GenBase: A Complex Analytics Genomics Benchmark

Rebecca Taft*
MIT CSAIL
rytaft@mit.edu

Manasi Vartak*
MIT CSAIL
mvartak@mit.edu

Nadathur Rajagopalan Satish
Intel Parallel Computing Lab
nadathur.rajagopalan.satish@intel.com

Narayanan Sundaram
Intel Parallel Computing Lab
narayanan.sundaram@intel.com

Samuel Madden
MIT CSAIL
madden@csail.mit.edu

Michael Stonebraker
MIT CSAIL
stonebraker@csail.mit.edu

ABSTRACT

This paper introduces a new benchmark designed to test database management system (DBMS) performance on a mix of data management tasks (joins, filters, etc.) and complex analytics (regression, singular value decomposition, etc.) Such mixed workloads are prevalent in a number of application areas including most science workloads and web analytics. As a specific use case, we have chosen genomics data for our benchmark and have constructed a collection of typical tasks in this domain. In addition to being representative of a mixed data management and analytics workload, this benchmark is also meant to scale to large dataset sizes and multiple nodes across a cluster. Besides presenting this benchmark, we have run it on a variety of storage systems including traditional row stores, newer column stores, Hadoop, and an array DBMS. We present performance numbers on all systems on single and multiple nodes, and show that performance differs by orders of magnitude between the various solutions. In addition, we demonstrate that most platforms have scalability issues. We also test offloading the analytics onto a coprocessor.

The intent of this benchmark is to focus research interest in this area; to this end, all of our data, data generators, and scripts are available on our web site.

1. INTRODUCTION

There have been many benchmarks proposed by the DBMS community over the years including the Wisconsin Benchmark [17] (general data management), Bucky [18] (data type support), Linear Road [16] (streaming support), as well as the TPC benchmarks [14]. All have served to focus the attention of the community on a specific class of DBMS issues.

At the present time we see a sea change occurring in DBMS analytics. Specifically, the previous focus has been on traditional business intelligence (BI) where commercial products offer an easy-to-use GUI for standard SQL aggregates (COUNT, SUM, MAX, MIN, AVERAGE etc., with an optional GROUP BY clause). This focus is epitomized by TPC-H and TPC-DS. Going forward, we see the DBMS analytics market shifting quickly and dramatically to more complex analytics. This is the domain of data scientists where common operations include predictive modeling, data clustering, regressions, probabilistic modeling, and the like. To focus the reader, we give three examples of this class of analytics.

Automobile insurance companies are following the lead of Progressive and putting sensors in clients' cars to measure how they drive (fast starts, abrupt braking, speeding, etc.), how much they drive, and what time of day they drive (4 AM is presumably less safe than 10 AM). Current systems report sensor measurements every second and data is kept for many months. Once this data is processed, a driver can be characterized by several thousand variables. These variables are added to current data (home address, make of car, etc.) and other possible information (credit score, marital status, etc.). The goal is mass personalization of rating. This task requires fitting some sort of risk model to (perhaps various subclasses of) drivers. This is clearly a task for a data scientist, one that entails both data management (filtering to the subclasses) and complex analytics (the modeling).

A second example concerns satellite imagery. Various satellites orbit the earth collecting sensor data of the area underneath. Because the sensor scans a swath of the earth as the satellite orbits, the collected data resembles a wide piece of scotch tape that is wrapped continuously around the earth. A workflow of processing steps turns a week or two worth of data into a vector of values for the various grid cells covering the surface of the earth. The "best" (e.g. minimum cloud cover) data is used from the multiple times the satellite passes over a given cell, and typical cell size is between 10 meters and 1000 meters on a side. An earth scientist wants to compute some surrogate from this vector to represent a study phenomenon (snow cover, vegetation index, etc.) for some area of interest. This represents data management activity. In addition, multiple satellites cover the same area, but are invariably gridded differently. Making use of multiple satellites requires a "regridding" operation whereby one co-ordinate system is transformed to the other one; in the process forming a vector of values for this derived cell structure from multiple overlapping physical cells. This interpolation/regridding is an example of a complex analytical calculation.

Finally, consider a "quant" on Wall Street charged with constructing an electronic trading program. This program will watch a feed of trades and/or bid/ask quotes, looking for patterns of inter-

*These authors contributed equally

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2595633.

est. If a pattern is seen, then trading is activated based on the pattern observed. Of course, the quant must discover interesting patterns to base his trades on, based on the closing price of every stock for every trading day for (say) the last 20 years. There are about 15,000 publicly traded securities in the US and about 4,000 trading days in 20 years. The data is thereby a 4,000 by 15,000 array, which we call S. As a first step, he might compute the covariance between the time series of all pairs of securities. This is a 15,000 by 15,000 matrix with cell values between -1 and +1. Stocks with substantial correlation (positive or negative) would then be subjected to further scrutiny. Ignoring a constant and the requirement of subtracting off the mean of each time series, this all-pairs covariance is:

$$S \times S^T$$

This is clearly complex analytics. In addition, a quant might want to focus on stocks with a market capitalization over $1B or on stocks of companies headquartered in New York State, again requiring traditional data management operations to be interspersed with complex analytics.

In summary, we believe it is important to focus the DBMS community on this new class of data science problems, which we can characterize by the following two requirements:

1. **Data management.** Support for traditional DBMS operations prevalent in previous benchmarks.

2. **Complex analytics.** Support for analytics including linear algebra and statistical operations on array data (as in the Wall Street example above).

Historically, such data science problems have been addressed by using a DBMS (or custom code) for the data management with some sort of custom code or a statistics package for the analytics. Therefore, first, software packages should provide better support for data scientists so they do not have to copy and reformat their data to make use of multiple packages to accomplish their objective. In other words, a single software system should be capable of performing both kinds of operations, and we hope the proposed benchmark will focus attention in this area. In addition to calling for an integrated approach, we also see a need for such software systems to scale to data sizes large enough to span multiple nodes in a datacenter/cluster. As data sizes grow dramatically with the advent of cheap sensors and measurement techniques, solutions that can handle large datasets and that scale well on both data management and complex analytics tasks are essential.

A second issue concerns performance. Linear algebra operations such as matrix multiply are cubic in the size of the arrays and performance can vary by several orders of magnitude depending on the choice of implementation language or matrix algebra library. For example, simulating linear algebra operations in SQL, as proposed in [16, 21] will result in code that is largely interpreted, and may have performance problems. Obtaining good performance on our proposed benchmark will require carefully optimizing both data management and statistical/array operations.

A third issue concerns specialized hardware. There has been recent interest in performing data management on accelerators such as GPUs, FPGAs or Intel® Xeon Phi™ coprocessors[1] [22]. Obviously, such accelerators are more adept at the computations found in complex analytics than they are at routine data management. Hence, there is a clear opportunity to involve specialized hardware in the process of turbocharging the analytics found in our benchmark.

[1] Intel, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

To focus the community attention on all three issues, this paper presents a data science benchmark composed of both data management and complex analytic operations. Although numerous DBMS benchmarks have been proposed over the years [17, 18, 16, 14], none has adequately served the scientific community. As noted above, use cases for data management and complex analytic operations exist in many domains, but we have chosen a genomics use case for exposition here. We chose genomics for a few reasons: (a) genomics is quickly becoming a significant source of petabytes of data (current sequencing facilities can generate 15 petabytes of compressed genetic data per year [24]), (b) current systems for genomic analysis cannot handle data at this scale, (c) the majority of analysis in genomics consists of a combination of data management and statistics or linear algebra (see examples in Section 3), and (d) genomics is a societally important application domain to which the database community can contribute. As such, in Section 2, we present a short primer on genomics, followed in Section 3 by a description of the data sets and queries that comprise our benchmark.

Then, we continue in Section 4 by presenting experimental results of running the benchmark on a variety of processing engines, including R, Postgres, a popular column store, SciDB, and Hadoop. We observe performance differences of several orders of magnitude between the various systems. Lastly, in Section 5 we show an example system with an assist from specialized hardware, and indicate the advantages of this approach, especially when large scale problems are tackled. Sections 6 and 7 conclude with some comments on the results and suggestions for future experiments.

2. GENOMICS PRIMER

DNA is commonly thought of as the blueprint for all living organisms. In humans, it is contained in the nucleus of each cell in our bodies, and encodes instructions for how each cell should grow and operate. Interest in genomics data is skyrocketing as the cost of sequencing the human genome decreases. Specifically, the cost of sequencing a human genome is approaching $1000, down from hundreds of thousands of dollars five years ago [6].

The DNA (or genome) for an individual is a sequence of about a billion values from an alphabet of size 4 (A, C, T, G). A, C, T and G stand for Adenine, Cytosine, Thymine and Guanine, which are the four molecules, also known as "nucleotides" or "bases", that make up DNA. These bases are linked together into a strand of nucleic acid by a phosphodiester backbone, and are bound to a complementary strand by hydrogen bonds. The sequences are complementary because each A in one strand is replaced with a T in the other strand, and each C is replaced with a G. These pairs, A-T and C-G, are often referred to as base pairs. The two complementary strands of DNA naturally spiral around each other, creating the well-known double helix.

Genes are special subsequences of base pairs interspersed throughout the genome that are responsible for encoding traits inherited from our ancestors. When activated, genes produce RNA and protein, which are the workhorses of cells and carry out the "instructions" of the DNA. RNA is a single-stranded nucleic acid similar to DNA, but with Thymine (T) replaced with Uracil (U). Inside the nucleus of cells, genes are "transcribed" into RNA, and in most cases, the RNA transcript is then "translated" into protein. Each sequence of three bases in the RNA represents one of 20 different amino acids, the building blocks of protein. Biologists have identified about 20,000 different genes in the human genome that encode such protein [19].

It is important to note that not every gene is transcribed and translated into protein in every cell. In reality, some genes are active

and create high volumes of RNA and protein, while other genes are less active. Which genes are active at any given time varies by cell type. For example, genes responsible for cell growth are likely to be more active in cancer cells than in normal cells. Likewise, genes responsible for producing the oxygen-carrying protein hemoglobin will be more active in blood cells than in skin cells. Even though all cells in the human body contain the entire genetic code, only a subset of genes are actually active at any given time in any given cell. The level of activity or "expression" of a gene can be almost as important as the gene itself for determining phenotype (the physical manifestation of a gene). As a result, mechanisms for regulating the expression of certain genes are an active area of research.

Microarray technology supports the measurement of the expression level of thousands of genes at the same time with a single chip. For example, Affymetrix [1] sells quartz chips containing tens of thousands of microscopic probes designed to detect the presence and relative amount of specific segments (subsequences) of RNA. Biologists often collect microarray data for thousands of tissue samples from different patients in order to perform some sort of statistical analysis. This data collection results in a large dense matrix of floating point values, in which the columns correspond to different genes and the rows correspond to tissue samples of different patients.

3. BENCHMARK SPECIFICATION

In this paper we present a genomics benchmark developed for microarray data. As described in the previous section, microarrays are used to measure expression values for several thousand genes at the same time. This data is then used for a variety of analyses to identify gene function, disease pathways (sequences of molecular actions involved in the manifestation of a disease) and potential drug targets. Queries selected for the benchmark have been identified in collaboration with biologists and bioinformaticians from Novartis and the Broad Institute. These queries represent operations such as SVD, regression and statistics that are most commonly executed on microarray data. While the benchmark does not include an exhaustive list of operations relevant in genomics, we believe that the queries are a good representative set. Finally, while this work focuses on microarray data, the data representations and operations discussed can be extended to include other types of genomic data such as sequencing data.

Genomics data can be extremely large scale. With 10^{4-5} gene expressions per sample and up to 10^{8-10} samples (multiple samples can be taken from one patient), we are looking at a problem that can scale to 10^{2-5} nodes of a large cluster. For the purpose of this paper, however, we look at much smaller datasets (with one sample per patient).

3.1 The Data

The benchmark consists of four types of data sets: microarray data, patient metadata, gene metadata and gene ontology data, as described below.

3.1.1 Microarray data

This is our main dataset for all operations and is commonly represented as a matrix with rows representing samples (or patients) and columns representing genes. An example matrix, M, is shown below, where $M_{i,j}$ contains the expression value for gene j in patient i. We represent this matrix as shown below. The array format contains the list of attributes stored per cell followed by the array dimensions.

1. *Relational form:* Microarray_data (gene_id, patient_id, experimental_value)
2. *Array form:* experimental_value[patient_id, gene_id]

	Gene_1	Gene_2	...	Gene_n
Patient_1	0.34	1.56	...	0.008
...	...	...	...	...
Patient_m	2.32	0.99	...	0.007

We use microarray data matrices of four different sizes to exercise the capabilities of the hardware and study how the algorithms work under varying workloads.

1. Small: 5K genes for 5K patients
2. Medium: 15K genes for 20K patients
3. Large: 30K genes for 40K patients
4. Extra large: 60K genes for 70K patients

We found that none of the systems could run on the extra large data set, so we do not provide experimental results for this data set. We supplement microarray data with the following additional datasets.

3.1.2 Patient metadata

For each patient whose genomic data is available in the microarray dataset, we store patient metadata including demographic and clinical information as noted below. Specifically, it contains the patient age, gender, zip code, and the drug response for their disease (assuming only a single disease). For ease of computation, disease is represented numerically, (e.g. diabetes = 1, bipolar disorder = 2) and our data set contains 21 diseases. We represent this matrix as follows:

1. *Relational form:* Patient_Metadata (patient_id, age, gender, zipcode, disease_id, drug_response)
2. *Array form:* (age, gender, zipcode, disease_id, drug_response)-[patient_id]

	Age	Gender	Zip Code	Disease	Drug Response
Patient_1	85	F	77071	1	0.45
...	...	...	...	...	...
Patient_m	90	M	01192	2	3.2

3.1.3 Gene metadata

For every gene in the microarray, we store gene metadata including the target of the gene (the id of another gene that is targeted by the protein from the current gene), chromosome number, position (number of base pairs from the start of the chromosome to the start of the gene), length (in base pairs) and function (coded as an integer, for example, cell division = 10 and photosynthesis=12). We represent this matrix as follows:

1. *Relational form:* Gene_metadata (gene_id, target, chromosome, position, length, function)
2. *Array form:* (target, chromosome, position, length, function)[gene_id]

	Target	Chromosome	Position	Length	Function
Gene_1	Gene_39	3	156	170	10
...	...	...	...	...	...
Gene_m	Gene_232	20	89	90	12

3.1.4 Gene Ontology (GO) data

Genes are organized into an ontology depending on the biological functions they serve. For instance Gene_1 and Gene_2 may be involved in respiration, while Gene_3 and Gene_4 deal with cell division. These categories form a tree structure, and a gene is placed at the appropriate place(s) in the tree. The gene ontology is represented as below:

1. *Relational form:* Gene_ontology(gene_id, go_id, 0 or 1) where 1 implies that the gene with the given gene_id belongs to the GO category with id=go_id while 0 indicates that it doesn't.
2. *Array form:* belongs_to[gene_id, go_id]

	GO_1	...	GO_k
Gene_1	0	...	1
...	...	...	...
Gene_m	1	...	1

The sizes of the supplementary datasets are chosen to match the sizes of the microarray matrix. To protect privacy and to ensure that datasets of all four sizes are consistent, we use synthetically generated data. This data has been modeled on existing microarray and patient data, and generated using the tool available on our benchmark website [7].

3.2 The Queries

Our benchmark consists of 5 queries, which are run against the schema described in the previous section. The queries represent a mix of data management, linear algebra and statistical operations that are representative of genomic workloads.

3.2.1 Query 1: Predictive Modeling

An important use case for genomic data is predicting drug response based on gene expression and using this information to determine drug regimens for patients. One way of determining drug response is to build a regression model predicting drug response based on gene expression data. In this query, we select the expression data for a subset of genes with a particular set of functions (an example is shown below). Then we construct a linear regression model where the target variable is the patient drug response and the independent variables are expression values. In our implementation, we use a QR decomposition technique to solve the linear regression problem.

	Gene_1	...	Gene_k	Drug Response
Patient_1	0.34	...	0.008	5.12
...	...	...	...	...
Patient_m	2.32	...	0.007	3.78

We adopt the following workflow:

1. Select genes with a particular set of functions (since functions are encoded numerically, for example, function < 250)
2. Join the result with the microarray and patient data and project out the patient drug response and microarray data values
3. Restructure the information as a matrix (if required)
4. Build a regression model to predict drug response

3.2.2 Query 2: Covariance

Genes that have similar expression patterns (e.g. they are under or over expressed together) and those that have opposing expression patterns (e.g. they are under expressed when others are over expressed) are likely to be functionally related (e.g. part of the same pathway, located near one another etc.) Therefore, the covariance between expression values can be used to identify biologically related genes. An example covariance matrix is shown below.

	Gene_1	...	Gene_n
Gene_1	0.34	...	5.89
...	...	...	...
Gene_m	2.32	...	3.78

The query workflow is as follows:

1. Select patients with some disease (e.g. cancer)
2. Join the selected patients with the microarray table
3. Compute the covariance between the expression levels of all pairs of genes
4. For all pairs of genes with covariance greater than a threshold (e.g. top 10%), join the results with the gene_metadata table to obtain gene information for further analysis.

3.2.3 Query 3: Biclustering

An important goal of genomic analyses is to identify groups of genes and patients that show similar behavior. Genes with similar behavior are likely to be biologically related, and therefore can help understand disease pathways and identify new drug targets. One way to identify genes with similar behavior is via biclustering. Biclustering allows the simultaneous clustering of rows and columns of a matrix into sub-matrices with similar patterns. For instance, biclustering a microarray matrix would cluster together all the instances of genes and patients with expression values less than normal, as shown in bold below.

	Gene_1	Gene_9	Gene_13	...	Gene_n
Patient_5	0.564	**0.005**	**0.001**	...	0.988
Patient_12	0.113	**0.003**	**0.0009**	...	0.185
...	...	...	...	...	...
Patient_m	0.655	0.008	0.556	...	0.340

The query structure is as follows:

1. Select patients with specific age and gender (e.g. male patients less than 40 years old)
2. Join the results with the microarray data on patient_id
3. Restructure the result into a matrix if required
4. Run the biclustering algorithm on the matrix

3.2.4 Query 4: SVD

As with much experimentally collected data, genomic data is quite noisy. In order to perform various analyses, e.g. comparisons between gene expressions across different diseases, we need to reduce the noise in the experimental data. A popular approach for performing this task is through the use of singular value decomposition (SVD) [13]. SVD is a factorization of a matrix M as:

$$M = U\Sigma V$$

where Σ is a diagonal matrix containing singular values, and U and V respectively contain the left and right-singular vectors. The top singular values in Σ represent the signal in the data. For the benchmark, we use the Lanczos algorithm, which is a power method that can iteratively find the largest eigenvalues of symmetric positive semidefinite matrices.

The query is structured as follows:

1. Select genes with specific functions (since functions are encoded numerically, select, for example, genes with function < 250)

2. Join the microarray_data table with the gene_metadata table on gene_id

3. Restructure the resulting subset of the microarray_data table as a matrix (if required)

4. Run the Lanczos SVD algorithm to find the 50 largest eigenvalues and the corresponding eigenvectors

3.2.5 Query 5: Statistical tests (enrichment)

An extremely common operation in genomic analysis is called enrichment. Consider a set of genes G that all participate in the same biological process (e.g. cell division). To determine if the set of genes G is related to a particular disease (e.g. cancer), the entire known set of genes (covering all biological functions) is ranked based on their expression values for cancer. Statistical tests are then used to find out where the members of G tend to occur in that ranking (i.e. at the top or bottom of the list [25]). If the members of G do occur at the top or bottom of the list, G is said to correlate with cancer and merits closer analysis for potential drug targets.

The query described below replicates the enrichment operation using GO ontology data. The Wilcoxon Rank-Sum statistical test is used to determine if a gene set ranks at the top or bottom of the ranked list.

1. Select a subset of samples from the microarray data (e.g. 0.25% of patients)

2. Join the results of the select query with the gene_ontology table based on gene_id

3. For each GO term g, separate the genes based on whether they belong to the GO term or not (value is 1 vs. 0)

4. Perform the Wilcoxon test to determine if the genes belonging to term g occur at the top or bottom of the ranked set of genes.

All of the code for all of the systems tested along with the various data sets appears on our website [7]. Our code (except for hardware-dependent configurations) can be run on a cluster of machines or in the cloud.

4. BENCHMARK RESULTS

In this section we describe the single-node and multi-node systems that we tested, and then present results for each of them.

4.1 Single-Node Systems Tested

The first system we evaluated was R [10]. This popular statistics package implements a variety of linear algebra operations on arrays. The current released version (3.0.2 as of this writing) assumes data is main memory resident and has a hard limit of $2^{31} - 1$ cells in an array. It also runs single threaded on one core, regardless of the number of CPUs in our test configuration. For linear algebra, R uses BLAS [5] and LAPACK, a state-of-the-art linear algebra package. It contains a join operation (called merge) that uses a hash join algorithm. One would expect the main issues with this solution would be scalability and its inability to operate on multi-node hardware. As such, there are no multi-node results for this configuration. In Figures 1 and 2 we refer to this configuration as *Vanilla R* in order to distinguish it from other configurations that combine R with separate data management systems.

The second system we tested is Postgres [9]. It is a conventional RDBMS, which will happily execute the data management portion of the benchmark. However, to execute the linear algebra, we need to augment Postgres with an analytics add-on. We tested two different approaches. The first one, which we refer to as *Postgres + Madlib*, uses Madlib [8] for the linear algebra operations that it supports. Madlib augments Postgres with UDFs for linear algebra and a limited set of other analytic functions. Some of the UDFs are written in C++, while some of them use a combination of plpython and SQL. As such, Postgres + Madlib will be able to perform only part of our benchmark. To execute the entire benchmark, we added *Postgres + R*. Here, the data management was performed in Postgres and then the data was exported and reformatted so the linear algebra could be performed in R. In effect, this is our first candidate system, with Postgres doing the data management instead of R. Of course, this system has no integration between the two subsystems, and a human must do the heavy lifting in this area. Moreover, the data will have to be reformatted and copied between the two systems, which will be costly. Lastly, Postgres does not have multi-node support, so there will be no multi-node results for Postgres + Madlib and Postgres + R configurations.

The third system we tested is a popular column store DBMS. It, of course, can execute the data management operations, but must be augmented with linear algebra support. *Column store + R* uses the same architecture as Postgres + R, exporting data to R for the analytics. The other column store configuration, which we term *Column store + UDFs*, combines the column store with user-defined functions inside the DBMS. These functions are also implemented in R. Hence, this system tests the desirability of a column store for our benchmark, as well as the benefits of calling R from inside the DBMS. This system runs both single- and multi-node.

Our next configuration uses *SciDB* [12]. This is a native array DBMS that will store all of the data in our benchmark as arrays and not as tables. Use of this system will test whether a native array system will outperform an RDBMS simulating arrays. It supports some of the linear algebra operations directly, relying on ScaLAPACK for the remainder. Like the column store, this system runs both single- and multi-node.

Our final configuration is *Hadoop* [4], which has substantial marketing buzz as a good solution for many analytic functions. We coded the data management in Hive [2], and analytics operations in Mahout [3]. Mahout executes in the MapReduce framework and does not benefit from a sophisticated linear algebra package, such as BLAS or ScaLAPACK. As such, performance may be quite poor. Lastly, with this configuration we can only run the portion of the benchmark that is possible in Mahout.

We are aware of many other systems that are capable of running GenBase, but with limited time and resources, we tried to choose a representative sample of many different classes of systems. We hope others will use our code [7] as a starting point to implement GenBase on other systems of interest and report the results.

4.2 Multi-Node Systems Tested

Of the systems discussed in Section 4.1, SciDB, Hadoop and the column store run multi-node, so we tested these three across clusters of size 2 and 4. In addition, we tested *pbdR* [23], a set of packages that extend R to run on a cluster and allow some of the analytic functions to call the matrix package ScaLAPACK [11]. These functions reformat and export data to ScaLAPACK, ingesting the returned answer. ScaLAPACK is a multi-node parallel version of LAPACK and BLAS. For pbdR, we evenly partitioned the data be-

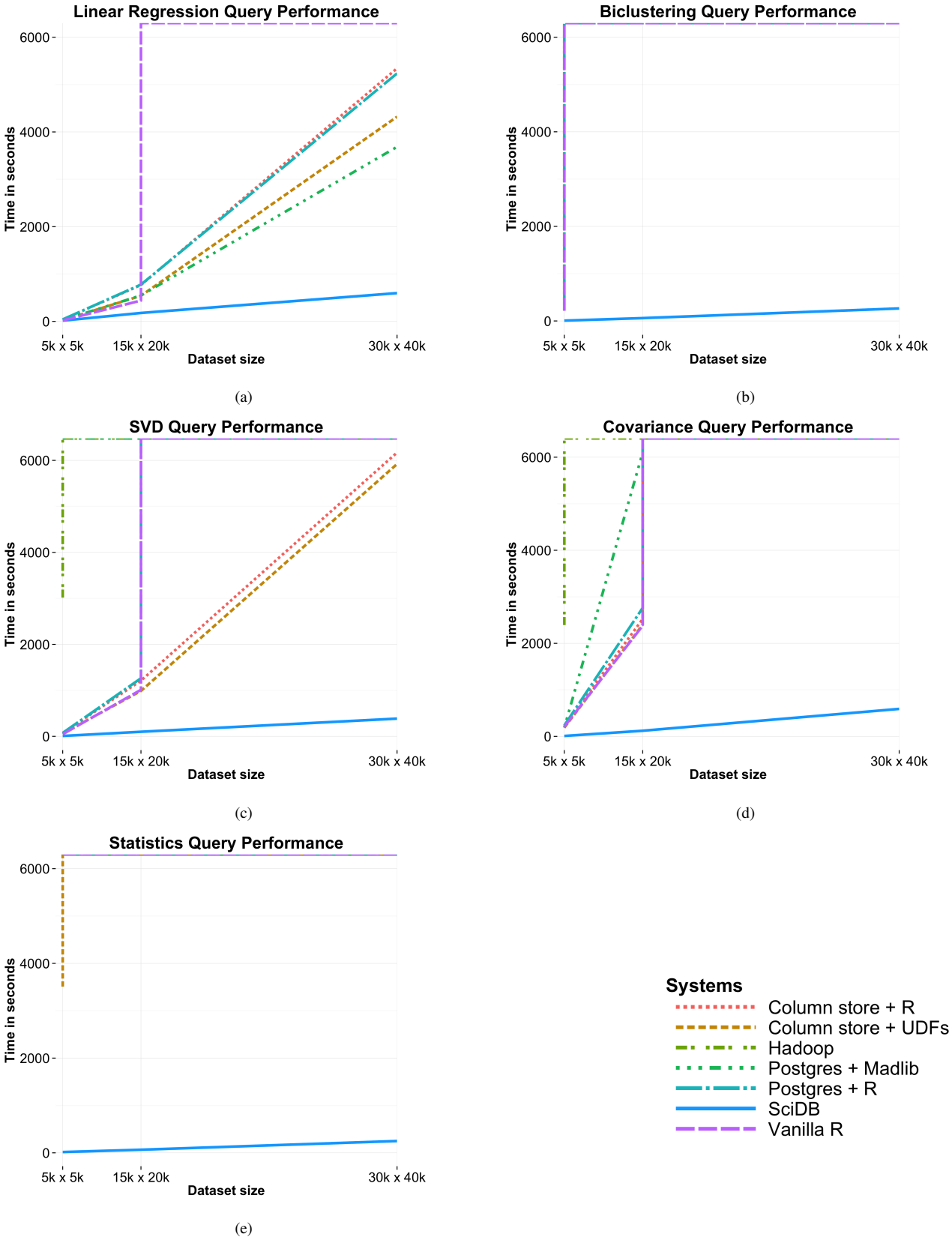

Figure 1: Overall performance of the various systems

tween nodes to allow the system to scale to the large dataset, and performed data management operations by combining local filters and joins on each node with MPI send and receive calls between nodes (built into pbdR). We also ran our column store augmented with pbdR for parallel analytics.

The next section presents single-node results for our benchmark, followed in Section 4.4 by multi-node results. In each chart, we plot results for all systems capable of running the given benchmark query. Therefore, some plots do not show results for systems in which the required functionality is missing. For example, Hadoop and Postgres + Madlib do not provide sufficient analytics functions to run the biclustering query. For all systems capable of running each query, we cut off all computation after two hours, as some of the systems would otherwise compute for a very long time. Also, in many systems, temporary space allocation failed on the large data sizes. We treat memory allocation failure and excessive computation length as "infinite" results, and indicate this with horizontal lines across the top of the charts. For all tests that completed successfully within the two-hour window, we ran each at least twice, and three times where practical (e.g., for run times less than one hour). Numbers presented in the figures below are an average over all runs.

Note that for each system tested, data was loaded from disk in a format optimal for that system. For example, Postgres data was stored in Postgres database tables, while R data was stored in .Rdata binary files. Time to convert the original .csv files to optimal system format is not included in our analysis.

Each of the four machines in our cluster had Intel® Xeon® E5-2620 processors with 2-sockets of 6 cores each and 48 GB of RAM. Each machine had 6 2-TB disks, configured as 3 4-TB virtual disks (RAID 0).

4.3 Single Node Results

Figures 1a through 1e plot the performance of the 7 systems tested on the three data set sizes for each of the benchmark tasks. In all cases, the y-axis measures elapsed (wall clock) time in seconds to complete the benchmark task. Figure 2 follows by breaking down the data management and analytics portions of the benchmark tasks separately for a sample task (regression). This breakdown is not available for Postgres + Madlib, so only results from the other systems are presented.

First, note the values for Vanilla R and Postgres + R. In general, R alone will perform well on small datasets, but cannot scale to the large dataset. The addition of Postgres as a backend generally increases total query time because of the cost of reformatting and exporting the data as well as the overhead of DBMS features. However, as data sets get larger (e.g., on the 30K x 40K data set), it is sometimes beneficial, for example in regression, to have a data management backend as R by itself cannot load the data into memory.

Now consider Postgres + Madlib. This configuration executes four of the five tasks, but only two within the 2 hour window. Compared to Postgres + R, it saves the cost of moving/reformatting data between systems, but in some cases it will have a less efficient execution of the analytics. The Madlib analytic functions written in C++, such as linear regression, tend to be faster than the corresponding functions in R. The others, such as SVD, in effect simulate matrix computations in SQL and plpython, rather than performing them natively. Hence, Postgres + Madlib is sometimes better and sometimes worse than Postgres + R. Of course, both have better scalability properties than R alone.

Moving on to the popular column store + R, we should note that this system has very similar performance to Postgres + R. The

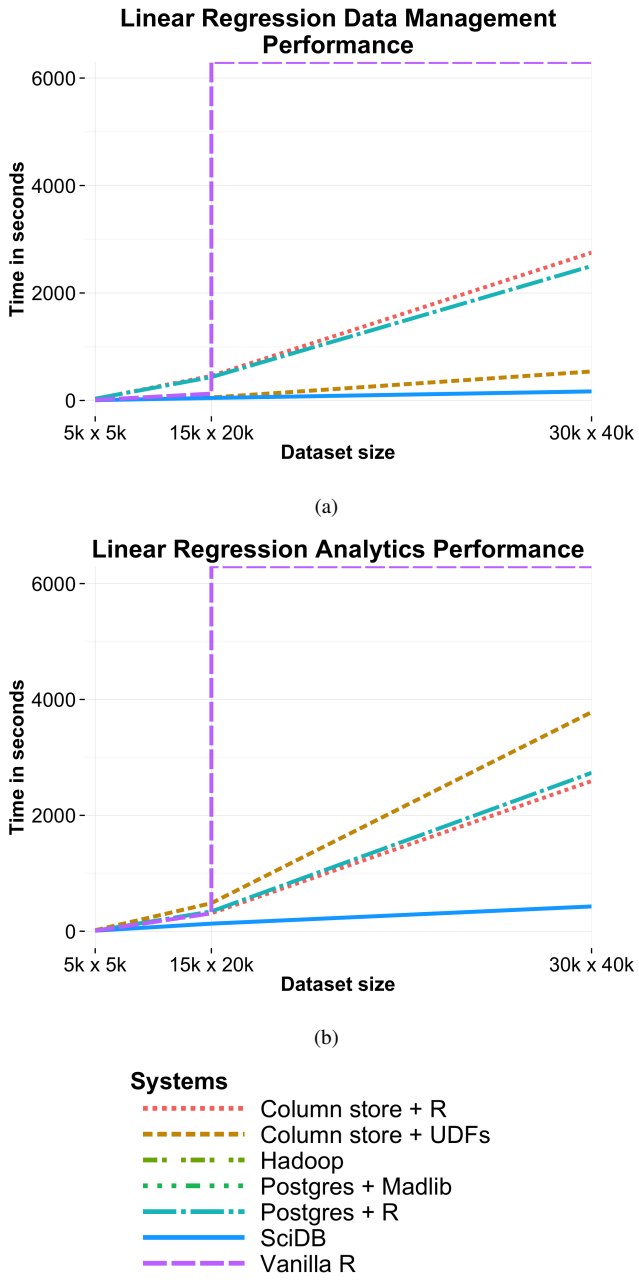

Figure 2: Data management and analytics performance of the various systems

conventional wisdom is that column stores beat row stores on data warehouse-style benchmarks. However, our tables are very narrow and we retrieve several columns in some of our tasks, a situation where column stores do not excel. Moving the analytics inside the DBMS as user-defined functions should always improve performance, and as expected, the column store + UDFs configuration generally performs slightly better than the column store + external R. This improvement is presumably due to the tighter coupling between the column store and R in the UDF interface. However, there seem to be some issues with this interface as there are a few situations, such as the biclustering query, in which the column store + UDFs configuration performs significantly worse.

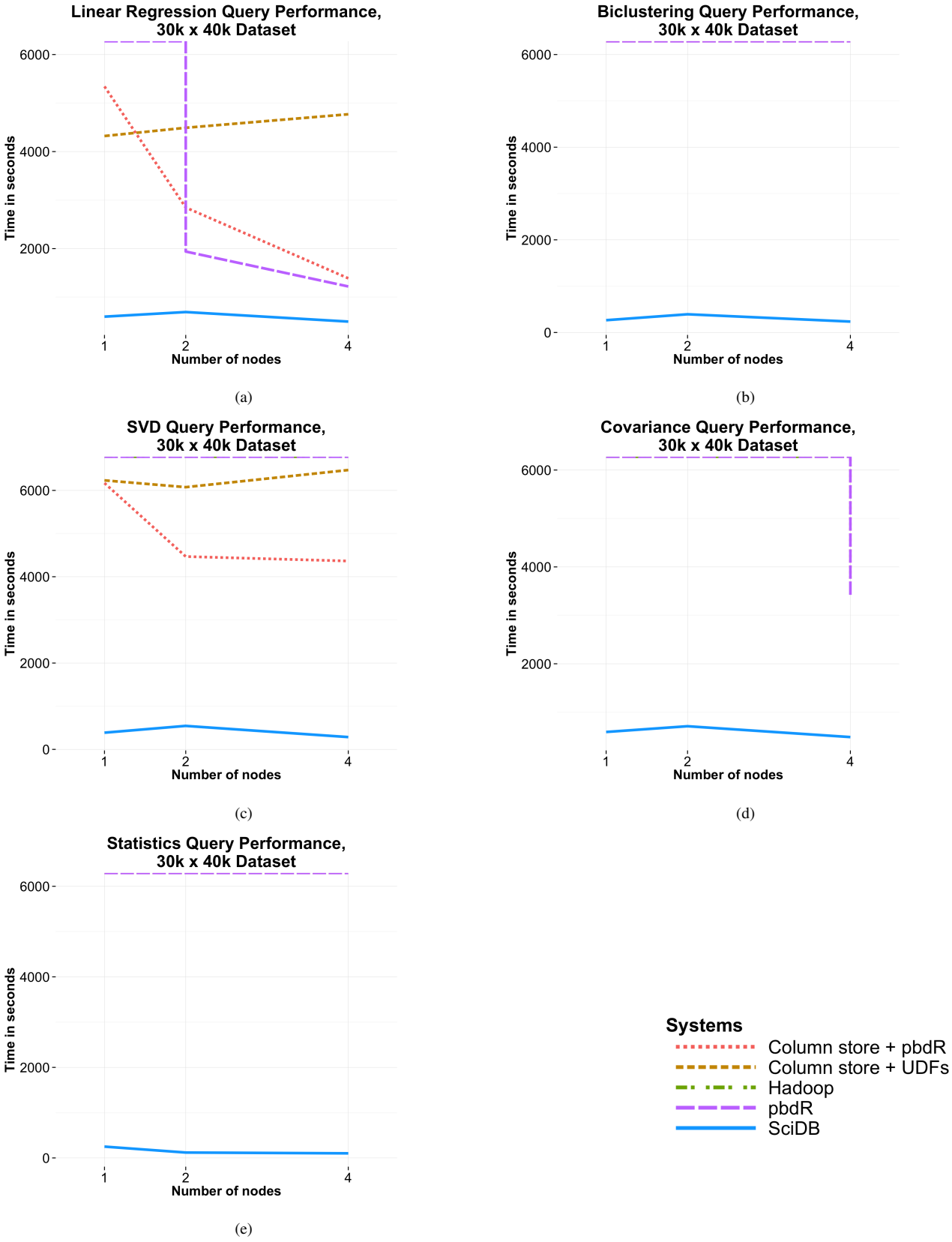

Figure 3: Overall performance of the various systems, varying num. nodes.

As you would expect, an array DBMS like SciDB is very competitive on this benchmark, since there is no need to recast tables to arrays and no data copying to an external system. Also, note that it performs some of the analytics tasks (Wilcoxon rank-sum test, biclustering) much faster than R. This is due to its use of custom code (that is more involved that just calling pre-existing ScaLAPACK routines). In general, there is a huge benefit from using high performance analytics.

Lastly, Hadoop is good at neither data management nor analytics. Data management is slow because Hive has only rudimentary query optimization and analytics are slow because matrix operations are not done through a high performance linear algebra package. In general, Hadoop runs only a subset of the tasks (SVD and Covariance) and offers between one and two orders of magnitude worse performance than the best system.

In most cases, the benchmarks are super-linear in the size of the data for all systems. SciDB appears to be approximately linear, but the plots for all other systems rise sharply as problem size is increased. In Figure 2a, we break out the data management time for the regression benchmark. Notice that all curves rise approximately linearly as problem size increases, but the slope varies considerably from system to system. Figure 2b shows a comparable plot for the analytics portion of the regression benchmark, and we note similar scalability.

Also, note that for most of the systems, analytics tends to increase with size faster than data management. Hence, many of the tasks spend the majority of their time in data management when problem size is small. However, as the problem size gets larger, the fraction of the time spent on analytics increases. To be competitive over all problem sizes, a system must be good at both kinds of operations and scale to problems that are larger than main memory.

4.4 Multi-Node Results

Figure 3 shows the performance of the various multi-node systems as we increase the number of nodes from one to four. To economize space, we present results only for the large data set. Again we break down the time into data management and analytics for the regression task in Figure 4.

Note that the scalability of all systems is less than ideal. If there is no locality between the data and computation, then scaling issues are almost guaranteed. SciDB often has worse performance on two nodes than on one, perhaps due to the increased movement of data when one goes from one node to two. In addition, no systems offered linear speedup between 2 and 4 nodes. Lastly, regression was the only task that all systems could finish within the allotted time for 2- and 4-node clusters. Even when we break out data management separately from analytics in Figure 4, we see a suboptimal scaling. Somewhat surprisingly, pbdR scales the best of the systems, because when moving to multiple nodes it is able to employ ScaLAPACK's parallelizing techniques to optimize analytics. Because we are testing specialized hardware as reported in the next section, we could only assemble a 4 node configuration.

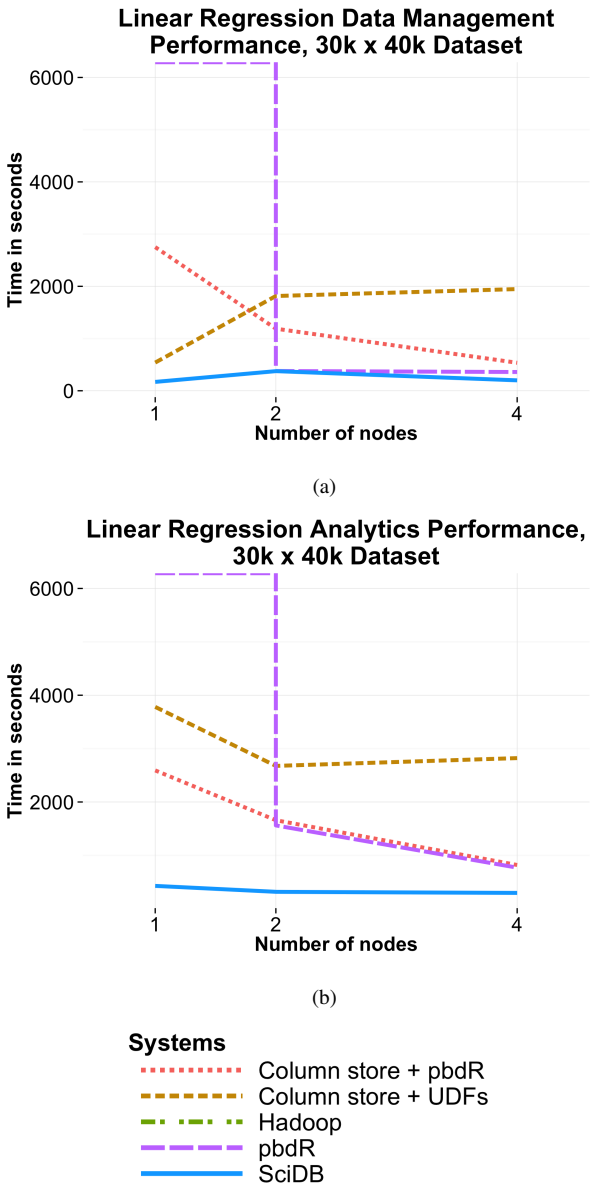

Figure 4: Data management and analytics performance of the various systems, varying num. nodes.

5. HARDWARE ACCELERATION

5.1 System Tested

We tested the recently announced Intel® Xeon Phi™ coprocessor from Intel. We use the Intel® Xeon Phi™ 5110P coprocessor that has 60 cores on a single chip with 8 GB of on-board memory and provides large computational bandwidth. We performed data management on SciDB, as in the previous section. The linear algebra operations are performed with routines specific to the Intel Xeon Phi coprocessor that are a mix of ScaLAPACK and custom code. Due to the current model of Intel Xeon Phi coprocessor, data must be copied into the memory of the Intel Xeon Phi coprocessor before it is operated on. Obviously, there is setup overhead to load the data. Hence, this system will show the acceleration of a state-of-the-art coprocessor, but only if the arrays are large enough to overcome the setup time. Since the percentage of analytics as a fraction of overall runtime goes up as data set size increases, this should be especially attractive on larger data configurations. At the same time, the memory available on the coprocessor is limited, and data sets that do not fit in this memory will suffer excessive data movement costs during computation. In our experiments, we find that the large data set can fit in the memory of a single Intel Xeon Phi coprocessor, and hence we show results for up to the large configuration. We compare these results with the Intel® Xeon® E5-

2620 system in the previous section. Future coprocessors will not have these limitations [20].

5.2 Benchmark Results

Figure 5 shows single-node results[2][3], for SciDB with an Intel Xeon Phi coprocessor compared to the execution on the Intel Xeon system previously described as problem size is increased. We implement the various operations using a combination of ScaLAPACK calls to the Intel® Math Kernel Library (Intel® MKL) [15] and custom code. We use an internal release of Intel® MKL11.1.x to allow automatic offloading of the ScaLAPACK pdgemm routine (used in covariance) to the coprocessor without any code changes. Linear regression is also available as a ScaLAPACK operator, but the Intel® MKL automatic offload of this operation is currently not fully supported and is a work-in-progress. Hence we do not show linear regression results here.

For the large data sets, the Intel Xeon Phi coprocessor-based system is faster than the Intel Xeon system by a problem-specific factor of up to 1.7X. For small data sets, the runtimes of the operations are small enough that memory management and data transfer overheads to the Intel Xeon Phi coprocessor dominate overall runtime and lead to minor fluctuations in performance. If we consider just the analytics measurements, the Intel Xeon Phi coprocessor-based system performs about 1.4-2.6X better than our Intel Xeon system in three of the four operations considered: covariance, SVD and statistics for the medium and large data sets. This speedup in analytics time is expected due to the computation and bandwidth resources available on the respective systems. The only exception is biclustering, which takes very little computation time and cannot be expected to show significant speedup on any accelerator.

These performance benefits clearly depend on the fraction of time spent in the analytics portion of each task. Among the five tasks this varies from almost all of the time (statistics task) to very little (biclustering). Hence, the advantage of specialized hardware depends on data set sizes and the fraction of the time spent on analytics. Of course, these tradeoffs will change considerably if an accelerator shares memory with the main processor, rather than being accessed through a bus. In summary, a task-specific co-processor is definitely worth including in a hardware configuration.

Table 1 shows the speedups of the analytics portions of the benchmark on the Intel Xeon Phi coprocessor-based system versus the Intel Xeon based system as number of nodes is increased from one to four on the large data set. For our multi-node runs, an additional bottleneck – the time spent in inter-node communication – can limit performance on certain operations. Wherever the time spent in actual computation is large enough, however, we still see benefits from using the Intel Xeon Phi coprocessors. These speedups, however, will be lower than on single node systems due to the communication time. The best improvements come for the large data set for 1 node, which gives the maximum amount of data per node. For this configuration, the Intel Xeon Phi coprocessor provides about 1.4-2.3X performance improvement (except for biclustering, which spends too little time in analytics for any coprocessor to accelerate significantly). In terms of overall time, the speedups are problem specific with speedups up to 1.5X with an average of around 1.3X. When we run the same data set on 4 nodes, the per-node computation falls, and hence the improvement due to the coprocessor is less pronounced. However, in reality, the genomics data should scale in size with the number of nodes in the cluster ("weak scaling").

Benchmarks	1 node	2 nodes	4 nodes
Covariance	2.60	1.55	1.54
SVD	2.93	2.30	1.37
Statistics	1.40	1.43	1.21
Biclustering	1.18	1.05	1.02

Table 1: Analytics speedup of the Intel® Xeon Phi™ 5110P coprocessor-based system versus an Intel® Xeon® E5 based system on SciDB+Scalapack

6. DISCUSSION

The above experiments raise a number of discussion points.

6.1 End-to-end Issues

In order to perform well on our benchmark, a platform must be good at both data management and analytics (specifically linear algebra). Our results show that systems that do only one of these tasks well will fail badly.

Specifically, if N is the size of the table holding the microarray information, then data management tasks are either $\mathcal{O}(N)$ or $\mathcal{O}(NlogN)$. Treating this table as a dense array yields a 2-D array with each dimension of size $N^{\frac{1}{2}}$. Hence analytics that are cubic in array dimension size run in $\mathcal{O}(N^{\frac{3}{2}})$. At a smaller scale, data management will dominate, because of the size of the constant terms and cost of I/O to fetch data, while at large scale analytics will quickly dominate due to asymptotic effects. Again to deal well with scale, any platform must be efficient at both kinds of tasks. Lastly, at large scale one must run in parallel over a collection of cores on multiple nodes, to avoid the super-linear decrease in performance as data sets grow. Any single-node system will fail badly.

This benchmark is designed to be representative of a broad class of applications that ingest and filter large quantities of data and then perform complex (quadratic or super-quadratic) analytics on it. Similar tasks arise in other sciences (e.g., astronomy), in social network analysis of large graphs, and in online settings like collaborative filtering and recommendation settings. These domains will show the same tradeoffs in terms of platform choice as in the genetics domain.

6.2 Analytics

BLAS/LAPACK/ScaLAPACK are known to be amongst the most efficient linear algebra packages, as they have been tuned over the years to be highly efficient. Similarly, processor-specific libraries such as the Intel® MKL libraries for Intel Xeon and Intel Xeon Phi processors are further tuned to take advantage of hardware features such as cache hierarchies and vectorization of operations. Without the same level of tuning, new versions of these routines will be much slower. Hence, the general wisdom is to use the existing codes rather than build new ones. This approach is being followed by most DBMSs that we are aware of.

However, use of these packages introduces a serious issue. Namely, DBMSs have their own formatting conventions for disk-based data. Tabular row stores invariably store relational tuples in highly encoded form on storage blocks. Column stores encode disk blocks in a different way for efficiency on their use cases. Similarly, SciDB

[2] Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance

[3] Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

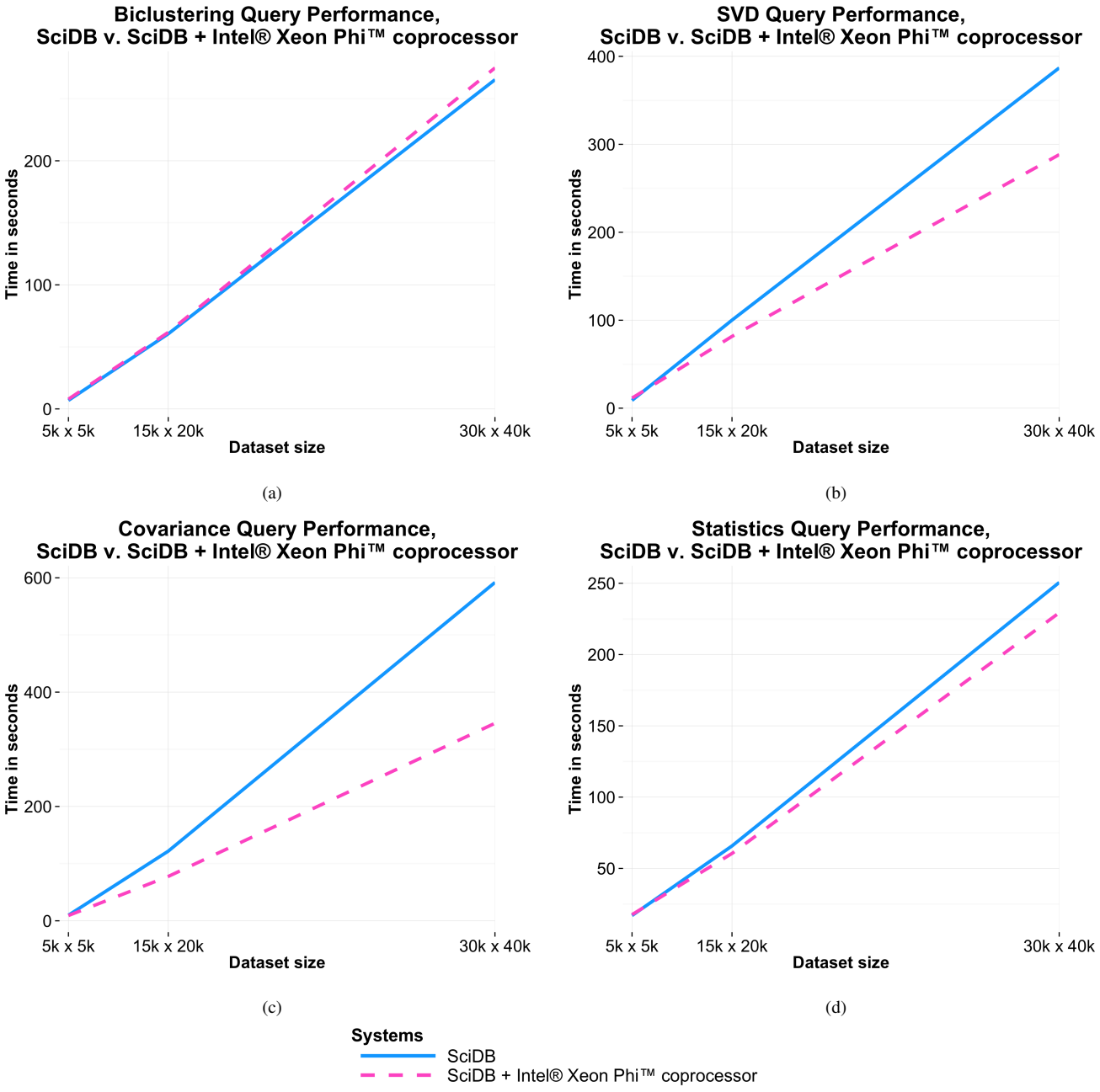

Figure 5: Overall performance of SciDB and SciDB + Intel® Xeon Phi™ coprocessor

heavily encodes rectangular "chunks" of data onto disk blocks. Moreover, SciDB and other platforms utilize range or hash-based partitioning schemes to allocate objects to compute nodes. Lastly, SciDB chunks are rather large, typically in the Mbyte range. In all cases, DBMSs employ a custom formatting scheme for storage of blocks, and carefully select both block size and partitioning strategy to maximize performance.

In contrast, ScaLAPACK operates on data arranged in a block-cyclic layout over a collection of nodes. Moreover, ScaLAPACK chunks are quite small, i.e. Kbytes, and are stored unencoded, so they can be unpacked and operated on easily.

Hence, there are good reasons for the DBMS and the linear algebra package to choose different storage formats. As such it is an $\mathcal{O}(N)$ operation to convert from one representation to the other. Since the constant is fairly large, this conversion can dominate computation time if the arrays are small to medium size.

As a consequence, we see two different analytics scenarios. When the arrays are large, it will pay to convert the data to ScaLAPACK format. With smaller arrays, an in-DBMS suite of analytics should be used to avoid the conversion cost. Of course, such an approach introduces a number of complexities: two codebases have to be maintained, and those codebases can produce inconsistent answers, particularly with respect to numerical stability and roundoff errors.

6.3 Algorithms

It is, of course, essential to have high-speed implementations of basic array operations. More important is the choice of algorithm. First, algorithms differ in their accuracy (approximate versus exact) and in their error bars (preciseness). Also, there are usually a substantial number of ways to compute any given quantity. A good algorithm can make a huge difference in the performance of any computation. Particularly for many matrix factorization and statistical optimization problems, there exist efficient approximate algorithms that parallelize well. It is likely that such algorithms will be critically important as dataset sizes grow – for example, in our benchmark, approximation algorithms may have allowed us to scale to the 60K x 70K dataset that none of the systems we tested could process in under two hours.

7. CONCLUSIONS

The purpose of this benchmark is to draw community attention to the needs of data scientists, namely for high performance, scalable data management and analytics. The results presented in this paper show that real-world systems have lots of room for improvement. We can easily conclude that only systems that are good at both kinds of tasks will excel at this benchmark. Moreover, some of the configurations we tested required "glue" code to copy/reformat data back and forth between multiple systems, and required several hours of programmer effort.

All of the code and data is available on our web site, noted earlier, and we hope other systems will try out this benchmark.

8. REFERENCES

[1] Affymetrix. http://www.affymetrix.com.

[2] Apache hive(tm). http://hive.apache.org.

[3] Apache mahout: Scalable machine learning and data mining. http://mahout.apache.org.

[4] Apache (tm) hadoop (r). http://hadoop.apache.org.

[5] Blas (basic linear algebra subprograms). http://www.netlib.org/blas/.

[6] Dna sequencing costs. http://www.genome.gov/sequencingcosts.

[7] Genbase. http://web.mit.edu/~mvartak/www/genmark.html.

[8] Madlib. http://madlib.net.

[9] Postgresql. http://www.postgresql.org.

[10] The r project for statistical computing. http://www.r-project.org.

[11] Scalapack - scalable linear algebra package. http://www.netlib.org/scalapack/.

[12] Scidb. http://www.scidb.org.

[13] Singular value decomposition for genome-wide expression data processing and modeling. www.pnas.org/content/97/18/10101.abstract.

[14] Tpc transaction processing performance council. www.tpc.org.

[15] Intel math kernel library (intel mkl) 11.1, 2013.

[16] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 480–491. VLDB Endowment, 2004.

[17] D. Bitton, C. Turbyfill, D. J. Dewitt, and D. J. Dewitt. Benchmarking database systems: A systematic approach. pages 8–19, 1983.

[18] M. J. Carey, D. J. DeWitt, J. F. Naughton, M. Asgarian, P. Brown, J. E. Gehrke, and D. N. Shah. The bucky object-relational benchmark. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, SIGMOD '97, pages 135–146, New York, NY, USA, 1997. ACM.

[19] M. Clamp, B. Fry, M. Kamal, X. Xie, J. Cuff, M. F. Lin, and E. S. Lander. Distinguishing protein-coding and noncoding genes in the human genome. *Proc. National Academy of Sciences of the United States of America*, 104(49):19428–33, 2007.

[20] R. Hazra. Driving industrial innovation on the path to exascale: From vision to reality, 2013. http://newsroom.intel.com/servlet/JiveServlet/download/6314-25051/Intel_ISC13_keynote_by_Raj_Hazra.pdf.

[21] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The madlib analytics library: or mad skills, the sql. *Proc. VLDB Endow.*, 5(12):1700–1711, Aug. 2012.

[22] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Designing fast architecture-sensitive tree search on modern multicore/many-core processors. *ACM Trans. Database Syst.*, 36(4):22, 2011.

[23] G. Ostrouchov, W.-C. Chen, D. Schmidt, and P. Patel. Programming with big data in r, 2012. http://r-pbd.org/.

[24] M. Schatz and B. Langmead. The dna data deluge. *IEEE Spectrum*, 50(7):28 – 33, 2013.

[25] A. Subramanian, P. Tamayo, V. K. Mootha, S. Mukherjee, B. L. Ebert, M. A. Gillette, A. Paulovich, S. L. Pomeroy, T. R. Golub, E. S. Lander, and J. P. Mesirov. Gene set enrichment analysis: A knowledge-based approach for interpreting genome-wide expression profiles. *Proc. National Academy of Sciences of the United States of America*, 102(43):15545–50, 2005.