A Big Data Platform Integrating Compressed Linear Algebra with Columnar Databases

by

Vishnu Gowda Harish

(Under the Direction of John A.Miller)

Abstract

Key foundational components of Big Data frameworks include efficient large-scale storage and high-performance linear algebra. We discuss 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 words: Data analysis; Data compression, Data mining, Linear algebra, Parallel programming;

A Big Data Platform Integrating Compressed Linear Algebra with Columnar Databases

by

Vishnu Gowda Harish

B.E., Visveswaraiah Technological University, 2008

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2017

A Big Data Platform Integrating Compressed Linear Algebra with Columnar Databases

by

Vishnu Gowda Harish

Acknowledgments

A special thank you to my major professor Dr John A. Miller. I have had a wonderful experience doing research under Dr Miller. His constant encouragement and technical guidance throughout my time at University of Georgia has been pivotal in completing my degree. I would also like to thank Dr Lakshmish Ramaswamy for being part of my committee. I have really enjoyed taking Distributed Systems and Advanced Data Intensive Computing courses under him and have learnt a lot. I would also like to thank Dr Ismailcem Budak Arpinar for agreeing to be part of my committe and for the timely help. I also thank Vinay Kumar Bingi for helping me with my thesis.

I would like to convey my deepest gratitude to my family. I am extremely grateful to my mother and my late father. Without their love and sacrifices I would have never got to where I am today. My wife has been my pillar of strength and has been very supportive in everything I do. My elder brother for always being there for me and constantly guiding me.

Table of Contents

Appendix

List of Figures

List of Tables

Chapter 1

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, 2, 3, 4, 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 document is organized as follows: Chapter 2 provides a brief overview of the recent work in integrating linear algebra in databases and the use of compression in column stores. In Chapter 3 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 Chapter 3 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 Chapter 4. Chapter 5 talks about how compressed linear algebra can be useful in advanced analytical applications. Chapter 6 discusses the performance results and Chapter 7 presents conclusions and future work.

Chapter 2

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).

The R language has an RLE class in its base package [16]. Using this class, atomic[1] 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. In ScalaTion, we store the starting position of the value along with the value and the length of contiguous occurence. The advantage of this is faster random access, while the disadvantage is it requires more space. For example, ScalaTion would require 33 percent more space than R for doubles.

[1]Not a list, see https://stat.ethz.ch/R-manual/R-devel/library/base/html/rle.html

Chapter 3

Integrating Compressed Linear Algebra

3.1 RLE Vectors

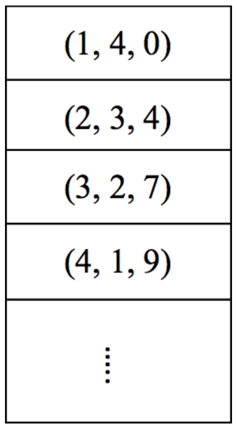
In run length encoding the original sequence of values is replaced with the value and number of times the value repeats. In ScalaTion we also record the starting position of the value along with the value and number of repetitions [1]. This information constitutes a Triplet and is represented as (*value, count, startPos*). An example of the Triplet is shown in Fig. 3.1 . 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*).

$$1,\ 1,\ 1,\ 1,\ 2,\ 2,\ 2,\ 3,\ 3,\ 4, \ldots$$

(a) Original run of values

$$(1, 4, 0)\ ,\ (2, 3, 4)\ ,\ (3, 2, 7)\ ,\ (4, 1, 9), \ldots$$

(b) Run of triplets



(c) ReArray

Figure 3.1: Run length encoding

Storing the values as Triplets will definitely save a substantial amount of space, but the real challenge is in making 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.3.1 (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

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu)^2 \tag{3.1}$$

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.

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}x_i^2 - \frac{S^2}{n} \tag{3.2}$$

The implementation in ScalSCALATIONaTion 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 logarithmic time. 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.3.2 shows the original run of values and the initial state of the ReArray.

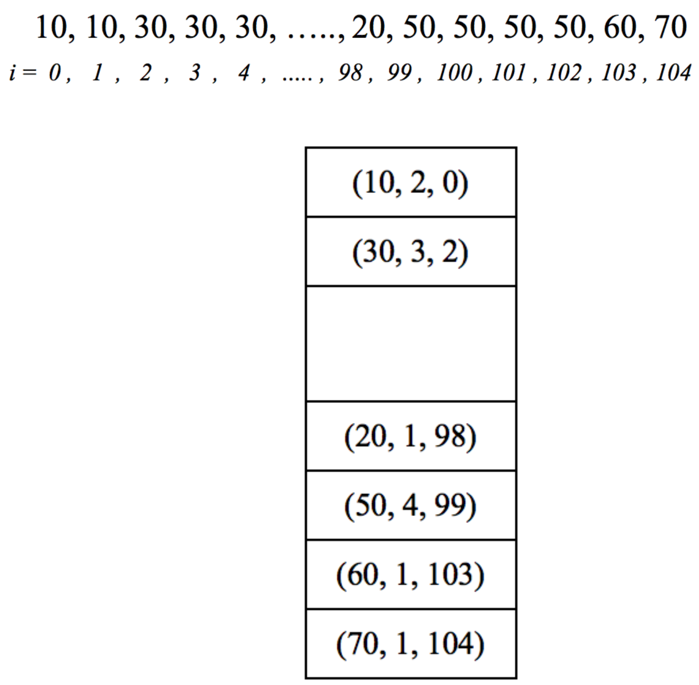


Figure 3.2: Data and Initial state of Rearray

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.3.3.

Fig.3.4 explains one of the scenarios where ReArray grows in size. Here we update index $i = 99$ to the value 30 and 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.

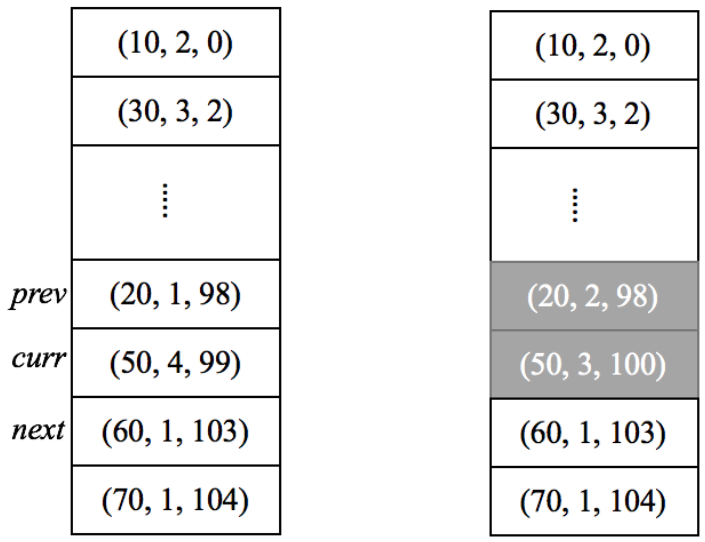


Figure 3.3: Update i = 99 to 20

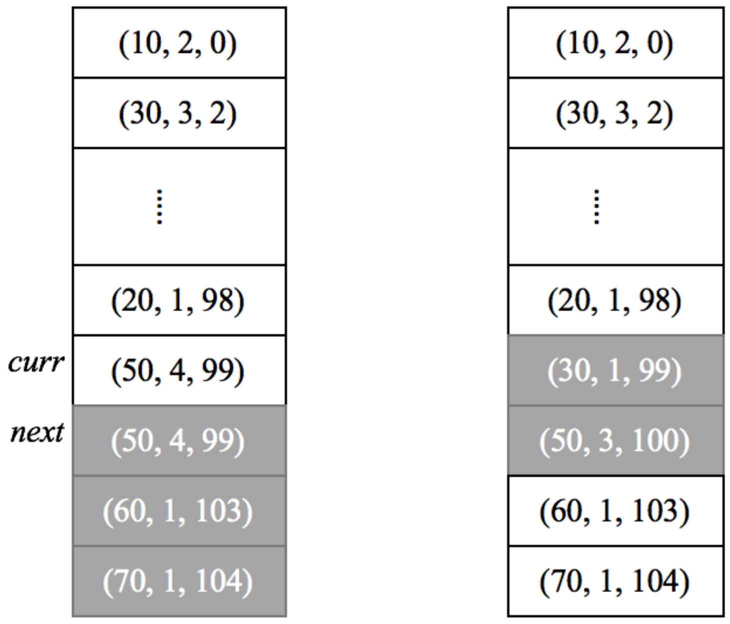


Figure 3.4: Update i = 99 to 30

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.3.5 depicts this scenario.

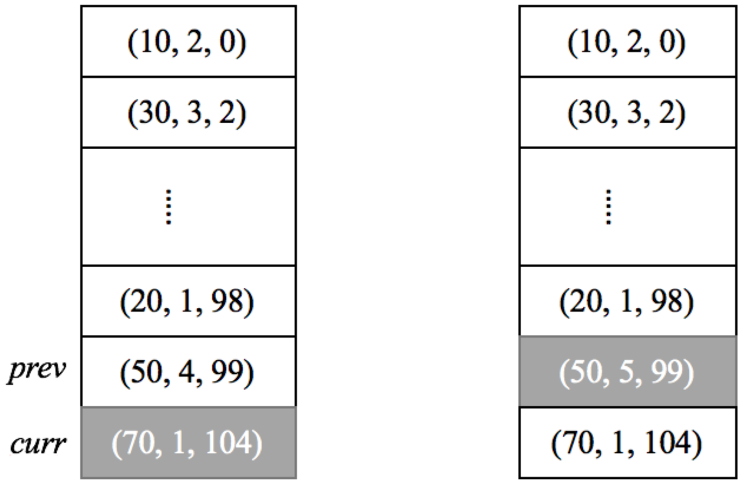


Figure 3.5: Update i = 103 to 50

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

3.2 RLE Matrices

RLE matrix contains a collection of RLE vectors that constitute a matrix. Fig.3.6 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.

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

Figure 3.6: A Relation and its equivalent run length encoded 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 $A = \left[a_{ij}\right]$ is an m-by-n matrix and $B = \left[b_{ij}\right]$ is an n-by-p matrix then the product AB $= \left[ab_{ij}\right]$ will be an m-by-p matrix and is defined as follows:

$$ab_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{3.3}$$

If you observe the above equation, you see that each element in the resultant matrix $\left[ab_{ij}\right]$ 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.

$$AB = A^T\ mdot\ B \tag{3.4}$$

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.

```
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))
    } // 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
```

More details about running and extending the code are given in Appendix A.

3.3 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 for calculating the revenue for the southern Atlantic states, columns 1 to 4 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 product of the x and y matrices. This gives the revenue per state for item1. The revenue for item1 is added to the revenue relation.

```
val sales_item1 = Relation ("Sales_Item1",
                            Seq ("Date", "FL", "GA", "NC", "SC"), ...
                            0,"SIIII")
val price_item1 = Relation ("Price_Item1",
                            Seq ("Date", "FL", "GA", "NC", "SC"), ...
                            0,"SDDDD")
val revenue     = Relation ("Revenue", -1, null, "Item",
                            Seq ("Date", "FL", "GA", "NC", "SC")
sales_item1.show()
price_item1.show()

val x = sales_item1.toMatriD (1 to 4, COMPRESSED)
val y = price_item1.toMatriD (1 to 4, COMPRESSED)
val z = x dot y
revenue.add ("Item1" +: z())
revenue.show ()
```

For the complete code see RelationTest5 object `http://www.cs.uga.edu/~jam/scalation_1.2/src/main/scala/scalation/relalgebra/Relation.scala`.

For a more complex real world example where we extract matrices from relations and perform operations refer to the Solar Radiation application (see Appendix B).

Chapter 4

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)
```

Chapter 5

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

$$\bar{X} = [\bar{X}_{*,1}, \ldots, \bar{X}_{*,n}] \tag{5.1}$$

$$\bar{\Sigma}(X) = \frac{(X - \bar{X})^{\mathrm{T}}(X - \bar{X})}{n - 1} \tag{5.2}$$

Chapter 6

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 compute nodes having 12 core Intel Xeon processors and 256GB of memory. To make sure that RLE vectors contains varied amounts of run lengths we adopted the below formulas.

$$\text{Max Repetitions} = \text{mrep} = (\text{n}^{rlp}).\text{toInt} \tag{6.1}$$

$$\text{Repetitions} = \text{Rep} = \text{RandomInteger}\ (1,\ \text{mrep}) \tag{6.2}$$

$$\text{Expected Repetitions} = \text{E}\ (\text{Rep}) = \frac{\text{mrep} + 1}{2} \tag{6.3}$$

$$\text{Expected Number of Triplets} = \text{n}_\text{c} = \frac{2\text{n}}{\text{mrep} + 1} \tag{6.4}$$

$$\text{Expected Compression Ratio} = \text{c}_\text{r} = \frac{\text{n}}{\text{n}_\text{c}} = \frac{\text{mrep} + 1}{2} \tag{6.5}$$

For example, if the run length parameter (rlp) is 0.2, then a vector of $n = 100$ million will have values whose Rep can be anything from 1 to 39 ($mrep$). Equations 6.4 and 6.5 gives the formulas to calculate theoretical values of n_c and c_r which would be 5 million and 20. Therefore, storing the dense vector requires 800 million ($8 * n$) bytes, while the Rle vector requires 80 million ($(8 + 4 + 4) * n_c$) bytes. The space compression ratio is $\frac{c_r}{2}$, which equals 10 in this case. We considered the rlp to be 0.2, 0.3 and 0.4 in our experiments. Each time value recorded is the average of 100 runs. Compute nodes whose load was more than 4 before running the job were not considered (see Appendix D).

6.1 Variance

Table 6.1 shows the time taken in milliseconds for variance operation and Fig 6.1 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 *rlp* of 0.4 performs better than 0.2 and 0.3 variants as it has a better compression ratio with fewer triplets.

Table 6.1: 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

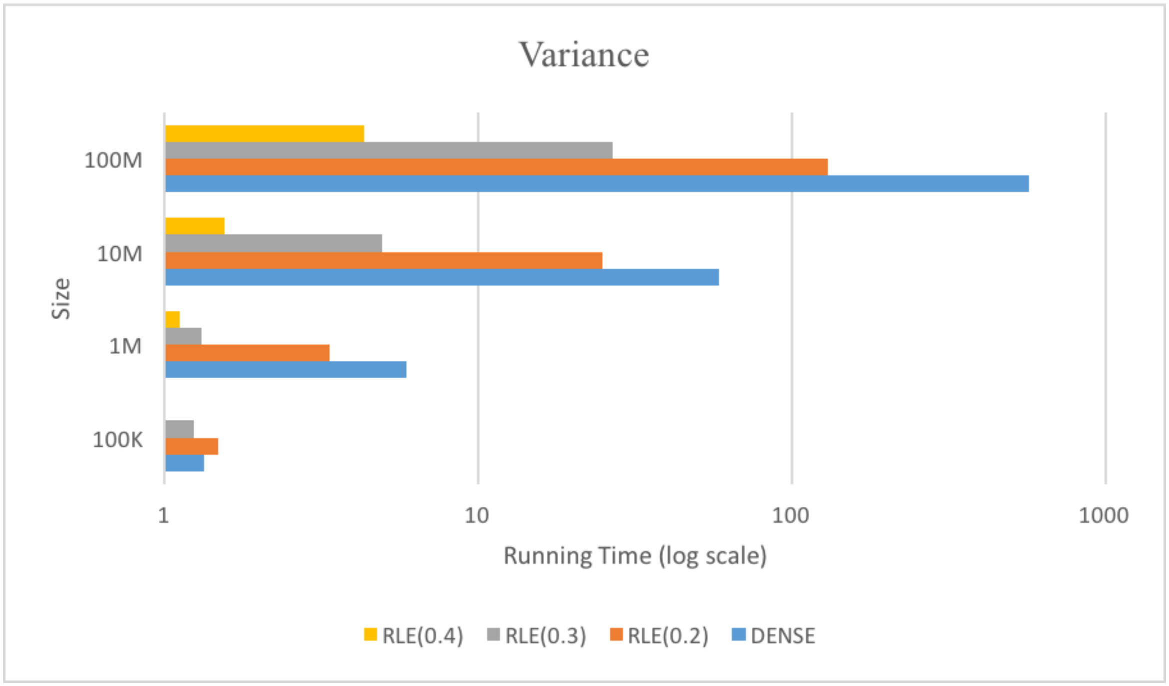


Figure 6.1: Running times of variance operation

Fig 6.2 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. More investigation needs to be done to understand the cause of this behaviour. For this analysis we considered the size to be 100M and the *rlp* to be 0.2. The *rlp* of 0.2 is picked to be conservative.

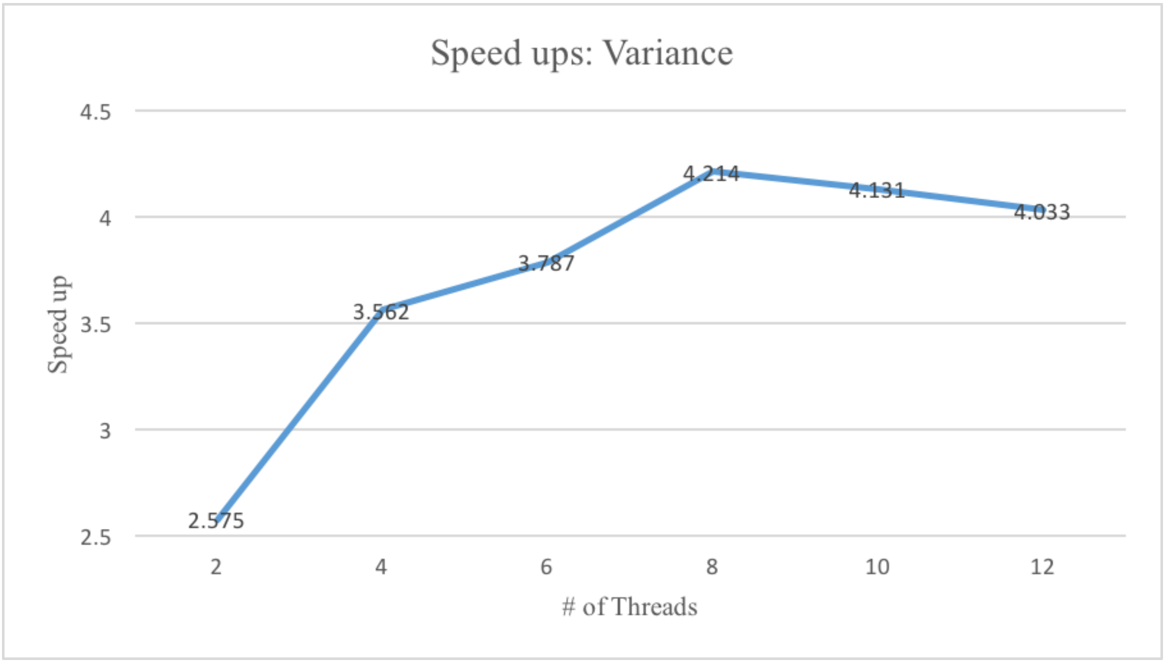


Figure 6.2: Speed up of variance operation in RLE vector where size = 100M and rlp = 0.2

6.2 Matrix Dot Product (dot and mdot)

The table below shows the time taken in milliseconds for the dot operation.

Table 6.2: 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

In certain cases, the dense fares better than the RLE (500k * 250 and 750K * 300 with rlp = 0.2). The load on the Zcluster nodes might be a reason for this behavior. The numbers also indicate that the run length encoded variants tend to perform better than the dense as the size increases. Fig 6.3 shows the graph corresponding to the running time.

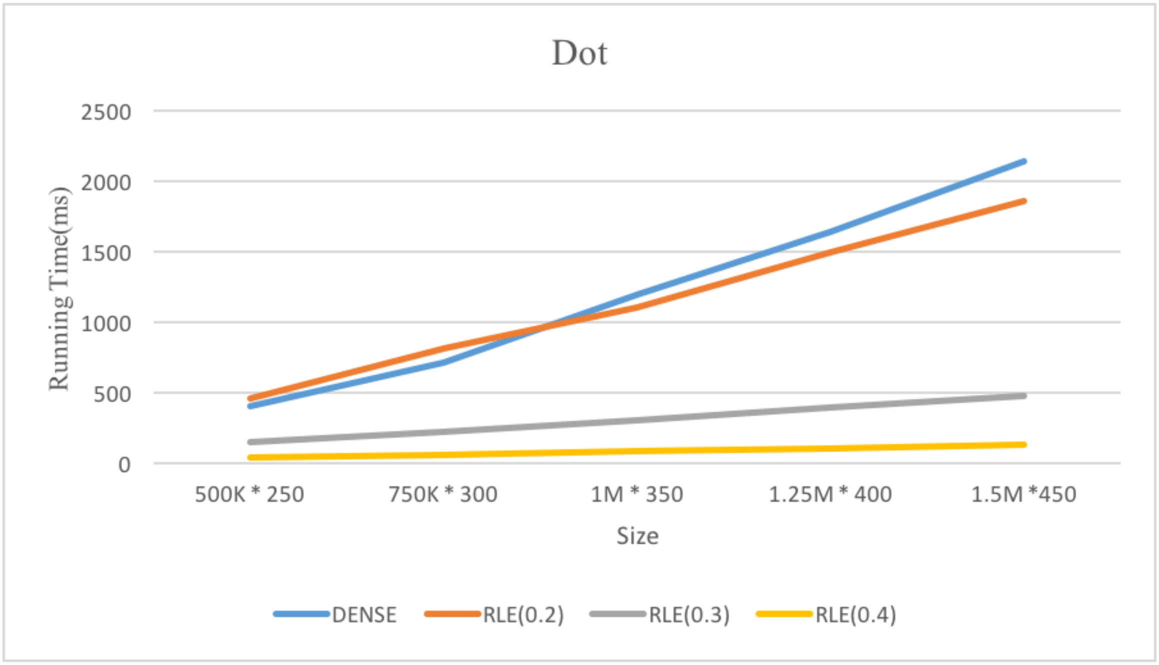


Figure 6.3: Running times of dot operation

Table 6.3 shows the time taken in milliseconds for the mdot operation. Unlike dot, in mdot we see that the run length encoded variant performs better than dense for all the sizes considered. Fig 6.4 shows the graph corresponding to the running time.

Table 6.3: 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

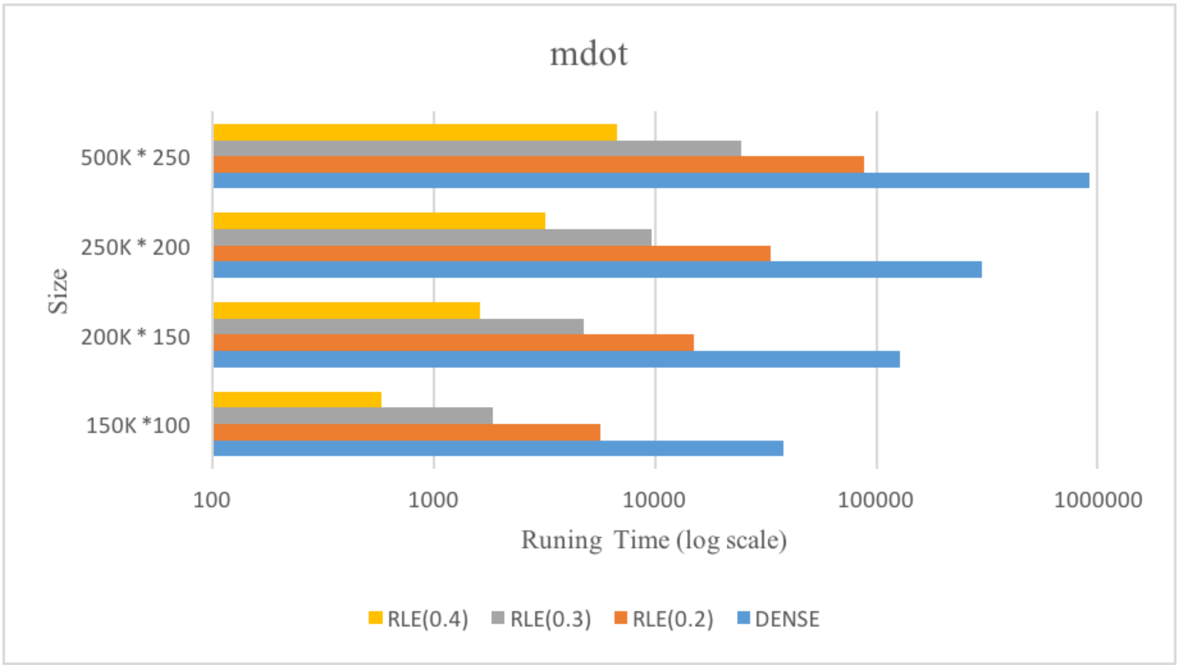


Figure 6.4: Running times of mdot operation

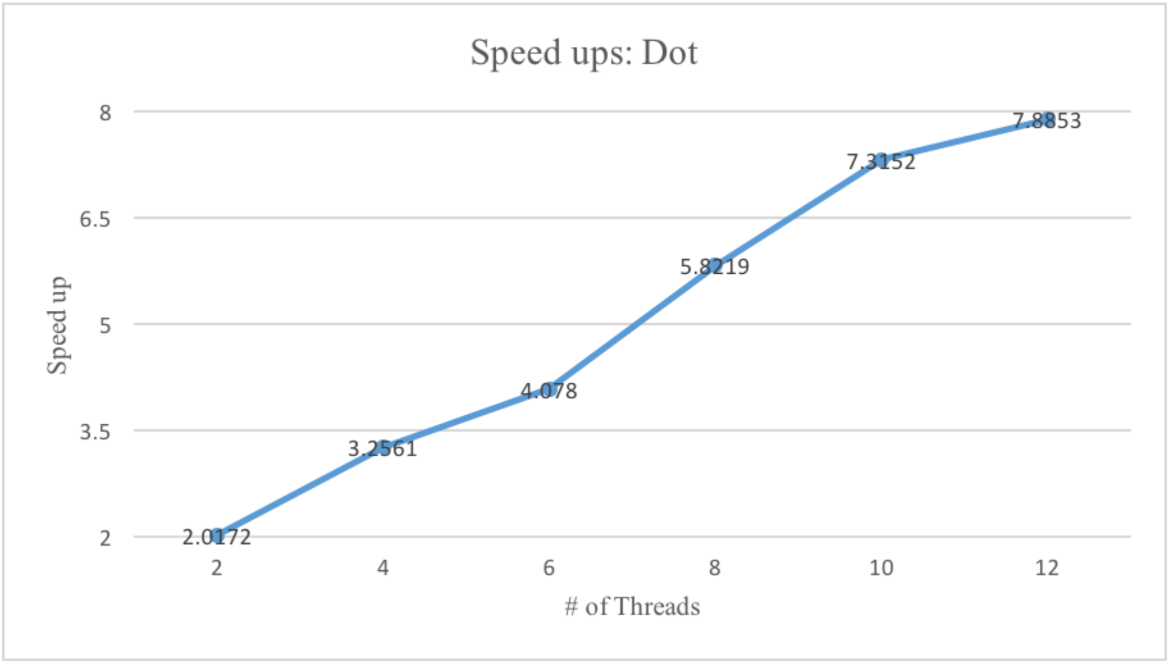


Figure 6.5: Speed up of dot operation in RleMatrix where size = 1.5M * 450 and rlp = 0.2

Fig 6.5 and 6.6 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.

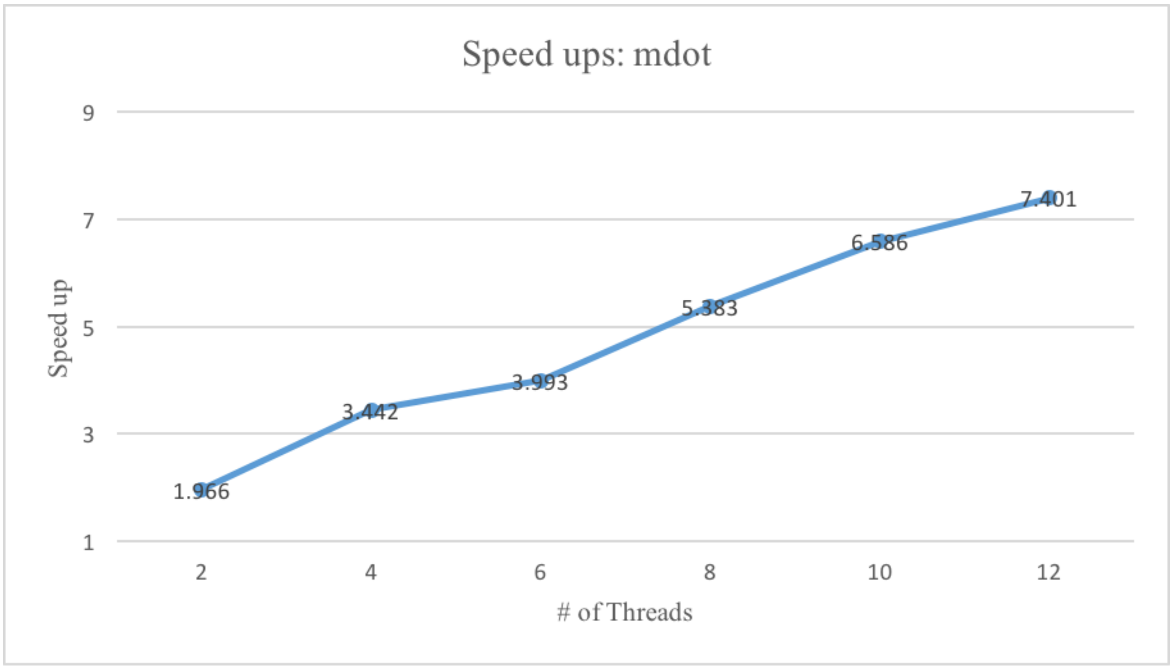


Figure 6.6: Speed up of mdot operation in RleMatrix where size = 500K * 250 and rlp = 0.2

6.3 Covariance Matrix

Table 6.4: 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 6.4 and 6.5 show the time in milliseconds taken to compute covariance matrix using dot and mdot operation. Computing the covariance matrix using mdot is faster than using dot. However, we see that the run length encoded variant performs better than the dense in either of the cases. For example, even with the case of rlp being 0.2, where there the number

Table 6.5: 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

of triplets is more, the RLE variant is around 8 to 12 times faster than dense when we use mdot to compute covariance matrix. While using dot to compute covariance matrix the RLE variant is around 6 to 8 times faster. Fig 6.7 and 6.8 show the graphs. The graphs show running times in log scale for ease of display.

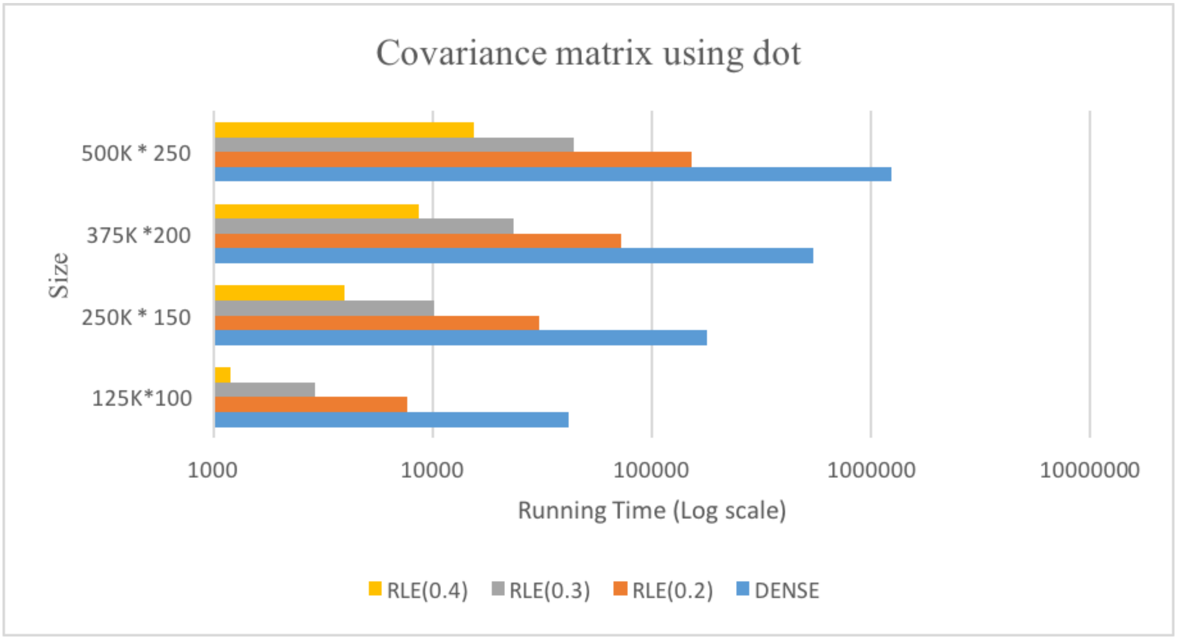


Figure 6.7: Running times of computing covariance RleMatrix

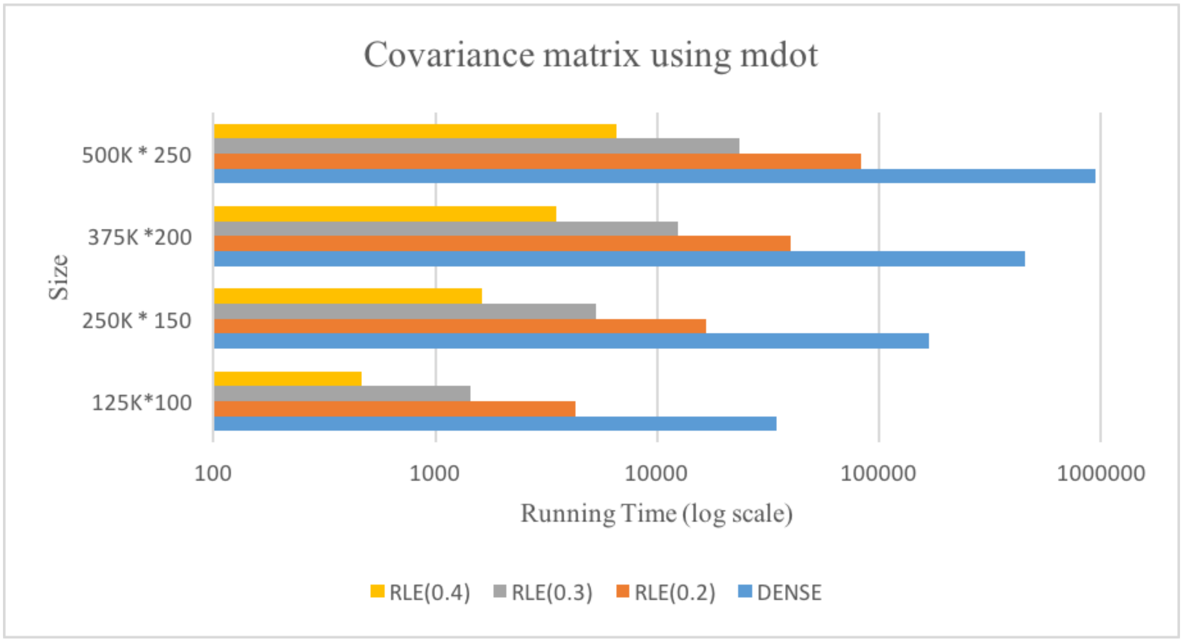


Figure 6.8: Running times of computing covariance RleMatrix

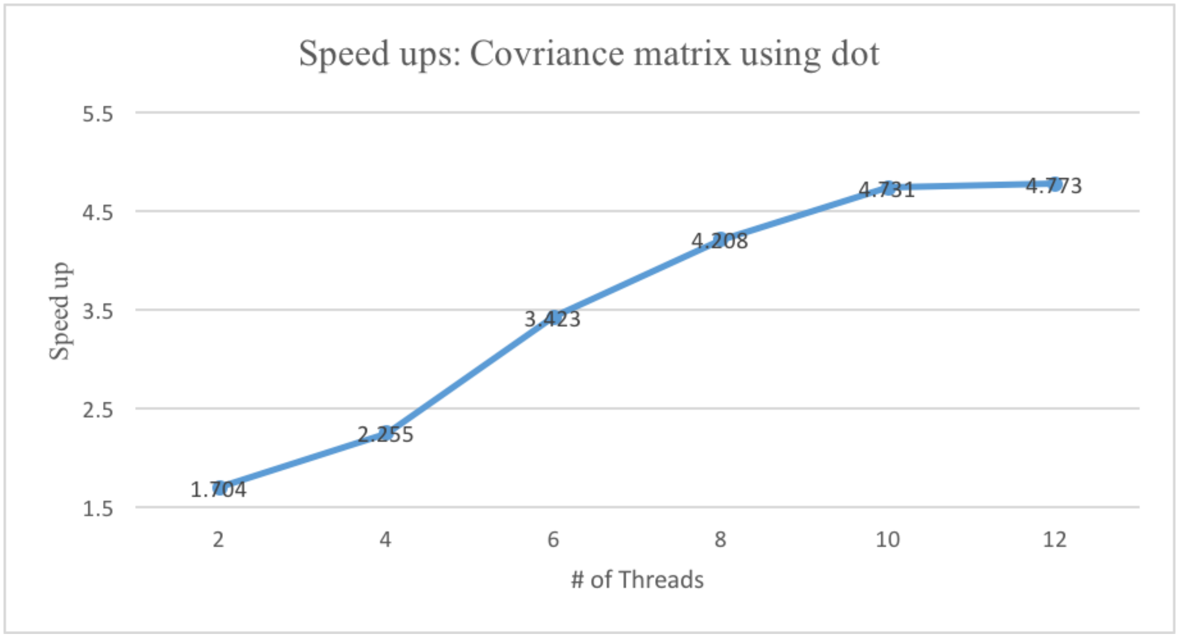


Figure 6.9: Speed up of computing covariance RleMatrix using dot where size = 500K * 250 and rlp = 0.2

Figures 6.9 and 6.10 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.

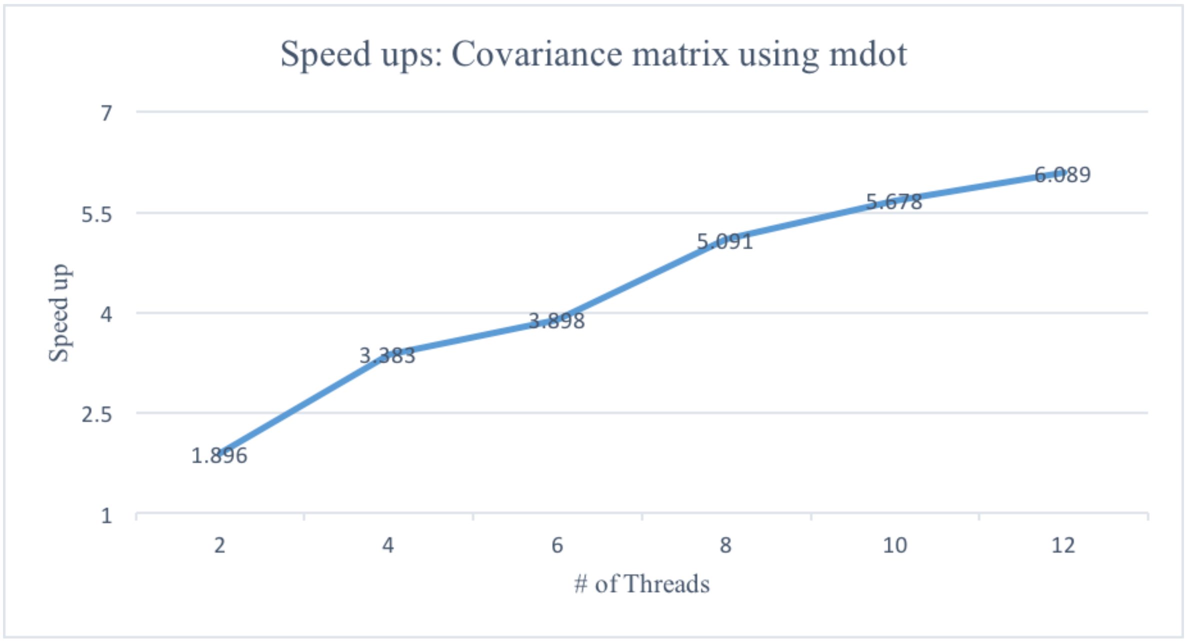


Figure 6.10: Speed up of computing covariance RleMatrix using mdot where size = 500K * 250 and rlp = 0.2

Besides syntheic data, preliminary testing of real world datasets has been conducted (see Appendix C).

Chapter 7

Conclusions and Future Work

In this research we show how Run Length Encoding (RLE) compression technique, commonly used in columnar relational databases, is added into 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. We also show how the use of the ScalaTion framework for parallel execution makes it easy to convert sequential codes to parallel implementations. The scope of extending compressed linear algebra to advanced analytical applications is shown by taking the example of a covariance matrix. Exceptional speed up is achieved in computing the covariance matrix sequentially as well as in parallel. We also show how integration of columnar relational and linear algebra provides efficient and convenient means for carrying out ad-hoc analytical studies.

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.

Bibliography

[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.

Appendix A

Developers Guide

ScalaTion is available under MIT license[1] and the source code for the current release (1.2) is available on the web.[2] ScalaTion uses sbt as the build utility tool and installation instructions are given in the home page. The ScalaDoc for the entire source code can be accessed from the home page[3]. The source code for dense and Run Length Encoded (RLE) variants of vectors and matrices is under the `scalation.linalgebra` package. The parallel flavor is present in `scalation.linalgebra.par` package.

Unit test code for the linear algebra implementations (e.g., vectors and matrices) is under the `testing.linalgebra` package. To run all the unit test cases via sbt we can use the `test` command. To run test cases of a specific implementation `test-only` command can be used. For example, the following sbt commands can be used to perform unit testing on RleVectorD and RleMatrixD classes. These classes store and operate on values of double datatype.

```
test-only testing.linalgebra.RleVectorD_T

test-only testing.linalgebra.RleMatrixD_T
```

[1] ScalaTion License File: `http://www.cs.uga.edu/~jam/scalation_1.x/LICENSE.html`
[2] ScalaTion Home Page: `http://www.cs.uga.edu/~jam/scalation_1.x/README.html`
[3] ScalaDoc: `http://www.cs.uga.edu/~jam/scalation_1.x/README.html#scaladoc`

Appendix B

Solar Radiation Application

A Solar Radiation application included as part of ScalaTion performs basic analytics on solar radiation and meteorological data. The data is provided by the National Solar Radiation Database (NSRDB[1]). Hourly Global Horizontal Radiation (GHR[2]) data of 40 sites from the years 1961 to 1990 is considered. For the dataset go to the ScalaTion home page. The following commands can be used to run the application.

```
$ unzip ~Download/solar-radiation-40.zip
$ mv solar-radiation-40.csv $SCALATION_HOME/data/analytics
$ sbt
> run-main apps.analytics.SolarRadiation
```

The code for the application is shown in B.1. Data is read from the csv file (*solar-radiation-40.csv*[3]) and a relation (*solarRel*) is created. Dense (*solarMat*) and Rle (*solarRleMat*) matrices are spun from the relation. The GHR mean value of every city is calculated from the dense matrix and stored in *mu*. Similarly *cmu* stores the GHR mean values calculated from the Rle matrix. Highest and lowest averages along with the corresponding site are printed. We calculate the time required to compute the mean vectors (*mu* and *cmu*). The average time for 5 iterations is considered. We also output the size of data for each column before and after compression.

[1]NSRDB Web Page: `http://rredc.nrel.gov/solar/old_data/nsrdb/`
[2]NSRDB Data: `http://rredc.nrel.gov/solar/old_data/nsrdb/1961-1990/hourly/`
[3]solar-radiation-40.csv: `scalation_1.x/data/analytics/solar-radiation-40.csv`

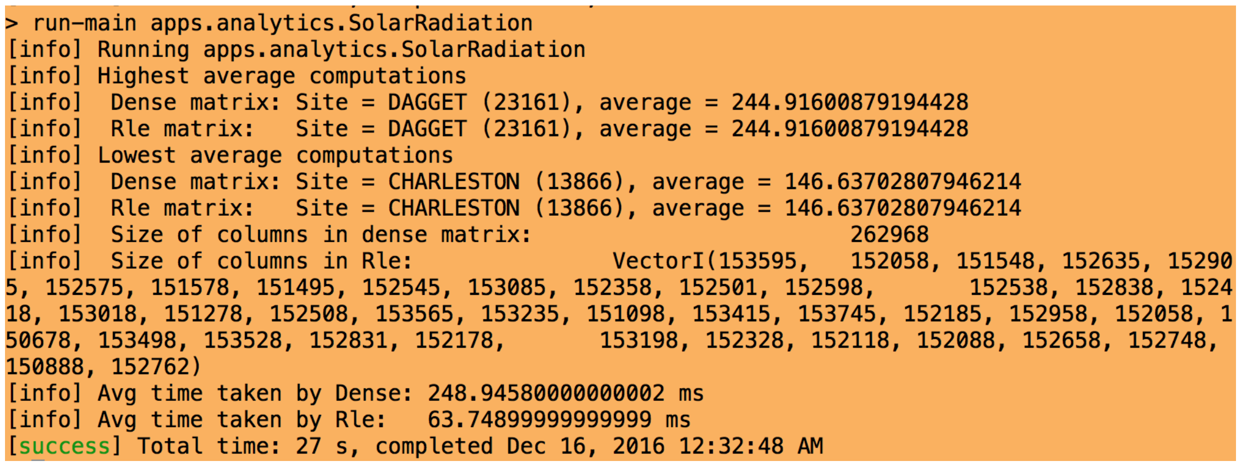
B.1 Code

```
object SolarRadiation extends App with PackageInfo
{
    val fName        = BASE_DIR + "solar-radiation-40.csv"
    val solarRel     = Relation (fName, "solarRel", -1, null, ",")
    val solarMat     = solarRel.toMatriD (0 to 39)
    val solarRleMat  = solarRel.toMatriD (0 to 39, MatrixKind.COMPRESSED)
                                .asInstanceOf [RleMatrixD]
    val siteInfo     = solarRel.colName

    val mu  = VectorD (for (j <- solarMat.range2) yield
                                  solarMat.col (j).mean)
    val cmu = RleVectorD (for (j <- solarRleMat.range2) yield
                                  solarRleMat.col (j).mean)

    println ("Highest average computations")
    println (s" Dense matrix: Site = ${siteInfo (mu.argmax ())},
                              average = ${mu.max ()}")
    println (s" Rle matrix:   Site = ${siteInfo (cmu.argmax ())},
                              average = ${cmu.max ()}")
    println ("Lowest average computations")
    println (s" Dense matrix: Site = ${siteInfo (mu.argmin ())},
                              average = ${mu.min ()}")
    println (s" Rle matrix:   Site = ${siteInfo (cmu.argmin ())},
                              average = ${cmu.min ()}")

    println (s" Size of columns in dense matrix: ${solarMat.dim1}")
    println (s" Size of columns in Rle: ${solarRleMat.csize}")

    val itr = 6
    val denseTimeVec  = new VectorD (itr)
    val rleTimeVec    = new RleVectorD (itr)
    for (i <- 0 until itr) {
        denseTimeVec (i) = timed { VectorD (for (j <- solarMat.range2)
                                   yield solarMat.col (j).mean) }._2
        rleTimeVec (i)   = timed { RleVectorD (for (j <- solarRleMat.range2)
                                   yield solarRleMat.col (j).mean) }._2
    } // for
    println ("Avg time taken by Dense: "+denseTimeVec.slice (1).mean+" ms")
    println ("Avg time taken by Rle:   "+rleTimeVec.slice (1).mean+" ms")

} // SolarRadiation object
```

B.2 Output

The output of the SolarRadiation application is shown in the figure below. We see that in this case, there is no gain in space after compression. For example, the size of the first column is 153595. Since each triplet takes 16 bytes, the total space required post compression for the first column would be 2457520 bytes. Without compression, the first column would require 2103744 bytes.

However, we see good improvement in computation time. Computating the mean vector from compressed data is around 4 times faster than computing from dense data.



```
> run-main apps.analytics.SolarRadiation
[info] Running apps.analytics.SolarRadiation
[info] Highest average computations
[info]  Dense matrix: Site = DAGGET (23161), average = 244.91600879194428
[info]  Rle matrix:   Site = DAGGET (23161), average = 244.91600879194428
[info] Lowest average computations
[info]  Dense matrix: Site = CHARLESTON (13866), average = 146.63702807946214
[info]  Rle matrix:   Site = CHARLESTON (13866), average = 146.63702807946214
[info]  Size of columns in dense matrix:                      262968
[info]  Size of columns in Rle:               VectorI(153595,   152058, 151548, 152635, 15290
5, 152575, 151578, 151495, 152545, 153085, 152358, 152501, 152598,       152538, 152838, 1524
18, 153018, 151278, 152508, 153565, 153235, 151098, 153415, 153745, 152185, 152958, 152058, 1
50678, 153498, 153528, 152831, 152178,       153198, 152328, 152118, 152088, 152658, 152748,
150888, 152762)
[info] Avg time taken by Dense: 248.94580000000002 ms
[info] Avg time taken by Rle:   63.74899999999999 ms
[success] Total time: 27 s, completed Dec 16, 2016 12:32:48 AM
```

Figure B.1: Output of SolarRadiation application

Appendix C

Census Application

Census application included as part of ScalaTion performs basic analytics on US census data of the year 1990. The dataset is available for download from the UCI Machine Learning Repository webpage[1]. The size of the dataset is around 360MB and the following sbt command can be used to run the application.

```
run-main apps.analytics.Census
```

The code for the application is shown in C.1. Data is read from the CSV file and a dense matrix (*censusMat*) is created. A Rle (*censusRleMat*) matrix is created from the dense matrix. Creating a relation directly currently gives slow performance and needs to be looked into to make it faster. We calculate the number of people who are born in the US and who already have a job. This computation is done via both the dense and the Rle matrices. We finally calculate the time required to do this calculation. The average time for 5 iterations is considered. The time required to build the Rle and the dense matrices are also recorded.

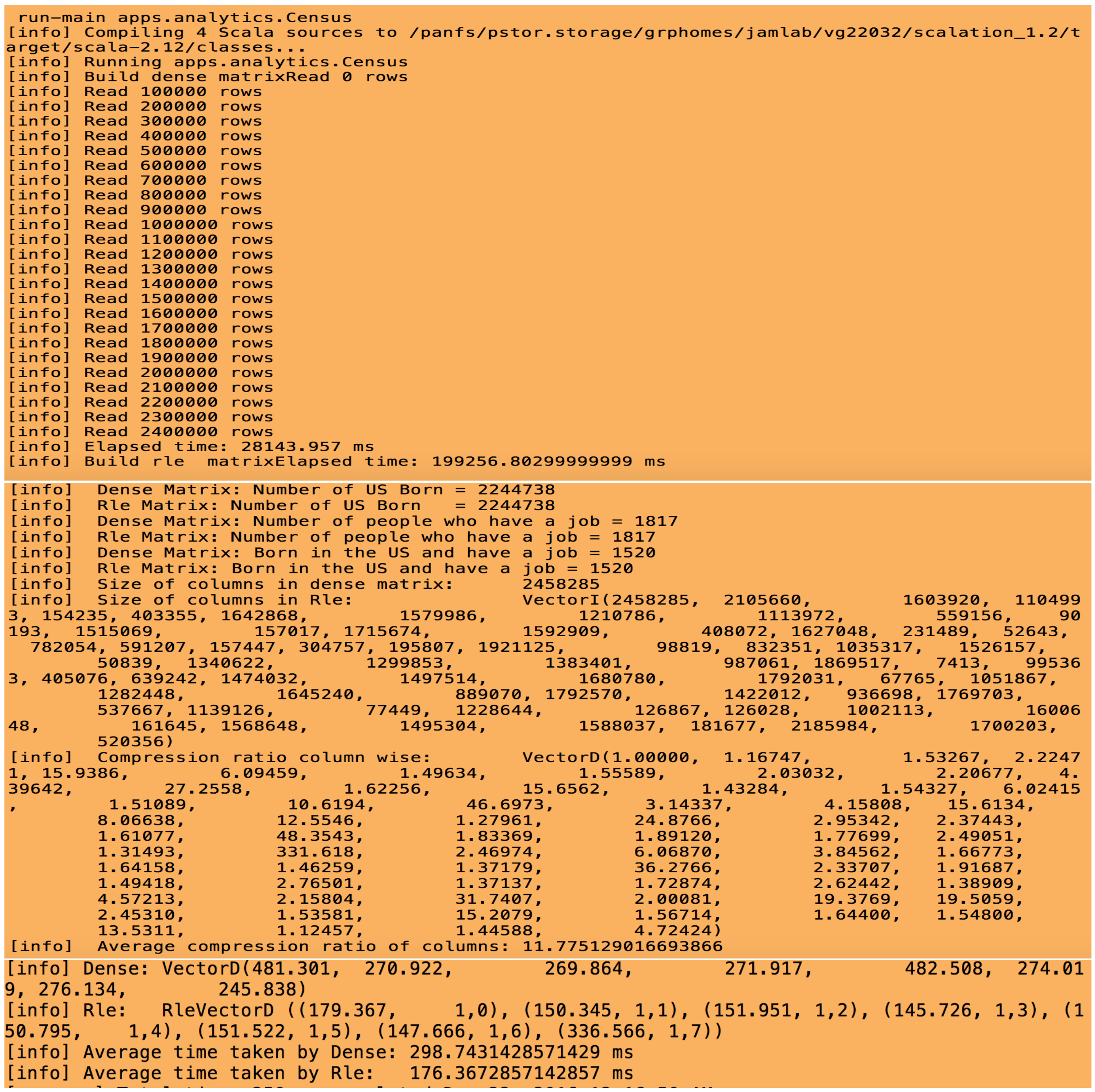
[1]Census Data: `https://archive.ics.uci.edu/ml/datasets/US+Census+Data+(1990)`

C.1 Code

```
object Census extends App
{
    val fName = "USCensus1990_data.csv"

    // FIX : to check why it is slow
    //  var censusRel = Relation (fName, "census", "D" * 69, -1, ",")
    //  val censusMat = censusRel.toMatriD (1 to 68)
    //  val censusRleMat = censusRel.toMatriD (1 to 68,
    //                              MatrixKind.COMPRESSED)
    //                              .asInstanceOf [RleMatrixD]

    print ("Build dense matrix")
    val censusMat = time {MatrixD (fName, 1)}
    print ("Build rle  matrix")
    val censusRleMat = time {RleMatrixD (censusMat)}

    val jobDense = censusMat.col (4). filterPos (x => x == 1.0)
    val jobRle   = censusRleMat.col (4). filterPos (x => x == 1.0)
    val usBornDense = censusMat.col (5). filterPos (x => x == 0.0)
    val usBornRle   = censusRleMat.col (5).filterPos (x => x == 0.0)

    // Print the statistics
    println (s" Dense Matrix: Number of US Born =
                            ${usBornDense.size}")
    println (s" Rle Matrix: Number of US Born =
                            ${usBornRle.size}")
    println (s" Dense Matrix: Number of people who have a job =
                                         ${jobDense.size}")
    println (s" Rle Matrix: Number of people who have a job =
                                         ${jobRle.size}")
    println (s" Dense Matrix: Born in the US and have a job =
                     ${usBornDense.intersect(jobDense).size}")
    println (s" Rle Matrix: Born in the US and have a job =
                       ${usBornRle.intersect(jobRle).size}")

    // Space information before and after RLE compression
    val compRatios =   (censusRleMat.csize.toDense.toDouble /
                        censusMat.dim1.toDouble).recip
    val avgcompRatios = compRatios.mean

    println (s" Size of columns in dense matrix: ${censusMat.dim1}")
    println (s" Size of columns in Rle: ${censusRleMat.csize}")
    println (s" Compression ratio column wise: ${compRatios}")
    println (s" Average compression ratio of columns: ${avgcompRatios}")

    //Compute the time
    val itr = 6
    val denseTimeVec  = new VectorD (itr)
    val rleTimeVec    = new RleVectorD (itr)

    for (i <- 0 until itr) {
         denseTimeVec (i) = timed { val usBornDense = censusMat.col (5)
                                                      .filterPos (x => x == 0.0)
                                    val jobDense = censusMat.col (4)
                                                   .filterPos (x => x == 1.0)
                               }._2
        rleTimeVec (i)   = timed { val usBornRle = censusRleMat.col (5)
                                                   .filterPos (x => x == 0.0)
                                   val jobRle = censusRleMat.col (4)
                                                . filterPos (x => x == 1.0)
                              }._2
    } // for

    println (s"Dense: $denseTimeVec")
    println (s"Rle:   $rleTimeVec")
    println ("Average time taken by Dense: "
              +denseTimeVec.slice(1).mean+" ms")
    println ("Average time taken by Rle: "
              +rleTimeVec.slice(1).mean+" ms")

} // Census object
```

C.2 Output

The output of the Census application is shown in the figure below. We see that the average compression ratio across all the columns is close to 12. We also acheive improvement in computation time. Computation via Rle is around 1.7 times faster than computation via dense.



```
 run-main apps.analytics.Census
[info] Compiling 4 Scala sources to /panfs/pstor.storage/grphomes/jamlab/vg22032/scalation_1.2/t
arget/scala-2.12/classes...
[info] Running apps.analytics.Census
[info] Build dense matrixRead 0 rows
[info] Read 100000 rows
[info] Read 200000 rows
[info] Read 300000 rows
[info] Read 400000 rows
[info] Read 500000 rows
[info] Read 600000 rows
[info] Read 700000 rows
[info] Read 800000 rows
[info] Read 900000 rows
[info] Read 1000000 rows
[info] Read 1100000 rows
[info] Read 1200000 rows
[info] Read 1300000 rows
[info] Read 1400000 rows
[info] Read 1500000 rows
[info] Read 1600000 rows
[info] Read 1700000 rows
[info] Read 1800000 rows
[info] Read 1900000 rows
[info] Read 2000000 rows
[info] Read 2100000 rows
[info] Read 2200000 rows
[info] Read 2300000 rows
[info] Read 2400000 rows
[info] Elapsed time: 28143.957 ms
[info] Build rle  matrixElapsed time: 199256.80299999999 ms
[info]  Dense Matrix: Number of US Born = 2244738
[info]  Rle Matrix: Number of US Born   = 2244738
[info]  Dense Matrix: Number of people who have a job = 1817
[info]  Rle Matrix: Number of people who have a job = 1817
[info]  Dense Matrix: Born in the US and have a job = 1520
[info]  Rle Matrix: Born in the US and have a job = 1520
[info]  Size of columns in dense matrix:      2458285
[info]  Size of columns in Rle:               VectorI(2458285,  2105660,        1603920,  110499
3, 154235, 403355, 1642868,        1579986,        1210786,        1113972,        559156,    90
193,  1515069,        157017, 1715674,        1592909,        408072, 1627048,  231489,  52643,
  782054, 591207, 157447, 304757, 195807, 1921125,        98819,  832351, 1035317,   1526157,
       50839,  1340622,        1299853,        1383401,        987061, 1869517,   7413,   99536
3, 405076, 639242, 1474032,        1497514,        1680780,        1792031,   67765,  1051867,
       1282448,        1645240,        889070, 1792570,        1422012,   936698, 1769703,
       537667, 1139126,        77449,  1228644,        126867, 126028,     1002113,        16006
48,       161645, 1568648,        1495304,        1588037,  181677,  2185984,        1700203,
       520356)
[info]  Compression ratio column wise:        VectorD(1.00000,  1.16747,        1.53267,  2.2247
1, 15.9386,        6.09459,        1.49634,        1.55589,        2.03032,        2.20677,   4.
39642,        27.2558,        1.62256,        15.6562,        1.43284,        1.54327,   6.02415
,        1.51089,        10.6194,        46.6973,        3.14337,        4.15808,   15.6134,
       8.06638,        12.5546,        1.27961,        24.8766,        2.95342,   2.37443,
       1.61077,        48.3543,        1.83369,        1.89120,        1.77699,   2.49051,
       1.31493,        331.618,        2.46974,        6.06870,        3.84562,   1.66773,
       1.64158,        1.46259,        1.37179,        36.2766,        2.33707,   1.91687,
       1.49418,        2.76501,        1.37137,        1.72874,        2.62442,   1.38909,
       4.57213,        2.15804,        31.7407,        2.00081,        19.3769,   19.5059,
       2.45310,        1.53581,        15.2079,        1.56714,        1.64400,   1.54800,
       13.5311,        1.12457,        1.44588,        4.72424)
[info]  Average compression ratio of columns: 11.775129016693866
[info] Dense: VectorD(481.301,  270.922,        269.864,        271.917,        482.508,  274.01
9, 276.134,        245.838)
[info] Rle:   RleVectorD ((179.367,     1,0), (150.345, 1,1), (151.951, 1,2), (145.726, 1,3), (1
50.795,    1,4), (151.522, 1,5), (147.666, 1,6), (336.566, 1,7))
[info] Average time taken by Dense: 298.7431428571429 ms
[info] Average time taken by Rle:   176.3672857142857 ms
```

Figure C.1: Output of Census application

Appendix D

Script to check system load and execute

The following Python script is used to check the system load before executing a Scala job. It uses the `uptime` command to know about the system load. It gives you the average load over the past 1, 5 and 15 minutes. The load over the past 1 and 5 minutes is checked to see whether it is less than the *cutoff*. If yes, the job is executed. If no, the script checks again after 5 seconds for a maximum of 10 times.

```
#!/usr/local/bin/python2.7

import subprocess
import time

command = "scala -cp $SCALATION_CLASSES testing.linalgebra.RleVectorD_T"
cutoff = 4
p = subprocess.Popen("uptime", stdout=subprocess.PIPE, shell=True)
(output, err) = p.communicate()
loadstr = (output.strip().split(":"))[-1]
load    = loadstr.split(",")       #delimeter for  linux
#load   = loadstr.split(" ")       #delimeter for mac

def checkAndRun(load1,load2):
    for i in xrange(10):
        print (i)
        if (load1 < cutoff and load2 < cutoff):
            q = subprocess.Popen(command,stdout = subprocess.PIPE,
                                 shell = True)
            (output,error) = q.communicate()
            print (output)
            return
        time.sleep(5)

checkAndRun(float(load[-3]),float(load[-2]))
```

Appendix E

Developer ToDo List

1. Storing the Relation in compressed form.
2. Produce transpose directly from the Relation.
3. Speed up loading of Relation.
4. Modify census example to include columns where it makes sense to apply operators like mean.
5. Provide examples that intermix relational algebra and linear algebra.
6. Ability to read from compressed files.
7. Ability to read from HDFS.
8. Ability to read from RDD.
9. Consider using Apache Arrow.
10. Place database on server with high speed connection for analytics. e.g. try RDMA.
11. Find a more efficient update method.
12. Compare various compression techniques. e.g. Dictionary Encoding, RLE and Sparsity.
13. Try sorting the data to improve RLE compression.
14. Investigate on the drop in speed up factor for variance operation when the number of threads is 10 or more.