JAWS: Job-Aware Workload Scheduling for the Exploration of Turbulence Simulations

Xiaodan Wang*, Eric Perlman*, Randal Burns*¶, Tanu Malik†,
Tamas Budavári‡¶, Charles Meneveau§¶ and Alexander Szalay‡¶
*Dept. of Computer Science, Johns Hopkins University, {xwang,eric,randal}@cs.jhu.edu
†Cyber Center, Purdue University, tmalik@purdue.edu
‡Dept. of Physics and Astronomy, Johns Hopkins University, {budavari,szalay}@jhu.edu
§Dept. of Mechanical Engineering, Johns Hopkins University, meneveau@jhu.edu
¶Institute for Data Intensive Engineering and Science, Johns Hopkins University

***Abstract*—We present JAWS, a job-aware, data-driven batch scheduler that improves query throughput for data-intensive scientific database clusters. As datasets reach petabyte-scale, workloads that scan through vast amounts of data to extract features are gaining importance in the sciences. However, acute performance bottlenecks result when multiple queries execute simultaneously and compete for I/O resources. Our solution, JAWS, divides queries into I/O-friendly sub-queries for scheduling. It then identifies overlapping data requirements within the workload and executes sub-queries in batches to maximize data sharing and reduce redundant I/O. JAWS extends our previous work [1] by supporting workflows in which queries exhibit data dependencies, exploiting workload knowledge to coordinate caching decisions, and combating starvation through adaptive and incremental trade-offs between query throughput and response time. Instrumenting JAWS in the Turbulence Database Cluster [2] yields nearly three-fold improvement in query throughput when contention in the workload is high.**

I. Introduction

Improved physical instruments, data pipelines, and large-scale simulations in the Sciences have led to an exponential growth of data. Gray and Szalay termed this the *data avalanche* [3]. For instance, direct numerical simulation of isotropic turbulence over a large eddy turnover time-scale (a two-second interval) yields 27 terabytes of data [2]. Similarly, the Panoramic Survey Telescope and Rapid Response System (Pan-STARRS) in Astronomy produce tens of terabytes daily [4]. As such, scientific repositories in various disciplines are turning to clusters of databases to manage petabytes of data. Database clusters achieve a high degree of parallelism and aggregate throughput by partitioning data, either spatially or temporally, across multiple nodes. These include the Turbulence cluster [2] for multi-scale simulations, the Graywulf cluster [4] for the Pan-STARRS Astronomy survey [5], and the Beowulf cluster [6] for geospatial databases.

Within scientific database clusters, a new class of data-intensive scientific workloads has emerged that correlate, mine, and extract features from vast amounts of data. These queries are long running (lasting hours or even days) and strain I/O resources by scanning large portions of the data. Throughput degrades precipitously when multiple queries execute concurrently. Furthermore, multiple queries may belong to the workflow sequence of a larger experiment, which complicates scheduling. Our goal is to alleviate performance bottlenecks from simultaneously executing workloads and enable scientists to explore data at a larger scale.

Data-driven, batch processing is motivated by the Turbulence Database [2], which typifies a data-intensive cluster with 27 terabytes distributed across multiple nodes. Individual experiments can last for several days by tracking, for example, the position or velocity of hundreds of thousands of particles distributed over the entire span of space and time. With over eight million queries accessing thirty-five billion points [7] to date, the Turbulence cluster must serve multiple data-intensive queries simultaneously. Because many queries access data from the same time spans or spatial regions, Turbulence workloads can benefit from data sharing.

Query scheduling needs to be re-examined to ensure high performance in data-intensive scientific clusters. Specifically, the scheduler must ensure high query throughput in the presence of concurrent accesses to peta-scale scientific datasets. As a first step, we implemented the LifeRaft [1] batch scheduler in Astronomy for the Sloan Digital Sky Survey (SDSS) [8] with promising results. However, LifeRaft left several problems open, which limits its applicability to other scientific databases. These include failure to account for constraints on the workflow execution order of queries and lack of coordinated caching decisions that exploit scheduling knowledge. We put forth a Job-Aware Workload Scheduler (JAWS) that inherits the benefits of LifeRaft and addresses its limitations.

Our goal is to provide a *data-driven, batch scheduler* in which scientists submit batch workloads that include long running, data-intensive queries. LifeRaft exploits opportunities for data sharing by reordering queries to (1) eliminate redundant accesses to the disk and (2) amortize the cost of data access over multiple queries. Scheduling is data-driven because execution order is based on the amount of contention for access to data regions rather than the arrival order of queries. LifeRaft achieves I/O amortization by co-scheduling queries that access the same data.

Contributions: JAWS provides *job-aware scheduling* that extends the data sharing benefits of batch scheduling to workflows which execute a large number of queries in an ordered sequence. Scientific workloads are grouped into jobs, which

SC10 November 2010, New Orleans, Louisiana, USA 978-1-4244-7558-2/10/$26.00

consist of a sequence of queries relating to the same experiment. Queries within a job exhibit data dependencies and must be executed in order, one after the other. LifeRaft does not capture ordering constraints, which exist for the Turbulence cluster and other scientific workflows. For example, when tracking the movement of particles over time in Turbulence, the position of particles at the next time step depends on the result of the query at the previous time step. JAWS provides a greedy algorithm that identifies data sharing between jobs and synchronizes the execution of jobs to realize data sharing.

We explore *cache replacement algorithms* that exploit knowledge from the scheduler to further improve performance. JAWS performance depends crucially on caching in which up to 54% of requests in Turbulence workloads are serviced from the cache. Furthermore, JAWS employs a two level framework that co-schedules access for groups of related data regions in order to exploit locality of reference in the computation. In turn, we coordinate caching decisions with scheduling to ensure that groups of data regions that are used together are cached together [9]. We present two cache replacement algorithms: (1) Utility Ranked Caching (URC), which incorporates full knowledge of pending requests in the workload, and (2) Segmented Least Recently Used (SLRU), which approximates URC based on little workload knowledge and minimal overhead. We instrument both algorithms and evaluate their benefit over SQL Server's page replacement algorithm, which is a variant of LRU-K [10].

We also describe an automated method for combating starvation that makes adaptive and incremental trade-offs between query throughput and response time performance based on workload saturation. Many queries in the Turbulence workload are short-lived (minutes or seconds), focus on a small spatial region, and are highly selective. A scheduler that maximizes query throughput will inevitably starve short-lived queries awaiting the completion of long running queries. Moreover, since queries may belong to a large job, delaying individual queries indefinitely negatively impacts the entire job. To realize starvation resistance, JAWS makes dynamic and incremental trade-offs between maximizing query throughput and lowering response time as workload saturation changes.

Instrumenting JAWS's data-driven batch scheduler for the Turbulence database cluster improves query throughput performance by nearly three-fold. Owing to higher throughput, data-driven scheduling also reduces response times. Our treatment includes a study of response time versus throughput trade-offs that demonstrates the system's ability to adapt automatically to changing workload saturation. In addition, we found that incorporating workload knowledge in cache replacement decisions further improves throughput by 16%.

II. Related Work

The query batching paradigm was studied for workloads against large datasets on tertiary storage in order to minimize I/O cost [11–13]. Yu and Dewitt [13] explored this in the Paradise system by reordering queries over data stored on magnetic tape. The reordering achieves sequential I/O by collecting data requirements during a pre-execution phase (without physically performing the I/O), reordering tape requests, and finally executing queries concurrently in one batch. Sarawagi et. al. [12] provide non-sequential processing by partitioning the data into fragments that are physically contiguous on the tertiary device and scheduling concurrent queries on a per fragment basis. Andrade et. al. [14] describe a general framework for caching and reusing intermediate results among analysis queries to reduce I/O. Although JAWS leverages some of these ideas, it also explores workflows in which queries exhibit data dependencies.

Google's Map-Reduce [15] is an attractive paradigm for parallel computation and scan-intensive queries. Yang et al. [16] extend the Map-Reduce paradigm to more efficiently support relational joins by adding a merge phase that processes heterogeneous datasets simultaneously. Moreover, Olston et al. [17] combined the procedural style of Map-Reduce with declarative SQL constructs.

Agrawal et al. [18] incorporated batch processing into Map-Reduce by identifying shared files among map tasks. Jobs are grouped into batches so that sequential scans of large files are shared among as many simultaneous jobs as possible. They also include an aging policy to avoid job starvation. Our work complements their results. However, a direct application to query scheduling is difficult for several reasons. Their model is sensitive to job arrival rates, which corresponds to a stationary process. As such, it is not suited for bursty workloads with no steady states. This is problematic for online systems that serve a continuous stream of queries from dozens of users such that a representative workload is not available. Shared files are also too large to fit in memory in the Map-Reduce environment: only a single file is scanned at a time. In contrast, cache replacement is crucial to the performance of JAWS. JAWS also realizes data sharing for queries that exhibit data dependencies within a workflow.

For Astronomy workloads, The CasJobs [19] system for the Sloan Digital Sky Survey [20] avoids the starvation of short queries from data-intensive scan queries by using a multi-queue job submission system in which queries from each class are assigned to different servers. The throughput of long running queries is further improved by partitioning the data and evaluating the queries in parallel across multiple servers. However, the distinction between long and short queries is arbitrary so that the longest short queries interfere with the short queue and the shortest long queries experience starvation. JAWS does not rely on ad hoc mechanisms to distinguish long and short running queries nor does it rely on multiple servers processing queries for the same data; queries of all sizes are supported in a single system. Also, JAWS automatically adapts between maximizing throughput and reducing response time for short queries as workload saturation changes.

III. LifeRaft Scheduling in Turbulence

In this section, we adapt our prior data-driven batch processing framework, LifeRaft, to query scheduling in the Turbulence database. LifeRaft evaluates queries based on the amount

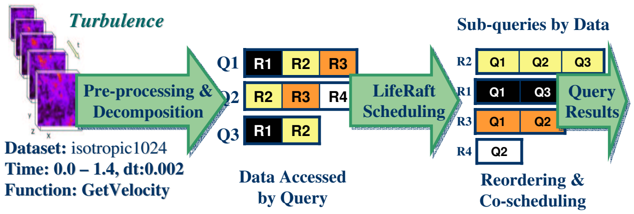


Fig. 1. Data-driven batching processing in LifeRaft.

of contention for data and provides system tunable techniques for starvation resistance.

A. Turbulence Database Cluster

The Turbulence Database Cluster [2] stores the complete space-time histories of Direct Numerical Simulation (DNS) on which scientists perform spatial and temporal exploration of turbulent flows using public Web Services interfaces. The cluster stores the result as a time series of three-dimensional structured grid data. The Turbulence workload consists of queries that perform computations in 4-D space/time over large amounts of data. These include (1) evaluating statistical arrays of turbulence quantities over the entire or parts of the volume, (2) tracking particles forward and backwards through time, and (3) identifying turbulent structures and tracking their formation and evolution.

The current version of the database consists of 27TB of turbulence simulation history: 1024 time steps over two seconds of simulation time. Each time step consists of the velocity vectors and pressure fields on a 1024^3 grid DNS of stationary hydrodynamic turbulent flow. The data are partitioned into fixed sized storage blocks or atoms of 64^3 voxels of roughly 8MB in size. (In practice, each atom is 72^3 in length with four units of replication on each side for performance reasons). In total, there are 4096 8MB atoms per time step, which serve as the fundamental unit of I/O in the database.

A hierarchical spatial index based on the Morton order is used to partition and index the space. In particular, it logically partitions the space into cubes of side 2^k for $k = 0, ..., log(n)$ for n atoms. The Morton index also acts as a space filling curve to provide a linear ordering of the atoms on disk while preserving spatial locality. That is, atoms which are close together in Morton order are also near each other in voxel space. (A clustered B+ tree access path, which is keyed on a combination of the Morton index and the time step, is used to retrieve each atom). Hence, both range and containment queries are efficient with respect to I/O. To improve cache reuse, points from each query are sorted and evaluated in Morton order so that each atom is read only once.

B. Data-Driven Batch Processing

We describe LifeRaft: a data-driven query scheduler that focuses on the data requirements of each query, instead of the arrival-order, to coordinate access to secondary storage and achieve large improvements in query throughput. LifeRaft is suited to scientific datasets that are indexed and partitioned spatially or temporally so that the data requirements of queries are known prior to execution and queries can be sub-divided. This framework, which we adapt for the Turbulence database, proved effective for Astronomy workloads [1].

LifeRaft evaluates queries in two stages (Figure 1). First, each query is pre-processed to identify its data access requirements. Turbulence queries provide a list of positions on which to perform computation. The pre-processor identifies the data atom that corresponds to each position, and hence the list of atoms accessed by the query. Pre-processing also outputs a list of sub-queries for each query that satisfies the following properties: each sub-query is a set of positions that fall within the same atom, the sub-queries can be executed in any order, and the result of the original query is obtained by combining the sub-query results.

The second step is to group sub-queries that access the same atom and co-schedule them together in a single pass over the data. Thus, to evaluate sub-queries whose positions fall within the same atom, LifeRaft reads the corresponding atom (a 64^3 region that forms the fundamental unit of I/O) and potentially nearby atoms for spatial interpolation queries. Moreover, the positions are sorted in Morton order prior to execution so that locations close in space are executed in close succession. This improves cache reuse because locations that reference the same set of atoms are evaluated together. Sorting the positions also amortizes disk seek times for dense queries in which multiple positions reside within the same atom. Li et al. [2] discuss query evaluation in the Turbulence database extensively.

C. Scheduling by Workload Throughput

LifeRaft favors atoms with more contention in terms of pending workload requests (*e.g.* number of queried positions) in order to maximize query throughput. A longer workload queue means that the cost of reading an atom from disk can be amortized over more queries. Thus, the scheduler evaluates data atoms in contention order. Let A_1, ..., A_n denote the list of atoms and Q_1, ..., Q_m denote the list of queries. A workload W_j^i represents the set of positions from Q_i that are contained within A_j and the workload queue for an atom A_j consists of the union of W_j^1, W_j^2, ..., and W_j^m. The amount of contention for an atom A_i is defined using the *workload throughput* metric as:

$$U_t(i) = \frac{\sum_{j=1}^{m} W_i^j}{T_b * \phi(i) + T_m * \sum_{j=1}^{m} W_i^j} \qquad (1)$$

$\sum_{j=1}^{m} W_i^j$ denotes the size of A_i's workload queue. T_b and T_m are constants which estimate the time cost of reading an atom from disk and the computation cost for a single position respectively. These costs can be derived empirically, and we assume uniform I/O cost for accessing each atom since they are equal-sized. Finally, ϕ is a function $[1, n] \rightarrow [0, 1]$ in which $\phi(i)$ is 0 if A_i is in memory and 1 otherwise. The numerator denotes the total size of all workload queues pending against atom A_i while the denominator sums both the I/O cost, which is amortized across multiple queries, and the computation cost. LifeRaft evaluates atoms greedily in order of decreasing workload throughput, which captures the rate in which the workload queue is consumed.

LifeRaft also provides a system tunable technique for combating starvation. While the greedy policy described above leads to high query throughput, it may starve individual queries. Starvation occurs for atoms with small workload queues that are accessed infrequently. Query response time increases due to the last mile bottleneck—a query cannot finish until every atom it accesses is processed. LifeRaft employs an *aged workload throughput* metric to balance the need to process queries in arrival order, which resists starvation, with maximizing throughput. The metric accounts for the age (queuing time) $E(i)$, in milliseconds, of the oldest sub-query in each atom A_i. The *aged workload throughput* metric for A_i is defined as:

$$U_e(i) = U_t(i) * (1 - \alpha) + E(i) * \alpha \tag{2}$$

$U_t(i)$ denotes the *workload throughput* of atom A_i and α is a real value between zero and one, which specifies how strongly biased the scheduler is towards processing queries in arrival order. The α parameter influences the completion order of concurrent queries. Biasing α to 0 selects the most contentious atoms first, whereas 1 processes queries in arrival order. Even when setting $\alpha = 1$, data sharing occurs among queries.

IV. Job-Aware Scheduling

Scientists often conduct an experiment through a sequence of related queries. We define a *job* as a collection of queries that belong to the same experiment, which may involve thousands of queries executing over several days. Prior knowledge of the access pattern of long running jobs is immensely valuable for scheduling. We categorize jobs into two types: batched and ordered. In batched jobs, each query can be executed independently and in any order (*i.e.* gathering aggregate statistics over the data). In a typical batched job, the number of queried positions remains constant or queries may look for changes in the same space over time with little movement. JAWS treats these jobs in a similar manner as other one-off queries. However, our focus is on optimizing for ordered jobs.

Ordered jobs require that each query execute one after the other in sequence because queries exhibit data dependencies and reuse results from their predecessor. Particle tracking experiments is one such example. To track the movement of particles over time, the positions of particles at the next time step depend on the state of the particles computed from the previous time step. In this section, we address the co-scheduling of queries from ordered jobs to realize data sharing.

A. Job Identification

In Turbulence jobs, scientists transform inputs to new queries based on the results of previous queries, making it difficult to predict data access pattern. A sample job, for instance, may scatter tens of thousands of points randomly in space and then track the diffusion of these points over time. This may run for several days in which the user collects results from a time step, calculates new positions outside the database, and then submits a new query for the next time step with the new positions. Moreover, multiple such long running jobs may execute simultaneously. We identify a sequence of queries as belonging to the same job using a combination of user IDs, spatial or temporal operation performed, time steps queried, and wall-clock time between consecutive queries. The techniques are heuristic, but highly accurate in practice (Section VI).

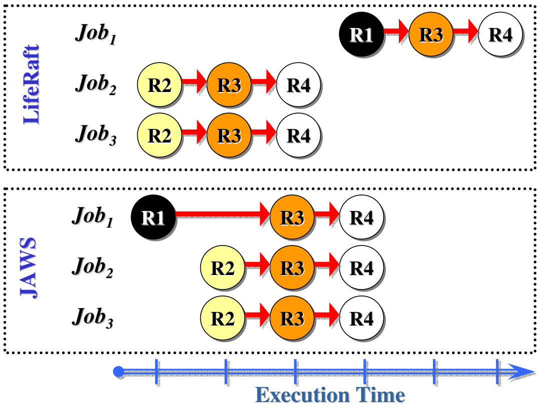


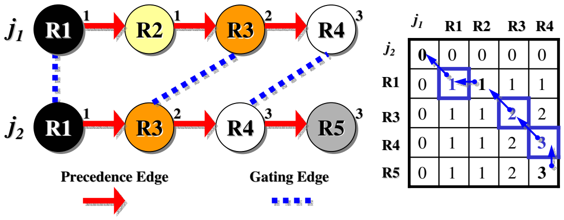
Fig. 2. Comparing Job Execution in JAWS with LifeRaft.

B. Gated Execution

JAWS employs a polynomial-time algorithm for the scheduling of ordered jobs in which queries exhibit data dependencies. The algorithm first identifies data sharing opportunities between pairs of jobs using a dynamic program. These data sharing positions are marked so that JAWS can synchronize the execution of different jobs to realize data sharing. Finally, the pairwise dynamic programming solutions are merged greedily to maximize data sharing over all jobs.

Being job-aware means that JAWS can identify data sharing opportunities that are missed by LifeRaft. Figure 2 compares JAWS with LifeRaft for the scheduling of three jobs. Each node denotes an individual query and the arrows indicate its execution order within the job. The value inside each node denotes the data region accessed. Thus, queries sharing the same value access the same set of atoms from the same time step. (For ease of illustration, we do not distinguish queries that only overlap partially in the data accessed). In Figure 2, JAWS completes 33% faster by only accessing data regions $R3$ and $R4$ once. Namely, it delays execution of Job_2 and Job_3 in order to align the execution of all jobs such that queries which access $R3$ and $R4$ are co-scheduled at the same time.

We first introduce some terminology before describing the job-aware framework. Let $\mathtt{J}$ be a list of ordered jobs j_1, ..., j_n. Each job j_i consists of a sequence of queries $q_{i,1}$, ..., $q_{i,m}$ in which a query $q_{i,k}$ can be evaluated only after queries $q_{i,1}$, ..., $q_{i,k-1}$ have completed execution. Define $A(q_{i,j})$ as the set of atoms accessed by query $q_{i,j}$ in which two queries $q_{i,j}$ and $q_{k,l}$ exhibit data sharing if $A(q_{i,j}) \cap A(q_{k,l}) \neq \emptyset$. Also, define $S(q_{i,j}) \rightarrow [\mathtt{WAIT}, \mathtt{READY}, \mathtt{QUEUE}, \mathtt{DONE}]$ as the state



j_2 \ j_1		R1	R2	R3	R4
	0	0	0	0	0
R1	0	1	1	1	1
R3	0	1	1	2	2
R4	0	1	1	2	3
R5	0	1	1	2	3

Fig. 3. Identifying data sharing through dynamic program.

```
AdmitGatingEdge($q_{l,n}$, $q_{k,m}$)
// new job $j_l$ ($q_{l,n}$), merged job $j_k$ ($q_{k,m}$)
  if $q_{k,m} \notin$GatingEdge($q_{l,n}$)
    admit←$q_{k,m} \cup$ GatingEdge($q_{k,m}$)
    MaxGatNum = $0$,
    for $1 \leq i < n$
      if |GatingEdge($q_{l,i}$)|$> 0$, MaxGatNum++
      for each $q_{x,y} \in$GatingEdge($q_{l,i}$)
        MaxGatNum = $max\{$MaxGatNum,G($q_{x,y}$)+1$\}$
    for each $q_{x,y} \in$admit
      if G($q_{x,y}$)>MaxGatNum, admit← $\emptyset$
      for each ($q_{l,a}$,$q_{x,b}$)$\in$DynamicProg($j_l$,$j_x$)
        if($a < n$ and $b > y$)or($a > n$ and $b < y$)
          or($l = a$ and $y \neq b$)or($l \neq a$ and $y = b$)
          admit← $\emptyset$
    GatingEdge($q_{l,n}$)=GatingEdge($q_{l,n}$)$\cup$ admit
```

Fig. 4. Pseudo-code for the admission of a new gating edge.

of query $q_{i,j}$. Namely, $q_{i,j}$ is in the WAIT state if it cannot be scheduled due to precedence constraints, the READY state if only gating constraints (describe below) are unsatisfied, the QUEUE state if all constraints are satisfied, and the DONE state if it has completed execution.

Furthermore, we maintain a precedence graph of all jobs in which each vertex is an individual query and each edge denotes a precedence constraint. Initially for a job j_i, there is a precedence edge (directed) from every query $q_{i,j}$ to $q_{i,j+1}$ for $1 \leq j < m$. JAWS can schedule immediately any query $q_{i,j+1}$ in the QUEUE state for which $S(q_{i,j}) =$ DONE. (The first query of each job is initialized in the QUEUE state). We introduce additional *gating* edges (undirected) between queries to identify data sharing opportunities across jobs. (A gating edge connects two vertices if the corresponding queries access the same data). When the scheduler encounters a query with gating edges in the precedence graph, it ensures that adjacent queries are co-scheduled at the same time. Formally, a gating edge can exist between queries $q_{i,j}$ to $q_{k,l}$ if $i \neq k$ and $A(q_{i,j}) \cap A(q_{k,l}) \neq \emptyset$. This gating edge is feasible (*i.e.* does not conflict with precedence constraints) if there does not exist another gating edge between a query $q_{i,x}$ and $q_{k,y}$ in which $x < j$ and $y > l$ or $x > j$ and $y < l$. Moreover, for any pair of jobs, each query from one job can have at most one gating edge to the other job. Thus, JAWS can schedule a query $q_{i,j+1}$ only if $S(q_{i,j}) =$ DONE and every adjacent (via a gating edge) query $q_{k,l}$ is in the READY state.

The first phase of JAWS takes as input the precedence relation for each job and identifies (through gating edges) the maximal possible data sharing between every pair of jobs. To accomplish this, we use a dynamic program based on the Needleman-Wunsch algorithm [21], which finds the best global alignment between two sequences. Let jobs j_i and j_k consist of n and m queries respectively. The algorithm aligns queries that exhibit data sharing between the two jobs using the following scoring system: for queries $q_{i,j}$ and $q_{k,l}$, let $s_{j,l}$ be 1 if they exhibit data sharing and 0 otherwise, while the penalty for skipping a query from either job is 0. The goal is to find an alignment between queries that maximizes this score. Each alignment translates into a gating edge, which indicates that JAWS should co-schedule the pair of queries to realize data sharing. Figure 3 illustrates the dynamic program solution for two jobs and the corresponding gating edges that are identified. (Note that the dynamic program is computed bottom up in which each entry $m_{i,k}$ in the matrix is computed from the maximum of $\{m_{i-1,k-1}+s_{i,k}, m_{i,j-1}, m_{i-1,k}\}$). Given n jobs with an average of m queries per job, the dynamic program phase incurs time complexity $\binom{n}{2}m^2$ or $O(n^2m^2)$.

Before describing the final merge phase, we introduce the concept of a gating number. Intuitively, the gating number $G(q_{i,j})$ is the minimum number of gating edges that the scheduler must evaluate before $q_{i,j}$ can be scheduled. We compute the gating number for every vertex in the precedence graph by performing a single, sequential pass over all jobs in execution order. This number is used to accept or deny the admission of new gating edges in the merge phase. Figure 3 notes the gating number of each query in the upper right hand corner of the corresponding vertex. For example, the gating number of the last query in j_1, which accesses data region $R4$, is three. This is due to the existence of two prior gating edges that ensure queries accessing $R1$ and $R3$ are co-scheduled.

The final phase greedily merges the gating edges between each pair of jobs from the dynamic program phase. This is accomplished by first sorting job pairs based on the number of gating edges found in the dynamic program solution. The merge begins by first picking the job pair j_i and j_k with the largest number of gating edges. Next, JAWS picks the job pair from the dynamic program phase with the most gating edges that involve a previously merged job (*i.e.* j_i or j_k) and a new job. Let this job pair be j_i and j_l. We merge j_l with the existing precedence graph by only admitting gating edges between j_i and j_l which do not cause a deadlock in scheduling. (Gating edges between j_k and j_l are added in a similar fashion). After the new job is merged, the gating number is updated over all vertices. We then pick subsequent jobs to merge iteratively until all jobs are included in the precedence graph. Provided n jobs with an average of m queries each, the time complexity of the merge phase is $O(n^3m^2)$. This overhead is low in practice given that the graph is sparse and completed queries are continually pruned.

Several conditions must be satisfied before each new gating edge is admitted to the precedence graph. Consider the admission of gating edges between a new job j_l and a previously merged job j_k. JAWS first evaluates each query in j_l in precedence order and determines, for each gating

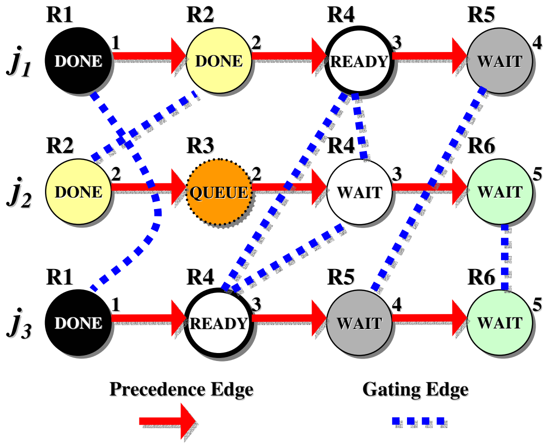


Fig. 5. Scheduling of three jobs with gating edges in JAWS.

edge encountered, whether to admit or omit the edge. Figure 4 details the pseudo-code for admitting a gating edge from $q_{l,n}$ (from the new job j_l) to $q_{k,m}$ (from the merged job j_k). Line 2 shows that $q_{l,n}$ inherits all gating edges incident to $q_{k,m}$ due to transitivity. Lines 3-7 determine the maximum gating number of prior queries in j_l. Lines 8-13 determine the feasibility of the new set of gating edges being considered. Line 9 checks for scheduling deadlocks using the gating number. Lines 10-13 ensure that new edges do not violate precedence constraints (*i.e.* each query in j_l has at most one gating edge to j_k).

Combining the concepts, Figure 5 illustrates the precedence graph after merging three jobs. Queries are shown in various states in which those marked DONE have completed execution and can be pruned from the graph. This includes $q_{1,1}$ from j_1 and $q_{3,1}$ from j_3 in which both queries access $R1$ and are co-scheduled by JAWS because of a shared gating edge. Query $q_{2,2}$ is marked QUEUE because it is awaiting execution. Queries $q_{1,3}$ and $q_{3,2}$ are in the READY state, which means they cannot be scheduled until all adjacent (via gating edges) queries is READY. Namely, query $q_{2,3}$ must be co-scheduled with queries $q_{1,3}$ and $q_{3,2}$ to realize data sharing on $R4$. Finally, queries marked WAIT await the completion of prior queries within the same job. Note that when a new job arrives, it can be added to the existing graph incrementally by computing new pairwise dynamic programs and then merging their solutions.

V. Extending LifeRaft in JAWS

Beyond job-aware scheduling, JAWS extends LifeRaft by (1) employing a two level scheduling framework to exploit locality of reference, (2) coordinating cache replacement with scheduling decisions, and (3) providing an automated solution for starvation resistance.

The two level scheduling framework is inspired by disk scheduling in Cello [22]. In Cello, disk bandwidth is first allocated to each application at a coarse level. Requests from different applications are then interleaved at a fine-grained level to meet quality of service guarantees. Similarly, JAWS first selects, at a coarse level, a single time step to evaluate. This selection is based on the mean workload throughput metric (Equation 1) computed over all atoms in a time step. JAWS then picks the time step with highest mean workload throughput, which tends to yield higher workload density and allows for the amortization of I/O over more queries. Next, up to k atoms with workload throughput metric greater than the mean are scheduled for execution. Here, k refers to the batch size or the maximum number of atoms that are co-scheduled per time step. The k atoms are sorted in Morton order and the corresponding sub-queries from each atom are evaluated in that order. Figure 6 illustrates two level scheduling in which atoms A_2, A_3, and A_4 are selected from time T_2 for $k = 3$.

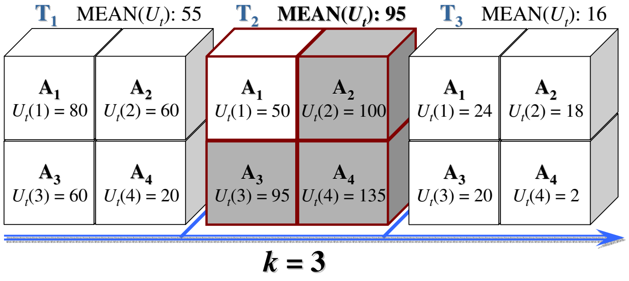


Fig. 6. Two level batch scheduling for $k = 3$.

Selection of batch size parameter k can significantly impact performance. A sufficiently large k ensures that the data access pattern conforms to the locality of reference. In particular, computations such as Lagrangian interpolation may require that a position accesses data from multiple atoms that are nearby in space. As such, sub-queries that access an atom as part of their kernel of computation should be scheduled together with sub-queries within that atom to avoid redundant data accesses later. Scheduling a batch of k atoms in a single pass accomplishes this. Moreover, a large batch size improves query response time by providing a balanced service to all queries within a time step rather than focusing on just a few contentious data regions across all time steps. However, too large a k can negatively impact throughput because execution order conforms less to the workload throughput metric. It may also flush the cache by processing a significant portion of each time step (roughly 20GB in size). We explore the parametrization of batch size k in Section VI.

A. Adaptive Starvation Resistance

JAWS provides an automated solution for starvation resistance by making adaptive and incremental trade-offs between query throughput and response time as workload saturation (*i.e.* query arrival rate) changes. Specifically, JAWS prefers to maximize data sharing and throughput when the workload is saturated in order to avoid exploding queuing times. In contrast, at low saturation, JAWS prefers to lower response times, relying on the system's additional capacity to preserve throughput. To do so, JAWS extends the parametrized *aged workload throughput* metric (Equation 2) to adapt automatically to workload saturation.

JAWS automatically and incrementally tunes the age bias, α, based on changing performance trade-offs as workload saturation varies. JAWS divides the workload into runs of r consecutive queries each, measures query performance for each run, and then adjusts α incrementally based on observed performance trade-offs compared with past runs.

Let $rt(i)$ and $tp(i)$ be the average response time and throughput of queries from the ith run consisting of Q_{ir}, ..., $Q_{(i+1)r-1}$. At the end of run i, α_i (the age bias during the ith run) is decreased (biased towards contention) or increased (biased towards age) in the following manner:

(1) if $\frac{rt(i)}{rt(i-1)} \geq 1$ and $\frac{tp(i)}{tp(i-1)} < \frac{rt(i)}{rt(i-1)}$, then
$\alpha_{i+1} = \alpha_i - min\{\frac{rt(i)}{rt(i-1)} - \frac{tp(i)}{tp(i-1)}, \alpha_i\}$

(2) if $\frac{rt(i)}{rt(i-1)} < 1$ and $\frac{tp(i)}{tp(i-1)} < \frac{rt(i)}{rt(i-1)}$, then
$\alpha_{i+1} = \alpha_i + min\{\frac{rt(i)}{rt(i-1)} - \frac{tp(i)}{tp(i-1)}, 1 - \alpha_i\}$

(1) indicates that if workload saturation rises (average response time increases between run $i-1$ and i) and query throughput did not increase at a commensurate rate, then the scheduler is biased towards contention. (2) increases α if saturation declines and query throughput is decreased, but there is no commensurate improvement in response time.

In practice, we need to ensure that α is adjusted incrementally and does not get "stuck" at a bad initial value. We avoid rapid variations in α between runs by incorporating performance of past runs. This is accomplished by calculating response time and throughput performance as $rt'(i) = 0.2rt(i)+0.8rt'(i-1)$ and $tp'(i) = 0.2tp(i)+0.8tp'(i-1)$ respectively for the ith run ($rt'(0) = rt(0)$ and $tp'(0) = tp(0)$). Also, it can be difficult to recover from a poor initial choice for α if workload saturation exhibits little change over an extended period. We vary the age bias to explore the performance curve if there is no change during two consecutive runs. (Note that it is not feasible to explore trade-off curves for multiple α values in real time).

B. *Cache Replacement*

Contention-based scheduling in JAWS benefits from cache reuse to avoid unnecessary I/O. In fact, our experience with Astronomy workloads [1] demonstrated that up to 40% of requests are serviced from the cache. We develop two algorithms for coordinating caching replacement with scheduling decisions to improve cache hit rates. We quantify the benefit they provide and the amount of computational overhead when compared with SQL Server's LRU-K based [10] page replacement algorithm in Section VI.

The first is a Segmented Least Recently Used (SLRU) based algorithm inspired by prior work [23, 24]. The SLRU approach exploits the notion that data atoms which are accessed frequently tend to be reused. For instance, particles with inertia may cluster in turbulent structures such as highly strained regions [25] such that these interesting data regions are repeatedly queried by multiple users. Thus, atoms that correspond to these regions of interest should not be flushed from the cache by queries that scan an entire time step only once. SLRU divides the cache into two segments: a *probationary* segment and a small (5 to 10% of the cache) *protected* segment. Cached atoms in each segment are ordered by recency of access. At the end of each run of the workload, SLRU promotes the most frequently accessed atoms into the protected segment. (Atoms evicted from this segment are inserted into the most recently used end of the probationary segment). Implementing this policy incurs almost no additional overhead.

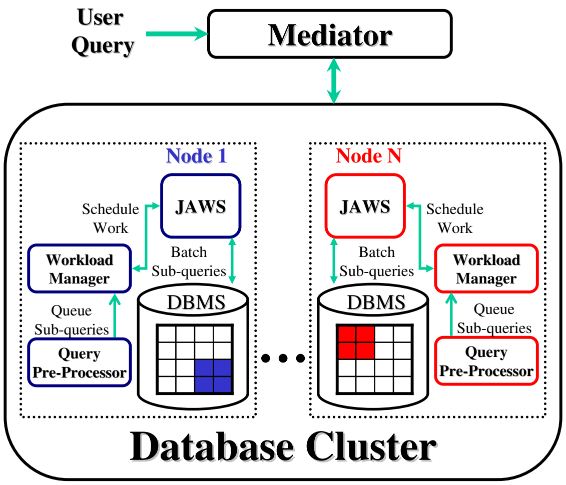


Fig. 7. JAWS architecture.

The second is a Utility Ranked Caching (URC) algorithm that incorporates full knowledge of workload access patterns and achieves the best cache hit ratio by evicting atoms that will likely be accessed farthest in the future. Specifically, URC ranks cached atoms in a priority queue based on their respective order in the two level scheduling framework. Because a batch of k atoms from the same time step are evaluated together in JAWS, the cache needs to coordinate with the scheduler and group atoms that are used together. Thus, atoms within the same time step are evicted in order of increasing workload throughput. Between two time steps t_i and t_j, if the mean workload throughput of t_j is greater, then atoms from t_i are evicted prior to those from t_j. Implementing this incurs significant maintenance overhead. After each new query or every time step that is processed, URC must update the ranks of all atoms in the corresponding time step.

C. *Architecture*

In the Turbulence cluster (Figure 7), data are partitioned spatially (as indicated by the shaded regions) and stored across different nodes, each running a separate JAWS instance. Incoming queries are first evaluated by the Query Pre-Processor, which takes as input a set of positions to evaluate. The positions are then assigned to the workload queues of the corresponding atoms. Workload queues are sorted by the *aged workload throughput* metric and organized in a two level hierarchy by the Workload Manager. In addition, the Workload Manager maintains state information which includes the age

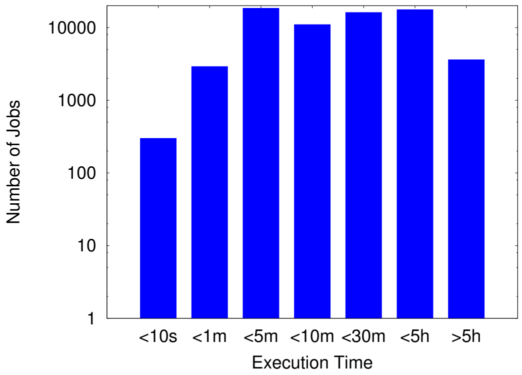


Fig. 8. Distribution of jobs by execution time.

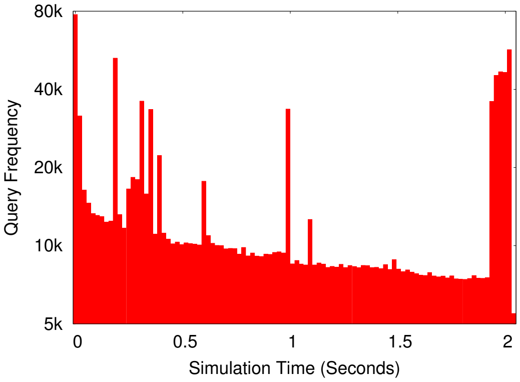


Fig. 9. Distribution of queries by time step accessed.

of the oldest query in each queue and a mapping of queries to their positions in the workload queues.

Finally, JAWS batches sub-queries of k workload queues from the time step with the highest mean *aged workload throughput* metric and submits them to the database. The list of positions are evaluated against the database and the results returned to JAWS. JAWS combines and buffers the sub-query results before delivering the final result to the user.

VI. Experiments

We implement JAWS for the Turbulence database and instrument its performance using workload derived from the SQL log on the Turbulence cluster. Our evaluation studies the performance benefits of job-awareness, two level scheduling, and adaptive starvation resistance. In addition, we show the scheduler's sensitivity to cache replacement and parameter selection. We compare JAWS to NoShare and LifeRaft with respect to query throughput and response time. NoShare evaluates each query independently (no I/O is shared) and in arrival order. While LifeRaft is adapted to Turbulence (Section III), the age bias α is not adaptive and is defined manually during initialization. Also, LifeRaft does not employ the two level scheduling framework (*e.g.* a single atom is scheduled at a time).

Queries are executed against a 800GB sample of the full 27TB Turbulence, which includes thirty-one time steps over 0.062 seconds of simulation time. Our experimental system only has capacity for a fraction of the production database. We do not run experiments on the production system because the workload is quite I/O intensive and would disrupt our scientist users. Experiments are conducted on a 2.33GHz dual-core Windows 2003 server with SQL Server 2005 and 4GB of memory. Data tables are striped across a set of four disks in RAID 5 configuration. (The log file is assigned to a separate disk to ensure sequential I/O). Next, we characterize the workload before presenting our main results.

A. Workload

Analysis of the Turbulence workload collected over the past two years (November 2007 to September 2009) revealed that a vast majority (over 95%) of queries belong to jobs. In total, there are over 80k unique jobs consisting of eight million queries that access nearly thirty billion positions. Figure 8 shows that jobs can vary greatly by execution time in which a majority (63%) persist between one and thirty minutes. While most jobs (88%) access data from only a single time step, 3% of jobs iterate over one hundred time steps (0.2 seconds of simulation time) or more, which covers more than 10% of the database. For evaluation, we employ a 50k query trace (roughly 1k jobs) from the week of July 20th in 2009, which exhibits a representative, bursty access pattern.

Figure 9 hints at the potential benefit from data-driven batch processing for the Turbulence workload. It illustrates query access frequency for each time step in which 70% of queries reuse data from a dozen time steps that are mostly clustered at the start and end of simulation time. (We observed similar reuse along the spatial dimension, although the skew is less pronounced). Significant reuse also occurs along other points, such as the spike between 0.25 and 0.4 seconds. In addition, there is a downward trend in access frequency that indicates many jobs which iterate over all time terminate mid-way through the experiment. A lower access frequency means that queries accessing time steps toward the end of simulation time are susceptible to starvation by the scheduler. Moreover, we found that queries which overlap in the time step accessed occur close temporally (*i.e.* concurrent experiments by the same user), which benefits caching.

B. Results

We compare the query throughput performance of JAWS against NoShare and LifeRaft. We implement two instances of LifeRaft, which explore two extreme values for the age bias α. $LifeRaft_1$ denotes a bias of 1, which schedules atoms in arrival order based on the age of the oldest request. It differs from NoShare in that queries referencing the same data as the current query in arrival order are co-scheduled. $LifeRaft_2$ denotes a bias of 0 or a contention-based, throughput maximizing scheduler. Moreover, to better quantify the benefits of various components in JAWS, we implement two versions: $JAWS_1$, which lacks job-awareness but includes two level scheduling and adaptive starvation resistance, and $JAWS_2$,

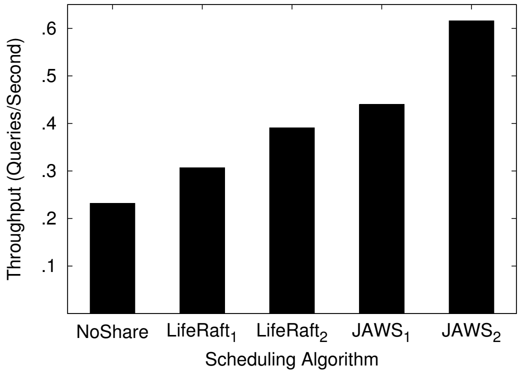


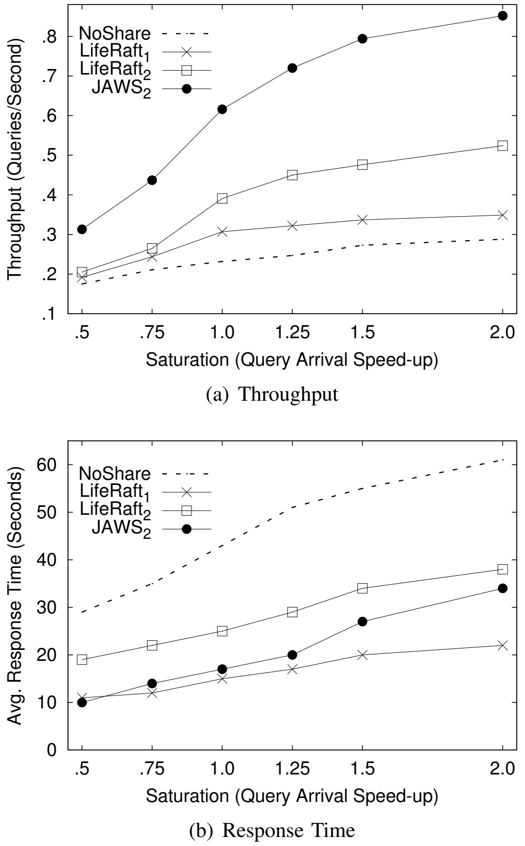
Fig. 10. Query throughput by scheduling algorithm.

which includes all components. We initialize α to 0.5 and set the batch size k (Section V) to 15 for JAWS. Figure 10 illustrates a 2.6 times improvement in query throughput for $JAWS_2$ over NoShare. As job-awareness is removed (*e.g.* not being able to exploit data sharing between jobs), $JAWS_2$ suffers a 30% performance drop. Transitioning from $JAWS_1$ to $LifeRaft_2$ incurs a less pronounced performance drop because much of the benefit is derived from contention-based batch processing. (Two level scheduling provides a modest 12% boost in performance). Finally, the 22% performance gap between $LifeRaft_1$ and $LifeRaft_2$ is attributed to improved cache reuse with contention-based scheduling.

Figure 11 explores the sensitivity of various algorithms to workload saturation. We measure workload saturation using a speed-up value, which modifies the arrival rate between consecutive jobs (*e.g.* the original workload has a speed-up of 1). Thus, if users submit job j_i two minutes following j_{i-1} in the original workload, then a speed-up of two indicates that j_i is now submitted in one minute. Throughput performance from figure 11(a) shows that both $JAWS_2$ and contention-based scheduler $LifeRaft_2$ scales as workload saturation increases. They benefit from higher concurrency in the workload in which there are more data sharing opportunities. In contrast, NoShare and $LifeRaft_1$ (arrival order scheduling) plateau much earlier at around 0.3 queries per second. Even though the algorithms converge at lower saturations (less data sharing opportunities), $JAWS_2$ still performs significantly better due to job-awareness.

The improved throughput performance of $JAWS_2$ and $LifeRaft_2$ comes from ignoring query arrival order, which leads to higher response times. Figure 11(b) shows that the response time performance gap remains fairly insensitive to changes in workload saturation. NoShare performs worst even when compared with $LifeRaft_2$ because the I/O overhead of evaluating each query independently leads to higher queuing times overall. $LifeRaft_2$ performs poorly even at lower saturations because it can delay queries indefinitely. The result also highlights the benefit of adaptive starvation resistance in JAWS. As the workload becomes saturated, JAWS decreases the age bias and aggressively maximizes throughput. This leads to large improvements in query throughput as evident from figure 11(a), but response time approaches that of $LifeRaft_2$. However, at lower saturations, increasing α becomes attractive because accepting a small drop in query throughput leads to large improvements in query response time. In fact, $JAWS_2$ outperforms $LifeRaft_1$ at the lowest saturation owing to better overall query throughput performance. Thus, JAWS makes effective trade-offs between throughput and response time based on workload saturation.

Figure 12 studies the sensitivity of JAWS to the batch size parameter k. Recall that k is used in the two level



(a) Throughput

(b) Response Time

Fig. 11. Sensitivity of performance to varying workload saturation.

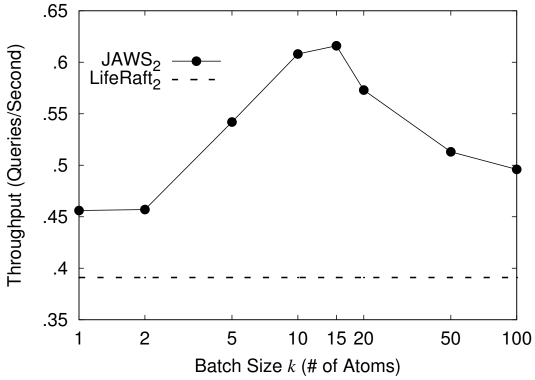


Fig. 12. Performance impact of varying batch size k in JAWS.

	Cache Hit	*Seconds/Qry*	*Overhead/Qry*
LRU-K	47%	1.62	-
SLRU	49%	1.56	< 1ms
URC	54%	1.39	7ms

TABLE I
PERFORMANCE AND OVERHEAD OF CACHING ALGORITHMS.

framework for scheduling a batch of k atoms per time step. The results indicate that the optimal batch size lies between ten and fifteen, which can vary based on the workload and amount of memory. If k is too small, the scheduler fails to exploit locality of reference in the computation and may access the same atom multiple times on different passes. However, even at $k = 1$, JAWS outperforms $LifeRaft_2$ due to job-awareness. At the other end, increasing batch size beyond 20 leads to rapid performance degradation as it negatively impacts cache reuse by evicting its contents. Moreover, too large a k means that scheduling conforms less to contention as measured by workload throughput. The impact beyond 50 is marginal because only atoms with workload throughput greater than the mean value are considered for scheduling.

We now evaluate our cache replacement algorithms, SLRU and URC, against SQL Server 2005's LRU-K based [10] page replacement, which lacks workload knowledge. We do not modify page replacement within SQL Server, but rather manage a 2GB cache externally from the database by first retrieving a set of atoms from the database and then flushing SQL Server's buffer. Computation cost is simulated in the database on a static set of atoms that are pinned in memory.

Table I illustrates the cache hit ratio, query performance, and overhead of each algorithm. (Note that overhead for SQL Server's page replacement is not measured). Exploiting workload knowledge improves cache hit by 7% for URC and a modest 2% for SLRU. (5% of the cache is allocated to the protected segment in SLRU). This translates into 16% and 4% improvement in query performance for URC and SLRU respectively. While the benefits are modest, the low overhead (a few milliseconds) incurred makes these optimizations attractive. Overhead is low for SLRU because atoms are assigned to the protected segment only once per run. For URC, since the time step with the lowest mean workload throughput does not change frequently, the overhead is low in practice. Moreover, metadata for both SLRU and URC is small. (Total metadata size is roughly 30MB even if all four million atoms fit in cache).

VII. DISCUSSION

We evaluated several techniques in JAWS that generalize the batch processing framework using the Turbulence database as a motivating application. These include job-aware scheduling for queries with data dependencies, which improves query throughput by 30%. A two level scheduling framework that exploits locality of reference in the computation to achieve a 12% improvement. A technique for starvation resistance that achieves the best of two worlds: lowering response time when the workload is less saturated and maximizing throughput during high saturation. Lastly, cache replacement algorithms that provide up to 16% performance benefit by exploiting workload knowledge. (The relative benefit of URC improves with increased workload saturation). Overall, JAWS achieves three-fold and 60% improvement in query throughput over NoShare and LifeRaft respectively.

In the future, we plan to provide quality of service guarantees for scientific workloads in a batch scheduling environment. The aged workload throughput metric enables starvation resistance through loose completion order guarantees, which may be insufficient for short-lived interactive queries. We are currently exploring techniques that provide predictable and fair completion time guarantees that are proportional to query size (*e.g.* short queries are delayed less than long queries). We observe that even with real-time constraints that bound the completion time of queries, there is still elasticity in the workload that permits the reordering of queries to exploit data sharing.

Another direction is to design declarative style interfaces that better support job-related optimizations during scheduling. Scientists that submit their workload through the interface can provide hints about the data requirements of their experiments and help the scheduler perform pre-fetching. This can be accomplished by, for instance, allowing users to explicitly link related queries or to pre-declare the time and space of interest. The scheduler can use this information to pre-fetch data atoms for a job and avoid page faults. For instance, we can extrapolate the trajectory of jobs in time and space (*i.e.* the velocity of the bounding box or time step delta between consecutive queries) to predict which data atoms are accessed by subsequent queries. This can also help mask the cost of random reads by pre-fetching large amounts of data.

Alternatively, scientists could encapsulate their jobs within the database to facilitate scheduling. Presently, users write a series of loops that iterate through each time step to gather data from the Turbulence cluster. The benefit is that scientists can arbitrarily customize their experiments, e.g. experimenting with particles of different masses. However, this computing paradigm makes it difficult to identify data sharing opportunities because much of the computation occurs outside of the database. One solution in Turbulence would implement the functionality to iterate over space and/or time inside the database. This provides the scheduler with a priori knowledge of all queries in a job at the expense of generality and flexibility. We plan to apply job-aware scheduling in JAWS to other data-intensive scientific workloads that, like Turbulence, involve complex workflows.

ACKNOWLEDGMENT

The authors wish to thank the Turbulence Research Group at Johns Hopkins University for their assistance with the data and workloads. This work is supported in part by CDI-II NSF grant CMMI-0941530.

References

[1] X. Wang, R. Burns, and T. Malik, “LifeRaft: Data-Driven, Batch Processing for the Exploration of Scientific Databases,” in *CIDR*, 2009.

[2] Y. Li, E. Perlman, M. Wan, Y. Yang, C. Meneveau, R. Burns, S. Chen, A. Szalay, and G. Eyink, “A Public Turbulence Database Cluster and Applications to Study Lagrangian Evolution of Velocity Increments in Turbulence,” *Journal of Turbulence*, vol. 9, no. 31, pp. 1–29, 2008.

[3] J. Gray and A. Szalay, “Where the Rubber Meets the Sky: Bridging the Gap Between Databases and Science,” *IEEE Data Engineering Bulletin*, vol. 27, no. 4, pp. 3–11, 2004.

[4] A. Szalay, G. Bell, J. Vandenberg, A. Wonders, R. Burns, D. Fay, and J. Heasley, “GrayWulf: Scalable Clustered Architecture for Data Intensive Computing,” in *HICSS*, 2009.

[5] J. N. Heasley, M. Nieto-Santisteban, A. Szalay, and A. Thakar, “The Pan-STARRS Object Data Manager Database,” *Bulletin of the American Astronomical Society*, vol. 38, no. 1, p. 124, 2007.

[6] M. Abdelguerfi, V. Mahadevan, N. Challier, and M. Flanagin, “A High Performance System for Viewing, Querying, and Retrieval of Geospatial Data Distributed Across a Beowulf Cluster,” in *HPCCS*, 2004.

[7] The JHU Turbulence Database Cluster. [Online]. Available: http://turbulence.pha.jhu.edu

[8] A. Szalay, J. Gray, A. Thakar, P. Kuntz, T. Malik, J. Raddick, C. Stoughton, and J. Vandenberg, “The SDSS SkyServer - Public Access to the Sloan Digital Sky Server Data,” in *SIGMOD*, 2002.

[9] E. Otoo, D. Rotem, and A. Romosan, “Optimal File-Bundle Caching Algorithms for Data-Grids,” in *SC*, 2004.

[10] E. J. O’Neil, P. E. O’Neil, and G. Weikum, “The LRU-K Page Replacement Algorithm for Database Disk Buffering,” in *SIGMOD*, 1993.

[11] J. Myllymaki and M. Livny, “Relational Joins for Data on Tertiary Storage,” in *ICDE*, 1997.

[12] S. Sarawagi, “Query Processing in Tertiary Memory Databases,” in *VLDB*, 1995.

[13] J.-B. Yu and D. J. DeWitt, “Query Pre-Execution and Batching in Paradise: A Two-Pronged Approach to the Efficient Processing of Queries on Tape-Resident Raster Images,” in *SSDBM*, 1997.

[14] H. Andrade, T. Kurc, A. Sussman, and J. Saltz, “Efficient Execution of Multiple Query Workloads in Data Analysis Applications,” in *SC*, 2001.

[15] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” in *OSDI*, 2004.

[16] H. C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, “Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters,” in *SIGMOD*, 2007.

[17] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, “Pig Latin: A Not-So-Foreign Language for Data Processing,” in *SIGMOD*, 2008.

[18] P. Agrawal, D. Kifer, and C. Olston, “Scheduling Shared Scans of Large Data Files,” in *VLDB*, 2008.

[19] W. O’Mullane, N. Li, M. Nieto-Santisteban, A. Szalay, and A. Thakar, “Batch is Back: CasJobs, Serving Multi-TB Data on the Web,” in *ICWS*, 2005.

[20] The Sloan Digital Sky Survey. [Online]. Available: http://www.sdss.org

[21] S. B. Needleman and C. D. Wunsch, “A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins,” *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970.

[22] P. Shenoy and H. Vin, “Cello: A Disk Scheduling Framework for Next Generation Operating Systems,” in *SIGMETRICS*, 1997.

[23] T. Johnson and D. Shasha, “2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm,” in *VLDB*, 1994.

[24] R. Karedla, J. S. Love, and B. G. Wherry, “Caching Strategies to Improve Disk System Performance,” *Computer*, vol. 27, no. 3, pp. 38–46, 1994.

[25] K. D. Squires and J. K. Eaton, “Preferential Concentration of Particles by Turbulence,” *Physics of Fluids*, vol. 3, no. 5, pp. 1169–1178, 1991.