**Annual Report for Period:** 09/2010 - 08/2011
**Submitted on:** 06/16/2011
**Principal Investigator:** Yang, Jun .
**Award ID:** 0916027
**Organization:** Duke University
**Submitted By:**
Yang, Jun - Principal Investigator
**Title:**
III: Small: RIOT: Statistical Computing with Efficient, Transparent I/O

## Project Participants

### Senior Personnel

**Name:** Yang, Jun
**Worked for more than 160 Hours:** Yes
**Contribution to Project:**
Jun Yang has been serving as the faculty lead on this project.

### Post-doc

### Graduate Student

**Name:** Zhang, Yi
**Worked for more than 160 Hours:** Yes
**Contribution to Project:**
Since the project's conception Yi Zhang has been the main student contributor. He has been involved in most of the research and development efforts, including the RIOT-DB system (CIDR 2009) and the next generation RIOT system (ICDE 2010, PVLDB 2011). Yi is currently focusing on RIOT execution and optimization issues.

**Name:** Thonangi, Risi
**Worked for more than 160 Hours:** Yes
**Contribution to Project:**
Risi Thonangi has been researching how to leverage the recently emerging SSDs (solid state drives) to improve performance. He has developed an index for SSDs that support efficient concurrent accesses despite the need for batch index organization. He is currently focusing on how to exploit SSD in RIOT workloads.

**Name:** Herodotou, Herodotos
**Worked for more than 160 Hours:** Yes
**Contribution to Project:**
Although not funded by this project, Herodotos Herodotou contributed to the development of the RIOT-DB system (CIDR 2009).

**Name:** Huang, Botong
**Worked for more than 160 Hours:** Yes
**Contribution to Project:**
Botong Huang is investigating how to run RIOT on data-parallel cloud computing platforms such as Hadoop MapReduce. He has experimented with various implementations of large dense matrix multiply on Hadoop, and is currently helping to build a multi-input and dynamic load balancing extension to Hadoop.

**Undergraduate Student**

**Name:** Zhang, Weiping

**Worked for more than 160 Hours:** Yes

**Contribution to Project:**

Shortly after the conception of the project, Weiping Zhang joined the RIOT team and worked for an REU-funded research internship and an independent study course (Spring 2010). He contributed to the development of RIOT's native array storage manager and is a co-author on of ICDE 2010 demo paper.

**Name:** Yan, Jiaqi

**Worked for more than 160 Hours:** Yes

**Contribution to Project:**

Jiaqi Yan joined the RIOT team in the summer of 2010 on an REU-funded research internship. He studied several information retrieval applications with scalability demands to see how RIOT might help. He is currently working on extending PostgreSQL with better support for matrix operations and leverage of modern GPU hardware.

**Technician, Programmer**

**Other Participant**

**Research Experience for Undergraduates**

## Organizational Partners

**HP Labs**

HP Labs provided additional funding to RIOT in Year 2 through their Innovation Research Program. The PI visited HP Labs in Beijing in July 2010 to help jump-start the collaboration. The collaboration focused on the issues of database extensibility and parallelization, at both GPU and cloud level. See the section of this report on research and education activities for details.

## Other Collaborators or Contacts

Huan Feng (intern at HP Labs China), Meichun Hsu (HP Labs Palo Alto), Xing-Xing Ju (intern at HP Labs China), Keyan Liu (HP Labs China), Kamesh Munagala (Duke University), Lei Wang (HP Labs China), Min Wang (HP Labs China).

## Activities and Findings

**Research and Education Activities:**

Recent technological advances have enabled collection of massive amounts of data in science, commerce, and society. These large datasets have brought us closer than ever before to solving important problems such as decoding human genomes and coping with climate changes. Meanwhile, the exponential growth in the amount of data has created an urgent challenge. Today, much of advanced analysis is done

with programs custom-developed by statisticians. Unfortunately, progress has been hindered by the lack of easy-to-use statistical computing environments that scale to large datasets. Many existing tools assume that datasets fit in main memory; when applied to large datasets, they are unacceptably slow because of excessive disk input/output (I/O) operations. There have been many approaches toward I/O-efficiency, but none has gained traction with the statistical computing community. Disk-based storage engines and I/O-efficient function libraries provide only a partial solution, because many sources of I/O-inefficiency in a program remain at a higher, inter-operation level: e.g., how large intermediate results are passed between operations, how much performance can be gained by deferring and reordering operations, etc. Database systems seem to be a natural solution, with I/O-efficiency and a high-level language (SQL) enabling many high-level optimizations. However, work in integrating databases and statistical computing has mostly remained database-centric, forcing statisticians to learn unfamiliar languages and deal with their impedance mismatch with the host language.

To make a practical impact on the statistical computing community, the RIOT project seeks to extend R---an open-source statistical computing environment widely used by statisticians---to transparently provide scalability over large datasets. Transparency means no SQL, or any new language to learn. Transparency means that existing code should run without modification, and automatically gain efficiency. RIOT is developing an end-to-end solution that addresses issues on all fronts: I/O-efficient and parallel algorithms, deferred evaluation, pipelined execution, cost-driven optimization, smart storage and materialization options, and seamless integration with database systems and the interpreted host language.

In Year 2 of the project, we have investigated the following specific research problems.

A) RIOT native store. We started this study in Year 1 and have now completed it, culminating in a paper in PVLDB 2011. Previous work has found databases inadequate in storing arrays, the main reason being that the relational model does not exploit the fact that arrays are ordered collections of data. Storing array indices for a dense array wastes storage and only slows down access. As for a sparse array, it is unnecessary to store the zero-valued elements that occupy most of the array; storing only the nonzero elements together with their array indices is more efficient. An index such as a B-tree can be used for fast access to elements. Even though database systems are better at handling sparse arrays, it is still difficult to support various array layouts, i.e., linearization of elements of multidimensional arrays on disk with proper clustering to facilitate efficient access. Furthermore, since dense and sparse arrays require dramatically different storage strategies, we need a storage manager flexible enough to handle matrices of arbitrary sparsity. Another complexity is that array sparsity may vary over time and over different regions of the array. For example, an initially sparse matrix may be gradually populated with new data until a subregion becomes dense. These problems motivates the design of a novel array storage manager for RIOT.

We have developed a B-tree variant called the Linearized Array B-tree,

or LAB-tree, which supports flexible matrix layouts and automatically adapts to varying sparsity across parts of a matrix and over time. We reexamine the leaf splitting strategies and batched update flushing policies, for which common practices have been rarely questioned. We obtained theoretical and empirical results that contribute to the fundamental understanding of these problems. These results challenge the common practices. For leaf splitting, exploiting the fact that the domain of array indices is bounded and discrete, we devise a strategy that naturally produces trees with 'no-dead-space,' often twice as efficient as those produced by split-in-middle. This advantage does incur a fundamental trade-off---in the worst-case, split-in-middle has competitive ratio 2, while this strategy has 3, which is the best possible for any 'no-dead-space' strategy. Nonetheless, on common workloads, this strategy consistently and significantly outperforms split-in-middle. For update batching, we give a flushing policy with competitive ratio O(log^3⁡K) in the worst case, beating Omega(sqrt(K)) for the standard 'flush-all' approach (which flushes the entire batch when the buffer is full). For common workloads, however, flush-all actually performs better in practice. On the other hand, starting from a simple policy with a poor competitive ratio of Omega(K), we devise a randomized variant that incurs fewer number of I/Os than flush-all for some workloads (and comparable numbers for others). Finally, we note that our techniques are easy to implement as they do not require intrusive modifications to the conventional B-tree. Also, many of our results generalize to other settings. The idea of 'no-dead-space' splitting makes sense for other discrete, ordered key domains. Both this idea and the idea of new batched update flushing policies can be applied to storage of matrix in chunks (as opposed to B-tree leaves). A paper presenting our findings has been accepted to PVLDB 2011.

B) Extending Database Systems (DBMS). In collaboration with HP Labs, we have made significant progress towards demonstrating that, with careful design and engineering, we can overcome many purported disadvantages of DBMS in storing and computing with matrices. Specifically, we reduce the overhead of element-oriented storage and iterator-based execution by storing and processing matrices in the unit of 'chunks' (submatrices). We work around the inefficiency of DBMS execution engine for numerical computation by invoking highly optimized numerical libraries (e.g., BLAS) on a per-chunk basis. We avoid the awkwardness of SQL in expressing linear algebra operations with user-defined functions that operate on entire matrices.

Going beyond earlier work in this area, we developed and evaluated alternative matrix linearization and chunking schemes, and show superior flexibility and performance than popular approaches. For example, for matrix multiply AxB, the popular approach of chunking A by rows and B by columns and multiplying a pair at a time is not only suboptimal, but also makes the rather restrictive assumption that an entire row and an entire column fit in memory. A better strategy is to divide A and B into square chunks (so that the memory can hold three such chunks), and multiply two input chunks at a time and accumulate the result in an output chunk. Surprisingly, an even better strategy, with optimal computation-to-I/O ratio and best performance, is to multiply a partial column of A and a partial row of B and accumulate the result in a square output chunk.

We also considered different strategies for implementing matrix operations within DBMS (specifically, PostgreSQL), and demonstrated their trade-offs. Some matrix operations (e.g., multiply) can be expressed in SQL using user-defined functions and aggregates that operate on chunks. However, more complex operations (e.g., QR factorization) require more elaborate control of execution. PL/pgSQL turned out to have high overhead. The strategy that we found to offer most flexibility and best performance is using C-language functions operating on entire matrices (but still breaking up execution in chunks). With large enough chunks, impressive speedups can be achieved by using optimized libraries (including those for GPUs; more on it below) to operate on chunks within user-defined functions.

Overall, we have demonstrated that the combination of chunking, user-defined functions, and use of optimized libraries makes highly efficient matrix computation possible within a DBMS. GPU acceleration is also effective within DBMS. Initial findings have been summarized in a paper titled 'MaSSA (Massive-Scale Statistical Analysis) DBMS,' presented at HP Tech Con 2011. We are working on polishing results and preparing a submission for publication in a database research conference.

C) Parallelization with GPU. As discussed above, we have demonstrated that DBMS can effectively leverage GPU for parallel processing, by chunking execution and invoking GPU libraries on a per-chunk basis. However, with the approach of expressing matrix operations in SQL using user-defined functions and aggregates, it is difficult to reuse results that reside in the GPU memory across function invocations, because it is unsafe to assume invocation orders of user-defined functions and aggregates or the preservation of GPU memory state across invocations. In the case of matrix multiply, for example, this approach must copy the result of multiplying each two chunks in the GPU out from the GPU memory and into main memory. An optimization, which avoids such copying and better suits the GPU library API, is to let the GPU pin a result chunk in its memory for accumulating multiplication results it produces. This optimization would be possible with the approach of implementing matrix operations as user-defined C-language functions operating on entire matrices. As mentioned earlier, initial results were reported in a paper in HP Tech Con 2011. The submission under preparation will include results on avoiding CPU/main memory copying.

D) Parallelization with Cloud. In collaboration with HP Labs China, we have been studying how to support statistical computing workloads in Hadoop, an open-source implementation of MapReduce that has been popular for data-parallel cloud computing. We investigated basic operations and computation patterns ranging from matrix multiply, join, all-pairs computation, to friends-of-friends clustering. We started with the constraint that we cannot modify Hadoop itself, but quickly realized that we had to resort to ad hoc implementation tricks for better performance. We identified a number of limitations of Hadoop and addressed two of them by extending Hadoop's programming model and execution framework.

The first feature we added to Hadoop is better support for multiple inputs. The basic MapReduce framework takes only a single dataset as

input. In practice, many operations require multiple inputs, and Hadoop programmers usually handle them by tagging each data item with its source, which is inconvenient. Furthermore, mappers partition their output disjointly across reducers. However, for many multi-input operations (e.g., matrix multiply and all-pairs computation), the work to be divided forms a multi-dimensional space; thus, one data item may need to be sent to multiple reducers. Hadoop programmers usually handle this case by having mappers duplicate their output, which is inefficient. We have developed a multi-input extension for Hadoop that removes this inefficiency and is upward compatible; i.e., single-input Hadoop programs will continue to function as usual.

The second feature is better load balancing. The current Hadoop deals with 'stragglers' by re-running their tasks from scratch on other nodes who have completed their tasks. There is no possible reuse of partial work completed. Furthermore, it does not further divide the remaining work among more than one node. Therefore, this approach cannot resolve load imbalance caused by uneven partitioning of work, which is difficult to avoid because perfectly even partitioning of keys (or even of data) does not imply even partitioning of work. Our extension avoids repeating partial work and allows remaining work to be further divided, thereby enabling more robust load balancing.

Our experimental evaluation has shown good results across a number of workloads, and we believe our extensions will benefit a wide range of applications, including RIOT. We are currently preparing a paper to submit to a database or cloud research conference.

E) RIOT Execution and Optimization Issues. Database queries exhibit a limited number of fairly simple data access patterns. However, linear algebra operations common in statistical computing give rise to a much larger space of complex access patterns typically expressed using nested loops. We are developing a framework that allows these access patterns to be captured---either through code analysis or user specification---in a form amenable to automatic optimization. Specifically, we would like to automatically and jointly choose processing algorithms and data layouts for a workflow involving multiple operations, inputs, and intermediate results. We also would like to automatically identify opportunities that allow multiple operations with common inputs to execute simultaneously and share accesses---either reads of disk-resident inputs or output generated from another operation. The first case reduces disk accesses; the second case avoids materializing and reading intermediate results on disk. The trade-off, however, is the higher memory requirement of running multiple operations simultaneously, which may result in a higher number of I/Os. An optimizer needs to evaluate this trade-off and decide whether to share using a cost-based analysis. Our study is currently underway and we expect to complete it in the first half of Year 3.

F) Working with Solid-State Drives (SSDs). Solid-state drives are becoming a viable alternative to magnetic disks for many workloads, but their performance characteristics, particularly those caused by their erase-before-write behavior, often render conventional algorithms and data structures suboptimal. We have been studying how to use them effectively for various database and linear algebra workloads.

The first problem, which we began to investigate in Year 1 and completed in Year 2, is indexing. There have been various proposals of indexes specialized for SSDs, but to make such indexes practical, we must address the issue of concurrency control. Good concurrency control is especially critical to indexes on SSDs, because they typically rely on batch updates, which may take long and block concurrent index accesses. We have designed, implemented, and evaluated an index structure called FD+tree and an associated concurrency control scheme called FD+FC. Our evaluation confirms significant performance advantages of our approach over less sophisticated ones. A paper describing our results is under submission.

We are currently working on two other problems---sorting and permuting on SSDs. A main application of these problems in RIOT is data layout conversion. Conventional I/O-efficient algorithms for these problems assume that disk reads and writes have equal costs. However, writes are more expensive for SSDs, in terms of performance, energy consumption, as well as wear. We have found that by doing more reads we can reduce the number of writes, and lower the costs overall. We expect to complete our study of these problems in the first half of Year 3.

The progress we have made thus far, as summarized above, are in line with the project plan outlined in our original proposal. In Year 3, we plan to wrap up our work on (B), (C), and (D) quickly, and complete (E) and (F) in the first half of Year 3. The remainder of Year 3 will be devoted to making a public code-release for RIOT, and continuing to explore cloud parallelization for RIOT beyond (D).

In terms of educational activities in Year 2, the PI has been on sabbatical, but has continued to closely supervise and train graduate and undergraduate researchers. The two undergraduate researchers have made solid contributions to RIOT: Weiping Zhang helped with the development of RIOT's native storage manager, and Jiaqi Yan studied RIOT applications and helped with the development and evaluation efforts of (B) and (D).

**Findings:**

In Year 1, we invested most of our effort in the prototyping and development activities, which provided us with much experience and insights. In Year 2, we expanded into several focused problems, including building a better storage engine and designing a better execution and optimization frameworks for RIOT, extending database systems to support matrix storage and computation, parallelization using GPU and cloud, and leveraging SSDs for better performance. Published results so far include: 1) the RIOT 'vision' paper in CIDR 2009, which also covers RIOT-DB; 2) a demonstration of a prototype for the next generation of RIOT in ICDE 2010; and 3) a paper on the RIOT storage engine in PVLDB 2011. A paper about indexing on SSDs is under submission. Two papers, one on extending database systems and one on extending Hadoop, are in preparation. We have released the code for RIOT-DB on the project website.

**Training and Development:**

Ph.D. students: Yi Zhang, Risi Thonangi, Botong Huang, Herodotos Herodotou.

Undergraduate students: Weiping Zhang, Jiaqi Yan

These students have gained research experience in algorithms, compilers, databases, high-performance computing, and programming languages.

**Outreach Activities:**

## Journal Publications

Yi Zhang, Kamesh Munagala, and Jun Yang, "Storing Matrices on Disk: Theory and Practice Revisited", Proceedings of the VLDB Endowment, p. , vol. , (2011). Published,

## Books or Other One-time Publications

Yi Zhang, Herodotos Herodotou, and Jun Yang, "RIOT: I/O-Efficient Numerical Computing without SQL", (2009). Conference Paper, Published
Collection: Proceedings of the 4th Biennial Conference on Innovative Data Systems Research (CIDR '09)
Bibliography: Asilomar, California, USA, January 2009

Yi Zhang, Weiping Zhang, and Jun Yang, "I/O-Efficient Statistical Computing with RIOT", (2010). Conference Demonstration Description, Published
Collection: Proceedings of the 26th International Conference on Data Engineering (ICDE '10)
Bibliography: Los Angeles, California, USA, March 2010

## Web/Internet Site

**URL(s):**
http://www.cs.duke.edu/dbgroup/Main/RIOT
**Description:**

## Other Specific Products

## Contributions

**Contributions within Discipline:**
At the end of Year 2 of the project, we have made a series of solid contributions toward enabling efficient statistical analysis over massive datasets. We have built two functional prototypes, RIOT-DB and RIOT, and published in CIDR 2009, ICDE 2010, and PVLDB 2011. A number of other papers are currently in preparation or under submission. For detailed descriptions of these contributions, please refer to the section of this report on research and education activities.

**Contributions to Other Disciplines:**
The PI has been part of two other interdisciplinary projects---one funded by NSF, which studies how to collect and analyze ecological

data from a sensor network, and another funded by NIH, which develops analytical and modeling tools for immunology. Some of the work in this project is motivated by the problems faced in the other projects, which will in turn benefit from our research results. As RIOT matures, it will be made available to the general statistical computing community for broader impact.

**Contributions to Human Resource Development:**

Ph.D. students: Yi Zhang, Risi Thonangi, Botong Huang.

Undergraduate students: Weiping Zhang, Jiaqi Yan.

**Contributions to Resources for Research and Education:**

The PI has integrated the research being carried out under this project into his education activities (see the section of this report on research and education activities for details). The RIOT tools are also publicly available in open source to other researcher and educators.

**Contributions Beyond Science and Engineering:**

## Conference Proceedings

## Special Requirements

**Special reporting requirements:** None

**Change in Objectives or Scope:** None

**Animal, Human Subjects, Biohazards:** None

## Categories for which nothing is reported:

Activities and Findings: Any Outreach Activities

Any Product

Contributions: To Any Beyond Science and Engineering

Any Conference