

# An Adaptive Tuning Strategy on Spark Based on In-memory Computation Characteristics

Yao Zhao\*, Fei Hu\*, and Haopeng Chen\*

\*School of Electronic Information and Electrical Engineering  
Shanghai Jiao Tong University, 800 Dongchuan Road, Shanghai, China  
GuiLong@sjtu.edu.cn, hu.fei@sjtu.edu.cn, chen-hp@sjtu.edu.cn

**Abstract**—We present an adaptive tuning method to improve Spark performance, especially for its in-memory computation. This manner serves one purpose: making a better use of memory reasonably through adaptively adopting suitable category based on Spark application runtime statistics on different working sets. This solution works in two steps. Firstly, it collects run-time statistics dynamically and stores them in round-robin structures to save memory. Secondly, it can change system storage category based on these statistics. Additionally we focus on serialization strategy optimization. For this purpose we test Spark integrated serialization algorithms: Java and Kryo serialization algorithms, and make a comparison of their performance. In order to gain flexibility we change Spark serialization mechanism by setting the default serialization unit from one RDD to one block. In this way, for the case which RDD has huge amount of blocks our solution can use different serialization algorithms to serialize different blocks in one RDD. We show that our solution is expressive enough to obtain 2x speedup than original Spark when there is inadequate memory resource.

**Keywords**—Spark, adaptive tuning, in-memory computation, serialization

## I. INTRODUCTION

Before Hadoop [1] framework comes out, there is no efficient distributed framework which can deal with big data excellently. Hadoop proposes MapReduce [2] computation model which divides computation process into two types of operations: Map and Reduce. In this way Hadoop simplifies programming and also provides high performance parallelism, fault-tolerant and load-balance. Although Hadoop framework performs so well in many types of computation jobs, when facing iterative machine learning algorithms and interactive data queries it incurs performance bottlenecks [6].

To improve performance in these two cases, Spark [3] is produced based on Hadoop. And it has been widely adopted to deal with big data due to its well-designed fault-tolerant and highly efficient data reuse mechanism. Spark implements a kind of in-memory computation abstraction called Resilient Distributed Datasets(RDDs), which can be persisted in memory. In this way it can reuse the cached in-memory intermediate result time-efficiently instead of getting it from disks which leads to substantial overheads. Spark outperforms so well in processing iterative machine learning algorithm, i.e. Graph X [7], and interactive data query that it can obtain 40x times speedups than Hadoop [8].

The key point of Spark memory computation is to store the intermediate result in memory [9]. More data cached in memory, faster the job will be processed. But in case that there is no enough space to cache all middle results in memory, spark has to spill the intermediate data into local disks or other external stable storage systems. This situation commonly occurs in spark applications, especially for data-intensive workloads. Because of this reason, spark applications are bottlenecked by memory size, network bandwidth, I/O speed, etc. So far Spark has not good solutions for this problem, it just gives some storage levels for developers to choose, we can choose to store RDD in memory, both memory and disk, some none-heap memory, i.e. Tachyon [10] when there is not enough space. If we choose MEMORY\_ONLY option the extra RDD partitions that can not be fit in memory will be lost and it will be recomputed on the fly each time they are needed. Additionally, before storing RDDs, Spark provides serialization libraries to serialize them. MEMORY\_ONLY\_SER is similar to MEMORY\_ONLY, except that it will store serialized Java objects in JVM. This is more space-efficient than deserialized objects, especially when using some efficient serializer, but more CPU-intensive to read. Developers have more freedom to make decision which storage categories will be used. Furthermore, Spark can compress RDDs with third-part compression algorithms, which will cause some CPU overhead, to save more space.

Spark is implemented by scala language which runs on JVM and it provides plenty of configuration parameters for developer. To make applications run faster and more efficiently, it is necessary to be clear of the capability of the cluster machines. After that we can make decisions how much memory the JVM heap will use, which Garbage Collector is the best choice for current working set in this computer, etc.

Although spark provides lots of tuning categories, it is hard to give a suitable solution for cluster resource allocation. To make a better use of the system resource, users must have good experience. In this paper we present an adaptive tuning strategy for Spark, concentrating on improving performance when there appears bottlenecks of system resources. The key point of this method is to make a choice whether it is necessary to change current category based on system resource at the time when there comes some bottlenecks for CPU or memory resource.

This paper begins with a overview of serialization algorithms used in Spark in section II. Then we propose Adaptive Serialization Strategy (ASS) in section III, the implementation of ASS in section IV, and the experimental results in section V. Finally, we discuss the related work in section VI, future work in section VII and conclusion in section VIII.

## II. COMPARISON OF SERIALIZATION ALGORITHMS

Serialization has an important influence on Spark performance. In this section we provide a comparison of two serialization algorithms integrated by Spark. Firstly, we introduce Java and Kryo serialization algorithms and then compare their performance.

### A. Serialization Algorithms

1) *Java serialization*: This is the default serialization option. In this mode, Spark can serialize objects using Java's ObjectOutputStream framework. Any class which implements java.io.Serializable interface can be used by Spark application. And java.io.Externalizable interface is used to control the performance of concrete serialization more closely. This serialization is flexible but often quite slow, and usually produces large serialized formats for many classes.

2) *Kryo serialization*: Kryo [5] is a fast and efficient object graph serialization framework for Java. It is faster and more compact than Java serialization, but more CPU-consuming and does not support all Serialization types. Beyond this, for best performance, target classes must be registered in Kryo.

### B. Comparison of Performance

Fig. 1 shows the comparison of these two serialization performance in time while Fig. 2 shows the performance in size. We can see that Kryo serialization algorithm obtains at least 7x times speedups than Java and saves at least 3x times memory space than Java.

## III. ADAPTIVE STRATEGY

We propose an adaptive serialization strategy in this section. At the first place we define a kind of fine-grained serialization concept and then introduce adaptive serialization strategy.

### A. Fine-grained serialization

Spark implements computation by establishing DAG Task Graph: when a job comes Spark will divide it into many stages, each stage contains lots of RDDs, each RDD consists of numbers of partitions. All RDDs in one stage make up a DAG scheduler graph based on their lineage, so are the stages. The last RDD in one stage is called final RDD, the partition number of this RDD decides how many tasks will be generated. Once a task is finished Spark will use the configured serialization method (default option is Java serialization, you can use `conf.set(spark.serializer, serializer-name)` command to set other serializer) to serialize the result. Then the serialized result

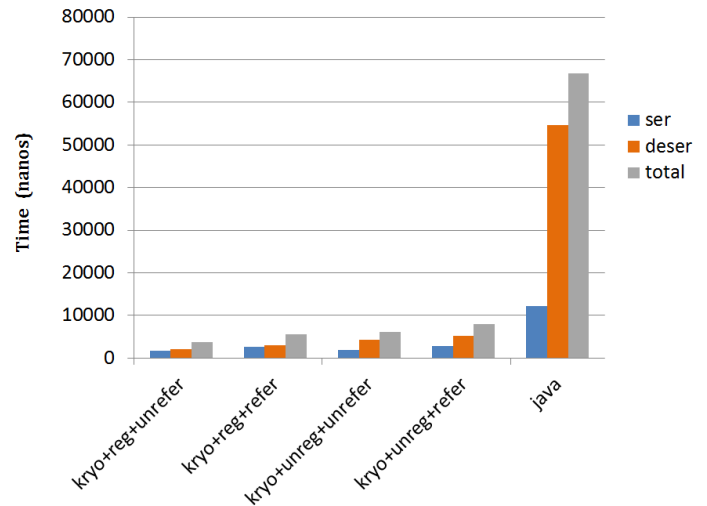


Fig. 1. Comparison of Java and Kryo serialization algorithm performance in time. "reg" shorts for registration and "refer" shorts for reference. We compare their serialization, deserialization and total time.

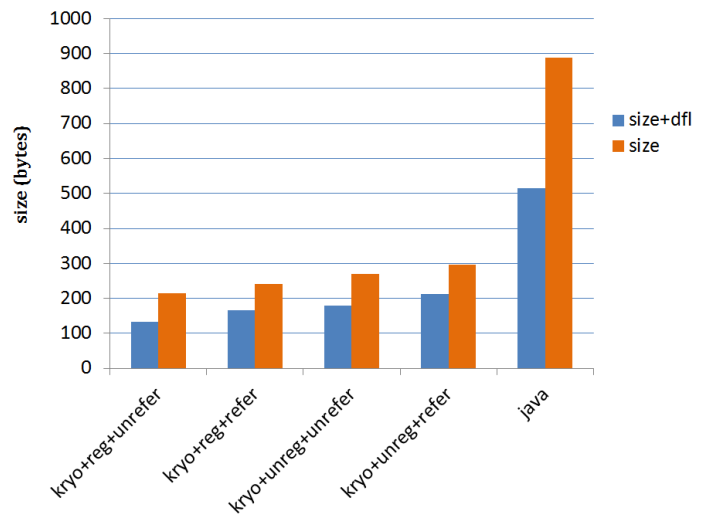


Fig. 2. Comparison of Java and Kryo serialization algorithm performance in size. "size+defl" means after serialization we use zlib compression library with level 6 to compress the result.

will be sent to driver node and used to computation. If this RDD is declared to be persisted it will be stored in memory. This is RDD write process. RDD read process is that the Reader gets a RDD and then reads each partition in it one by one.

Spark uses one RDD as serialization unit, but in the above process we can see that the smallest serialization unit can be assigned to one partition which means we can use different serialization methods to serialize different partitions in one RDD. In this way we can obtain more flexible choices in RDD serialization.

### B. Adaptive Serialization Strategy

It will produce huge statistics at Spark run-time, such as memory usage, CPU usage, size of RDD, serialization and deserialization cost, etc. These statistics can be used to

optimize Spark running mechanism, i.e. Shark [4] implements a Partial DAG Execution mechanism to alter DAG graph based on these statistics. Similarly, to take advantage of these data we provide an adaptive serialization tuning method called Adaptive Serialization Strategy (ASS). ASS will collect statistics while a job is running. To support ASS, we expanded Spark to support dynamic changing serialization algorithm based on statistics collected at run-time.

We currently adopt ASS during the lifetime of a Spark application and change serialization algorithm at the time a task finished, when a partition would be generated, since one partition corresponding to one block that we set to the smallest serialization unit in Spark. By default, Spark uses configured serializer to serialize RDDs. ASS modifies this mechanism in three ways. Firstly, it gathers statistics at run-time. Secondly, it can alter the configured serializer based on these statistics. Finally, it changes the smallest serialization unit from one RDD to one partition.

The statistics are collectable through using an accumulator API. Some types of these statistics include:

- 1) JVM heap information, i.e., memory usage
- 2) Cached middle output size and CPU usage, which can be used to serialization optimization.
- 3) Partition size, I/O speed and serialization performance, which can be used to estimate cost about different strategies.

In order to save space we use round-robin structure to store these statistics. Adaptive Serialization Strategy provides an automatic serialization selection mechanism based on data collected dynamically instead of manual configuration optimization.

#### IV. IMPLEMENTATION

While implementing ASS, we discover that the amount of memory used to store statistics has significant impacts on system performance. To reduce memory overhead, we use round-robin structure to store these statistics.

##### A. Storage Management

We use a HashMap to store blockId to serializerId mappings and another HashMap to hold serializerId to serializer mappings. Since we just have two serializer (so far, Spark just integrates Java and Kryo serialization methods) and we can remove a blockId-serializerId mapping after corresponding blockId recycled, it will save much more space. At Spark application run-time there will be great number of statistics to be collected. Beyond doubt these statistics will cost large number of memory to persist. For space-efficiency, we use lossy compression to record these statistics, limiting their size to 3-4KB per task.

##### B. Strategy Transformation

Since we set one block as the smallest serialization unit in Spark during task run time current serialization category

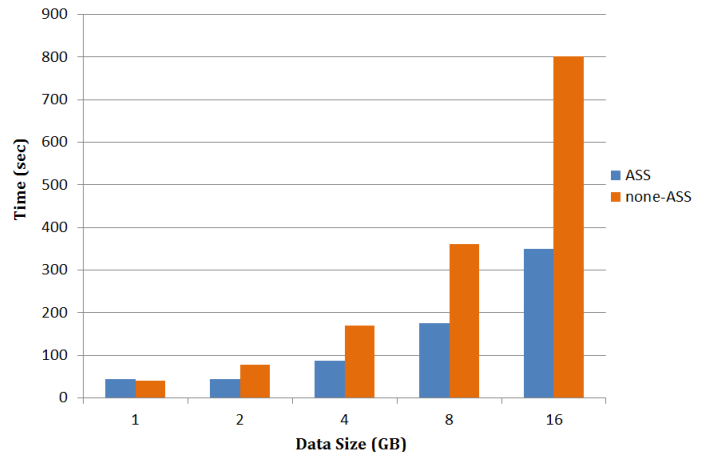


Fig. 3. A simpleCount algorithm performance comparison.

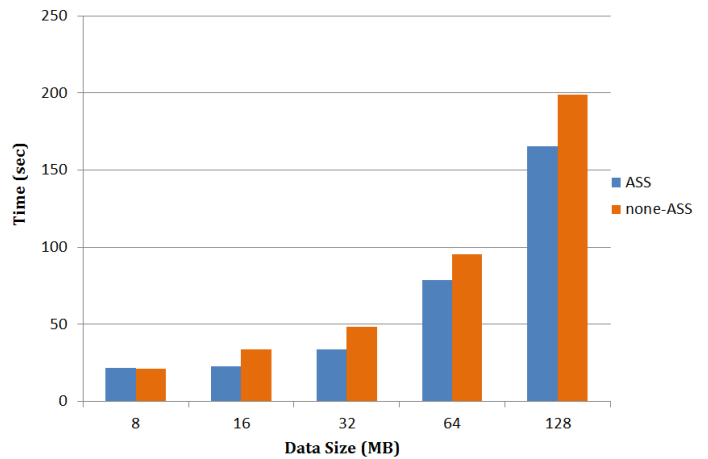


Fig. 4. PageRank performance comparison.

cannot be changed. After a task complemented Adaptive Serialization Strategy will work: using collected statistics ASS can detect the time when system resources bottlenecks appears, at that time it will choose a suitable serializer.

#### V. EVALUATION

We have implemented an expansion with Adaptive Serialization Strategy on the version of Spark(1.5.1). We evaluated its performance with experiments conducted on 4-nodes cluster. Each node is a virtual machine with one Intel Xeon Processor E5 CPU, 2GB RAM and 64GB disk allocation on VMware Workstation. We run two use cases with MEMORY\_ONLY storage level.

##### A. SimpleCount

We take a simple count algorithm, which finds top 10 most popular IP, for example and run sample data with different sizes which are 1GB, 2GB, 4GB, 8GB and 16GB. Each node has 1GB executor memory for all applications. The algorithm shows in algorithm 1. Fig. 3 demonstrates that with the increment of data Adaptive Serialization Strategy solution performance exceeds original Spark, especially when the data

size grows to 16GB our solution gets about 2x times speedup than the original one. We attribute the speedup to Kryo excellent performance both in serialization and deserialization. Additionally, fine-grained serialization method makes contribution to the speedup too.

---

**Algorithm 1** Simple Count
 

---

**Require:** FilePath

**Ensure:** top 10 most popular IP

```

1: val output = sc.textFile(FilePath)
2: .map(l=>l.split("\t")).map(l=>(l(2),1))
3: .cache()
4: .reduceByKey(_+_2).filter(x=>x._2>100)
5: .collect()
6: .sortWith(_._2>_._2)
7: .take(10)
8: .map(x=>(x._1,x._2))

```

---



---

**Algorithm 2** PageRank
 

---

**Require:** FilePath Iteration

**Ensure:** pagerank

```

1: val lines = sc.textFile(FilePath, 1)
2: val links = lines.map { s =>
3:   val parts = s.split("\\s+")
4:   (parts(0), parts(1))
5: }.distinct().groupByKey().cache()
6: var ranks = links.mapValues(v => 1.0)
7:
8: for (i <- 1 to Iteration) {
9:   val contribs = links.join(ranks).values.flatMap {
10:     case (urls, rank) =>
11:       val size = urls.size
12:       urls.map(url => (url, rank / size))
13:   }
14:   ranks = contribs.reduceByKey(_ + _)
15:   .mapValues(0.15 + 0.85 * _)
16: }
17:
18: val output = ranks.collect()

```

---

### B. PageRank

We run 10 iterations of the PageRank algorithm provided by Spark official as another use case to test the performance of Adaptive Serialization Strategy solution on different data sets with sizes ranging from 8MB to 128MB. As the results shows in Fig. 4 we can see that our solution does not outperform much better than original Spark in running PageRank algorithm. We give three reasons to explain this phenomenon: First reason is that PageRank is a classical iterative machine learning algorithm whose performance relies on the speed of reading middle output files. From PageRank algorithm shows in algorithm 2 we can see it uses groupByKey and reduceByKey transformations that will produce great number of middle output files. The second reason is said to be that there

will be many shuffle operations produced by reduceByKey and groupByKey transformation which is an expensive operation since it creates in-memory data structures to organize records. When these intermediate output do not fit in memory Spark will spill them to disk or got lost(in this experiment because we use MEMORY\_ONLY storage level, the data will be lost), incurring the additional overhead of recomputing the lost data. Last but not at least, in order to serialize these huge amount of middle output data Kryo will cost more CPU resource than Java. But on the whole our solution exceeds original Spark in memory-intensive workloads.

## VI. RELATED WORK

### A. Memory Tuning

1) *Cache Size Tuning*: The amount of memory used for caching RDDs plays an important role in Spark memory tuning. Spark divides the memory usage into two types: cache usage and computation usage. By default, Spark uses 60% of the configured executor memory(set by spark.executor.memory) to persist RDDs. This means that the rest 40% of configured memory is used for objects created during task execution.

If JVM GC becomes frequently at run time it suggests memory used for computation usage is insufficient. In this case we can reduce the ratio of memory used for caching RDDs to low GC overhead. Conversely, if GC is not operated so frequently we can improve the memory used for caching RDDs. To change this, we can call conf.set("spark.storage.memoryFraction", Ratio) on Spark-Conf.

2) *Advanced GC Tuning*: As the most concerned bottleneck in spark computation, the utilization of memory dominates the performance of spark clusters, especially for iterative machine learning and interactive data query cases. Spark application runs on JVM which maintains JVM Heap ,a data structure which be used to store runtime-needed data and computation jobs. Also, JVM Heap space is divided into two parts: Young Gen and Old Gen. Furthermore, the Young Gen consists of three regions[Eden,Survivor1,Survivor2].If Young Gen is full, JVM executes a minor Garbage Collection(minor GC) to clean some space for new-generated objects. If Old Gen is full, JVM takes a Full Garbage Collection(full GC).Minor GC happens a lot but each GC takes a little time, on the contrary Full GC occurs unfrequently but each costs much more time. Obviously full GC gives bad influence on system performance. Spark memory tuning focuses on reducing Full GC times. If Full GC happens frequently we can turn up the old Gen size.

### B. Compressed RDD Storage

We can choose to use Spark integrated compression algorithm to reduce more space overhead or not. There are three well-performed compression libraries delivered by Spark: Snappy [11], LZ4 [12] and LZ4 [13]. Compression is more

space-efficiency but more CPU-intensive. To make a balance between memory and CPU resource we test the performance of above three compression algorithms based on different data types and give a solution based on different workloads.

## VII. FUTURE WORK

For best performance Kryo requires users to register the classes they will use in advance. So far, Spark doesn't support automatic registering user self-defined or other Third-Party Libraries classes, this is the only reason why Kryo serialization method has not been set to the default one. Next step we will change Spark Kryo mechanism to make it support common user self-defined or Third-Party Libraries classes self-registration. Similarly, we can use adaptive tuning method to compression and memory advanced tuning. These are the works we will research next.

## VIII. CONCLUSION

With the increment of Big Data the requirement of real-time performance is becoming more and more important. To meet the real-time demand memory computation will make very big difference. In this paper We adopted Adaptive Serialization Strategy on Spark to get more efficient memory utilization. We compared original Spark and Spark with Adaptive Serialization Strategy performance on two workloads. From the result we can see that our solution appeared more efficient than original Spark in memory-intensive workload.

## REFERENCES

- [1] K. Shvachko, K. Hairong, S. Radia. *The Hadoop Distributed File System*, Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on 2010, 3(7):1-10.
- [2] D. Jeffrey, G. Sanjay. Communications of the ACM - 50th anniversary issue: 1958 - 2008. MapReduce: simplified data processing on large clusters, 2008, 51(1):107-113
- [3] Z. Matei, C. Mosharaf, D. Tathagata. NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation[C]. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing, 2012, 2-2.
- [4] C. Engle, A. Lupher, R. Xin. Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. ACM[C]. Shark: fast data analysis using coarse-grained distributed memory, 2012: 689-692.
- [5] Kryo - fast, efficient java serialization and cloning. <http://code.google.com/p/kryo/>. Accessed: 6 Jun 2012.
- [6] G. Katarina, H. Michael, A. H. Wilson. *Challenges for MapReduce in Big Data*, Proceeding of the IEEE 10th 2014 World Congress on Services (SERVICES 2014)[J].
- [7] J. E. Gonzalez, R. S. Xin, A. Dave. *GraphX: Graph processing in a distributed dataflow framework*, Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)[C].
- [8] M. Zaharia, M. Chowdhury, T. Das. *Fast and interactive analytics over Hadoop data with Spark*, USENIX[C].
- [9] T. Jiang, Q. Zhang, R. Hou. *Understanding the behavior of in-memory computing workloads*, Workload Characterization (IISWC), 2014 IEEE International Symposium on. IEEE[C].
- [10] H. Li, A. Ghodsi, M. Zaharia. *Tachyon: Reliable, memory speed storage for cluster computing frameworks*, Proceedings of the ACM Symposium on Cloud Computing:1-15.
- [11] J. H. Um, S. Lee, T. H. Kim, C. H. Jeong, K. Seo, J. Park, H. Jung. *H MapReduce-based Bulk-Loading Algorithm for Fast Search for Billions of Triples*, In Computer Science and its Applications (pp. 1139-1145). Springer Berlin Heidelberg.
- [12] T. Gagie, P. Gawrychowski, S. J. Puglisi. *Approximate pattern matching in LZ77-compressed texts.*, of Discrete Algorithms, 2015, 32, 64-68.
- [13] R. D. Masram. *Efficient Selection of Compression-Encryption Algorithms for Securing Data Based on Various Parameters*, Doctoral dissertation, College of Engineering, Pune, 2014.



Yao Zhao. He was born in GuiZhou Province, China, in 1989 and admitted to Xi'an Jiao Tong university with passing the undergraduate entrance examination in 2008. He got his Bachelor degree in computer science and technology, Xian Jiao Tong University in 2012 Xian, Shanxi Province, China. And now he is working for Master degree in software engineering, Shanghai Jiao Tong University, Shanghai, China. He is interested in machine learning algorithm, natural language identification, distributed computer system and big data processing. Mr. Zhao is working for his Master degree in Shanghai Jiao Tong University.



Fei Hu. He received his Bachelor degree from Department of computer software, Northwest University, Xian, Shanxi Province, China in 1990 and received his Master degree of computer science and engineering and Ph.D of Precision Guidance and Control both from Northwest Polytechnical University, Xian, Shanxi Province, China in 1993 and 1998.

He has worked in Department of Computer Science and Engineering, Northwestern Polytechnical University lecturer, from 1993 to 2006. From 2006/9 to now he has worked in School of Software, Shanghai Jiao Tong University. His Publications are as follows: Zhiyang Zhang, Fei Hu and Jian Li, Autonomous Flight Control System Designed for Small-Scale Helicopter Based on Approximate Dynamic Inversion, The 3rd IEEE International Conference on Advanced Computer Control (ICACC 2011), 18th to 20th January 2011, Harbin, China.



Haopeng Chen. He received his Ph.D degree from Department of Computer Science and Engineering, Northwestern Polytechnical University, Xian, Shanxi Province, China in 2001. He has worked in School of Software, Shanghai Jiao Tong University since 2004 after he finished his two-year postdoctoral research job in Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China. He got the position of Associate Professor in 2008. In 2010, he studied and researched in Georgia Institute of Technology as a visiting scholar.

His research group focuses on Distributed Computing and Software Engineering. They have kept researching on Web Services, Web 2.0, Java EE, .NET, and SOA for several years. Recently, they are also interested in cloud computing and researching on the relevant areas, such as cloud federation, resource management, dynamic scaling up and down, and so on.