A Big Data Platform Integrating Compressed Linear Algebra with Columnar Databases

Vishnu Gowda Harish, Vinay Kumar Bingi, John A. Miller
Department of Computer Science
University of Georgia
Athens, GA, USA
{vishnu.gowdahari25@, vinaykuma.bingi25@, jam@cs.}uga.edu

***Abstract*—Key foundational components of Big Data frameworks include efficient large-scale storage and high-performance linear algebra. This paper discusses efficient implementations that utilize compression techniques inspired by columnar relational databases for improving space and time profiles for vector and matrix operations. In addition, linear algebra operations are integrated with columnar relational algebra operations both in dense and compressed forms. For several of the operations substantial speedups are obtained by operating directly on the compressed relations, vectors and matrices. Advantages of mixing and matching relational and linear algebra operations are also pointed out. Both serial and parallel implementations are provided in the ScalaTion Big Data Analytics Framework.**

***Index Terms*—Data analysis; data compression, data mining, linear algebra, parallel programming;**

I. Introduction

Data is growing at a rapid rate and there is an increasing demand for efficient storage and faster analytics. Due to this ever-increasing amount of data, big data frameworks are exploiting parallel and distributed processing and integration of databases with computational frameworks as well as novel approaches to data storage and computational algorithms. Compression techniques used in columnar relational databases have been used to speed up Online Analytical Processing (OLAP) operations. In this paper, we extend this work to more advanced operations on vectors and matrices.

Data compression can be very helpful in meeting these goals. The obvious advantage of compression is the savings in storage space. Besides that, it can also be used to improve analytical performance. In traditional disk-based systems where disk I/O performance has been a pinch point, compression is used to reduce the size of data traveling to and from disk and improve the overall performance of the system. However, the availability of a large amount of main memory at low prices has paved the way for in-memory platforms to do analytics. Here the data are stored in main memory, and disk is used for persistence and recovery. Thus the disk I/O performance is less of a concern for modern in-memory systems. In these platforms, compression techniques are used in in-memory to reduce the memory footprint and also to speed up performance by operating directly on compressed data.

Compression is widely used by big data platforms that adopt a column-wise approach to store data on disk or memory [1]–[5]. It results in efficient compression because in general there is greater repetition in columns than in rows. Some queries when working on huge data sets require the output to only contain a few columns. In these kinds of queries, row-wise storage exhibits poor performance. Since the data are stored in rows, all the columns have to be read but it is not the case with column storage.

Once data is compressed, we need to take into account the time spent on decompressing during the execution of the query. This approach tends to exhibit poor performance as decompression can take a substantial amount of time when the data sets are huge. However, with certain compression techniques, we can operate directly on compressed data without having to decompress [1]. This leads to a performance gain as executing analytical workloads directly on compressed data will be faster compared to that of raw data, since the size of data in contention is lesser. Run length encoding and dictionary encoding are examples of such compression techniques.

Compression further improves the performance as it exhibits better cache utilization. Cache is a small memory area that lies between the processors and main memory. It stores data that are frequently accessed from main memory. Good utilization of cache memory can help improve performance. Fetching data from the cache is faster than that from main memory. The data requested is first checked to see if it is available in the cache. If data is not present in the cache, it is fetched from main memory (cache miss). The performance of in-memory algorithms tends to be better with fewer cache misses. With compression, more content is packed in each cache line which reduces the number of cache misses. The same is illustrated in [3].

Previously only certain types of analytics (mainly descriptive) were performed on top of the databases and data scientists had to rely on other tools like R, SAS to perform advanced statistical learning. The data set sizes may be huge (up to terabytes) and the procedure of taking these data to the computation side turns out to be very slow [6]. Therefore, sampling on the database has to be performed and a subset of actual data has to be extracted to the statistical packages, which would result in loss of detail and affect prediction accuracy. Also, another point to note is that statistical package and database running on the same node leads to performance degradation as both are trying to acquire the same computing resources

[6]. To mitigate these problems, the database community has taken a couple of approaches. For example, in HP Vertica, a fast parallel transfer mechanism exists for moving data to distributed R [6]. Initially, only metadata information is transferred to R which will be used for subsequent data transfers. Post the metadata transfer, User Defined Functions (UDFs) are used for data transfer. Each UDF transfers a certain segment of the table which is partitioned based on the number of R instances available. UDFs do not read the table content from disks as it uses memory buffers to store the content. There has also been work trying to embed packages like R into data stores and executed via UDFs. However, this workflow is single threaded [7].

An alternative approach is to provide the necessary linear algebra foundation at the database level itself. Linear algebra forms the basis of statistical and analytical computations and providing support of fundamental datum objects like vectors and matrices in the database opens up the possibility of computations at the database itself. By adopting this approach, we eliminate the need for data transfers to an external statistical system and avoid duplication of data across disparate systems. Matrices and vectors can also be manipulated (read, updated and deleted) in the same way as database tables and columns.

The amount of sparse data being generated today is huge mainly due to the contribution of various applications. As illustrated in [8], analytical platforms like column stores are well equipped to handle the sparse nature of data. The wideness of data is not a concern as only the columns required are read and processed. Sparseness can be handled as we can employ different null compression techniques for each column based on its sparsity level. Another important observation is that sparse data might contain columns that are highly sparse and might not contribute to the analytics outcome. As illustrated in [9], in certain datasets, we see that the size and variables run into millions. The design matrix for such data is huge and results in reduced computational efficiency while applying analytic techniques like regression. Dimensionality reduction techniques like min-wise hashing can be used to trim the design matrix [9]. Regression on this reduced matrix increases the computational efficiency without compromising on accuracy. Compression along with the dimensionality reduction can be an effective tool for improving performance.

The rest of this paper is organized as follows: Section II provides a brief overview of the recent work in integrating linear algebra in databases and the use of compression in column stores. In Section III we dig deeper into how compressed linear algebra is integrated into columnar databases provided in ScalaTion [10], a platform for analytics and simulation written in Scala. We pick run length encoding as the compression technique and explain why and how it is adopted in ScalaTion. We provide a new efficient mutable data structure for supporting run length encoding and provide examples as for how random access and updates are performed. We also talk about the support of matrices and vectors as storage structures and how they can be extracted from relations, updated and then saved back as relations. In Section III we also touch on linear algebra operations that operate directly on compressed data. Discussion of how these operations can be made faster using parallelism is given in Section IV. Section V talks about how compressed linear algebra can be useful in advanced analytical applications. Section VI discusses the performance results and Section VII presents conclusions and future work.

II. Related Work

Data compression is widely used in different analytical platforms. Shark [4] is an example of such platform that provides a uniform bed for relational and analytical processing. It is built on top of Spark and includes several modifications to the out-of-box Spark engine for optimal execution of queries. An example of such modification is an additional in-memory columnar store. This enables Shark to use light-weight compression techniques like run length encoding and dictionary encoding to enhance performance and save space. Spark SQL [2], similar to Shark but with additional capabilities, also uses compression in its in-memory columnar cache. Druid [5] is another open source, distributed big data platform that leverages compression. It is mainly used for OLAP processing on real-time streaming data. Storage units known as segments are replicated across the nodes in the cluster and are stored column-wise. This helps Druid to apply efficient compression techniques to enhance performance.

Data compression is widely adopted in the NoSQL databases world mainly column stores as they tend to achieve higher compression ratio. For example, [1] shows how compression techniques are adopted in C-Store and come up with a query executor to operate directly on compressed data. It also comes up with a tree based model that helps in deciding which compression algorithm should be adopted. The model checks as to how the data are distributed in the column to come up with a decision.

Data compression is also used in in-situ (in the same place) analytics. In-situ analytics has been widely adopted by the scientific community and it involves providing simulation time analytics. Since running simulation and analytics on the same computing environment would cause performance degradation, it is advisable to perform both these tasks in parallel on independent computing resources. However, this approach comes with an I/O bottleneck. [11] shows how compression is adopted to mitigate the costs of data transfer between simulation and analysis environments. It provides insights as to where and what compression mechanism should be used during the workflow.

There has been prior work on tightly integrating analytics with data stores mainly by providing a linear algebra foundation. SciDB [12], [13] is an example of the same. It is a database that is fine tuned to the scientific users. The data model used is an Array and this is useful as the base for several advanced analytical algorithms like Regression, Classification are linear algebra computations over arrays. SciDB provides a variety of ways to interact with the data model. Array Query Language (AQL) is one of them where a user can execute linear algebra operations via a SQL-like interface.

Array Functional Language (AFL) gives a procedural interface where primitive linear algebra operations can be grouped together to provide a desirable result. Given the popularity of R, SciDB also provides a wrapper on which R programs can run and access data residing in SciDB.

[14] shows how to integrate linear algebra into an in-memory columnar database. Integrating the matrix data structure is done in a variety of ways. For example, an entire relation with m rows and n columns is considered to be a matrix (m by n). Each of the n columns of the matrix is stored separately in column storage. Another approach is where entire relation content is put into a continuous sequence of values. The entire relation can also be converted to a 3 column table containing the value, row, and column index. Each of these 3 columns is again stored in separate column containers. It also supports Compressed Row Storage (CSR) representation of matrices. In this case, it also embeds matrices as part of the DDL and offers built in DML commands to process matrices.

MAD skills [15] is another research work related to this area. Here SQL and UDFs are used to formulate linear algebra expressions. It provides a layer of C++ on top of DBMS. This layer helps in type mapping between C++ and database, better error handling and also provides the mathematical library to perform advanced learning algorithms (e.g., *k*-Means, Regression).

R language has an RLE class in its base package [16]. Using this class, the atomic vectors can be stored in run length encoding format. RLE in R stores the elements in an atomic vector as a series of pairs of a value and the length of its contiguous occurrence. The lengths and values can also be accessed as individual lists. In R, `rle()` is extended with `seqle()` where contiguous elements with a common slope or delta are compressed. This helps in encoding linear sequences.

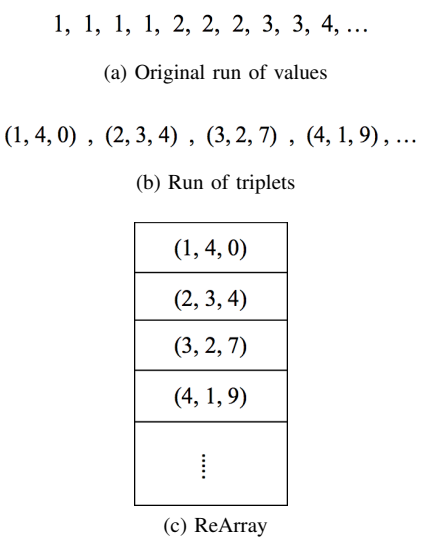
III. Integrating Compressed Linear Algebra

In this section, we see how compressed linear algebra is adopted in ScalaTion. We consider run length encoding as the compression technique and take a look at the data structures implemented to support the same. Since vectors and matrices are fundamental objects in linear algebra systems, we see how run length encoded variants of the same are supported in ScalaTion. We look at various analytical and linear algebra operations on these compressed data objects.

A. RLE Vectors

In run length encoding the original sequence of values is replaced with the value and number of times the value repeats. In ScalaTion we also record the starting position of the value along with the value and number of repetitions [1]. This information constitutes a Triplet and is represented as (*value, count, startPos*). An example of the Triplet is shown in Fig. 1. We see that the sequence of 1's in the original run of values is replaced by a triplet (*1, 4, 0*). Similarly, the sequence of 2's is replaced by (*2, 3, 4*).

Storing the values as Triplets will definitely save a substantial amount of space, but the real challenge is in making



(a) Original run of values

(b) Run of triplets

(c) ReArray

Fig. 1. Run length encoding

the operations faster. In order to speed up execution of linear algebra and analytical operations, we need an efficient mutable storage structure that provides fast random access and updates. To satisfy this requirement we use ReArray to store the triplets. ReArray is a modified version of Resizable Array implementation provided in Scala. An example of ReArray is shown in Fig.1 (c). An RLE vector is nothing but a run length encoded vector. It internally contains a ReArray to hold all the triplets corresponding to the actual values.

Random access of a triplet in an RLE Vector is very straightforward and fast as it involves just an array lookup. Fast random access to an individual value in an RLE vector is more challenging. Since the triplets are ordered based on the startPos in the ReArray we take the binary search approach to find the first triplet whose startPos is greater than the index. Once we have this triplet we just pull out the value associated with it. By taking this approach we provide random access with logarithmic complexity.

Finding the mean of values is an important analytical operation and is nothing but the sum divided by the number of values. For doing a sum operation on an RLE vector, we go through every triplet summing the product of its *value* and *count*. This approach exhibits better performance when compared to the dense counterpart where we have to go through every value in the vector. The implementation of sum (S) operation in the RLE vector is shown below.

```
def sum = v.foldLeft (0.0)
          ((s, a) => s + (a.value*a.count))
```

foldLeft is a functional combinator provided by Scala and the function it takes is applied to each element of the data structure it is called on. The data structure here is *v* and the function is summing the product of value and count of each triplet *a*.

Variance is a critical operation in statistical analysis. Given a sample, variance gives you information on how far the values

are from the mean. Let s^2 be the sample variance (unbiased) and is calculated as follows

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu)^2$$

where $(x_1, x_2, ..., x_n)$ are samples of random variable X, n is the sample size and μ is the mean. The above equation can be reduced as follows which is more efficient for computation.

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}x_i^2 - \frac{S^2}{n}$$

The implementation in ScalaTion to calculate the sample variance (unbiased) for the elements of a vector is shown below.

```
def variance = (normSq − sum ∗ sum / nd) /
               (nd − 1.0)
```

sum gives the sum of elements in the vector and nd the number of elements in the vector as a double. $normSq$ gives the Euclidean norm squared and is the dot product of the vector with itself. For calculating the dot product of two dense vectors, we simply iterate through both the vectors adding up the product of corresponding elements in both the vectors. However, in the case of RLE vector, we take advantage of triplets. Due to highly repetitive values, the number of triplets is very less compared to the actual values. Thus iterating through the triplets take lesser time than through actual values. We take the approach of a 2-way merge during the calculation of the dot product as the intervals of corresponding triplets might overlap. The more the repetitive nature in the values, the faster the dot product of RLE vector will be when compared to dense vector.

Update operation on a RLE vector is a complex operation encompassing several scenarios. [17] shows how RLE data are updated using count indexes. In ScalaTion, updating RLE data would cause the ReArray to grow, shrink or stay the same in size. *shiftLeft*, *shiftRight*, *shiftLeft2* and *shiftRight2* implementations are provided in ScalaTion (as part of ReArray) to achieve the same. If the dense vector is compressed to a RLE vector of size C, where C is the number of triplets, then the complexity of the shift operations are $O(C)$. Let us go through few of the update scenarios in detail here.

Fig.2 shows the original run of values and the initial state of the ReArray. Let us consider the case where we update index *i* = 99 to the value 20. First, the triplet that contains the index position is determined using a binary search approach. Let us call this triplet as *curr* and the one before and after as *prev* and *next*. Since the index we are trying to update is the startPos of *curr* and its count is greater than 1, we just do a check with the *prev*. We see that the new value 20 is equal to the value of *prev*. Thus the count of *curr* is decremented by 1 while the count of *prev* and startPos of *curr* is incremented by 1. This is shown in Fig.3.

Fig.4 explains one of the scenarios where ReArray grows in size. Here we update index *i* = 99 to the value 30 and

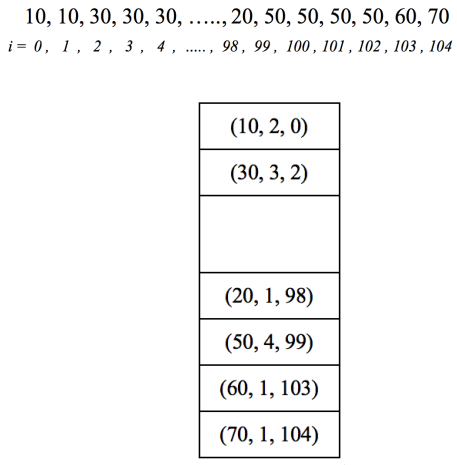


Fig. 2. Data and Initial state of Rearray

it is not equal to the value of *prev*. In this case we do a *shiftRight* where we shift every triplet right by one position starting from *curr*. This causes the ReArray to grow in size by 1 making *(50, 4, 99)* as the *next* now. After the shift, we modify *curr* (value, count) and *next* (count, startPos). The triplets that are modified and undergo a change in position due to the shift are highlighted.

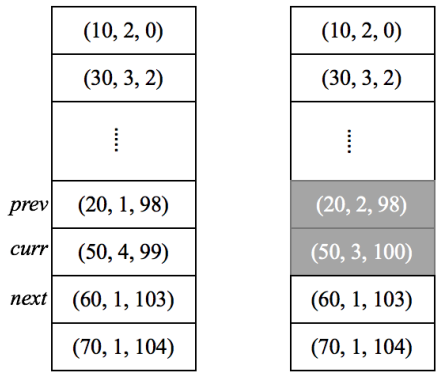


Fig. 3. update *i = 99 to 20*

When we update index *i* = 103 to 50 the ReArray shrinks by 1. *curr* is now *(60, 1, 103)*. Since the count of *curr* is 1, the new value is compared to the value of both *prev* and *next* as it can be equal to either one or both. Since it is only equal to the value of *prev* we do a *shiftLeft* on *curr*. It shrinks the ReArray by removing *curr* and making the triplet *(70, 1, 104)* as the new *curr* now. Once the shift is done we increment the count of *prev* to complete the update. Fig.5 depicts this scenario.

There exist update scenarios that cause the ReArary to grow or shrink in size by 2. *shiftRight2* and *shiftLeft2* caters to such scenarios.

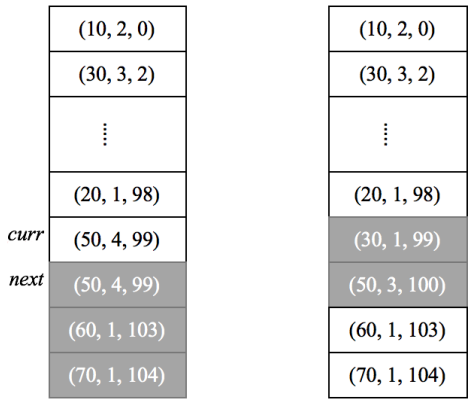


Fig. 4. update *i = 99 to 30*

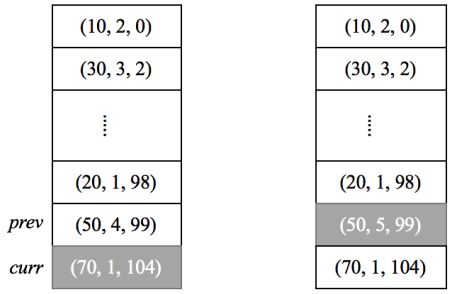


Fig. 5. update *i = 103 to 50*

B. RLE Matrices

RLE matrix contains a collection of RLE vectors that constitute a matrix. Fig.6 shows a relation and its equivalent RleMatrixD. The RleMatrixD consists of four RleVectorDs each representing a column of the relation. The RleMatrixD and RleVectorD operate on data of type double. RLE matrix internally consists of an array to hold the RLE vectors. This internal structure helps in providing constant time access to every column of a matrix

Matrix multiplication is a critical linear algebra operation used in various fields of science. Let us first take a look at the general definition of matrix multiplication. If $A = [a_{ij}]$ is an m-by-n matrix and $B = [b_{ij}]$ is an n-by-p matrix then the product AB = $[ab_{ij}]$ will be an m-by-p matrix and is defined as follows:

$$ab_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

If you observe the above equation, you see that each element in the resultant matrix $[ab_{ij}]$ is a dot product of i^{th} row and j^{th} column of A and B, respectively. In ScalaTion, we can use an alternative linear algebra operator ($mdot$) to yield the results of matrix multiplication.

$$AB = A^T \ mdot \ B$$

The same fundamental rule applies when multiplying two RLE matrices too. As seen previously, each RLE vector contained within the RLE matrix is a column vector. Creating an RLE matrix during extraction will make each RLE vector within the matrix a row vector. Then matrix multiplication is reduced to $mdot$ of two RLE matrices and the code snippet below shows the implementation of the same.

Col_1	Col_2	Col_3	Col_4
1.7	2.1	6.2	5.8
1.7	3.5	6.2	5.8
1.7	3.5	6.2	6.4
2.1	3.5	4.2	6.4
2.1	3.5	4.2	6.4
⋮	⋮	⋮	⋮
3.5	6.2	1.5	2.5
3.5	6.2	1.5	2.5

(a) Relation with 4 columns and 100 rows

$$\begin{bmatrix} RleVectorD\ ((1.7, 3, 0),\ (2.1, 2, 3), \ldots (3.5, 2, 98)) \\ RleVectorD\ ((2.1, 1, 0),\ (3.5, 4, 1), \ldots (6.2, 2, 98)) \\ RleVectorD\ ((6.2, 3, 0),\ (4.2, 2, 3), \ldots (1.5, 2, 98)) \\ RleVectorD\ ((5.8, 2, 0),\ (6.4, 3, 2), \ldots (2.5, 2, 98)) \end{bmatrix}$$

(b) RleMatrixD

Fig. 6. A Relation and its equivalent run length encoded matrix

```
def mdot(b: RleMatrixD): RleMatrixD =
{
  val vv = Array.ofDim[RleVectorD](b.dim2)
  for (j <- b.range2) {
    vv(j) = RleVectorD (
              for (i <- range2) yield
                v(i) dot b.v(j)
            ) // RleVectorD
  } // for
  new RleMatrixD (dim2, b.dim2, vv)
} // mdot
```

The result of the above dot product is another RLE matrix. It will have the number of columns equal to that of the second matrix. Thus initially we create an empty array(*vv*) of size *b.dim2* to hold the RLE vectors that constitute the resultant product matrix. *dim1* and *dim2* gives the number of rows and columns of the RLE matrix where *range2* is an ordered sequence from 0 until *b.dim2*. Since we have taken the transpose of the first relation before extracting the RLE matrix, *v(i)* gives us the i^{th} row and *b.v(j)* gives the j^{th} column RLE vectors respectively. Each column of the resultant matrix is compressed to a RleVectorD and then assigned accordingly to the array *vv*. Finally, we return the new matrix that has the same number of rows (*dim1*) as the first matrix and the same number of columns (*b.dim2*) as the second matrix.

ScalaTion also supports a slightly different variant of the dot product similar to MATLAB [18]. This dot product yields a vector as the result and is nothing but the dot product of corresponding column vectors of both the matrices. The code below shows the implementation of the same.

```
def dot (b: RleMatrixD): VectoD = {
    if (dim1 != b.dim1)
        flaw ("dot", "incompatible")
    RleVectorD(for (j <- range2) yield
        v(j) dot b.v(j))
} // dot
```

C. Interoperability of Relations and Matrices

ScalaTion has the ability to extract matrices and vectors from relations, perform operations and then transform the result back to relations. In the example below, columns 1 to 50 of sales_item1 relation (dates by states) are converted to a RleMatrixD *x* by specifying the kind as COMPRESSED. Similarly price_item1 relation is converted to a RleMatrixD *y*. *z* is the resulting vector after performing the dot of *x* and *y* matrices. This gives the revenue per state for item1. The revenue for item1 is added to the revenue relation.

```
val x = sales_item1.toMatriD (1 to 50,
                              COMPRESSED)
val y = price_item1.toMatriD (1 to 50,
                              COMPRESSED)
val z = x dot y
revenue.add (z)
```

IV. Parallelization

Scala's library contains parallel collections [19]. This inbuilt parallelism within the collection helps programmers easily embed parallelism into their code. For example, in *dot* operation on RLE matrix we use the Range collection (*range2*) for traversal in the for loop. This for loop can be made parallel by using ParRange collection as below.

```
for (j <- range2.par) yield v(j) dot b.v(j)
```

As shown above calling *par* on the sequential range will give a reference to the parallel range. The entire collection is broken down to into subsets of smaller elements and each subset is assigned a thread to execute. Since thread creation is expensive, creating a new thread for each subset is not optimal. Scala uses Java Fork/Join Framework [20] to achieve better performance. This framework enables scalable and efficient parallelism in performing the computations. In this framework, computation is broken down into tasks and each task is assigned to a worker thread. The result from all these tasks are combined/joined to produce the final result. However, in multicore environments, the performance is determined by how well the tasks are scheduled among different processors. The Fork/Join framework uses work stealing for task scheduling wherein once a worker's queue is empty, it tries to grab a task from any other random worker's queue to achieve faster computations. We can also specify the desired amount of parallelism or number of threads in the fork/join pool. For example, the below line of code sets the parallelism level to 12.

```
new java.util.concurrent.ForkJoinPool(12)
```

V. Extending Compressed Linear Algebra to Advanced Analytical Applications

Advanced analytics usually deal with large amounts of data demanding an efficient approach in each step of the analytics. Compressed linear algebra can be used in these applications to get performance gains.

A covariance matrix is a structure capturing covariance in multidimensional data. Covariance determines how much two variables change with respect to each other. Computing the covariance matrix is required by advanced analytical techniques like Principal Component Analysis (PCA) [21], which can use either Eigenvalue analysis or Singular Value Decomposition (SVD). PCA has applications in various fields including but not limited to finance, pharmacy, biology. Covariance matrix also finds its use in the area of portfolio management. The active portfolio management discussed in [22], for example, considers covariance matrix as a measure of risk when considering risk-return tradeoffs. Covariance matrix which when implemented using compressed linear algebra in ScalaTion optimizes not just in computation but also in storage.

Given a data matrix $X \in \mathbb{R}^{m \times n}$, $\bar{X}$ is an estimator of the mean vector and $\bar{\Sigma}(X)$ is an estimator of the covariance matrix and are calculated as follows

$$\bar{X} = [\bar{X}_{*,1}, \ldots, \bar{X}_{*,n}]$$

$$\bar{\Sigma}(X) = \frac{(X - \bar{X})^{\mathrm{T}}(X - \bar{X})}{n - 1}$$

VI. Performance Evaluations

Following the philosophy of open science [23] we have set up the experiments in such a way to facilitate other researchers to build upon our results by executing on different platforms and even implementing additional operators (for details see www.cs.uga.edu/~jam/scalation_1.2/README.html). We used UGA Zcluster do the performance testing. Zcluster is the computing environment provided by the Georgia Advanced Computing Resource Center (GACRC). The tests were run on nodes having 12 core Intel Xeon processors and 256GB of memory. To make sure that RLE vector contains a varied amount of run lengths we adopted the below formula.

$$(size^{runLengthval}).toInt$$

For example, if the runLengthVal is 0.2, then a vector of size 100 million will have values whose run lengths can be anything from 1 to 39. We considered the runLengthVal to be 0.2, 0.3 and 0.4 in our experiments. Each time value recorded is the average of 100 runs. Nodes whose load was more than 4 before running the job were not considered.

A. Variance

Table I shows the time taken in milliseconds for variance operation and Fig 7 shows the graph. The performance of RLE and dense is very similar when the size is 100K. However, we can see that the RLE vector performs better with increasing size. RLE vector with runLengthVal of 0.4 performs better than 0.2 and 0.3 variants as it has a better compression ratio with fewer triplets.

TABLE I
VARIANCE

	100K	1M	10M	100M
Dense	1.333	5.910	58.902	571.656
Rle (0.2)	1.482	3.365	25.022	130.731
Rle (0.3)	1.238	1.308	4.921	26.903
Rle (0.4)	0.706	1.120	1.560	4.315

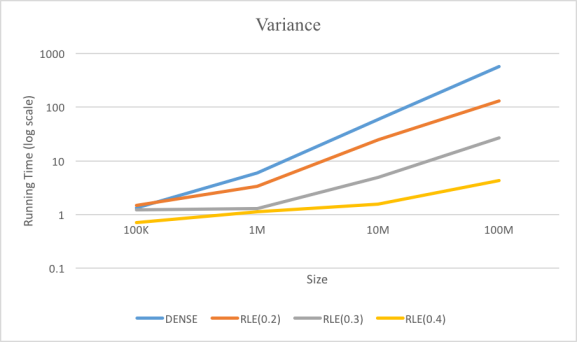


Fig. 7. Running times of variance operation

Fig 8 shows the speed up in parallel implementation of variance in RLE vector. There is a good amount of speed up as we increase the number of threads. We see an increase in the speedup factor as the number of threads increases. However, we see the speed up factor tends to drop a bit when the number of threads is 10 or more. Load on the Zcluster nodes can be a factor that causes this behavior. For this analysis we considered the size to be 100M and the runLengthVal to be 0.2. The runLengthVal of 0.2 is picked to be conservative.

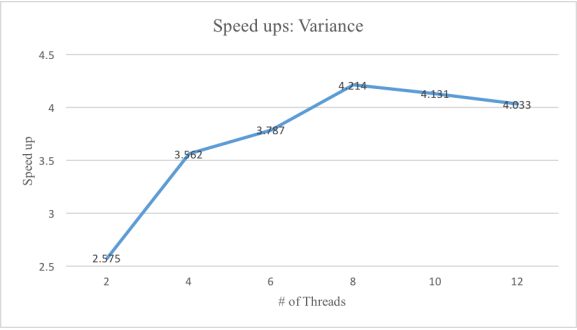


Fig. 8. Speed up of variance operation in RLE vector where *size = 100M and runLengthVal = 0.2*

B. Matrix Dot Product (dot and mdot)

Table II and III show the time taken in milliseconds for the dot and mdot operation, respectively. In dot, the run length encoded variants tend to perform better than the dense as the size increases. In mdot, we see that the run length encoded variant performs better than dense for all the sizes considered. The dense variant performs better than the compressed variant for sizes 500K * 250 and 750K * 300. The load on the Zcluster nodes might be a reason for this behavior. Fig 9 and 10 shows the graph corresponding to the running time.

TABLE II
DOT

	500K*250	750K*300	1M*350	1.25M*400	1.5M*450
Dense	405.638	714.498	1197.545	1647.284	2142.262
Rle (0.2)	461.318	818.163	1107.707	1497.648	1865.432
Rle (0.3)	147.938	224.670	301.713	399.585	473.636
Rle (0.4)	43.085	60.868	81.587	100.199	130.547

TABLE III
MDOT

	125K*100	200K*150	250K*200	500K*250
Dense	38028.92	126918.615	298669.35	922944.293
Rle (0.2)	5629.317	14854.119	33083.823	88135.299
Rle (0.3)	1851.427	4719.289	9629.486	24308.794
Rle (0.4)	580.6	1604.591	3177.852	6678.272

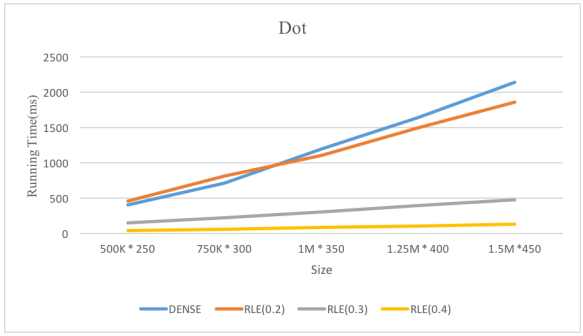


Fig. 9. Running times of dot operation

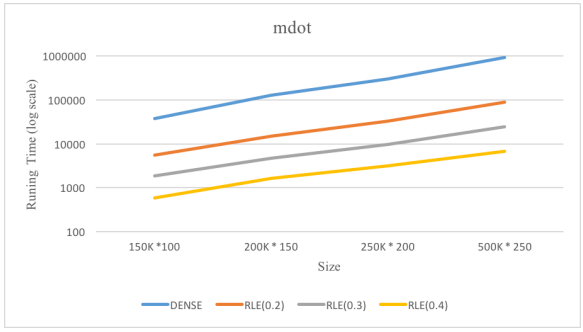


Fig. 10. Running times of mdot operation

Fig 11 and 12 show the speed up achieved in parallel implementations of dot and mdot operations. We achieve close

to 8 times speed up and see there is an increase in speed up as the number of threads increase.

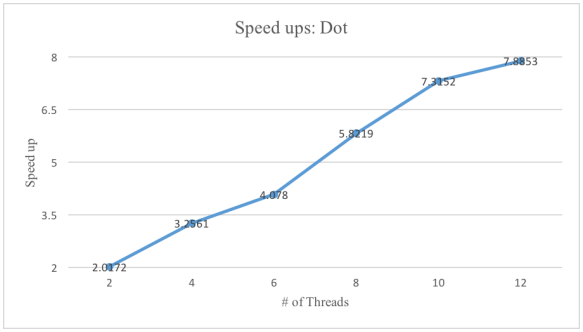


Fig. 11. Speed up of dot operation in RleMatrix where *size = 1.5M * 450 and runLengthVal = 0.2*

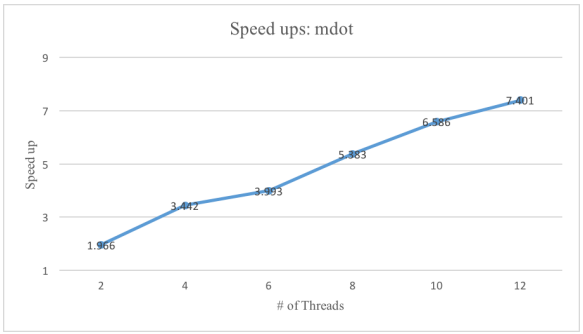


Fig. 12. Speed up of mdot operation in RleMatrix where *size = 500K * 250 and runLengthVal = 0.2*

C. Covariance Matrix

Table IV and V show the time in milliseconds taken to compute covariance matrix using dot and mdot operation. We see that here too the run length encoded variant performs better than the dense for the sizes considered. Fig 13 and 14 show the graphs. The graphs show running times in log scale for ease of display.

TABLE IV
COVARIANCE MATRIX USING DOT

	125K*100	250K*150	375K*200	500K*250
Dense	41480.927	178702.386	548189.092	1241232.450
Rle (0.2)	7628.197	30503.018	72769.150	152110.244
Rle (0.3)	2899.550	10165.394	23363.203	44020.599
Rle (0.4)	1192.539	3955.631	8604.610	15456.106

TABLE V
COVARIANCE MATRIX USING MDOT

	125K*100	250K*150	375K*200	500K*250
Dense	34361.954	167792.864	456690.646	948004.948
Rle (0.2)	4262.414	16638.372	40070.643	82930.099
Rle (0.3)	1437.823	5326.554	12374.461	23354.191
Rle (0.4)	466.196	1615.642	3512.591	6577.679

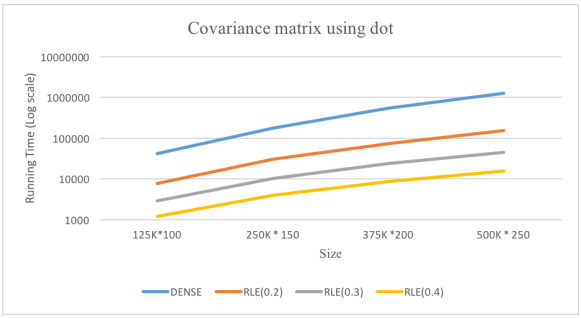


Fig. 13. Running times of computing covariance RleMatrix

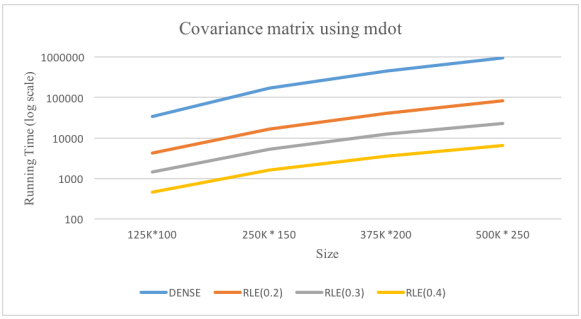


Fig. 14. Running times of computing covariance RleMatrix

Figures 15 and 16 show the speed up in the parallel implementations. We see that using mdot operation to compute covariance matrix tends to give better speedup when compared to using dot operation. While using dot we see that the speed up is close to 5 times whereas for mdot it is 6 times.

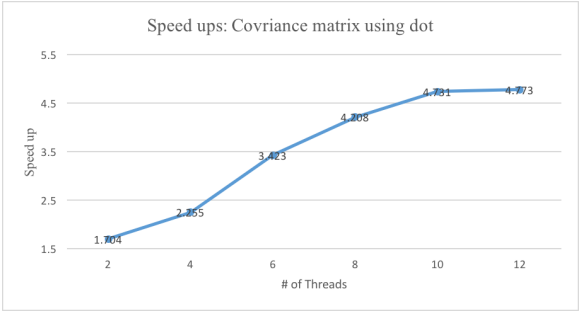


Fig. 15. Speed up of computing covariance RleMatrix using dot where *size = 500K * 250 and runLengthVal = 0.2*

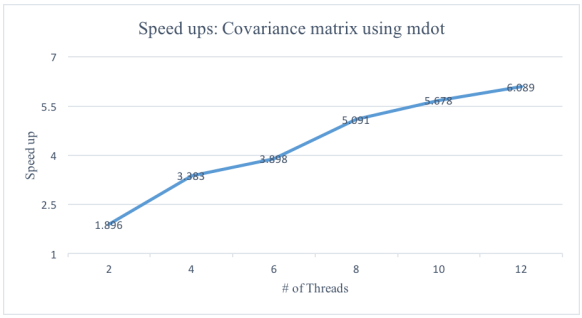


Fig. 16. Speed up of computing covariance RleMatrix using mdot where *size = 500K * 250 and runLengthVal = 0.2*

VII. Conclusions and Future Work

Compression techniques commonly used in columnar relational databases are added to a comprehensive linear algebra package provided by the ScalaTion open source big data framework. This allows matrices and vectors to be stored with considerably less space and in some cases provided exceptional speed up.

Some of the contributions of this paper include the use of Run Length Encoding (RLE) to speed up vector and matrix operations that are foundations for big data analytics. Integration of columnar relational and linear algebra provides efficient and convenient means for carrying out ad-hoc analytical studies. The paper also shows how the use of the ScalaTion framework for parallel execution makes it easy to convert sequential codes to parallel implementations.

In the future, we plan to support handling of distributed data in ScalaTion. We would work on exploring the tradeoffs of using sparsity versus compression and also see how to utilize these techniques in advanced predictive analytics. Also, a faster form of update on RLE vector will be explored. ScalaTion can also be made to adopt Apache Arrow [24], an in-memory columnar data layer that can be shared across systems. This would make ScalaTion compatible with other platforms making it easier to transfer analytical data to and from these platforms. This would result in saving of computing resources which would otherwise be spent on serializing and deserializing data. Open source big data projects like Hadoop, HBase, and Spark are working with Apache Arrow.

References

[1] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006, pp. 671–682.

[2] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1383–1394.

[3] M. L. H. P. Jens Krger, Johannes Wust, "Leveraging compression in in-memory databases," in *Proceedings of the DBKDA 2012, The Fourth International Conference on Advances in Databases, Knowledge, and Data Applications*, 0 2012, pp. 147 – 153.

[4] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: Sql and rich analytics at scale," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*. ACM, 2013, pp. 13–24.

[5] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli, "Druid: A real-time analytical data store," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 157–168.

[6] S. Prasad, A. Fard, V. Gupta, J. Martinez, J. LeFevre, V. Xu, M. Hsu, and I. Roy, "Large-scale predictive analytics in vertica: fast data transfer, distributed model creation, and in-database prediction," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1657–1668.

[7] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson, "Ricardo: integrating r and hadoop," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 987–998.

[8] D. J. Abadi *et al.*, "Column stores for wide and sparse data." in *CIDR*, 2007, pp. 292–297.

[9] R. D. Shah and N. Meinshausen, "Min-wise hashing for large-scale regression and classification with sparse data," *arXiv preprint arXiv:1308.1269*, 2013.

[10] J. A. Miller, C. Bowman, V. G. Harish, and S. Quinn, "Open source big data analytics frameworks written in scala," in *Big Data (BigData Congress), 2016 IEEE International Congress on*. IEEE, 2016, pp. 389–393.

[11] H. Zou, Y. Yu, W. Tang, and H. M. Chen, "Improving i/o performance with adaptive data compression for big data applications," in *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. IEEE, 2014, pp. 1228–1237.

[12] M. Stonebraker, P. Brown, D. Zhang, and J. Becla, "Scidb: A database management system for applications with complex analytics," *Computing in Science & Engineering*, vol. 15, no. 3, pp. 54–62, 2013.

[13] P. G. Brown, "Overview of scidb: large scale array storage, processing and analysis," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 963–968.

[14] D. Kernert, F. Köhler, and W. Lehner, "Slacid-sparse linear algebra in a column-oriented in-memory database system," in *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*. ACM, 2014, p. 11.

[15] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li *et al.*, "The madlib analytics library: or mad skills, the sql," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1700–1711, 2012.

[16] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2015. [Online]. Available: https://www.R-project.org/

[17] A. Mohapatra and M. Genesereth, "Incrementally maintaining run-length encoded attributes in column stores," in *Proceedings of the 16th International Database Engineering & Applications Sysmposium*. ACM, 2012, pp. 146–154.

[18] M. MATLAB and S. T. Release, "Natick," *Massachusetts, United States: The MathWorks Inc*, 2012.

[19] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky, "A generic parallel collection framework," in *European Conference on Parallel Processing*. Springer, 2011, pp. 136–147.

[20] D. Lea, "A java fork/join framework," in *Proceedings of the ACM 2000 conference on Java Grande*. ACM, 2000, pp. 36–43.

[21] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 2, no. 4, pp. 433–459, 2010.

[22] I. Bolshakova, E. Girlich, and M. Kovalev, *Portfolio optimization problems: A survey*. Univ., Fak. für Mathematik, 2009.

[23] J. C. Molloy, "The open knowledge foundation: open data means better science," *PLoS Biol*, vol. 9, no. 12, p. e1001195, 2011.

[24] "Apache arrow," https://arrow.apache.org, accessed: 2016-11-14.