ROBUS: Fair Cache Allocation for 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 the 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 we also define envy freeness via the notion of *core* from cooperative game theory. Our cache management platform, 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 an industry-standard workload shows that our algorithms score high on both performance and fairness metrics across a wide variety of practical multi-tenant setups.

1. INTRODUCTION

Two recent trends in data processing are: (i) the aggressive use of memory to speed up processing by caching datasets [10, 33], and (ii) the use of multi-tenant clusters for analyzing large and diverse datasets [72]. The growing popularity of systems like Apache Spark [10] and SAP HANA [33] highlight these trends.

*This research is supported by NSF grants CNS-1423128 and IIS-1423124.

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'17, May 14-19, 2017, Chicago, IL, USA

© 2017 ACM. ISBN 978-1-4503-4197-4/17/05...\$15.00

DOI: <http://dx.doi.org/10.1145/3035918.3064018>

Caching has become ubiquitous.

Traditionally, caches were only used internally in database systems as a buffer pool [61] for recently accessed data pages from disk. Modern data analytics systems additionally expose APIs for data caching to application developers and data analysts.

```
1 // read sales data (id, year, product, city, sales)
2 val sales = sc.textFile("sales.txt").map(_._split(",")).
3 // find sales of the current year and cache
4 val salesThisYear = sales.filter(_(1)=="2016").cache
5 // find sales in the city "SF" this year
6 salesThisYear.filter(_(3)=="SF").map(_(4).toInt).sum
7 // find sales of the product "iCar" this year
8 salesThisYear.filter(_(2)=="iCar").map(_(4).toInt).sum
```

Listing 1: Example Spark program showing user-directed caching

For example, Spark introduces an abstraction called *Resilient Distributed Dataset (RDD)* to represent any data relevant to modern analytics: files (on a local or distributed file-system), tables (horizontally or vertically partitioned), vertices or edges of graphs, statistical models learned from data, etc. A user can create an RDD directly from data residing on a local or distributed file-system, or by applying a transformation to one or more other RDDs. The user can then direct the system to cache the RDD in memory. Listing 1 gives an example.

As we show in Figure 1, computations done on RDDs cached in memory run one to two orders of magnitude faster than when the data resides on disk [10]. The experiment in Figure 1 was carried out on a 10-node cluster using three queries of varying complexity on TPC-H data stored in text format on the Hadoop Distributed FileSystem (HDFS). Two different architectures for caching were considered in Figure 1: (i) caching using Spark's internal RDD cache store, and also (ii) caching RDDs on an external memory store called Alluxio [54]. Spark's RDD cache provides much bigger speedups because it stores data in an optimized columnar format, whereas Alluxio simply mimics the data layout on disk (which, in this case, is row-based) in memory.

Caching under multi-tenancy is nontrivial.

Cache is always a limited resource since the total size of memory in a cluster is usually orders of magnitude smaller than the data sizes stored and queried in the cluster. As we saw in Figure 1, in a multi-tenant cluster that has multiple users, massive performance speedups will be experienced by tenants who get to use the in-memory cache. Thus, it is highly desirable in multi-tenant clusters that low-priority tenants should not be able to hog the available cache, and prevent other tenants from getting the performance benefits they deserve. Unlike a resource like a CPU core which is used by one tenant at a time, a cached data item can simultaneously benefit a high-priority and a low-priority tenant. Furthermore, different tenants will have different utilities for objects that could be placed in the cache.

Figure 1: Comparing performance of different caching alternatives for TPC-H queries

When faced with such challenges, traditional cache allocation policies can lead to user dissatisfaction, poor or unpredictable performance, and low resource utilization. We will illustrate the problems and opportunities through an example.

Multiple tenants: The predominant practice in the industry is to group similar users—e.g., all users submitting batch jobs—into queues (or, pools). Each queue forms a tenant in the cluster. Our example uses three tenants: (i) *Analyst*, the business analysts in the company, (ii) *Engineer*, the developers in the company who build data-driven applications such as recommendation models, and (iii) *VP*, the top-level management in the company such as the CEO and the Chief Security Officer who look at hourly and daily reports.

Cacheable entities: These three tenants will benefit from caching one or more of three views—*R*, *S*, and *P*—each of size *M* bytes. Throughout this paper, “view” refers to any data item that can be cached to give a performance benefit. For SQL workloads, a view corresponds to a SQL expression, like any candidate view generated by a materialized view selection algorithm [41, 74, 15, 51]. For broader data analytics—e.g., graph processing—a view corresponds to a dataset on which the user has put a cache directive (recall the example Spark program from Figure 1).

Tenant	<i>R</i>	<i>S</i>	<i>P</i>
Analyst	2	1	0
Engineer	2	1	0
VP	0	1	2

Table 1: Utilities of cached views to tenants

Utilities: The matrix in Table 1 shows the *utility* that each tenant gets if the corresponding view were to be cached in memory. A simple definition of utility we will use in this paper is the savings in I/O because data is read from the in-memory cache versus disk. For example, if view *R* is cached in memory, then tenant *Analyst* will get a utility of two units. One common pattern in multi-tenant clusters that we bring out in Table 1 is that view *R* could be the detailed logs that business analysts and developers access quite often; view *P* could be a table that only the top-level management has access to; while view *S* could be a materialized view with aggregated information shared by all tenants.

Scenario 1: Suppose the in-memory cache has a total size of *M* bytes. If a static and equal partitioning of the cache is used, each tenant will be entitled to $\frac{M}{3}$ bytes of cache memory. Recall that each of the views *R*, *S*, and *P* are *M* bytes each; so none of them will fit in their $\frac{M}{3}$ bytes of cache. If, instead, the cache is kept unpartitioned, and a *Least Recently Used (LRU)* policy is used for cache allocation, the view *R* will likely remain cached for the most time on account of it being the most used. Thus, the Analyst and Engineer tenants will see performance speedups. However, the VP tenant’s workload will not see any benefits.

Scenario 2: To satisfy the VP tenant, suppose the cluster admin decides to give the VP tenant 50% higher priority than the other tenants. So, she assigns weights to the Analyst, Engineer, and VP tenants in the ratio 1 : 1 : 1.5. The cache is now allocated based on the weighted utility of the tenants. But, even with this change,

view *R* will be the only one cached since it has the highest weighted utility of 4 ($= 2 \times 1 + 2 \times 1$); higher than view *S*’s ($1 \times 1 + 1 \times 1 + 1 \times 1.5 = 3.5$), and view *P*’s ($2 \times 1.5 = 3$).

Scenario 3: To improve the poor performance seen by the VP tenant, the cluster admin now decides to double the size of cache memory. Now two views will fit in the 2*M*-sized cache. However, even after this massive investment, the VP tenant will only see a minor increase in performance compared to the Analyst and Engineer tenants: views *R* and *S* will now be cached since they together have the highest weighted utility of 7.5 (4 for *R* + 3.5 for *S*); higher than 7 for *R* and *P*, and 6.5 for *S* and *P*.

Better scenarios: Let us consider what would have been an *ideal* cache allocation. An alternative in Scenarios 1 and 2 is to cache view *S* instead of *R*. While *S* has a slightly lower weighted utility, all three tenants will benefit from caching *S*. An alternative in Scenario 3 is to cache *R* and *P* which will also give performance benefits to all three tenants while only being slightly lower in overall weighted utility than caching *R* and *S*. In particular, the *VP* tenant will now see major benefits from doubling the cache size.

1.1 Contributions

- Our example shows key tradeoffs which create the need to make principled choices during memory allocation for caching in multi-tenant clusters. In this paper, we develop cache allocation algorithms that can speed up a multi-tenant workload while guaranteeing *fairness* in terms of the tenants’ performance.
- In Section 2, we propose ROBUS, a platform to optimize multi-tenant workloads in an *online* manner using the cache. ROBUS groups queries in small time-based batches and employs randomized cache allocation on each batch.
- Section 3 considers shared resource allocation within a batch, and enumerates properties desired 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 *proportionally fair* policy 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. 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 describe our implementation of ROBUS on a multi-tenant Spark cluster. Motivated by practical use cases, we develop a 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 across a comprehensive spectrum of scenarios.

2. THE ROBUS PLATFORM

ROBUS (Random Optimized Batch Utility Sharing), shown in Figure 2, is the cache management platform we have developed for multi-tenant data-parallel workloads.

Multi-tenant cluster.

ROBUS is designed to easily fit in modern data analytics systems like Hadoop and Spark. These systems support multi-tenancy. Each tenant submits its workload in an online fashion to a designated queue which is characterized by a weight indicating the ten-

Figure 2: ROBUS platform

ant’s *fair* share of system resources. A fair share scheduler [4, 36] is typically employed to schedule tasks (smallest unit of work) for the workload.

Memory allocated to an application is divided into two parts: a heap space for run-time objects and a data cache. While the heap is divided across all concurrently running tasks and is allocated by the task scheduler (e.g., Hadoop fair scheduler [4], DRF [36]), the cache is shared by all queries in execution simultaneously and is managed by ROBUS. Referring back to Figure 1, ROBUS uses the Spark RDD cache architecture to implement the shared cache because of its superior performance.

ROBUS workflow.

ROBUS is designed as a middleware between application tenants and the computation platform. Queries submitted by tenants to queues are processed by ROBUS in *batches* of a fixed time interval. Queries within a batch are optimized together and are scheduled for execution at the same time. This framework is motivated from prior multi-query optimization works such as Shared Scans [14] where a job/query is allowed to wait for a maximum time bounded by a preset *batch size* before it gets scheduled. ROBUS uses a similar notion where queries will never have to wait for more than the batch interval before being scheduled. Furthermore, ROBUS distinguishes between *interactive tenants* and *batch tenants*. Interactive tenants run short queries for which they want to see immediate response. On the other hand, batch tenants are more throughput-sensitive. That is, they would like to see their workload finished as quickly as possible, and care less about how soon individual queries in the workload are finished. If a tenant declares itself as interactive to ROBUS, then ROBUS sets the batch interval to zero for that tenant. Thus, queries from this tenant will be scheduled immediately and will not be considered by ROBUS’s optimization algorithm. We will discuss a more sophisticated batching model later in Section 5.4.

Figure 2 illustrates ROBUS workflow. ROBUS carries out the following four steps in a repeatedly-running loop. Step 1 removes a batch of queries from the tenants’ queues. Step 2 runs an algorithm over this entire batch to select a set of views to cache from a predetermined set of candidate views. (The set of candidate views can be computed separately using any state-of-the-art algorithm for materialized view selection from a query workload [15, 51]; and this computation is not part of ROBUS’s repeatedly-running loop.) This algorithm to select cached views from the candidate views simultaneously optimizes performance and fairness. Designing this algorithm is the main focus of this paper. Step 3 brings the views recommended in Step 2 to the cache (if they are not already in the cache) before rewriting queries to use the views. This step also directs the system to *uncache* any previously cached views not required for this batch. The entire batch of the rewritten queries is then submitted for execution in Step 4.

Life-cycle of a query in this workflow consists of the following three phases broadly:

Batching Delay: Time from the query arrival to ROBUS submitting it for execution; Includes both the wait time before Step 1 as well as the time spent in ROBUS optimization.

Scheduling Delay: The wait the query experiences in a scheduler queue before the first task of it is launched.

Execution Time: Time span from the first task start to the last task completion for the query.

The batch size configuration drives the trade-offs in Batching Delay and Execution Time. For example, a very low setting of batch size ensures a lower Batching Delay, but at the same time, lowers the chance of savings in Execution Time. This happens because a smaller batch results in many cache updates in order to get locally-optimal views for each batch into the cache; thereby making most of the queries disk-bound. We evaluate these tradeoffs and also give some guidelines in tuning the batch size in Section 5.

Views and Utilities.

Step 2 takes three inputs: (i) a set of candidate views, (ii) a utility estimation model for cached views, and (iii) total cache budget (i.e., memory available for caching). The candidate view generation in ROBUS is a pluggable module. By default, the candidate views for a SQL query are the base tables accessed by the query. The ROBUS prototype we evaluate in Section 5 uses a predetermined set of candidate views consisting of different vertical projections of the input tables. This set can be made richer using the pluggability of the module. For workloads like machine learning and graph processing, the candidate views are the datasets on which the user has put a cache directive. A view management algorithm from the database literature [15, 51] can be employed to dynamically build and maintain a set of candidate views on disk to suit the workload. Such an algorithm, if employed, would run in the background. ROBUS also supports a pluggable Step 3 where a query is rewritten to use the views selected for caching in Step 2. In future, we plan to extend Step 3 to support re-optimization of the query based on the cached views, which could change the query plan entirely.

ROBUS view selection module employs a pluggable utility estimation model to estimate the utility provided to a query by any cached view. ROBUS makes polynomial (in the number of tenants) calls to this model while optimizing a batch. ROBUS currently models these utilities as savings in disk I/O costs if the view were to be read off of the in-memory cache versus disk. This approach keeps the models simple and widely applicable. It could be extended to richer utility models required to support more complex candidate view selection algorithms from literature that consider interactions among views [53]. Total utility of a cache configuration for a tenant is computed by adding up estimated utilities of the queries submitted by the tenant. The tenant utilities thus computed are used in recommending an optimal set of views to cache. The view selection policy attempts to come up with a cache configuration that provides a weight-proportional performance speedup to the tenants’ workload. The guarantees provided are in expectation, i.e., over a number of batches. This is due to the fact that a cache directive is specified over an entire view. In other words, a view is treated indivisible at the program interface level. While individual partitions of a view on a node are managed by a standard LRU cache manager by the system underneath, ROBUS operates at a higher level of data abstraction.

3. FAIRNESS PROPERTIES AND POLICIES FOR SINGLE BATCH

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 [25, 13, 23], 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 [26] 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 that $\sum_{S \in Q} x_S = 1$. We define the seminorm $\|\cdot\|$ on an allocation $\mathbf{x}$ as $\|\mathbf{x}\| = \sum_{S \in (Q \setminus \emptyset)} x_S$.

In other words, $\|\mathbf{x}\|$ is the total quantity of probability mass that allocation $\mathbf{x}$ places on “desireable” (nonempty) configurations. 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. In classical economic theory, a *fair* allocation is one that is *Pareto-efficient* and *envy-free* [71]. Also, there has been much recent work in economics

and computer science on heterogeneous resource allocation problems that consider these properties and the notion of *Sharing Incentive* [36, 63, 48]. 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 having access to all of the resources with probability $\frac{1}{N}$. More formally, 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.

The above desiderata omit a formal definition of envy-freeness (informally, that no tenant should prefer the allocation to another tenant) which is something we revisit later. 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) [64, 36]. In our case, since the queries are seen by the query optimizer, strategy-proofness is not an issue.

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 (STATIC). Static partitioning is the algorithm that 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 thus trivially not Pareto efficient. As mentioned previously, such examples motivate the randomization framework to start with.

Random Serial Dictatorship (RSD). Random serial dictatorship (RSD) is widely considered [23, 13] for problems like 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 so the allocation is $\langle x_R = \frac{1}{3}, x_S = \frac{1}{3}, x_P = \frac{1}{3} \rangle$, where each tenant has the same utility in expectation.

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, RSD fails to capture the shared nature of our problem. Consider Table 3; RSD computes the same allocation as in the example in Table 2, $\langle x_R = \frac{1}{3}, x_S = \frac{1}{3}, x_P = \frac{1}{3} \rangle$. On this example, although 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 $\langle x_R = 0, x_S = 1, x_P = 0 \rangle$ 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 (OPTP). Next, consider the mechanism that 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 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 the allocation $\langle x_R = \frac{1}{2}, x_S = \frac{1}{2} \rangle$. This allocation is both SI and PE.

3.3 Envy-freeness and the Core

SI and PE are necessary conditions for fair solutions, but are not sufficient. The above discussion omits the notion of *envy free*, meaning no tenant envies how the allocation treats another tenant. Defining envy free is straightforward in settings where resources are partitioned between tenants: No tenant should derive higher utility from the allocation to another tenant. However, in our setting, resources (views) are shared between tenants and thus envy-freeness is ill-defined a priori. We want to reconsider envy in the public goods setting as a notion of proportionality and stability, a sort of group sharing-incentive. In order to develop the intuition behind this, 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, intuitively because the outcome is highly disproportional to the interests of the agents.

Randomized Core. In order to formalize this intuition, we borrow the notion of core from cooperative game theory and exchange market economics [38, 67, 20, 31]. 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 probability that the more demanded view R is cached is increased by a proportionally larger amount. In societal terms, each user perceives his tax dollars as being spent fairly.

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 [34, 55] attempts to find per-tenant prices that implement a Walrasian equilibrium, while the second, termed ratio equilibrium [47] 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 (PF)

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 [52]. The proof also follows easily from the classic first order optimality condition of PF [59]; 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 condi-

tions, we have:

$$\begin{aligned} x_S > 0 &\implies \sum_i \frac{U_i(S)}{U_i(\mathbf{x})} = d \\ x_S = 0 &\implies \sum_i \frac{U_i(S)}{U_i(\mathbf{x})} \leq d \end{aligned}$$

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

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

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 $\|\mathbf{x}\| \leq 1$.

Utilities under MMF and PF. (PF) always has larger total utility than MMF in certain canonical scenarios. Define a *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 [46] 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 $\alpha\beta \geq 1$. 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 5 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.

Algorithm	SI	PE	CORE
STATIC	✓		
RSD	✓		
OPTP		✓	
MMF	✓	✓	
PF	✓	✓	✓

Table 5: 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 only give the results for these precise methods here; the technical proofs are included in the appendix. We also 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 [18, 35] 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. In our presentation, we assume WELFARE solves the welfare maximization problem exactly. A discussion on modeling inaccuracies in the utility model is left to Appendix C. Our algorithms, described next, will make polynomially many calls to WELFARE.

Multiplicative Weight Method. We first mention the multiplicative weight method, which will serve as a common subroutine to all our provably good algorithms. This classical framework [18, 35] uses a Lagrangian update to decide feasibility of linear constraints to arbitrary precision. We formally present the Arora-Hazan-Kale (AHK) procedure [18] for solving this problem in Appendix A.

4.1 Proportional Fairness

Our PF algorithm uses the AHK algorithm as a subroutine and considers dual weights to find an additive ϵ approximation solution. The full details of the analysis can be found in Appendix B. The primary result is the following theorem:

THEOREM 3. *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.*

4.2 Max-min Fairness

We present an algorithm SIMPLEMMF in Appendix D 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 [37]. We note that the idea of applying the multiplicative weight method to compute max-min utility also appeared in [28]. The algorithm gives the following result:

THEOREM 4. *An approximation algorithm for SIMPLEMMF 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 3), 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 the random optimal configurations and the solution to procedure given in Algorithm 3.

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 .

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

Given a configuration space $\mathcal{S}$, we can solve the program (2) using gradient descent, as shown in Algorithm 1. As precomputation, for each configuration $S \in \mathcal{S}$, we precompute $V_i(S)$. Then $V_i(\mathbf{x}) = \sum_{S \in \mathcal{S}} V_i(S)x_S$.

Max-min Fairness. Using the precomputed configuration space $\mathcal{S}$, we solve SIMPLEMMF 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 [21]). In order to

---

Algorithm 1 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
-

compute the MMF allocation, we iteratively compute the lexicographically max-min allocation using the above LP. The details are standard; see for instance [37]. 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

We evaluate ROBUS caching policies on some of the practical multi-tenant setups listed below.

- **Analysts:** Tenants correspond to various BI analysts in an enterprise that all run a similar workload. A few datasets are frequently accessed by all tenants suggesting a good opportunity for shared optimization.
- **ETL+Analysts:** All analysts have similar data access patterns as above. But additionally, a tenant runs ETL workload that may touch different datasets.
- **Workflow+Engineering:** Engineering workload is of bursty nature. Depending on the time of day, engineering queues have small bursts of work whereas workflow queues have pre-scheduled well-balanced workload.

5.1 Setup and Methodology

Figure 2 has presented the architecture of ROBUS. We use Apache Spark [10] to build a system prototype. A long running Spark context is shared among multiple queues, each queue corresponding to a tenant. The Spark context has access to the entire RDD cache in the cluster. Spark's internal fair share scheduler is configured with a pool dedicated to each queue; the fair share properties of the pool are set proportional to the weight of the corresponding queue.

Spark version	1.6.1
Number of worker nodes	10
Total number of cores	80
Total executor memory	100GB
Memory used for storage (Cache)	10GB

Table 6: Test cluster setup

Table 6 presents our test cluster setup. We generate two types of data: (a) A set of 30 datasets with varying sizes each matching schema of the “sales” tables—*store_sales*, *catalog_sales*, and *web_sales*—from TPC-DS benchmark [11], and (b) All datasets from TPC-H benchmark [12] generated at scale 10.

The first category of data represents raw fact/log data that comes into the cluster from the OLTP/operational databases in a company. This data is processed by synthetically-generated ETL and exploratory SQL queries, each performing scans and aggregations over a dataset. We refer to this category of queries as the Sales workload. Total size of Sales data on disk is 600GB. We treat a

Figure 3: Workload generation for ROBUS

vertical projection of each dataset consisting of the most popularly accessed columns as a candidate view. Sizes of these views when loaded to cache range from 118MB to 3.6GB.

The second category of data, TPC-H, represents a typical warehouse. This data is queried by standard TPC-H benchmark queries which involve more complex operations, such as joins, compared to the Sales workload. Like in the case of the Sales workload, we support materialization of a vertical projection on each dataset. Sizes of the cached views range from 10MB to 3GB.

We set the cache size to 10GB, 10% of the total executor memory, leaving aside the rest as the heap space for execution objects.

The tenant utility model we use to estimate the utility of a cache configuration is a simple approximation to the I/O savings. If a view in a cache configuration can answer a query, the utility for the query is set equal to the size of the dataset on disk from which the view is created. The utility for a batch of queries is the sum of utilities for each query within the batch minus the cost of building the configuration. The dataset sizes are stored in a hashtable to speed up the utility evaluation.

ROBUS workflow is described in Section 2 already. The cache update phase in our prototype first *uncaches* any views not required for the current batch and then materializes each newly selected view in the configuration by running a *count(*)* query over it. While the materialization queries are running, we also submit the queries from the batch which are not dependent on any view being materialized in order to make full use of resources.

Workload Arrival and Data Access. Figure 3 shows our workload generation process. Several studies have established that query arrival times follow a Poisson distribution [40, 66]. We use the same in our prototype. Previous studies have also indicated that the data accessed by analytical workloads follows a Zipf distribution [40, 65]: A small number of datasets are more popular than others, while there is a long tail of datasets that are only sporadically accessed. To replicate such data access, our synthetic Sales workload generator picks a dataset from a Zipfian distribution provided at the time of configuration and adds grouping and aggregation predicates from a probability distribution over a set of predicates defined over the datasets at the time of configuration. The TPC-H workload generator, on the other hand, picks a benchmark query from a probability distribution over the 22 benchmark queries provided at the time of configuration. Queries used in evaluation are all submitted using SparkSQL APIs.

5.2 Performance gains due to batching

We first analyze the performance obtained on different granularity of data caching. We compare three policies here:

1. **BASELINE:** The case where no view is ever cached.
2. **GREEDY ONLINE:** Each query greedily materializes a view. The LRU policy at data partition level takes care of cache management.
3. **ROBUS OPTP:** Queries in a fixed time batch are analyzed to

Figure 4: Workload throughput on different cache granularities

materialize a *configuration* (refer Definition 1) that provides maximum utility to the query batch.

We use a setup with four tenants generating queries with mean inter-arrival time of 10 seconds each. While three of the tenants generate queries over TPC-H data, the fourth tenant is configured with Sales workload generator. The query generators are kept active for 600 seconds while the ROBUS OPTP policy is configured to use 40 second batches. Figure 4 compares the policies based on the overall throughput. Figure 5 plots for all queries from a tenant the end-to-end (E2E) latency and the time breakup of the three phases of query life-cycle (as defined in Section 2).

Surprisingly, GREEDY ONLINE caching fares worse than BASELINE. Due to a large variability in the data accessed by tenants' workload, greedily caching views leads to too many cache transitions making most queries I/O bound. Furthermore, the extra overhead in running cache queries results in a longer Scheduling Delay as can be seen in Figure 5(b). While the batched caching approach used in ROBUS causes an increase in Batching Delay by a few seconds (capped by 40s in this case), the execution times, as seen in the graph, drop by one or two orders of magnitude for most of the queries. This triggers a significant drop in the Scheduling Delay as well. The overall throughput gain, as seen in Figure 4, is over 3x.

5.3 Evaluation of batched caching policies

The batched caching policy evaluated above is designed with a goal of utility maximization (OPTP) and is unfair in theory (Section 3). Here we compare it with max-min fair and proportional fair policies, and also with a commonly used static partitioning.

- **STATIC:** Cache is partitioned in proportion to weights of the tenants with each partition optimized individually.
- **MMF:** Max-min fairness implementation described in Section 4.3.
- **FASTPF:** Proportional fairness implementation described in Section 4.3.
- **OPTP:** Workload from a batch is treated as belonging to a single tenant – a special case of either MMF or FASTPF.

In order to compare these policies, 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)
3. Number of tenants (Section 5.3.3)
4. Batch size and Cache state (Section 5.3.4)

We first list the performance metrics we gathered.

Performance Metrics.

1. Throughput Gain. Throughput is simply the number of queries served per unit time. The *throughput gain* for a cache setup corresponds to the ratio of the throughput of the setup to the throughput of BASELINE, wherein no cache is used, under the same workload.

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 [43]. As our work is about speeding up queries, we use relative speedups across queries while deriving fairness. The

Figure 5: Comparing cache policies in terms of E2E latency of queries including the breakup across the three phases of query lifetime

Figure 6: Effect of data sharing changes on four equi-paced tenants

baseline again is the case of not using cache. Here, X_i is the mean speedup for tenant i , and λ_i is the weight of tenant i .

$$\text{Fairness index} = \frac{(\sum_{i=1}^n \frac{X_i}{\lambda_i})^2}{n \sum_{i=1}^n (\frac{X_i}{\lambda_i})^2} \quad (4)$$

3. Average Cache Utilization. This is simply the average fraction of cache in use during workload execution.

4. Hit Ratio. The fraction of queries served off cached views in a workload.

We emphasize that these metrics are computed over long time horizons. Some of the other metrics we collect include flow time, mean execution time, mean wait time, and wait time fairness index. These, however, are not presented due to space constraints.

5.3.1 Effect of data sharing among tenants

To study the impact of different data sharing patterns on the performance of algorithms, we create two different setups with four tenants. In setup $\mathcal{G}_1$, all four tenants generate queries from a single probability distribution over Sales datasets. Setup $\mathcal{G}_2$, however, showcases a big heterogeneity among the tenants: Two of the tenants generate queries over TPC-H data; The other two generate Sales queries with datasets picked from different probability distributions. Each tenant generates queries with mean query inter-arrival time set to 10 seconds. Batch size is set to 30 seconds. We run 30 batches of workload for every data point.

Figure 6 shows how algorithms perform on the two setups. The homogeneous setup shows a throughput gain upwards of 2 on all algorithms except STATIC. OPTP, in particular, gives a high throughput by always caching views on the most frequently accessed datasets. All algorithms, including OPTP, score high on the fairness index as well. The heterogeneous setup distinguishes the algorithms better. Firstly, throughput goes down with heterogeneity in all cases. The gap in throughput of STATIC with the shared cache policies is small despite the shared policies caching significantly more data. This is due to the frequent updates to cache configuration per batch in this setup. MMF, in particular, showcases very small gain in throughput. We revisit this phenomenon later in Section 5.4. OPTP scores poorly on fairness index in $\mathcal{G}_2$: It gives preferential treatment to the two TPC-H tenants due to data sharing between them while deny-

ing cache space to the other two tenants. Both MMF and FASTPF get an almost perfect fairness score irrespective of data sharing pattern among tenants.

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 two setups, $\mathcal{G}_3$ and $\mathcal{G}_4$, as described in Table 7. While the first setup leads to roughly equal number of queries from each tenant in a batch, the second setup generates three queries of tenant 2 for every query of tenant 1 on average. Here, the first tenant is configured to generate TPC-H workload while the second one generates Sales workload.

Setup	$\mathcal{G}_3$	$\mathcal{G}_4$
Number of tenants	2	2
Batch size (s)	36	36
Mean query inter-arrival rates (s)	(6, 6)	(12, 4)

Table 7: Query inter-arrival rates for different setups

Figure 7 shows the impact of variance in query arrival. Throughputs are much better in setup $\mathcal{G}_4$ because of a greater proportion of queries over Sales workload: These queries are all scan bound and get big speedups due to caching compared to some complex TPC-H queries that are not as heavily dependent on scans.

The average cache utilization and hit ratio, however, go down in the second setup. This, too, is due to the nature of the workload wherein the most popular views for the Sales workload submitted by tenant 2 are smaller in size compared to the most popular views for the TPC-H workload. OPTP favors the faster tenant in the second setup *unfairly*: It serves 8x queries off cache for the faster tenant compared to the queries served off cache for the slower tenant. Both MMF and FASTPF serve only about 3x queries off cache for the faster tenant which is in proportion to the number of queries submitted.

5.3.3 Effect of number of tenants

We present results on two setups: $\mathcal{G}_5$ with four tenants and $\mathcal{G}_6$ with 8 tenants. Each tenant generates Sales queries from a single probability distribution over datasets under both the setups. We try to keep the number of queries per batch the same by doubling the query inter-arrival rate with the number of tenants, batch size remaining the same. 30 batches of workload are run in each case. Figure 8 shows the results.

Figure 7: Effect of variance in query arrival rates

Figure 8: Effect of changing number of tenants

Figure 9: Effect of batch size on four equi-paced tenants setup

As the number of tenants goes up, both the average cache utilization and the hit ratio under STATIC drop sharply, whereas the average cache utilization of the other algorithms remain largely stable. The fairness index stays high for all algorithms in both setups since the tenants show a great deal of homogeneity in data access. Throughput gains by MMF and FASTPF are about 2x over those by STATIC, while OPTP improves it further by only about 30%.

5.3.4 Batch Size and Cache State

Our batched policies introduce additional optimization choices. Primarily, there are two tunable parameters in a view selection algorithm: 1. Batch size, and 2. State of cache across batches.

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

Batch size	0(BASELINE)	10	20	30	40	50
Mean Batching Delay	0s	5s	10s	15s	20s	25s
Mean Execution Time	83s	64s	67s	44s	48s	56s

Table 8: Query time statistics on stateless cache optimizations

We empirically compared how the algorithms react to these parameters. Figure 9 shows the effect of a change in batch size on two versions of FASTPF: One treating the cache as stateless(PFSL), and the other treating it as stateful(PFSF), with $\gamma = 2$. The two

versions are compared on a four tenant setup. Mean Batching Delay and Mean Execution Time for the stateless setups are included in Table 8. The best performance is seen on the stateless cache with batch sizes of 30 and 40 seconds. With queries arriving at a mean inter-arrival time of 10 seconds, each batch contains 10-18 queries here. Upon inspecting the cache configurations across batches, it is noticed that the cache fits 5-6 views on average. Each of the cached views, in most cases, is observed to serve at least two queries thus amortizing the cost of data loading well. Smaller batch sizes (10s and 20s) result in too many cache transitions adversely affecting performance speedups. Stateful cache optimizations help here by retaining the more popular views but only by a small margin. Larger batch sizes (50s and up) also show lower performance gains because they allow a relatively smaller fraction of queries the cache benefits, the cache budget remaining the same.

PFSL setups are observed to score low on fairness index in comparison to PFSF. This is due to an implementation choice we made: Do not evict a view requested for a new batch if it is already cached. While the stateful optimization accounts for such views during utility estimation, the stateless optimization does not. This creates a small gap in the estimated utilities and the actual performance speedups for tenants in the stateless optimization which is reflected in the observed fairness index.

5.4 Discussion

Comparing caching policies. Our experiments show that among all setups, FASTPF policy shows the most consistent numbers both in terms of throughput and fairness. However, in many cases, MMF is also observed to do just as well. We believe this is a second order difference that a more precise cost model and implementations of the exact algorithms (for instance, the algorithm in Appendix B for proportional fairness) will bring out. However, even given similar empirical results, PF has the advantage of the core property as a succinct and easy to explain notion of fairness and should, therefore, be used as the default policy. The following example of a four tenant setup further illustrates the difference among the policies.

We set the query generator for three tenants to a distribution g_1 over Sales datasets while the fourth tenant is made to gener-

Figure 10: Fraction of time popular views cached by algorithms

Figure 11: Fairness index as a function of number of batches

ate queries from a different distribution g_2 . Recollecting the example presented in Table 4, MMF tries to share the cache (probabilistically) equally between the two sets of tenants, producing an inefficient allocation outside of the core. We include a chart showing the duration for which the most popular views were cached by MMF, FASTPF, and OPTP in Figure 10. The top three views in each of g_1 and g_2 serve 25%, 13%, and 8% of the queries respectively. 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 MMF ends up favoring the fourth tenant. OPTP favors top views from g_1 unreasonably more thus being unfair. FASTPF provides more favorable tradeoffs here.

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

Convergence Properties. As our algorithms are randomized, it is important to study how long they take to converge to fair solutions. In our experiments, we find that the number of batches to achieve this convergence is very small, to the tune of 10-15. In Figure 11, we present results of a 8 tenant workload with 40 batches, optimized once using MMF and once using FASTPF. The fairness index was computed after every 2 batches. It can be seen that both algorithms converge to their respective optimal values in less than 15 batches. We plan to systematically study which parameters define the rate of convergence of the algorithms as a future work.

Batch Size. The analysis presented in Section 5.3.4 shows that the best performance is achieved when a query batch contains a sufficient number of queries so that a cached view can help two or more of the queries. A small profiling run could help learn the characteristics of query arrival in order to set the right batch size. Having said this, the Batching Delay could be a bigger factor in determining the batch size in practical ROBUS setups. Our evaluation so far focussed on the throughput gains over a given workload wherein queries arrived for a set time period. Caching policies were compared on how fast (and on how fairly) they processed

Figure 12: Comparing Batching Delay and Execution Time when Batch Size=10 seconds

this query workload. We consistently noticed that scheduling delays dominated batching delays (refer Figure 5) so much so that the Batching Delay was a very small factor in the E2E latency. A long running industrial cluster, however, may want to enforce certain latency sensitive SLAs [29] on the queries which would bound the Batching Delay.

We first attempt to understand the impact of Batching Delay in our current setup. We compare the Batching Delay with Execution Time on a workload processed in fixed time batches of 10 seconds in Figure 12. The Scheduling Delay is not considered here as it is not a factor ROBUS can control directly. One can notice a number of queries spending a significant proportion of time in Batching Delay. Most of these are short running queries even when disk-bound, and therefore, the Batching Delay harms their E2E latency. Going forward, we would like ROBUS platform to incorporate SLAs on individual query wait times.

Code Base. The code base of ROBUS has been open-sourced [8] and our entire experimental setup can be replicated by following a simple set of instructions provided with the code.

6. RELATED WORK

Multi-tenant database-as-a-service architectures.

Traditionally, the notion of multi-tenancy in databases deals with consolidating DBMS resources, namely, hardware, process, schema, among tenants [44, 19, 58, 30]. Commercial database engines use adaptive memory managers to split server memory dynamically among various pools including the buffer pool [70, 9]. Buffer pool is used to cache recently accessed data pages for performance gains. However, the model of multi-tenancy used here supports exclusive ownership of data by tenants and therefore, no consideration is given to a possibility of data sharing among tenants. Emerging multi-tenant big data architectures, on the other hand, allow for the entire cluster data to be shared among tenants representing different teams within an enterprise [7, 5]. This sharing of data is exploited by ROBUS in making the best use of cache.

A recent work on buffer pool management proposes a multi-tenancy aware buffer pool page replacement algorithm [57]. Techniques developed here could potentially apply to big data analytics setups, e.g., Spark’s RDD store can be treated as a pool of data partitions/blocks. ROBUS, though, manages cache at the application layer because: 1. Newer programming interfaces like Spark allow cache directives to be specified only on the objects/views accessible through programs, 2. Cache “all-or-nothing” property observed in data-parallel workloads [16] calls for decision-making on an entire view, especially for iterative machine learning workloads, and 3. While hit ratio metering required in [57] is supported in mature database-as-a-service installations, it would require big infrastructure upgrades in an emerging data analytics framework like Spark.

Multi-tenant big data analytics architectures.

Big data analytics platforms support a model of multi-tenancy where large volumes of data is accessed by analysts, data scien-

Policy	Frequency	Data abstraction	Multi-tenant	Multi-tenancy properties			
				Pareto efficient	Strategy free	Fair shares	Fair speedups
Isolated caches(LRU)	Online	Block	✗	-	-	-	-
MT-LRU [57]	Online	Block	✓	✓	does not apply	✓	✗
Shared offline [2]	Offline	File/MV/RDD	✓	✗	does not apply	✗	✗
Greedy online(LRU)	Online	File/MV/RDD	✓	✓	✗	✗	✗
Fair Ride [64]	Online	File	✓	near-optimal	✓	✓	✗
ROBUS	Online batches	File/MV/RDD	✓	✓	does not apply	near-optimal	near-optimal

Table 9: Summary of cache management policies in data analytics clusters

tists, and developers alike for functionalities ranging from periodic report generation to business intelligence [1, 10]. A critical component of such platforms is a fair scheduler or a resource allocator [4, 36, 43]. These schedulers do not differentiate the cache resource from the heap resource and, as a result, partition cache among tenants. As seen in our work, partitioned/isolated cache setups severely reduce optimization opportunities.

Most data analytics platforms support data objects to be cached either in application memory or in an external caching service (e.g. Alluxio [54] and DDM [3]). ROBUS optimizer fits naturally in both scenarios: While data sharing comes naturally with the use of external cache store, the application memory is also shared among multiple jobs submitted through a job server [7, 5].

Memory-based analytics infrastructures such as SAP Hana [33], RAMClouds [62] allow fast processing of memory-resident data. However, many commercial clusters today have considerably large amount of storage on disk than total memory (e.g., Facebook [6]). Choice of cache directives is critical for performance and fairness in such setups and ROBUS is designed to provide an automated solution to this problem.

Table 9 summarizes how ROBUS compares to existing caching policies. LRU policies typically operate at a lower granularity of data on individual nodes. Among the policies used to manage data at a higher granularity, i.e., a file, a materialized view, or an RDD, an offline policy could be used to materialize views that are expected to maximize benefits with some prior knowledge on data access [2]. This policy, however, does not provide any fairness guarantees. On the other extreme, an online policy could greedily materialize every cache directive using LRU for evictions. As we saw in Figure 4, this results in performance inefficiencies when tenants have big variability in data access. Additionally, this policy is not strategy proof as pointed out in FairRide [64] with a phenomenon of ‘free-ride’: a tenant getting *free* access to a data object brought in cache by another tenant. FairRide brings in strategy-proofness to this setup by probabilistically blocking access to cached files. However, FairRide does not give any guarantees on expected speedup to tenants, which is an important metric from fairness standpoint. Furthermore, it operates on files cached in an external store unlike ROBUS, which supports materialization of any view of data in the application memory as well. As ROBUS operates on batches of workload already submitted by tenants, the question of strategic tenants does not arise. Finally, ROBUS supports multiple algorithms to optimize a workload batch which allow for different trade-offs in the performance and the fairness.

Physical design tuning and Multi-query optimization.

Classical view materialization algorithms in databases [41, 74, 15, 39, 56] 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 [24, 51, 32, 53], on the other hand, adapt to changes in query workload by modifying 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 through the pluggability provided in the framework (Section 2).

Batched optimization of queries was proposed in [68] and is used in many work sharing approaches [75, 14, 60, 73]. ROBUS employs batched query optimization likewise, but crucially also ensures that each tenant gets their fair share of utility from the cache.

Fairness theory.

The proportional fairness algorithm is widely studied in Economics [59, 45, 25] as well as in scheduling theory [36, 63, 50, 49, 42, 69, 17]. In the context of resource partitioning problems (or exchange economies) [20, 31], it is well-known that a convex program, called the Eisenberg-Gale convex program [45] 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 for which solutions implement proportional fairness in a randomized sense.

In scheduling theory, the focus is on analyzing delay properties [50, 49, 42] assuming jobs have durations. Our focus is instead on utility maximization, which has also been considered in the context of wireless scheduling in [69, 17]. 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 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.

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., cached data can be accessed by all tenants. In this paper, we presented ROBUS, a cache management platform for achieving a fair allocation of cache 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 using industry-standard workload show that our algorithms are capable of providing desirable performance-fairness trade-offs.

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] Apache Hadoop, <http://hadoop.apache.org/>.
- [2] Building the Periscope Data Cache with Amazon Redshift. <https://goo.gl/c68EzG>.
- [3] Discardable distributed memory: Supporting memory storage in HDFS, <http://hortonworks.com/blog/ddm/>.
- [4] Hadoop fair scheduler, http://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html.
- [5] Livy job server. <https://github.com/cloudera/livy>.
- [6] Presto: Interacting with petabytes of data at Facebook. <http://on.fb.me/LnFVAM>.
- [7] REST job server for Apache Spark. <https://github.com/spark-jobserver/spark-jobserver>.
- [8] ROBUS Cache Planner, <https://github.com/dukedbgrou/ROBUS>.
- [9] The self managing database: Automatic SGA memory management. Oracle white paper, November 2003. <http://goo.gl/7in5Vb>.
- [10] Spark: Lightning fast cluster computing, <https://spark.apache.org/>.
- [11] TPC-DS benchmark, <http://www.tpc.org/tpcds/>.
- [12] TPC-H benchmark, <http://www.tpc.org/tpch/>.
- [13] A. Abdulkadiroglu and T. Sonmez. Random serial dictatorship and the core from random endowments in house allocation problems. *Econometrica*, 66(3):689–701, 1998.
- [14] P. Agrawal, D. Kifer, and C. Olston. Scheduling shared scans of large data files. *Proc. VLDB Endow.*, 1(1):958–969, Aug. 2008.
- [15] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. VLDB '00, pages 496–505, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.
- [16] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated memory caching for parallel jobs. NSDI'12, pages 20–20, Berkeley, CA, USA, 2012. USENIX Association.
- [17] M. Andrews. Instability of the proportional fair scheduling algorithm for hdr. *Wireless Communications, IEEE Transactions on*, 3(5):1422–1426, 2004.
- [18] S. Arora, E. Hazan, and S. Kale. The multiplicative weights update method: A meta algorithm and applications. *Theory of Computing*, 8:121–164, 2005.
- [19] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-tenant databases for software as a service: Schema-mapping techniques. SIGMOD '08, pages 1195–1206, New York, NY, USA, 2008. ACM.
- [20] R. J. Aumann. Markets with a continuum of traders. *Econometrica: Journal of the Econometric Society*, pages 39–50, 1964.
- [21] M. Berkelaar, K. Eikland, and P. Notebaert. *lpsolve : Open source (Mixed-Integer) Linear Programming system*.
- [22] A. Bhalgat, 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.
- [23] A. Bogomolnaia and H. Moulin. A new solution to the random assignment problem. *Journal of Economic Theory*, 100(2):295–328, 2001.
- [24] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *ICDE*, 2007.
- [25] E. Budish. The combinatorial assignment problem: Approximate competitive equilibrium from equal incomes. *Journal of Political Economy*, 119(6):1061–1103, 2011.
- [26] 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.
- [27] Y. Cai, C. Daskalakis, and S. M. Weinberg. Optimal multi-dimensional mechanism design: Reducing revenue to welfare maximization. In *FOCS*, 2012.
- [28] 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.
- [29] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based scheduling: If you're late don't blame us! In *Proceedings of the ACM Symposium on Cloud Computing, SOCC '14*, pages 2:1–2:14, New York, NY, USA, 2014. ACM.
- [30] C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD '11*, pages 313–324, New York, NY, USA, 2011. ACM.
- [31] G. Debreu and H. Scarf. A limit theorem on the core of an economy. *International Economic Review*, 4(3):pp. 235–246, 1963.
- [32] I. Elghandour and A. Aboulmaga. Restore: Reusing results of mapreduce jobs. *Proc. VLDB Endow.*, 5(6):586–597, Feb. 2012.
- [33] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: Data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, Jan. 2012.
- [34] 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.
- [35] Y. Freund and R. E. Schapire. Adaptive game playing using multiplicative weights. *Games and Economic Behavior*, 29(1–2):79 – 103, 1999.
- [36] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. NSDI'11, pages 323–336, Berkeley, CA, USA, 2011. USENIX Association.
- [37] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *EuroSys*, pages 365–378. ACM, 2013.
- [38] D. Gillies. *Some Theorems on n-Person Games*. PhD thesis, Princeton University, 1953.
- [39] J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: A practical, scalable solution. SIGMOD '01, pages 331–342, New York, NY, USA, 2001. ACM.
- [40] J. Gray, K. Baclawski, P. Sundaresan, and S. Englert. Quickly generating billion-record synthetic databases. Association for Computing Machinery, Inc., January 1994.
- [41] 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.
- [42] S. Im, J. Kulkarni, and K. Munagala. Competitive algorithms from competitive equilibria: Non-clairvoyant scheduling under polyhedral constraints. STOC '14, pages 313–322, 2014.

- [43] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. *SOSP '09*, pages 261–276, New York, NY, USA, 2009. ACM.
- [44] D. Jacobs, S. Aulbach, and T. U. München. Ruminations on multi-tenant databases. In *BTW Proceedings, volume 103 of LNI*, pages 514–521. GI, 2007.
- [45] 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.
- [46] 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.
- [47] M. Kaneko. The ratio equilibrium and a voting game in a public goods economy. *Journal of Economic Theory*, 16(2):123–136, 1977.
- [48] I. Kash, A. D. Procaccia, and N. Shah. No agent left behind: Dynamic fair division of multiple resources. *AAMAS '13*, pages 351–358, Richland, SC, 2013. International Foundation for Autonomous Agents and Multiagent Systems.
- [49] 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.
- [50] 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.
- [51] Y. Kotidis and N. Roussopoulos. DynaMat: A dynamic view management system for data warehouses. *SIGMOD '99*, pages 371–382, New York, NY, USA, 1999. ACM.
- [52] 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.
- [53] 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. *SIGMOD '14*, pages 1591–1602, New York, NY, USA, 2014. ACM.
- [54] 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.
- [55] A. Mas-Colell and J. Silvestre. Cost share equilibria: A lindahl approach. *Journal of Economic Theory*, 47(2):239–256, 1989.
- [56] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. *SIGMOD '01*, pages 307–318, New York, NY, USA, 2001. ACM.
- [57] V. Narasayya, I. Menache, M. Singh, F. Li, M. Syamala, and S. Chaudhuri. Sharing buffer pool memory in multi-tenant relational database-as-a-service. *Proc. VLDB Endow.*, 8(7):726–737, Feb. 2015.
- [58] V. R. Narasayya, S. Das, M. Syamala, B. Chandramouli, and S. Chaudhuri. SQLVM: performance isolation in multi-tenant relational database-as-a-service. In *CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, 2013.
- [59] J. Nash. The bargaining problem. *Econometrica*, 18(2):155–162, 1950.
- [60] 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.
- [61] E. J. O’Neil, P. E. O’Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 297–306, New York, NY, USA, 1993. ACM.
- [62] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, Jan. 2010.
- [63] D. C. Parkes, A. D. Procaccia, and N. Shah. Beyond dominant resource fairness: Extensions, limitations, and indivisibilities. *EC '12*, pages 808–825, New York, NY, USA, 2012. ACM.
- [64] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica. FairRide: Near-optimal, fair cache sharing. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 393–406, Santa Clara, CA, Mar. 2016. USENIX Association.
- [65] 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.
- [66] 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.
- [67] H. E. Scarf. The core of an n person game. *Econometrica*, 35(1):pp. 50–69, 1967.
- [68] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, Mar. 1988.
- [69] A. L. Stolyar. On the asymptotic optimality of the gradient scheduling algorithm for multiuser throughput allocation. *Oper. Res.*, 53(1):12–25, 2005.
- [70] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra. Adaptive self-tuning memory in DB2. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 1081–1092. VLDB Endowment, 2006.
- [71] H. Varian. Two problems in the theory of fairness. *Journal of Public Economics*, 5(3-4):249–260, 1976.
- [72] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O’Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.
- [73] X. Wang, C. Olston, A. D. Sarma, and R. Burns. CoScan: Cooperative scan sharing in the cloud. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 11:1–11:12, New York, NY, USA, 2011. ACM.
- [74] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. *VLDB '97*, pages 136–145, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.
- [75] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: Dynamic bandwidth sharing in a DBMS. *VLDB '07*, pages 723–734. VLDB Endowment, 2007.

APPENDIX

A. MULTIPLICATIVE WEIGHT METHOD

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 [18] 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 2 AHK Algorithm

```

1: Let  $K \leftarrow \frac{4\rho^2 \log r}{\delta^2}$ ;  $y_1 = \bar{1}$ 
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:      $M_{it} = a_{it} x_t - b_i$  ▷ 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:   ▷ Multiplicatively update  $y$ .
12:   end for
13:   Normalize  $y_{t+1}$  so that  $\|y_{t+1}\|_1 = 1$ .
14: end for
15: Return  $x = \frac{1}{K} \sum_{t=1}^K x_t$ .
```

This procedure has the following guarantee [18]:

THEOREM 5. *If $\text{LP}(A, b, P)$ is feasible, the AHK procedure never declares infeasibility, and the final x satisfies:*

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

B. PROPORTIONAL FAIRNESS

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 [27, 22]. 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 Appendix A, 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 an 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 $\text{PFFEAS}(Q)$ is infeasible, or we find $(\mathbf{x}, \gamma)$ such that for all i , we have:

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

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

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

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

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

Thus, we arrive at the result of theorem 3.

C. APPROXIMATE COST MODEL

In practice, cost models are difficult to develop and may not perfectly capture the true utilities of caching a view. We briefly consider the case of an approximate cost model and argue proportional fairness still provides some guarantees in this instance.

DEFINITION 7. A utility estimation model is δ -approximate if for every tenant i and configuration S , the utility $\hat{U}_i(S)$ estimated by the model is within a $(1 + \delta)$ multiplicative factor of the true utility $U_i(S)$, i.e., $\frac{1}{1+\delta} U_i(S) \leq \hat{U}_i(S) \leq (1 + \delta) U_i(S)$.

Note that since the utility for tenant i of a configuration S is defined as the sum of utilities from i 's individual queries in S , if the utility estimation model is a multiplicative approximation on queries, this implies that it is a multiplicative approximation on configurations. Furthermore, the δ -approximation on configurations guarantees δ -approximation on the expected utility of randomized allocations, following directly from the linearity of expectation. Our primary guarantee of the proportional fairness algorithm is that it will result in a core allocation; we therefore argue

that running proportional fairness with an approximate utility estimation model will result in an approximate core allocation.

DEFINITION 8. An allocation $\mathbf{x}$ is said to lie in the (randomized) α -approximate 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 $U_i(\mathbf{y}) \geq \alpha U_i(\mathbf{x}), \forall i \in T$ and $U_j(\mathbf{y}) > \alpha U_j(\mathbf{x})$ for at least one $j \in T$.

LEMMA 5. Allocations which are proportionally fair with respect to a δ -approximate utility estimation model lie in the (randomized) $(1 + \delta)^2$ -approximate core with respect to the true utilities.

PROOF. The lemma follows directly from observing the guarantee of a core allocation on the estimated utilities $\hat{U}_i(\mathbf{x})$ and applying the bound given by definition of the δ -approximate utility estimation model on both sides of the inequality, noting that it applies to randomized allocations by virtue of the linearity of expectation. $\square$

D. MAX-MIN FAIRNESS

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 $\delta = \epsilon/N$. We can now compute K from the AHK procedure, so that $K = \frac{4N^2 \log N}{\epsilon^2}$ in order to approximate λ^* to a factor of $(1 - \epsilon)$. The procedure is described in Algorithm 3.

Algorithm 3 Approximation Algorithm for SIMPLEMMF

- 1: Let ϵ denote a small constant < 1 .
  - 2: $T \leftarrow \frac{4N^2 \log N}{\epsilon^2}$
  - 3: $\mathbf{w}_1 \leftarrow \frac{1}{N}$ $\triangleright$ Initial weights
  - 4: $\mathbf{x} \leftarrow \mathbf{0}$ $\triangleright$ 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: $w_{i(k+1)} \leftarrow w_{ik} \exp(-\epsilon \frac{U_i(S)}{U_i})$
  - 8:   Normalize $\mathbf{w}_{k+1}$ so that $\|\mathbf{w}_{k+1}\|_1 = 1$.
  - 9: $x_S \leftarrow x_S + \frac{1}{T}$ $\triangleright$ 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 $O\left(\frac{4N^4 \log N}{\epsilon^2}\right)$ invocations of WELFARE .