I/O Streaming Evaluation of Batch Queries for Data-Intensive Computational Turbulence

Kalin Kanov, Eric Perlman, Randal Burns, Yanif Ahmad, Alexander Szalay
Department of Computer Science
Johns Hopkins University
Baltimore, Maryland 21218
{kalin, eric, randal, yanif, szalay}@cs.jhu.edu

ABSTRACT

We describe a method for evaluating computational turbulence queries, including Lagrange Polynomial interpolation, based on partial sums that allows the underlying data to be accessed in any order and in parts. We exploit these properties to stream data from disk in a single pass and concurrently evaluate batch queries. The combination of sequential I/O and data sharing improves performance by an order of magnitude when compared with direct evaluation of each query. The technique also supports distributed evaluation of queries in a database cluster, assembling the partial sums from each node at the query mediator. Interpolation is fundamental to computational turbulence, over 95% of queries use these routines, and the partial sums method allows the JHU Turbulence Database Cluster to realize scale and throughput for our scientists' data-intensive workloads.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Scientific databases, Statistical databases*; H.2.4 [**Database Management**]: Systems—*Query Processing*

General Terms

Design, Performance

Keywords

Data-intensive computing, I/O streaming, query evaluation, query optimization, database clusters, software for high-throughput computing.

1. INTRODUCTION

Data-intensive computing has revolutionized access to the computational simulation of turbulence. Previously, high-resolution data were available only to researchers using high-performance computing facilities. Researchers perform large simulations that are analyzed during the computation with only a small subset of data stored for subsequent analysis. As a result, the same simulations must be repeated after new questions arise that were not initially obvious; most breakthrough concepts cannot be anticipated in advance. In order to provide public access to world-class simulations, we have built the Turbulence Database Cluster at Johns Hopkins. This data-intensive service allows users to query the entire space-time evolution of the direct numerical simulation (DNS) of forced isotropic turbulence [7, 10]. It supports ad-hoc querying and data mining. Experiments execute against stored data and may be varied or repeated without re-simulation. The service currently occupies 67 TB in two databases managed by the 50 node, 1.1 PB Graywulf cluster [14].

A clustered database approach to computational turbulence has revealed performance issues for data-intensive workloads. Query throughput and system scalability limit the utility of the service, restricting the number of concurrent queries and increasing the time needed to complete experiments. Heavy usage can slow down the service by a factor of 10 to 20. The first iteration of query evaluation techniques [7] were adapted from simulation code and exhibit poor access locality at all levels of the memory hierarchy and incur substantial storage overhead associated with the replication of data across nodes. The predominant turbulence query performs spatial interpolation of a vector field at a specific point based on high-order Lagrange Polynomials. To evaluate each point query requires a kernel computation (Figure 2) using data points from the simulation, e.g. 8^3 data points for 8^{th} order Lagrange Polynomial interpolation. Typically, scientists request batches of up to 100,000 point queries from the same simulation timestep. Evaluation of these queries consists of embedded loops that iterate over the kernel of computation. This does not access data coherently; it retrieves data from the same cache line or page in multiple loop iterations. It also accesses the same data values multiple times when the kernels of multiple queries overlap.

To improve memory and I/O performance, we define a data-driven partial-sums method for the concurrent evaluation of multiple high-order Lagrange interpolation queries in a single, streaming pass over the data. Since the Lagrange interpolation is a linear combination of the data values and the Lagrange polynomials, which only depend on the target position and the resolution of the grid, the queries can be evaluated incrementally and in parts. Therefore, we perform the computation for all queries at the same time by maintaining a partial sum for each target point and accessing the data sequentially. This makes memory accesses coherent in

all levels of the cache hierarchy, regardless of the cache line size. Sequential access patterns use the full parallelism of the memory hardware, avoid associativity or bank conflicts, and make processor, I/O, and database prefetching effective. Most importantly, I/O Streaming reduces the overall I/O requirements. Batches of concurrent queries share I/O; a single value from the database contributes to all queries for which the kernel of computation contains the point.

Evaluating by partial sums supports distributed evaluation and eliminates the need for data replication among database nodes. Previous approaches have replicated a kernel half-width at the boundaries of a partition in order to avoid transferring data among nodes (Section 2). When a computation kernel spans data partitions, the Turbulence Database Cluster mediator splits a single query into two or more partial-sum queries to the multiple database nodes and combines the results. Distributing queries has minimal performance overhead and reclaims the 42% space used for replicated data that localized queries to single nodes [10]. It also supports arbitrarily large kernels; replication restricted us to 8^{th} order Lagrange interpolation previously.

I/O streaming and evaluating by partial-sums applies to a broad class of decomposable functions in the realm of data-intensive science. This includes all computations that are expressed as a linear combination of the data samples, such as differentiation, integration, and filtering. We have implemented I/O streaming and distributed evaluation in the second generation Turbulence Database Cluster. This database stores the output of a magneto-hydrodynamic astrophysics simulation, requiring 40 TB of storage. We have evaluated our technique on user workload traces from the JHU Turbulence Database Cluster. We have realized a speedup of at least 6 times and up to 50 times in the overall performance of queries that compute Lagrange interpolation when compared with a direct method of evaluation. Since Lagrange interpolation is the most commonly used function this translates into improved performance for scientific experiments.

2. BACKGROUND

We describe two versions of the Turbulence Database Cluster that define the *immersive turbulence* approach [10] in which we store the output of world-class, high-resolution simulations in a database cluster and allow scientists to explore, analyze, and visualize the data through Web services. The first version of the database hosts 27 TB of publicly-available data from a direct-numerical simulation (DNS) of forced isotropic turbulence over 1024 timesteps of a 1024^3 grid [7]. We explore some of the design flaws and limitations of our first design, which motivated the current line of research. We also discuss the magneto-hydrodynamic (MHD) data set stored in the second version of the database that implements I/O streaming. This second version hosts 40TB of data representing 1024 timesteps on a 1024^3 grid. It is currently available to our collaborators only as we validate the data and test and debug functionality.

In both versions of the database, data are partitioned spatially and temporally across the database cluster and are accessed through a Web server. The Web server acts as a mediator that divides user requests according to the spatial partitioning of the data and submits them for execution to

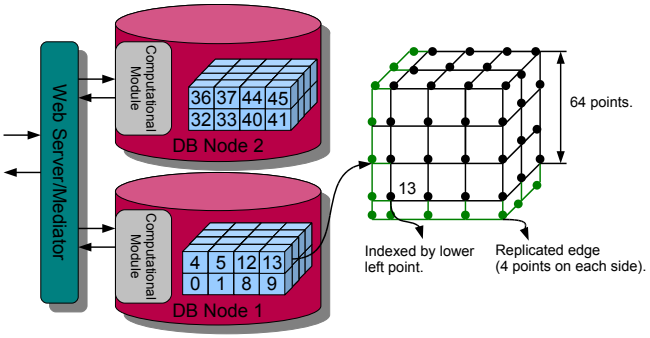


Figure 1: Diagram of the JHU Turbulence Database Cluster architecture deployed on two nodes.

the appropriate database nodes (Figure 1). Li et al. [7] and Perlman et al. [10] describe the architecture in detail.

Simulation data are partitioned into data cubes or "atoms" of fixed size (represented as blobs in the database), which are also the fundamental unit of I/O. We use blobs in order to amortize I/O and reduce the storage overhead associated with the generation of an index. All computations, such as Lagrange interpolation, require a cube of data as a kernel (512 points for 8^{th} order interpolation). Additionally, indexing each individual point would nearly double the storage requirements.

The Morton z-order space-filling curve governs the spatial partitioning and organization of the data. The curve provides the mapping of the three-dimensional data to the linear ordering in which the atoms are laid out on disk. The Morton z-order curve was chosen because it preserves spatial locality well and the indexes are easy to compute. The spatial access method used to store and retrieve the data atoms to and from disk is a standard B+ tree clustered index, which is keyed with a combination of the time step and the Morton z-curve index.

Our experience deploying the first version of the database revealed several design errors. We stored all attributes in a single table, which reduces I/O performance. For example, scientists would be interested in just one of the fields (velocity or pressure). However since velocity and pressure were stored together in the same atom data for both would be retrieved when each particular query was evaluated. We also chose to replicate data on the edges of partitions in order to localize the computation of kernels to single database nodes. Since the highest order interpolation supported was 8^{th} order Lagrange interpolation, which has a kernel of size 8^3, the required replication was 4 data points in each dimension (a kernel half-width). This introduced ∼42% storage overhead.

The second version of the database amends these decisions in part due to the I/O streaming techniques we develop. We employ vertical partitioning in order to improve the specificity of reads. The MHD data set consists of 3 vector fields, velocity, magnetic field, and magnetic vector potential, and the scalar pressure field. These data are partitioned into 4 tables respectively, and thus when a request for a particular field is made we only have to read the specific data instead

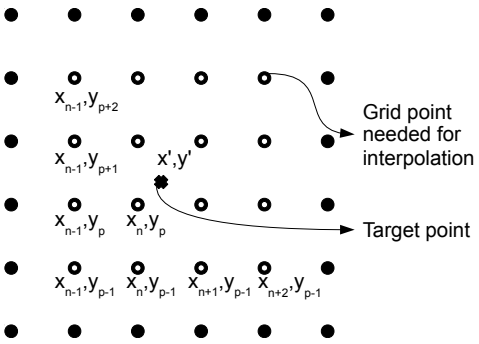


Figure 2: Data requirements for Lagrange Polynomial interpolation at a target point (x', y'). A 4^{th} order interpolation in two dimensions (shown) accesses a square of size 4 around the target point.

of all of them. This vertical partitioning is reminiscent of column-store databases [3, 2, 13]. In order to reduce wasted data transfer and to improve memory performance, we reduce the size of a data atom to 4^3 (768B of storage for vector fields, e.g. storing V_x, V_y and V_z together). Small data atoms do not pollute the cache and have small transfer times from disk that outweigh the potentially higher costs of more disk seeks. Evaluating by partial-sums allows us to distribute the computation for kernels that cross node boundaries and work with smaller atoms eliminating the need for replication as described in Section 3.3.

3. I/O STREAMING

We have implemented a data-driven partial-sums method for the computation of high-order Lagrange interpolations in the JHU Turbulence and MHD Database Clusters. We perform the computations by means of an I/O stream that takes a single pass over the data. For each target point, we maintain a partial sum of the results that we update incrementally as we access relevant data points. We also use partial sums to implement distributed evaluation of interpolation kernels across multiple database nodes. Each database node executes its part of the computation and maintains a partial sum. The mediator combines the partial sums and returns the final result to the user.

3.1 Partial-Sums

Lagrange polynomial interpolation uses a cubic grid of data surrounding the target point as the kernel of computation. The N^{th} order Lagrange polynomial interpolation of any point in space $\mathbf{p'}$ is given by

$$f(\mathbf{p'}) = \sum_{k=1}^{N} l_z^{q-\frac{N}{2}+k}(z') \sum_{j=1}^{N} l_y^{p-\frac{N}{2}+j}(y') \sum_{i=1}^{N} l_x^{n-\frac{N}{2}+i}(x') \cdot f(x_{n-\frac{N}{2}+i}, y_{p-\frac{N}{2}+j}, z_{q-\frac{N}{2}+k}) \quad (1)$$

in which $\mathbf{p'} = (x', y', z')$ is the position of the target point in 3-dimensional space and $f(x_i, y_j, z_k)$ represents the data stored at the node on the grid at location (x_i, y_j, z_k). The computation requires the data from an $N \times N \times N$ cube around the target point (Figure 2). The data grid location (x_n, y_p, z_q) is computed as $n = int(\frac{x'}{\Delta x}+\frac{1}{2})$, $p = int(\frac{y'}{\Delta y}+\frac{1}{2})$, $q = int(\frac{z'}{\Delta z}+\frac{1}{2})$, where Δx, Δy, Δz are the widths of the grid in the x, y and z dimensions. The Lagrange coefficients l are given below in which θ can be x, y or z and α can be n, p or q, respectively.

$$l_\theta^i(\theta') = \frac{\prod_{j=\alpha-\frac{N}{2}+1, j\neq i}^{\alpha+\frac{N}{2}} (\theta' - \theta_j)}{\prod_{j=\alpha-\frac{N}{2}+1, j\neq i}^{\alpha+\frac{N}{2}} (\theta_i - \theta_j)} . \quad (2)$$

We observe that Equation 1 can be computed incrementally and in parts. It is a linear combination of the data values and the coefficients $l_\theta^i(\theta')$ and the coefficients are independent of the data values at grid nodes.

We leverage partial sums to evaluate multiple point queries concurrently, using a single, sequential pass over the data. For each data atom, we update all of the interpolation kernels that include data from it, evaluating the portion of the computation contributed by the data points $f(x_{n-\frac{N}{2}+i}, y_{p-\frac{N}{2}+j}, z_{q-\frac{N}{2}+k})$ that are part of this atom. We allocate space for the partial sum and compute the Lagrange coefficients on the first data point in the kernel. Interpolation queries remain active until the last data access.

We compute the Lagrange polynomial coefficients efficiently based on Purser and Leslie's procedures [11]. Purser and Leslie observed that the values in the denominator of Equation 2 are constant and do not depend on the coordinates of the target point. Thus, the values in the denominator are pre-computed and reused for the entire batch. Also, careful coding of the formulas eliminates redundant multiplications. These two optimizations reduce the time complexity of the computation of the coefficients to $O(N)$ from $O(N^3)$. In our system, these techniques result in small performance gains, because I/O, not computation, bounds data-intensive workloads.

3.2 I/O Streaming

Direct approaches to the evaluation of turbulence queries produce cache faults and perform redundant I/O, accessing the same data multiple times. These approaches gather the data points in the kernel of a given query in their entirety before evaluating the interpolation function. Kernels that span data atoms (partitions) perform multiple I/Os to collect the kernel data. As many as 27 I/Os are needed for points that span three atoms in all three dimensions. These I/Os produce seeks in the disk system and perform incoherent accesses in cache memories, making unaligned requests for small amounts of data in each cache line. Atoms that cover multiple kernels are read multiple times by different queries. This results in cache misses at all levels in the hierarchy, including to disk. The Morton z-order (and other space-filling curves) cluster data well. However, no linear ordering of data localizes all computations, because the data are 3-dimensional and partitioned.

Evaluating by partial-sums allows us to stream I/O, performing a sequential scan of the data that reads each data atom only once. We execute a multi-point query (or a batch query) as follows. For each query, we create hash table en-

tries for each of the data atoms needed by the query's interpolation kernel. The hash table is keyed by the Morton z-curve index of the data atoms and looks up the partial sum of the query. Thus, each hash table entry contains the list of point queries that need to be updated when reading the corresponding data atom. We maintain a small amount of metadata for each query in addition to the partial sum, including the number of data atoms that have been read, the total number of atoms required, and the Lagrange coefficients needed for interpolation. To retrieve data, we build a temporary table that stores the z-indexes of the needed data atoms and perform a join between this temporary table and the data table. The database chooses its preferred I/O plan for this join, which always accesses the underlying data table sequentially.

The results of this join form an I/O stream on which we evaluate partial sums. As a data atom arrives, we perform a hash table lookup to determine all queries that require points within the atom and update their partial sums (Figure 3). The first time we update a partial sum for a query, we compute the Lagrange coefficients for that target point. We cache these coefficients for reuse on subsequent partial sums. This is a time/space tradeoff; the coefficients could be recomputed for every data atom were the system memory constrained. When all of the needed data atoms have been processed, we return the result to the mediator.

The benefits of I/O streaming come at the expense of modest memory consumption. For each query, we create a hash table entry for every data atom it accesses, typically around 20 per query. Other metadata for the queries, including the partial sum and the Lagrange coefficients, are allocated dynamically and only retained between the first and last update of its partial sum. Our evaluation shows that I/O streaming requires only tens to hundreds of megabytes of memory.

We sort the temporary table for batches smaller than 100,000 queries and when more than 1.1 queries access each data atom on average. In these cases, the database chooses a nested loops join and performance improves when both relations are sorted on the join key. Sorting larger batches has a negative impact on query performance, because the database chooses a sort merge join, which sorts the temporary table again.

Our final optimization performs loop unrolling to ensure that data are accessed sequentially within each cache line. The original direct computation of Lagrange interpolation loops over the x then y then z dimensions for the entire kernel. Because vector components are stored as tuples in row major order, e.g. velocity is $\langle V_x, V_y, V_z \rangle$, these loops perform strided access to individual velocity components, which reduces the coherency of memory access and negatively impacts memory throughput. I/O streaming accesses multiple data points in each data atom. For I/O streaming, we unroll the vector component loop and access the data sequentially in memory. Cache lines are consumed in their entirety and the necessary coefficients are loaded and used only once. Iterating in the appropriate order scans data sequentially and eliminates random memory access.

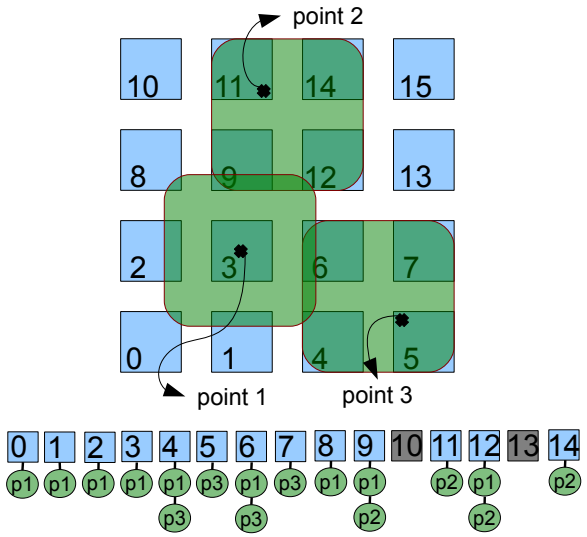


Figure 3: Processing a batch query. The kernel computation of each query is shown in green. The data atoms (blue) that are intersected by the kernel of each query need to be accessed. The arrival order of each atom and the query that needs data from it is shown on the bottom, atoms in gray are not needed and are not read.

3.3 Distributed Evaluation

Partial sums evaluation makes it possible to evaluate Lagrange interpolation queries across nodes of the database cluster. When interpolation kernels span multiple database nodes, each of the nodes computes the partial sum for the data points that it stores and returns it to the mediator. The mediator assembles the sums to complete the computation and returns results.

We use distributed evaluation of queries to eliminate replication among database nodes. The first version of our database was constrained to evaluating each query on a single database node. Direct evaluation requires all of the data to be available and we wanted to avoid the complexity of inter-node queries. As a consequence, we replicated a kernel half width of data around every data partition. Queries near the boundary of the partition access data in the replicated half width within the interpolation kernel. This resulted in a 42% storage overhead [10]. Using partial sums, we eliminate this overhead and reclaim the storage space. Our evaluation shows that distributed evaluation incurs little overhead.

4. EXPERIMENTS

We evaluate I/O streaming query evaluation by partial-sums using microbenchmark workloads characteristic of user patterns and on user query traces from the Turbulence Database cluster. Microbenchmarks isolate the performance benefits by query type, data sharing, and data access pattern. User query traces show that I/O streaming realizes an order of magnitude performance increase in practice.

Our evaluation compares **I/O Streaming** against several other evaluation techniques that allow us to isolate the important performance factors. These include:

- **Direct:** The direct method of evaluation that processes queries in arrival order and executes a *SELECT* query for each target point in order to retrieve all of the data atoms that cover its kernel of computation. The interpolation computation consists of nested loops that evaluate one component of a vector field after another (e.g. V_x first then V_y then V_z).

- **Sorting:** An improvement on Direct that also executes a *SELECT* query for each target point, but sorts the target points in Morton z-curve order before processing them. We sort the input in Morton z-curve order since the data atoms are organized in this order on disk and we expect the number of disk seeks to be reduced when reading them in this order. For the interpolation computation, we implement the optimizations described in Section 3: iterating in the correct order and evaluating the components of a vector field at the same time. This reduces random memory access and improves the cache locality of the computation.

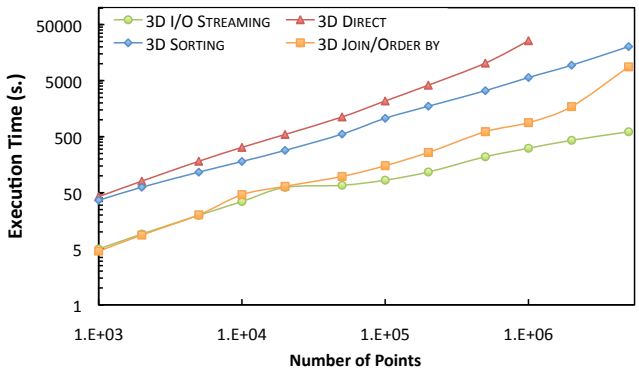
- **Join/Order By:** A direct method that was redesigned to make use of a join. This eliminates the overhead of executing multiple queries and the database query execution engine can take advantage of efficient read-ahead and prefetching techniques. The method uses an *ORDER BY* clause on the sequence id of the input queries in order to ensure that all of the data for a query have been read-in before performing the Lagrange interpolation. The interpolation computation is performed in the same fashion as for the Sorting method.

All methods compute 8^{th} order Lagrange interpolation. The Join/Order By strategy executes a single query and takes advantage of database read-ahead and prefetching. Join/Order by is a substantial improvement over the query at a time evaluation of Direct and Sorting. However, Join/Order By does not benefit from data sharing and its performance degrades with respect to I/O streaming for queries that are large or dense in space.

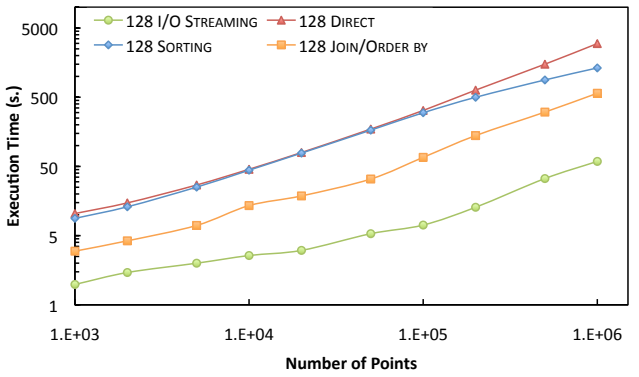
Experiments use a dedicated version of the MHD database cluster that stores ∼300 time-steps of the velocity fields of the MHD DNS (∼0.6 seconds of simulation time). The data are evenly partitioned across two databases based on splitting the z dimension. Database nodes are 2.33 GHz dual quad-core Windows 2003 servers with SQL Server 2008 and 8GB of memory. Data tables are striped across seven disks on each of the nodes. We do not perform experiments on the production version of the Turbulence Database cluster as our results would be affected by queries run by the users and would negatively impact scientists' workloads.

4.1 Microbenchmarks

To define microbenchmarks, we analyzed the query logs and selected two query patterns that are common among users, but have differing data requirements, sharing, and spatial extent. The **3D** workload distributes points randomly throughout the entire cubic volume of a specific timestep. Users employ this query pattern to generate unbiased global statistics. The **128** workload randomly distributes points within a randomly chosen sub-volume of size 128^3. Random points within sub-volumes are useful for particle tracking



(a) 3D



(b) 128

Figure 4: Execution time for randomly distributed points in the entire 1024^3 space (3D) and in a 128^3 subset of the entire space (128).

along field lines in a region of interest and creating animations of 3-d turbulence within a memory/space budget. We vary the number of points in a batch query between 1000 and 5 million in order to examine I/O scalability and data sharing. Batch sizes of 100,000 or more are typical of user workload.

Our principal finding is that I/O streaming improves the performance of direct evaluation by an order of magnitude over point query evaluation techniques (Direct and Sorting). Figure 4 compares the execution time of all methods, displayed in log scale. Sorting improves performance by up to a factor of two, because it generates I/O patterns that are more sequential. However, Sorting evaluates each query one at a time and reads the same data multiple times when it is accessed by multiple kernels, incurring cache misses. I/O streaming reads each element only once. It never takes a cache miss and accesses data more coherently, in storage and memory order. This results in a further ten times performance gain.

For queries without data sharing, the performance benefits come from evaluating queries as joins. Join/Order By requests the data atoms needed by each target point and executes the query as a single join. This is much more efficient than the multiple selection queries used by query-at-a-time

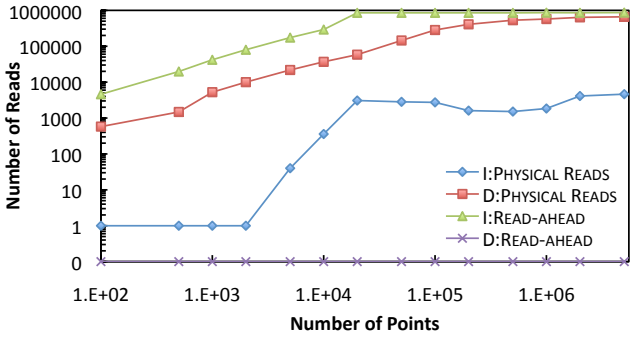
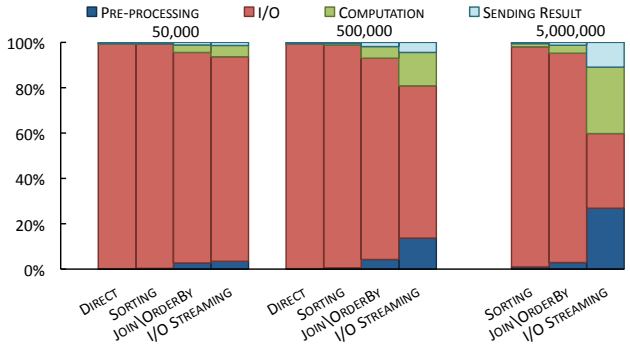
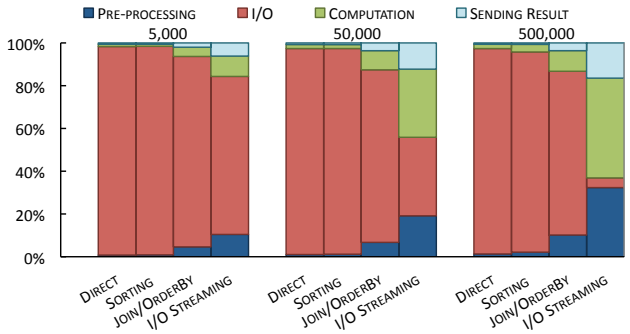


Figure 5: Physical and read-ahead reads of I/O streaming (I) and the direct evaluation (D).



(a) 3D



(b) 128

Figure 6: Breakdown of the execution time for randomly distributed points in the entire 1024^3 space (3D) and a 128^3 subset of the entire space (128). For more than 10^6 points Direct was not executed because the time exceeds reasonable bounds.

methods. Join/Order By results track those of I/O streaming for smaller queries in the 3D workload. Because 3D randomizes target points over the entire 1024^3 volume, there is essentially no data sharing for fewer than 10,000 queries. The sort induced by the *ORDER BY* clause also does not have a significant impact in those cases. Joins allow the database query plan generator to pick the most efficient plan for the execution of the query. For example, for batches of up to 10,000 target points a nested loops join with unordered prefetch is executed. For batches of 20,000 target points, a sort merge join is executed. For 50,000 and more points, the database chooses a hash match join. At this point, prefetching and read-ahead become very effective and we see the execution time increasing at a lower rate. Beyond 50,000 points, the physical I/O remains more or less constant, because read-ahead prefetches the entire time step.

For larger queries and queries with data sharing, I/O streaming benefits from more effective cache usage. Each database atom is accessed only once even if it is needed by multiple queries. This accounts for the differences between I/O Streaming and Join/Order By. If an atom is needed by more than one query it will be accessed more than once due to the *ORDER BY* clause in the *JOIN* statement, and if the atom was evicted from the database cache it will have to be read from disk again. In the 128 workload, even a small number of queries share data, because the volume is restricted. Therefore, the negative effects of accessing the same data multiple times degrades performance for as few as 1000 point queries.

A closer look at I/O shows that the strictly sequential access pattern of I/O streaming makes prefetching effective, which helps account for the large performance improvement over direct evaluation. Figure 5 shows the aggregate I/O statistics generated by SQL Server 2008 for the 3D queries. Executing a single join query results in substantially fewer physical reads (up to 500 times). Instead, the database performs efficient read-ahead, which populates the cache with data. Database reads to prefetched data are logical (cache hits) and do not generate physical I/O. On the other hand the direct methods evaluate queries one at a time and do not take advantage of read-ahead. All database reads result in physical I/O, which explains the poor performance.

Decomposing queries into their component costs reveals that I/O streaming alleviates the I/O bottleneck for large data-intensive turbulence queries and computation of the interpolation function emerges as the most costly operation. Figure 6 shows the breakdown of execution time into query preprocessing, I/O, computation of the interpolation function, and transmission of query results for 3D and 128 workloads respectively. For 3D, the computation and I/O costs are roughly the same at 5 million query points. When target points become dense enough, the query reads the entire data space (12 GB for a timestep) and I/O costs stop increasing. The 128 workload accesses a smaller data region (25 MBs) and computation dominates for much fewer than 1M target points. I/O streaming incurs moderate pre-processing costs to determine the Morton z-indexes of the data atoms required by each query and generate the hash table that maintains them along with the partial-sums as described in Section 3.

As computation becomes a bottleneck, efficient interpolation and memory coherency become important. Loop unrolling to reorder memory requests so that they are sequential and pre-computing Lagrange coefficients [11] cuts computation costs by 15% on average and as much as 35% (Figure 7). Moreover, evaluating by partial-sums effectively exploits the data sharing opportunities that are available. It allows us to iterate over a data atom that is read into memory directly

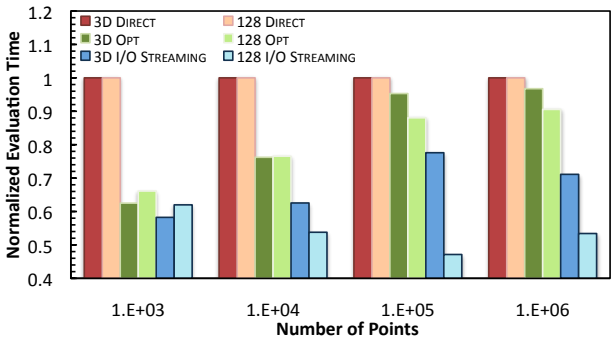


Figure 7: Computation time for interpolation comparing the direct method (Direct), loop unrolling and precomputing coefficients optimizations (Opt), and optimization plus I/O streaming data sharing (I/O Streaming). Computation times are normalized to Direct.

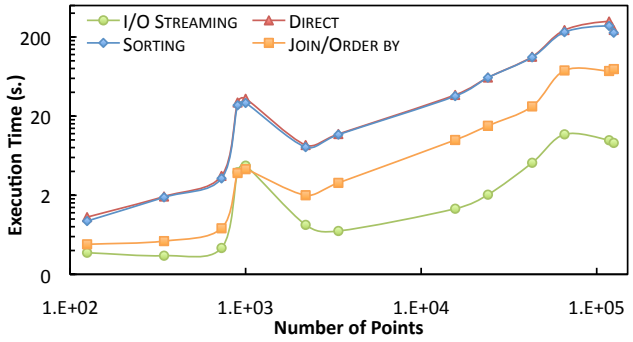


Figure 9: Execution time for queries derived from the usage log of the Turbulence Database cluster.

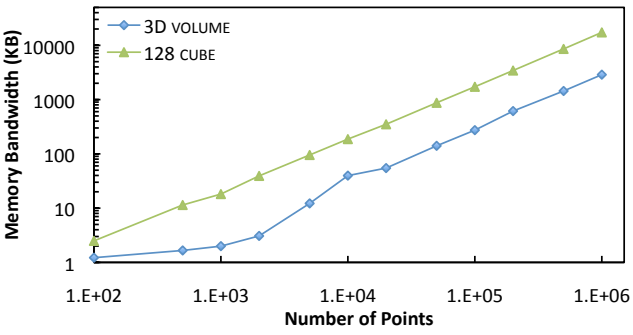


Figure 8: Memory bandwidth of an I/O streaming evaluation of Lagrange interpolation.

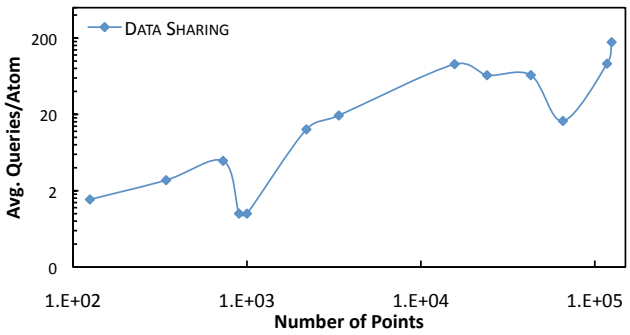


Figure 10: Amount of available data sharing for batch queries derived from the usage log of the Turbulence Database cluster (computed as average number of queries per atom for each batch query).

without having to copy data from it into a separate buffer. Because we do this for all queries that need data from the particular data atom, the computation costs are reduced up to a further 40%. We chose the size of a data atom to be 782B so that it fits within L1 cache and all partial sums using that data are evaluated as L1 cache hits.

The benefits of I/O streaming come at the cost of memory consumption to store partial results. I/O streaming requires space for the results to be allocated when the I/O stream encounters the first partial sum and retained until query completion when the stream reaches the last partial sum. Queries with interpolation kernels that span Morton z-order partition boundaries may have to wait quite awhile.

We define the maximum memory needed by active queries as the computation's memory bandwidth (Figure 8). Even the largest 3D computations use a negligible amount of memory: 3MB for 1 million query points. Points are spread out across the entire data space and only a small set of queries are active at any one time. The 128 workload has higher memory bandwidth, because the density of query points in small data regions means that a large fraction of queries can be active concurrently. However, the 10+ MBs needed for 1 million query points still fits easily within cache. Each query uses ∼150 bytes, including 96 bytes of cached coefficients for the Lagrange interpolation. To save space, the coefficients could be recalculated for each partial sum, but that is unnecessary given the small memory consumption in practice.

4.2 User Workload

We perform a similar evaluation on workload gathered from the usage log of the Turbulence Database cluster. The workload was chosen from a representative 10 day period beginning 05/21/2009. Figure 9 compares the execution time for queries of different sizes. The results are similar to the microbenchmarks with I/O streaming performing up-to 8 times better than Join/Order By. The bump in the execution time for the batches of 900 and 1000 queries is due to the fact that target points in these queries were distributed randomly in the entire space, whereas for most of the other queries the target points were densely clustered together.

Figure 10 shows the amount of data sharing as the average number of queries per atom for the user workload. The correlation between available data sharing and execution time

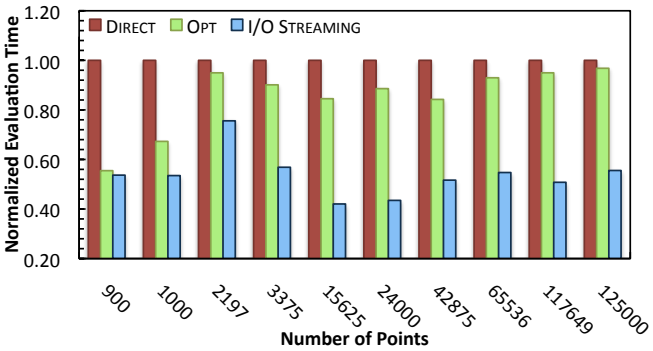


Figure 11: Execution time compared for the direct method (Direct), loop unrolling and precomputing coefficients optimizations (Opt), and optimizations plus I/O streaming (I/O Streaming) for queries derived from the usage log of the Turbulence database. Computation times are normalized to Direct.

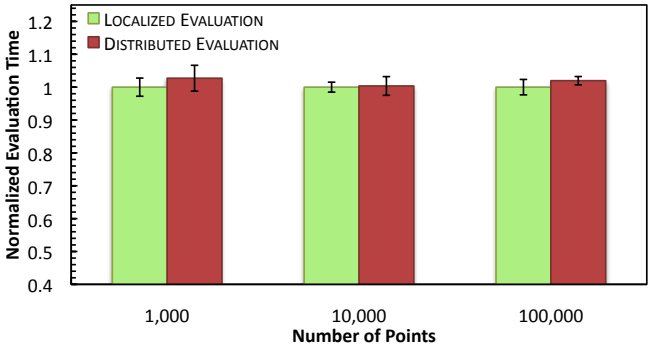


Figure 12: Performance of local and distributed computation of partial sums. (Normalized to local execution time).

is evident as less data sharing leads to larger execution times and more data sharing to smaller execution times.

Figure 11 shows the time to perform the Lagrange interpolation computation on the user workload. The result is consistent with the microbenchmark evaluation. The optimized version of the computation reduces the time by ∼15% and the I/O streaming reduces this by a further 40% on average. This result is more closely aligned with the results presented for the 128 workload as opposed to the 3D workload. This is due to the fact that most of the user queries were densely clustered in a small region of space, as is the case in the 128 workload.

4.3 Distributed Evaluation

Partial sums computation of Lagrange interpolation eliminates the need to localize each computation to an individual database node, which reclaims the 42% storage overhead of replicating an overlapping data region along partition boundaries. Instead, partial sums are used to perform distributed evaluation on multiple nodes, evaluating multiple partitions of the Lagrange interpolation kernel on different databases and combining their contributions on the mediator. This incurs some runtime overhead, because the mediator performs additional computation.

A microbenchmark experiment shows the overhead of distributed evaluation to be about 2%. We configure a cluster of eight virtual database nodes deployed on the two node experimental cluster. We compare two configurations: one that stores an entire timestep on a single node (no distributed evaluation) and the other that divides each timestep into eight partitions by splitting each dimension in two. We then query a 2-d plane across the middle of the timestep on a warm cache, which takes 2% longer on average in the distributed case (Figure 12). The warm cache is necessary to reveal the performance difference, otherwise I/O costs dominate. This particular 2-d query has poor distribution performance, because the interpolation kernel of every target point spans 2, 4, or 8 partitions. We conclude that distribution overheads are negligible.

5. RELATED WORK

Batch Processing: Data-intensive computing relies on the workload properties of batch computations to realize high-throughput. This is a fundamental tenet of Map/Reduce frameworks [20], which encapsulates computation and data access patterns in the functional abstractions of map and reduce. Shared scans between jobs that access the same files improve the I/O performance of map/reduce [1]. The same principles have been applied in databases [17, 18] that merge queries that share data. All-Pairs [8] and Wavefront [22] use declarative abstractions to computing functions over set combinations and in recurrences respectively. Programs in these abstractions are parallelized automatically and data and computation are placed to minimize I/O. These systems optimize I/O across multiple jobs based on co-scheduling jobs that share data. I/O streaming operates at a finer granularity within a single job, reformulating overlapping kernel queries as a single join and processing data in a strictly sequential fashion.

Paradise [23] uses *query batching* to execute multiple queries that access tape-resident data. Queries are grouped into batches based on the tapes that they access and reordered in order to perform sequential I/O in a pre-execution step. Our I/O streaming evaluation also consumes data in the order in which they arrive and has the additional benefit that it does not impose any restriction on this arrival order. In contrast to *query batching* in the Paradise system, data do not have to be buffered and are consumed immediately upon retrieval. This lowers the memory consumption significantly and does not pollute caches with buffered data.

Crescando [16] executes multiple queries and updates (operations) by means of a join over sets of operations and a table. We adopt a similar strategy in pre-processing queries into a temporary table and joining the temporary table against a data table. We extend their model by performing the computation in parts because each query point requires multiple data items for completion.

Stream Processing: Scientific applications have been mapped to stream processors that are required to process data sequentially [5, 21]. Yang et al. [21] extract the inter-dependence of arrays and loops and perform optimizations

that include coarse-grained program transformations (loop reordering and fusion, unifying arrays) and fine-grained program transformations (reordering computation and data inside loops). Our streaming approach extends to I/O as well as computation and applies to more complex overlapping kernel functions.

Romein et al. [12] present a software approach to process streaming telescope data on a supercomputer. They reduce the number of memory references during the correlation of signals by keeping correlations in registers and reusing the samples. This exploits the data sharing opportunities that arise during the computation similar to I/O streaming.

Querying Continuous Functions: Extensions to SQL [9] and an algebra for scientific data sets [19] have proposed integrated interpolation, including query optimization. I/O streaming is a promising technique for evaluating such interpolation procedures were the extensions adopted.

MauveDB [4] defines model-based views that represent sparse and irregular raw data as a "uniform grid-based approximation." They also discuss interpolation-based views in which the values at target points are a function of neighbors. Grumbach et al. [6] develop a query language that extends the standard relational framework to include interpolation. FunctionDB [15] defines a query language and algebraic query processor for the creation and querying of function views, interfaces to continuous functions based on regression. None of these works focus on batch queries.

6. CONCLUSIONS AND FUTURE WORK

We present an I/O streaming technique for the evaluation of data-intensive workloads that consist of decomposable kernel computations. Example functions include differentiation, integration, and filtering, but we focus on Lagrange interpolation, which is used by 95% of queries to the Turbulence Database Cluster. Users submit multiple point queries to be executed in a batch. I/O streaming computes the entire batch in a single, sequential pass over the data by maintaining a partial sum of the result for each point. The partial sum of a query is updated whenever the I/O stream produces data within the point's kernel. When compared with direct methods of computation of interpolation, I/O streaming performs an order of magnitude faster.

The single I/O streaming pass over the data improves cache locality and reuse at all levels of the memory hierarchy. The method takes advantage of the efficient execution of joins in modern database systems. It makes full use of read-ahead and prefetching and accesses data sequentially in memory and on disk. I/O streaming exploits the data sharing opportunities that arise during the evaluation of multiple queries and achieves execution times that increase at lower rates for large batches of points and batches whose points are densely clustered.

The focus of this work has been the optimization of a single job containing a batch of target points. We require the single job so that we can represent the batch as a join and preprocess the kernels to order the data access and identify data sharing among kernels. However, data sharing also exists among multiple unrelated jobs that can be leveraged to reduce I/O and improve throughput for map/reduce systems [1], scientific databases [17], and even specifically in the Turbulence Database Cluster [18]. Turbulence workloads often have multiple jobs that analyze overlapping sets of timesteps and could share I/O, i.e. read the data once for all jobs. The job-aware workload scheduler (JAWS) [18] detects this sharing and co-schedules jobs and timesteps. However, JAWS does not manipulate query scheduling and I/O ordering at a fine granularity and realizes much more modest performance benefits than does I/O streaming. We are currently investigating how to integrate partial sums, I/O streaming, and distributed evaluation into a multi-job scheduling framework, such as JAWS for turbulence data and shared-scans for Map/Reduce [1].

7. REFERENCES

[1] P. Agrawal, D. Kifer, and C. Olston. Scheduling shared scans of large data files. *Proceedings of VLDB*, 1(1):958–969, August 2008.

[2] P. A. Boncz, W. Quak, and M. L. Kersten. Monet and its geographic extensions: A novel approach to high performance GIS processing. In *Extending Database Technology*, 1996.

[3] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *SIGMOD*, 1985.

[4] A. Deshpande and S. Madden. MauveDB: supporting model-based user views in database systems. In *SIGMOD*, 2006.

[5] M. Erez, J. H. Ahn, J. Gummaraju, M. Rosenblum, and W. J. Dally. Executing irregular scientific applications on stream architectures. In *International Conference on Supercomputing*, 2007.

[6] S. Grumbach, P. Rigaux, and L. Segoufin. Manipulating interpolated data is easier than you thought. In *Conference on Very Large Data Bases*, 2000.

[7] 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*, 9:31–+, 2008.

[8] C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, and D. Thain. All-pairs: An abstraction for data-intensive computing on campus grids. *IEEE Trans. Parallel Distrib. Syst.*, 21:33–46, January 2010.

[9] L. Neugebauer. Optimization and evaluation of database queries including embedded interpolation procedures. In *SIGMOD*, 1991.

[10] E. Perlman, R. Burns, Y. Li, and C. Meneveau. Data exploration of turbulence simulations using a database cluster. In *Supercomputing*, 2007.

[11] R. J. Purser and L. M. Leslie. An Efficient Interpolation Procedure for High-Order Three-Dimensional Semi-Lagrangian Models. *Monthly Weather Review*, 119:2492–+, 1991.

[12] J. W. Romein, P. C. Broekema, E. van Meijeren, K. van der Schaaf, and W. H. Zwart. Astronomical real-time streaming signal processing on a Blue Gene/L supercomputer. In *Symposium on Parallel Algorithms and Architectures*, 2006.

[13] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden,

E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented DBMS. In *Conference on Very large Data Bases*, 2005.

[14] A. S. Szalay, G. Bell, J. Vandenberg, A. Wonders, R. Burns, D. Fay, J. Heasley, T. Hey, M. Nieto-Santisteban, A. Thakar, C. van Ingen, and R. Wilton. Graywulf: Scalable clustered architecture for data intensive computing. *Hawai'i International Conference on System Sciences*, 2009.

[15] A. Thiagarajan and S. Madden. Querying continuous functions in a database system. In *SIGMOD*, 2008.

[16] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *Proceedings of Very Large Data Bases*, 2(1), August 2009.

[17] X. Wang, R. C. Burns, and T. Malik. Liferaft: Data-driven, batch processing for the exploration of scientific databases. In *CIDR*, 2009.

[18] X. Wang, E. Perlman, R. Burns, T. Malik, T. Budavári, C. Meneveau, and A. Szalay. Jaws: Job-aware workload scheduling for the exploration of turbulence simulations. In *Supercomputing*, 2010.

[19] R. H. Wolniewicz and G. Graefe. Algebraic optimization of computations over scientific databases. In *Conference on Very Large Data Bases*, pages 13–24, 1993.

[20] 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.

[21] X. Yang, J. Du, X. Yan, and Y. Deng. Matrix-based streamization approach for improving locality and parallelism on ft64 stream processor. *J. Supercomput.*, 47(2):171–197, February 2009.

[22] L. Yi, C. Moretti, S. Emrich, K. Judd, and D. Thain. Harnessing parallelism in multicore clusters with the all-pairs and wavefront abstractions. In *High Performance Distributed Computing*, 2009.

[23] 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 *Conference on Scientific and Statistical Database Management*, 1997.