O(\log N)$ iterations.

Thus, we arrive at the result of theorem 4.

4.2 Max-min Fairness

We present an algorithm **SIMPLEMMF** that computes an allocation \mathbf{x} maximizing $\min_i V_i(\mathbf{x})$. The MMF allocation can be computed by applying this procedure iteratively as in [26]; we omit the simple details from this version. We note that the idea of applying the multiplicative weight method to compute max-min utility also appeared in [20].

We write the problem of deciding feasibility as $\text{SIMPLEMMF}(\lambda)$:

$$(F) = \left\{ \sum_S V_i(S) x_S \geq \lambda \quad \forall i \right\}$$

subject to the constraints:

$$(P) = \left\{ \sum_S x_S \leq 1, x_S \geq 0 \quad \forall S \right\}$$

We have $\lambda^* \in [1/N, 1]$, where $\lambda^* = \max_{\mathbf{x}} \min_i V_i(\mathbf{x})$. Therefore, the width $\rho \leq 1$. Further, we can set $\delta = \varepsilon/N$. We can now compute K from the AHK procedure, so that $K = \frac{4N^2 \log N}{\varepsilon^2}$ in order to approximate λ^* to a factor of $(1 - \varepsilon)$. The procedure is described in Algorithm 2.

Algorithm 2 Approximation Algorithm for SIMPLEMMF

```

1: Let  $\varepsilon$  denote a small constant  $< 1$ .
2:  $T \leftarrow \frac{4N^2 \log N}{\varepsilon^2}$ 
3:  $\mathbf{w}_1 \leftarrow \frac{1}{N}$  ▷ Initial weights
4:  $\mathbf{x} \leftarrow \mathbf{0}$  ▷ Probability distribution over set of views
5: for  $k \in 1, 2, \dots, T$  do
6:   Let  $S$  be the solution to  $\text{WELFARE}(\mathbf{w}_k)$ .
7:    $w_{i(k+1)} \leftarrow w_{ik} \exp(-\varepsilon \frac{U_i(S)}{U_i})$ 
8:   Normalize  $\mathbf{w}_{k+1}$  so that  $\|\mathbf{w}_{k+1}\| = 1$ .
9:    $x_S \leftarrow x_S + \frac{1}{T}$  ▷ Add  $S$  to collection
10: end for
```

In order to compute MMF allocations, we use a similar idea to decide feasibility, except that we have to perform $O(N^2)$ invocations. This blows up the running time to $O\left(\frac{4N^4 \log N}{\varepsilon^2}\right)$ invocations of **WELFARE**.

The algorithm gives the following result:

THEOREM 5. *An approximation algorithm for SIMPLEMMF (Algorithm 2) finds a solution \mathbf{x} such that $\min_i V_i(\mathbf{x}) \geq \lambda^*(1 - \varepsilon)$ using $\frac{4N^2 \log N}{\varepsilon^2}$ calls to **WELFARE**.*

4.3 Fast Heuristics

In this section, we present heuristic algorithms that directly work with the exponential size convex programs. We directly implement these algorithms in software to gather our experimental results.

Configuration Pruning.

For $M = O(N^2)$, generate M random N -dimensional unit vectors $w_k, k = 1, 2, \dots, M$. For each w_k , let S_k be the configuration corresponding to $\text{WELFARE}(w_k)$. Denote this set of configurations by \mathcal{S} . We restrict the convex programming formulations of PF and MMF to just the set of configurations \mathcal{S} , and solve these programs directly, as we describe below. The intuition behind doing this pruning step is the following: The approximation algorithms for PF and MMF find convex combinations of configurations that are optimal for $\text{WELFARE}(w)$ for some w 's that are computed by the multiplicative weight procedure. Instead of this, we generate random such Pareto-optimal configurations, giving sufficient coverage so that each tenant has a high probability of having the maximum weight at least once.

We compared two algorithms for **SIMPLEMMF**, one using the multiplicative weight procedure (Algorithm 2), and the other solving the linear program (Program (3) below) restricted to random optimal configurations. When run on 200 batches with five tenants, using 5 weight vectors gives a 10.4% approximation to the objective of **SIMPLEMMF**. With 25 random weight vectors, the approximation error is 1.4%, and using 50 random weights, the approximation error drops to 0.6%. This shows that a small set \mathcal{S} of configurations that are optimal solutions to $\text{WELFARE}(w)$ for random vectors w is sufficient to generate good approximations to our convex programs. In our implementation, we set \mathcal{S} to be the union of these configurations along with the configurations generated by the **SIMPLEMMF** algorithm (Algorithm 2).

Proportional Fairness.

We first note that (PF) is equivalent to the following; the proof of equivalence follows from Theorem 2, where the dual variable corresponding to the constraint $\sum_S x_S = 1$ is precisely N .

$$\text{Max } g(\mathbf{x}) = \sum_{i=1}^N \log(V_i(\mathbf{x})) - N\|\mathbf{x}\| \quad \text{s.t.: } \mathbf{x} \geq \mathbf{0} \quad (2)$$

Given a configuration space \mathcal{S} , we can solve the program (2) using gradient descent, as shown in Algorithm 3. As precomputation, for each configuration $S \in \mathcal{S}$, we precompute $V_i(S)$. Then $V_i(\mathbf{x}) = \sum_{S \in \mathcal{S}} V_i(S) x_S$.

Algorithm 3 Proportional Fairness Heuristic

```

1: Let  $M = |\mathcal{S}|$ . Set  $t = 1$ .
2: Let  $\mathbf{x}_1 = (1/M, 1/M, \dots, 1/M)$ .
3: repeat
4:    $\mathbf{y} = \nabla g(\mathbf{x})$  evaluated at  $\mathbf{x} = \mathbf{x}_t$ .
5:    $r^* = \arg \max_r (g(\mathbf{x}_t + r\mathbf{y}))$ 
6:    $\mathbf{x}_{t+1} = \mathbf{x}_t + r^*\mathbf{y}$ 
7:   Project  $\mathbf{x}_{t+1}$  as:  $x_d = \max(x_d, 0)$  for all dimensions  $d \in \{1, 2, \dots, M\}$ .
8: until  $\mathbf{x}_t$  converges
```

Max-min Fairness.

Using the precomputed configuration space \mathcal{S} , we solve SIM-PLEMMF using the following linear program:

$$\max \left\{ \lambda \mid \sum_{S \in \mathcal{S}} V_i(S) x_S \geq \lambda \quad \forall i, \mathbf{x} \geq \mathbf{0} \right\} \quad (3)$$

This can be solved using any off-the-shelf LP solver (our implementation uses the open source lpsolve package [13]). In order to compute the MMF allocation, we iteratively compute the lexicographically max-min allocation using the above LP. The details are standard; see for instance [26]. Briefly, in each iteration a value of λ is computed. All tenants whose rate cannot be increased beyond λ without decreasing the rate of another tenant are considered saturated and the rate of λ for these tenants is a constraint in the next iteration of the LP. The solution to the final LP for which all tenants are saturated is the MMF solution.

5. EVALUATION

The primary goal of evaluation is to test different cache allocation policies on a variety of practical setups of multi-tenant analytics clusters. The setups could differ in terms of number of tenants, workload arrival patterns, or data access patterns among other things. Some of the example setups are listed below.

1. BI: Tenants correspond to BI user groups that have similar workload to process. There are certain datasets that are frequently accessed by all tenants suggesting a good opportunity for shared optimizations.
2. ETL+BI: All BI tenants have similar data access patterns as above. But additionally a tenant runs ETL workload which might touch entirely different datasets than the BI tenants.
3. Heterogeneous: Each tenant accesses different datasets. There is less scope for shared optimization.
4. Production+Engineering: Engineering workload is of bursty nature. Depending on the time of the day, engineering queues will have differing amount of work; whereas production queues, running pre-scheduled workflows, have roughly similar amount of work throughout.

We replicate various combinations of these setups on a small-scale Spark cluster and run controlled experiments using synthetically generated query workloads on TPC-DS [5] data. The methodology and the results are explained next.

5.1 Setup and Methodology

Figure 1 has presented the architecture of ROBUS. We use Apache Spark [4] to build a system prototype. Spark is a natural choice for the evaluation since it supports distributed memory-based abstraction in the form of Resilient Distributed Datasets (RDDs). In our prototype, a long running Spark context is shared among multiple queues, with each queue corresponding to a tenant. The Spark context has an access to the entire RDD cache in the cluster. Spark’s internal fair share scheduler is configured with a dedicated pool for each queue; the fair share properties of the pool are set proportional to weight of the corresponding queue.

Table 7 presents our test cluster setup. We use TPC-DS data generated over 10 different scale factors and store it in HDFS, amounting to 1.2TB in toto. We only focus on the three “sales” tables—*store_sales*, *catalog_sales*, and *web_sales*—since they provide a linear scaleup. The total data size queried, amounting to the sales

Spark version	1.1.1
Number of worker nodes	10
Instance type of nodes	c3.2xlarge
Total number of cores	80
Executor memory	80GB

Table 7: Test cluster setup on Amazon EC2

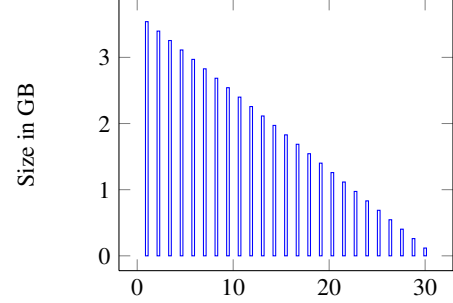


Figure 2: Cache size estimates of candidate views

tables, is 600GB. We create 30 candidate views on disk, each being a vertical projection of a sales table. Each query we generate can be translated to use one of the views. Total size of the views on disk is 60GB, the smallest with a size of 118MB and the largest with a size of 3.6GB. Figure 2 shows cache size estimates of the candidate views.

We set the cache size to 8GB, 10% of the total executor memory, leaving aside the rest as a heap space. Only 6GB of the cache is used to carry out our optimizations in order to avoid memory management issues our Spark installation experienced while evicting from a near-full cache.

Query workload consists of exploratory SQL queries, each performing scans and aggregations over a dataset. A significant chunk of the big data analytics workload falls under this category. This choice of workload allows for a simplistic model to estimate utility provided by cached data. A lookup table is populated with size estimates of the views which also represents utilities in terms of disk input cost savings. The WELFARE function defined in Section 4 refers to this lookup table when needed. All the queries are submitted using SparkSQL APIs.

Workload Arrival and Data Access.

Figure 3 shows our workload generation process. Several studies have established that query arrival times follow a Poisson distribution [29, 51]. We use the same in our prototype. Previous studies have also indicated that the data accessed by analytical workloads follows a Zipf distribution [29, 50]: A small number of datasets are

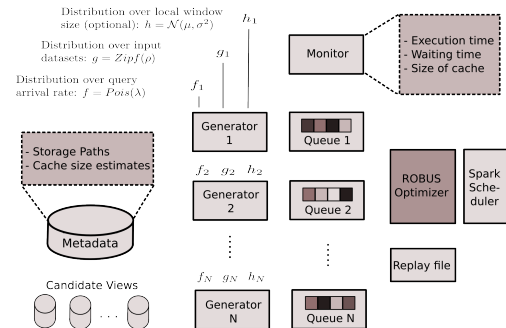


Figure 3: Workload generation for ROBUS

Distribution	Ranks
g_1	{ 30, 29, 28, 8, 27, 26, 25, 1, 24, 23, 22, 7, 21, 20, 19, 6, 18, 17, 16, 5, 3, 14, 15, 4, 12, 11, 10, 13, 9, 2 }
g_2	{ 30, 29, 7, 28, 27, 26, 1, 25, 24, 23, 6, 22, 21, 20, 5, 19, 18, 17, 4, 16, 15, 14, 3, 13, 12, 11, 2, 10, 9, 8 }
g_3	{ 30, 7, 29, 28, 27, 1, 26, 25, 24, 6, 23, 22, 21, 5, 20, 19, 18, 4, 17, 16, 15, 3, 14, 13, 12, 2, 11, 10, 9, 8 }
g_4	{ 8, 30, 29, 28, 1, 27, 26, 25, 7, 24, 23, 22, 6, 21, 20, 19, 5, 18, 17, 16, 4, 15, 14, 13, 3, 12, 11, 10, 2, 9 }

Table 8: Ranks used in Zipfian distributions

more popular than others, while there is a long tail of datasets that are only sporadically accessed. Each query generator picks datasets from a Zipfian distribution provided at the time of configuration. Additionally, grouping and aggregation predicates are picked from a probability distribution defined for the chosen dataset.

Further, [50] also shows that 90% of recently accessed data is re-accessed within next hour of first access. This makes a lot of sense because users typically want to drill down a dataset further in response to some interesting observation obtained in the previous run. In order to support such scenarios, we pick a small window in time from a Normal distribution (denoted by h in Figure 3). Over this window, a small subset of datasets is chosen from the Zipfian g . This subset forms candidates for the duration of the window. Each query to be generated picks one dataset from the candidates uniformly at random. This technique is taken from [29] which terms the values used in local window as “cold” values to differentiate them from globally popular “hot” values. The generated workload still follows the Zipfian g globally. As indicated in the figure, distribution h is optional; If not provided, datasets are picked from Zipfian g at all times.

5.2 Performance Metrics

We gather several performance metrics while executing a workload. They are defined next. We emphasize that these metrics are over long time horizons.

1. **Throughput.** This is simple to define:

$$\text{Throughput} = \frac{\text{number of queries served}}{\text{total time taken}} \quad (4)$$

2. **Fairness Index.** For job schedulers, a performance-based fairness index is defined in terms of variance in slowdowns of jobs in a shared cluster compared to a baseline case where every job receives all the resources [32]. As our work is about speeding up queries, we use relative speedups across queries while deriving fairness. The baseline is the case of statically partitioned cache. Here, X_i is the mean speedup for tenant i , and λ_i is the weight of tenant i .

$$\text{Fairness index} = \frac{(\sum_{i=1}^n \frac{X_i}{\lambda_i})^2}{n \sum_{i=1}^n (\frac{X_i}{\lambda_i})^2} \quad (5)$$

3. **Average Cache Utilization.** This is simply the average fraction of cache utilized during workload execution.
4. **Hit Ratio.** The fraction of queries served off cached views.

Setup	Distributions used by four tenants
\mathcal{G}_1	{ g_1, g_1, g_1, g_1 }
\mathcal{G}_2	{ g_1, g_1, g_1, g_2 }
\mathcal{G}_3	{ g_1, g_1, g_2, g_3 }
\mathcal{G}_4	{ g_1, g_2, g_3, g_4 }

Table 9: Data access distributions used in evaluation

Parameter	Value
Query inter-arrival rates (sec)	{20 \forall tenant}
Batch size (sec)	40
Number of batches	30

Table 10: Data sharing experiment setup

Some of the other metrics we collect include flow time, mean execution time, mean wait time, and wait time fairness index. They are not included due to space constraints.

5.3 Algorithm Evaluation

In this section, we evaluate four view selection algorithms on various setups. Each algorithm processes a batch of query workload in an offline manner as detailed in Section 2.1. Section 3 discussed several possible algorithms. Here, we compare the following:

1. **STATIC:** Cache is partitioned in proportion to weights of the tenants. We treat this as baseline when evaluating fairness index.
2. **MMF:** Max-min fairness implementation described in Section 4.3.
3. **FASTPF:** Proportional fairness implementation described in Section 4.3.
4. **OPTP:** The only goal is to optimize for query performance; Workload from a batch is treated as if belonging to a single tenant – a special case of either MMF or FASTPF.

In order to compare these algorithms across various settings, we vary the following parameters independently in our experiments.

1. Data sharing among tenants (Section 5.3.1);
2. Workload arrival rate (Section 5.3.2); and
3. Number of tenants (Section 5.3.3).

5.3.1 Effect of data sharing among tenants

To study the impact of different data sharing patterns on the performance of algorithms, we first create four different Zipf distributions over candidate views: g_1, g_2, g_3, g_4 . Each of the distributions is skewed towards a different subset of views as can be seen in Table 8. We create four test setups, each allowing a different level of data sharing, as listed in Table 9. The other common parameters are listed in Table 10.

Figure 4 shows how different algorithms perform in each of these setups. Throughput goes down with heterogeneity in data access. STATIC performs poorly in all the setups, the performance being between 30%-40% worse of the others. Its lower cache utilization and lower hit ratio are further indicators of why STATIC is not the right choice for cache allocation. There is very little to distinguish among the three cache-sharing algorithms. This shows that our fair algorithms can provide a throughput close to the optimal. In terms of fairness, OPTP algorithm gives the most inconsistent

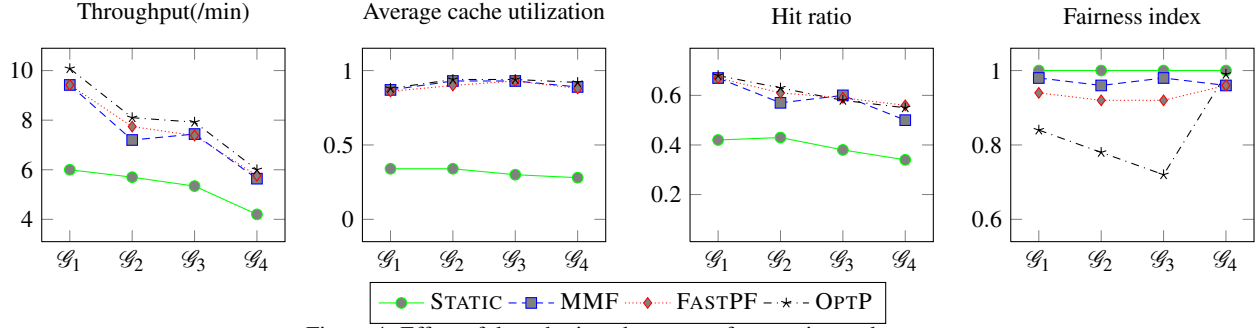


Figure 4: Effect of data sharing changes on four equi-paced tenants

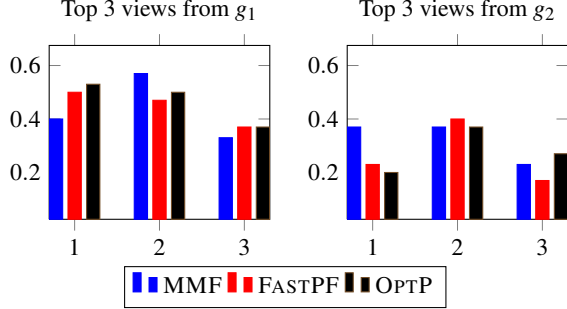


Figure 5: Fraction of time the popular views in setup \mathcal{G}_2 were cached

performance. It scores high in the setup with most heterogeneity, but fails when data sharing is involved. MMF and FASTPF, on the other hand, score high in all the setups.

The performance of MMF interestingly falls alarmingly low in the second setup. This is clearly an outcome of the data sharing pattern wherein three of four tenants largely share the same subset of views. Recollecting the example presented in Table 4, MMF tries to share the cache (probabilistically) equally between the two sets of tenants effectively producing an allocation off the core. We include a chart showing the duration the most popular views were cached for by MMF, FASTPF, and OPTP. (Figure 5) Top three views in each of g_1 and g_2 serve 25%, 13%, and 8% of the queries respectively. It can be seen that while MMF caches the topmost view from the distributions roughly equally, FASTPF and OPTP favor the topmost view from g_1 more since it is shared by three tenants. MMF tries to compensate the three tenants by caching their second best view more, but this view has a lower utility both due to lower access frequency and smaller size. So the overall performance of MMF suffers in this case.

5.3.2 Effect of variance in query arrival rates

To replicate the bursty tenants scenarios, we vary query inter-arrival rates of tenants in a two-tenant setup. We create three setups—*low*, *mid*, and *high*—with query inter-arrival rates as listed in Table 11. The other parameters used in each of the setups are listed in Table 12.

Setup	Poisson mean, λ_1	Poisson mean, λ_2
<i>low</i>	12	12
<i>mid</i>	18	8
<i>high</i>	24	6

Table 11: Query inter-arrival rates for different setups

Parameter	Value
Data access distributions	$\{g_1, g_2\}$
Batch size (sec)	72
Number of batches	30

Table 12: Query arrival rate experiment setup

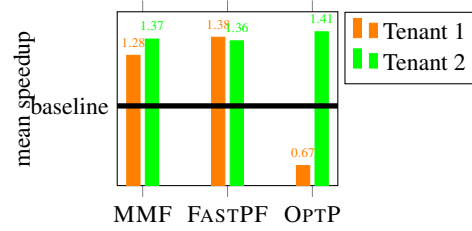


Figure 7: Mean speedups provided by different algorithms over STATIC policy for the two tenants in the setup *high*

Figure 6 shows the impact of variance in query arrival rate on various metrics. The performance of STATIC remains below the other three algorithms as can be seen from the first three graphs. The performance gap, however, is small because the cache is partitioned in only two parts for STATIC each part being large enough to serve 80% of the queries that could be served off unpartitioned cache. When it comes to the fairness index, all the algorithms except OPTP get a near-perfect score. OPTP favors the faster tenant in both *mid* and *high* setups so much that the slower tenant's performance degrades. Figure 7 shows the speedups for MMF, FASTPF, and OPTP relative to STATIC under the setup *high*. It can be seen that the first tenant sees a performance degradation with OPTP empirically proving the fact that OPTP is not sharing incentive.

5.3.3 Effect of number of tenants

Setup	Poisson mean, λ
2	10
4	20
8	40

Table 13: Query inter-arrival rates for a tenant under different setups

To further stress the utility of optimizing the entire cache as a shared resource, we experiment with increasing number of tenants. Specifically, we consider scenarios with 2, 4, and 8 tenants, all using the same distribution over dataset access. We try to keep the number of queries per batch the same by doubling query inter-arrival rate with doubling of the number of tenants, batch size remaining the same across the setups. Table 13 lists the query inter-

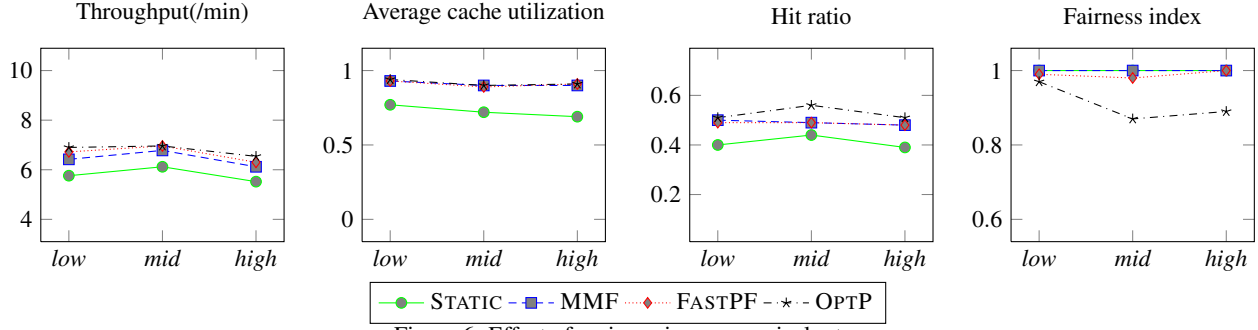


Figure 6: Effect of variance in query arrival rates

Parameter	Value
Data access distributions	$\{g_i \forall \text{ tenant}\}$
Batch size (sec)	40
Number of batches	30

Table 14: Number of tenants experiment setup

arrival rates we used. The other parameters common across the setups are listed in Table 14.

Figure 8 shows behavior of the algorithms under these scenarios. The gap in throughput between STATIC and the other algorithms is large (35%-45%). As the number of tenants goes up, the average cache utilization of STATIC drops sharply, whereas the average cache utilization of the other algorithms remain largely stable. This can be attributed to the static partitioning of cache in STATIC. The hit ratio shows a similar pattern again showing why STATIC is not the best choice. In terms of fairness index, OPTP finds it increasingly harder to provide a fair solution. With an increase in the number of tenants, the number of queries per tenant per batch goes down which makes the locally optimal choices of OPTP more unlikely to provide equal speedups. In contrast, MMF and FASTPF, with their randomized choices, score over 0.9 in all the scenarios exhibiting their superiority.

5.4 Discussion and Future Work

Our evaluation on practical setups brings up some interesting insights that opens up multiple possibilities for the future. We discuss some of the challenges and the directions here.

We first note that our experiments mainly show first-order differences between algorithms. In order to bring out second order differences, we will need a better cost model that matches query execution more closely, as well as implementations of the exact algorithms (for instance, the algorithm in Section 4.1 for proportional fairness). Our experiments show that across all setups, the fair algorithms provide far better trade-offs in throughput and fairness compared to STATIC and OPTP: they never perform worse than 90% of OPTP and they consistently score over 90% on the fairness index compared to STATIC. When comparing max-min fair and proportional fair implementations, there is no clear winner. This contrasts the theoretical results obtained in Section 3 which showed proportional fair to be the superior of the two in many cases. We believe this is a second order difference that a better implementation and cost model will bring out.

We next note that the **running time** of our algorithms is polynomial in number of tenants. In most typical industry setups, the ones we evaluated, there is only a handful number of tenants. Therefore, we expect our algorithms to be fast even in the wild. Just to quantify the query wait times, we observed them to be of the order of tens of milliseconds in most cases.

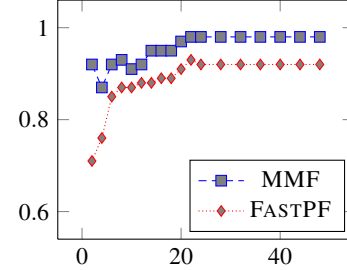


Figure 9: Fairness index as a function of number of batches

Convergence Properties.

As our algorithms are randomized in nature, it is important to study how long they take to converge to solutions that yield fairness across time. After running several workloads, we find that the number of batches to achieve convergence is very small, of the order of 15-25. In Figure 9, we present results of a four tenant workload with 50 batches, optimized once using MMF and once using FASTPF. The fairness index was computed after every 2-4 batches. It can be seen that both algorithms converge to their respective optimal values at around 20 batches. As a future work, we plan to systematically study which parameters define rate of convergence of the algorithms.

Batch Size and Cache State.

Our batched processing architecture introduces additional optimization choices. Primarily, there are two ways of tuning a view selection algorithm:

1. Controlling batch size, and
2. Managing state of cache across batches.

The first option needs no elaboration. The second option is whether the cache is treated as *stateful* or as *stateless* when optimizing a batch. In the former case, the estimated benefit of views that are already in cache is boosted by a factor $\gamma > 1$. This influences the next cache allocation, and makes it more likely for these views to stay in the cache. The latter case ignores the state of the cache when considering the next batch. All the results presented so far have used stateless cache.

We empirically compared how the algorithms react to these parameters. Figure 10 shows effect of change in batch size on two versions each of MMF and FASTPF: one treating the cache as stateless (MMFSL and FASTPFSL), and the other treating it as stateful (MMFSF and FASTPFSF), with $\gamma = 2$. It can be seen that both versions provide similar throughput in all the cases. It can be observed that the stateful algorithms score higher on fairness for the smallest batch size but there is no clear pattern seen when the batch size is larger. It makes sense since the lower batch sizes do not

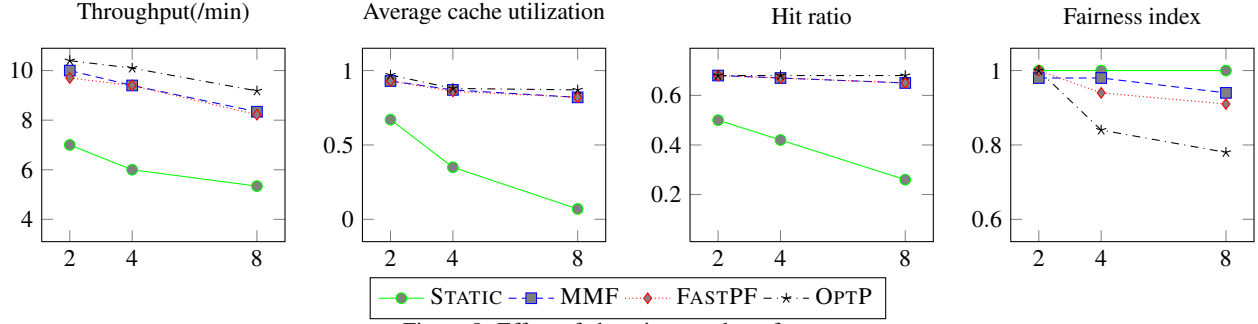


Figure 8: Effect of changing number of tenants

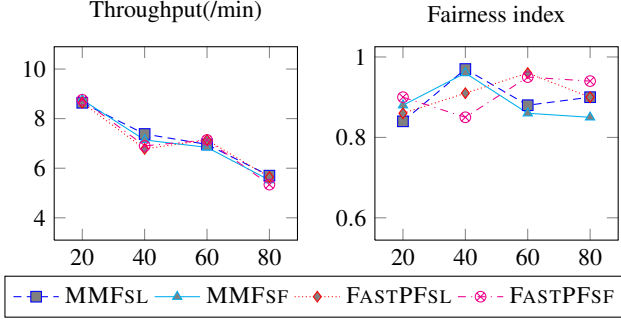


Figure 10: Effect of batch size on four equi-paced tenants setup

give enough choices for fair configurations of cache and maintaining the state results in an artificial increase of the batch size thereby providing better configurations. As a future work, we plan to explore these trade-offs on a larger scale to devise better guidelines on parameter tuning.

Engineering issues.

We now highlight some challenges in scaling up our experiments to industry scale. These challenges are tied to engineering issues in current implementations of systems such as Spark, and will get ironed out over time. Most common multi-tenant Spark setups use a separate Spark *context* for each tenant, effectively partitioning cache. In fact, most current multi-tenant data warehouse systems recommend splitting memory across queues. This is in part due to multi-thread management challenges that result in weird situations such as evicting cached partitions prematurely. Another engineering issue, specific to Spark, is the inordinately long delays in garbage collection when cluster scales up. We should be able to see a much better impact of ROBUS optimization once these practical issues get resolved.

Code Base.

The code base of ROBUS has been open-sourced [3] and our entire experimental setup can be replicated following a simple set of instructions provided with the code.

6. RELATED WORK

Physical design tuning and Multi-query optimization.

Classical view materialization algorithms in databases [30, 55, 8, 28, 45] treat entire workload as a set and optimize towards one or more of the flow time, space budget, and view maintenance costs. Online physical design tuning approaches [16, 40, 22, 42], on the other hand, adapt to changes in query workload by modify-

ing physical design. None of the afore-mentioned approaches support multi-tenant workloads and therefore cannot be used in selecting views for caching. However, some of the techniques used, in particular candidate view enumeration, view matching, and query rewrite, can be applied in ROBUS framework.

Batched optimization of queries was proposed in [53] and is used in many query sharing approaches [57, 7, 48]. ROBUS employs batched query optimization likewise, but crucially also ensures that each tenant gets their fair share of benefit.

Fairness theory.

The proportional fairness algorithm is widely studied in Economics [47, 34, 17] as well as in scheduling theory [25, 49, 39, 38, 31, 54, 9]. In the context of resource partitioning problems (or exchange economies) [12, 21], it is well-known that a convex program, called the Eisenberg-Gale convex program [34] computes prices that implement a Walrasian equilibrium (or market clearing solution). Our shared resource allocation problem is different from allocation problems where resources need to be partitioned, and it is not clear how to specify prices for resources (or views) in our setting. Nevertheless, we show that there is an exponential size convex program using configurations as variables, whose solution implements proportional fairness in a randomized sense.

In scheduling theory, the focus is on analyzing delay properties [39, 38, 31] assuming jobs have durations. Our focus is instead on utility maximization, which has also been considered in the context of wireless scheduling in [54, 9]. The latter work focuses on *long-term* fairness for partitioned resources, where utility of a tenant is defined as sum of discounted utilities across time. The resulting algorithms, though simple, only provide guarantees assuming job arrivals are ergodic, and if tenants exist forever. They do not provide per-epoch guarantees. In contrast, we focus on obtaining per-epoch fairness in a randomized sense without ergodic assumptions, and on defining the right fairness concepts when resources are shared. We finally note that [37] presents dynamic schemes for achieving envy-freeness across time; however, these techniques are specific to resource partitioning problems and do not directly apply to our shared resource setting.

Multi-tenant architectures.

Traditionally, the notion of multi-tenancy in databases deals with sharing database system resources, viz. hardware, process, schema, among users [33, 11, 46]. Each tenant only accesses data owned by them. Emerging multi-tenant big data architectures, on the other hand, allow for entire cluster data to be shared among tenants. This sharing of data is critical in our work as it allows the cache to be used much more efficiently.

A critical component of modern multi-tenant architectures, such

as Apache Hadoop, Apache Spark, Cloudera Impala, is a fair scheduler/ resource allocator [2, 25, 32]. The resource pool considered by these schedulers do not differentiate the cache resource from the heap resource and as a result divides the cache among tenants. As seen in our work, partitioned cache setups severely dampen optimization opportunities. Our prototype uses Spark in a single application mode running multi-tenant workloads through the same application. However, there has been ongoing work in supporting a distributed cache storage shared across tenants [43, 1]. ROBUS optimizer will be a natural fit for such systems.

7. CONCLUSION

Emerging Big data multi-tenant analytics systems complement an abundant disk-based storage with a smaller, but much faster, cache in order to optimize workloads by materializing views in the cache. The cache is a shared resource, i.e., the views materialized can be accessed by all tenants. In this paper, we presented ROBUS, a framework for achieving both a fair allocation of cache and a near-optimal performance in such architectures. We defined notions of fairness for the shared settings using randomization in small batches as a key tool. We presented a fairness model that incorporates Pareto-efficiency and sharing incentive, and also achieves envy-freeness via the notion of core from cooperative game theory. We showed a proportionally fair mechanism to satisfy the core property in expectation. Further, we developed efficient algorithms for two fair mechanisms and implemented them in a ROBUS prototype built on a Spark cluster. Our experiments on various practical setups show that it is possible to achieve near-optimal fairness, while simultaneously preserving near-optimal performance speedups using the algorithms we developed.

Our framework is quite general and applies to any setting where resource allocations are shared across agents. As future work, we plan to explore other applications of this framework.

8. REFERENCES

- [1] Discardable distributed memory: Supporting memory storage in HDFS, <http://hortonworks.com/blog/ddm/>.
- [2] Hadoop fair scheduler, http://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html.
- [3] ROBUS Cache Planner, <https://github.com/dukedbgrou/ROBUS>.
- [4] Spark: Lightning fast cluster computing, <https://spark.apache.org/>.
- [5] TPC-DS benchmark, <http://www.tpc.org/tpcds/>.
- [6] A. Abdulkadiroglu and T. Sonmez. Random serial dictatorship and the core from random endowments in house allocation problems. *Econometrica*, 66(3):689–701, 1998.
- [7] P. Agrawal, D. Kifer, and C. Olston. Scheduling shared scans of large data files. *Proc. VLDB Endow.*, 1(1):958–969, Aug. 2008.
- [8] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *Proceedings of the 26th International Conference on Very Large Data Bases, VLDB '00*, pages 496–505, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.
- [9] M. Andrews. Instability of the proportional fair scheduling algorithm for hdr. *Wireless Communications, IEEE Transactions on*, 3(5):1422–1426, 2004.
- [10] S. Arora, E. Hazan, and S. Kale. The multiplicative weights update method: A meta algorithm and applications. *Theory of Computing*, 8:121–164, 2005.
- [11] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-tenant databases for software as a service: Schema-mapping techniques. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08*, pages 1195–1206, New York, NY, USA, 2008. ACM.
- [12] R. J. Aumann. Markets with a continuum of traders. *Econometrica: Journal of the Econometric Society*, pages 39–50, 1964.
- [13] M. Berkelaar, K. Eikland, and P. Notebaert. *lpsolve : Open source (Mixed-Integer) Linear Programming system*.
- [14] A. Bhargat, S. Gollapudi, and K. Munagala. Optimal auctions via the multiplicative weight method. In *Proceedings of the fourteenth ACM conference on Electronic commerce*, pages 73–90. ACM, 2013.
- [15] A. Bogomolnaia and H. Moulin. A new solution to the random assignment problem. *Journal of Economic Theory*, 100(2):295–328, 2001.
- [16] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2007.
- [17] E. Budish. The combinatorial assignment problem: Approximate competitive equilibrium from equal incomes. *Journal of Political Economy*, 119(6):1061–1103, 2011.
- [18] E. Budish, Y.-K. Che, F. Kojima, and P. Milgrom. Designing random allocation mechanisms: Theory and applications. *American Economic Review*, 103(2):585–623, 2013.
- [19] Y. Cai, C. Daskalakis, and S. M. Weinberg. Optimal multi-dimensional mechanism design: Reducing revenue to welfare maximization. In *FOCS*, 2012.
- [20] D. Chakrabarty, A. Mehta, and V. Vazirani. Design is as easy as optimization. In M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, editors, *Automata, Languages and Programming*, volume 4051 of *Lecture Notes in Computer Science*, pages 477–488. Springer Berlin Heidelberg, 2006.
- [21] G. Debreu and H. Scarf. A limit theorem on the core of an economy. *International Economic Review*, 4(3):pp. 235–246, 1963.
- [22] I. Elghandour and A. Aboulnga. Restore: Reusing results of mapreduce jobs. *Proc. VLDB Endow.*, 5(6):586–597, Feb. 2012.
- [23] D. K. Foley. Lindahl’s solution and the core of an economy with public goods. *Econometrica: Journal of the Econometric Society*, pages 66–72, 1970.
- [24] Y. Freund and R. E. Schapire. Adaptive game playing using multiplicative weights. *Games and Economic Behavior*, 29(1–2):79 – 103, 1999.
- [25] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, NSDI'11*, pages 323–336, Berkeley, CA, USA, 2011. USENIX Association.
- [26] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 365–378. ACM, 2013.
- [27] D. Gillies. *Some Theorems on n-Person Games*. PhD thesis, Princeton University, 1953.

- [28] J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 331–342, New York, NY, USA, 2001. ACM.
- [29] J. Gray, K. Baclawski, P. Sundaresan, and S. Englert. Quickly generating billion-record synthetic databases. Association for Computing Machinery, Inc., January 1994.
- [30] H. Gupta and I. S. Mumick. Selection of views to materialize in a data warehouse. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):24–43, 2005.
- [31] S. Im, J. Kulkarni, and K. Munagala. Competitive algorithms from competitive equilibria: Non-clairvoyant scheduling under polyhedral constraints. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, STOC '14, pages 313–322, 2014.
- [32] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.
- [33] D. Jacobs, S. Aulbach, and T. U. MÄijnchen. Rumination on multi-tenant databases. In *BTW Proceedings, volume 103 of LNI*, pages 514–521. GI, 2007.
- [34] K. Jain and V. V. Vazirani. Eisenberg-gale markets: Algorithms and structural properties. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 364–373. ACM, 2007.
- [35] R. K. Jain, D.-M. W. Chiu, and W. R. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical report, Digital Equipment Corporation, Sept. 1984.
- [36] M. Kaneko. The ratio equilibrium and a voting game in a public goods economy. *Journal of Economic Theory*, 16(2):123–136, 1977.
- [37] I. Kash, A. D. Procaccia, and N. Shah. No agent left behind: Dynamic fair division of multiple resources. In *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '13, pages 351–358, Richland, SC, 2013. International Foundation for Autonomous Agents and Multiagent Systems.
- [38] F. Kelly, L. Massoulié, and N. Walton. Resource pooling in congested networks: proportional fairness and product form. *Queueing Systems*, 63(1-4):165–194, 2009.
- [39] F. P. Kelly, A. K. Maulloo, and D. K. H. Tan. Rate control for communication networks: Shadow prices, proportional fairness and stability. *The Journal of the Operational Research Society*, 49(3):pp. 237–252, 1998.
- [40] Y. Kotidis and N. Roussopoulos. Dynamat: A dynamic view management system for data warehouses. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 371–382, New York, NY, USA, 1999. ACM.
- [41] H. W. Kuhn and A. W. Tucker. Nonlinear programming. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492, Berkeley, Calif., 1951. University of California Press.
- [42] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, and M. J. Carey. Miso: Souping up big data query processing with a multistore system. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1591–1602, New York, NY, USA, 2014. ACM.
- [43] H. Li, A. Ghodsi, M. Zaharia, E. Baldeschwieler, S. Shenker, and I. Stoica. Tachyon: Memory throughput i/o for cluster computing frameworks. *LADIS*, 18:1, 2013.
- [44] A. Mas-Colell and J. Silvestre. Cost share equilibria: A lindahl approach. *Journal of Economic Theory*, 47(2):239–256, 1989.
- [45] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 307–318, New York, NY, USA, 2001. ACM.
- [46] V. Narasayya, S. Das, M. Syamala, S. Chaudhuri, F. Li, and H. Park. A demonstration of sqlvm: Performance isolation in multi-tenant relational database-as-a-service. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1077–1080, New York, NY, USA, 2013. ACM.
- [47] J. Nash. The bargaining problem. *Econometrica*, 18(2):155–162, 1950.
- [48] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare: Sharing across multiple queries in mapreduce. *Proc. VLDB Endow.*, 3(1-2):494–505, Sept. 2010.
- [49] D. C. Parkes, A. D. Procaccia, and N. Shah. Beyond dominant resource fairness: Extensions, limitations, and indivisibilities. In *Proceedings of the 13th ACM Conference on Electronic Commerce*, EC '12, pages 808–825, New York, NY, USA, 2012. ACM.
- [50] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's adolescence: An analysis of hadoop usage in scientific workloads. *Proc. VLDB Endow.*, 6(10):853–864, Aug. 2013.
- [51] Z. Ren, X. Xu, J. Wan, W. Shi, and M. Zhou. Workload characterization on a production hadoop cluster: A case study on taobao. In *IISWC'12*, pages 3–13, 2012.
- [52] H. E. Scarf. The core of an n person game. *Econometrica*, 35(1):pp. 50–69, 1967.
- [53] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, Mar. 1988.
- [54] A. L. Stolyar. On the asymptotic optimality of the gradient scheduling algorithm for multiuser throughput allocation. *Oper. Res.*, 53(1):12–25, 2005.
- [55] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 136–145, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.
- [56] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.
- [57] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: Dynamic bandwidth sharing in a dbms. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 723–734. VLDB Endowment, 2007.

APPENDIX

A. EXPERIMENT RESULTS

A.1 Results of experiments on effect of data sharing

Metric	STATIC	MMF	FASTPF	OPTP
Throughput(/min)	6.00	9.42	9.42	10.08
Avg cache util.	0.34	0.87	0.86	0.88
Hit ratio	0.42	0.67	0.67	0.68
Fairness index	1.00	0.98	0.94	0.84

Table 15: Performance of algorithms on setup \mathcal{G}_1

Metric	STATIC	MMF	FASTPF	OPTP
Throughput(/min)	5.70	7.20	7.44	5.64
Avg cache util.	0.34	0.93	0.90	0.94
Hit ratio	0.43	0.57	0.61	0.63
Fairness index	1.00	0.96	0.92	0.78

Table 16: Performance of algorithms on setup \mathcal{G}_2

Metric	STATIC	MMF	FASTPF	OPTP
Throughput(/min)	5.34	7.44	7.38	7.92
Avg cache util.	0.30	0.93	0.93	0.94
Hit ratio	0.38	0.60	0.59	0.58
Fairness index	1.00	0.98	0.92	0.72

Table 17: Performance of algorithms on setup \mathcal{G}_3

Metric	STATIC	MMF	FASTPF	OPTP
Throughput(/min)	4.20	5.64	5.76	6.00
Avg cache util.	0.28	0.89	0.88	0.92
Hit ratio	0.34	0.50	0.56	0.55
Fairness index	1.00	0.96	0.96	0.99

Table 18: Performance of algorithms on setup \mathcal{G}_4

A.2 Results of experiments on effect of query arrival rate

Metric	STATIC	MMF	FASTPF	OPTP
Throughput(/min)	5.76	6.42	6.72	6.90
Avg cache util.	0.77	0.93	0.93	0.94
Hit ratio	0.40	0.50	0.49	0.51
Fairness index	1.00	1.00	0.99	0.97

Table 19: Performance of algorithms on setup *low*

Metric	STATIC	MMF	FASTPF	OPTP
Throughput(/min)	6.12	6.78	6.96	6.96
Avg cache util.	0.72	0.90	0.89	0.90
Hit ratio	0.44	0.49	0.49	0.56
Fairness index	1.00	1.00	0.98	0.87

Table 20: Performance of algorithms on setup *mid*

Metric	STATIC	MMF	FASTPF	OPTP
Throughput(/min)	5.52	6.12	6.30	6.54
Avg cache util.	0.69	0.90	0.91	0.91
Hit ratio	0.39	0.48	0.48	0.51
Fairness index	1.00	1.00	1.00	0.89

Table 21: Performance of algorithms on setup *high*

A.3 Results of experiments on effect of number of tenants

Metric	STATIC	MMF	FASTPF	OPTP
Throughput(/min)	7.00	10.00	9.70	10.40
Avg cache util.	0.67	0.93	0.93	0.97
Hit ratio	0.50	0.68	0.68	0.68
Fairness index	1.00	0.98	1.00	1.00

Table 22: Performance of algorithms on setup with 2 tenants

Metric	STATIC	MMF	FASTPF	OPTP
Throughput(/min)	6.00	9.40	9.40	10.10
Avg cache util.	0.34	0.87	0.86	0.88
Hit ratio	0.42	0.67	0.67	0.68
Fairness index	1.00	0.98	0.94	0.84

Table 23: Performance of algorithms on setup with 4 tenants

Metric	STATIC	MMF	FASTPF	OPTP
Throughput(/min)	5.34	8.34	8.22	9.18
Avg cache util.	0.07	0.82	0.82	0.87
Hit ratio	0.26	0.65	0.65	0.68
Fairness index	1.00	0.94	0.91	0.78

Table 24: Performance of algorithms on setup with 8 tenants