Execution of Recursive Queries in Apache Spark

Pavlos Katsogridakis[1 2], Sofia Papagiannaki[1], and Polyvios Pratikakis[1]

[1] Institute of Computer Science, Foundation for Research and Technology — Hellas
[2] Computer Science Department, University of Crete, Greece

Abstract. MapReduce environments offer great scalability by restricting the programming model to only map and reduce operators. This abstraction simplifies many difficult problems occuring in generic distributed computations like fault tolerance and synchronization, hiding them from the programmer. There are, however, algorithms that cannot be easily or efficiently expressed in MapReduce, such as recursive functions. In this paper we extend the Apache Spark runtime so that it can support recursive queries. We also introduce a new parallel and more lightweight scheduling mechanism, ideal for scheduling a very large set of tiny tasks. We implemented the aformentioned scheduler and found that it simplifies the code for recursive computation and can perform up to 2.1× faster than the default Spark scheduler.

1 Introduction

Modern analytics queries consist of complex computations operated on massive amounts of data. By restricting the programming model to only map and reduce, or equivalent operators, MapReduce [5] clusters scale out because they do not need to track task dependencies, have simpler communication patterns, and are tolerant to executor and even master node failures. However, this simplified programming model cannot easily express some applications, including applications with nested parallelism or hierarchical decomposition of the data. When faced with such algorithms, programmers often develop iterative versions that translate recursion into worklist algorithms. This may be inefficient as it introduces unnecessary barriers from one iteration to the next, and can be unintuitive and complicated to code.

The Barnes-Hut simulation [3] is an approximation algorithm for particle simulation with nested parallelism that cannot be easily expressed using flat map-reduce operators. In its simple two-dimensional version, the simulation first recursively splits the space into four quads and computes the center of mass for each, resulting in a tree structure that represents the whole space. In its second phase, it uses the tree of all the centers of mass to compute the forces applied to each body in the space. That reduces the N-Body problem complexity from $O(n^2)$ to $O(nlogn)$, by grouping all objects in distant quads into one force.

Fig. 1 shows a simplified version of the recursive query that implements the second phase of the algorithm. Function `calcForces` traverses the tree computed during the first phase, to calculate all the forces applied to a single `particle`.

```
1 def calcForces(particle, tree) = {
2   if(isFar(particle, tree, THETA))
3       Array(force(particle, tree))
4   else
5       tree.map( child => {
6        calcForces(particle, child)
7       }).flatten
8 }
```

Fig. 1. N-Body recursive query

If the particle is far enough from all particles in the tree, then the total force can be computed using the center of mass of the whole space represented by the tree (lines 2–3). If the particle is near that space, then the function recurses to compute all forces applied to the input particle by each sub-tree (lines 5–7). This computation cannot be executed using the classic MapReduce abstraction, because it allows only flat map-reduce operations on the dataset. Assuming the `tree` argument is a distributed dataset, the map function would need to recursively apply a map-reduction to directly code the above algorithm.

In this paper we extend the Apache Spark MapReduce engine [18] to directly support such nested and recursive computations. Spark is an implementation of the MapReduce model that outperforms Hadoop [2] by packing multiple operations into single tasks, and by utilizing the RAM memory for caching intermediate data. We target Apache Spark because it is a widely used, efficient, state-of-the-art platform for data analytics, and currently the fastest-growing such open-source platform [4, 13].

Spark expresses and executes in-memory fault-tolerant computations on large clusters using Resilient Distributed Datasets (RDD). RDD instances are immutable partitioned collections that can be either stored in an external storage system, such as a file in HDFS, or derived by applying operators to other RDDs. RDDs support two types of operations: (i) transformations, which create a new dataset from an existing one, and (ii) actions which return a value to the driver program after running a computation on the dataset. Examples of RDD transformations are *map* and *filter* operations, whereas *reduce* and *count* operations are typical actions. All transformations in Spark are lazy, which means that the result is not computed right away. Instead, Spark keeps track of all the transformations applied to the base dataset and they are only materialized when an action requires a result to be returned to the driver program.

Each RDD operator uses a User Defined Function (UDF) that manipulates the data. By default, this UDF is not itself allowed to operate on RDDs in Spark, as RDD objects and their dependency graph are allocated in the master node containing the Spark *scheduler* and *driver*, where the main program is executed, whereas UDFs are executed by the worker nodes containing the Spark *executors*. This restriction does not affect a large set of programs that do not use recursive computations. Moreover, even recursive computations can almost always be transformed to use a worklist and iteratively fixpoint, to bypass this restriction. That is, however, often ineffective in time and space, e.g., when not all recursive computations need to go to the same recursive depth, or when the created tasks are few and not load-balanced. Finally, refactoring a simple recursive compu-

```
1 val file1 = sc.textFile("hdfs://file1")
2 val file2 = sc.textFile("hdfs://file2")
3 file1.map(word1 =>
4   file2.filter(word2 =>
5     (word1.length > word2.length))
6       .collect())
7   .collect()
```

Fig. 2. Example of nested RDD operations

tation into a worklist algorithm often introduces complexity and, with it, the possibility of errors. Barnes-Hut is an example of such a recursive application that cannot directly be expressed using the "vanilla" RDD abstraction, because it needs nested RDD operators to express the recursive function shown in Fig. 1.

This paper extends the Spark programming model and scheduler to support nested RDD operations, to facilitate expressing recursive and hierarchical computations. We implemented this by modifying the RDD scheduling mechanism in Spark and measured its performance. We found that recursive RDD operations can greatly simplify the code for algorithms of a recursive nature.

The current Spark driver does not optimally schedule such fine-grain tasks as it introduces comparatively large latency from the time one task finishes to the time another task is scheduled to execute on that executor node. As recursive and hierarchical decomposition of work tends to create small tasks, we designed and implemented an extension to the Spark scheduler that supports parallel, lightweight scheduling better suited for jobs with fine-grain tasks.

Overall, this paper makes the following contributions:

- We added support for nested RDD queries in the Spark scheduler and compare it against built-in operators implemented without nesting. To demonstrate the usability of the programming model extension we implement an N-Body particle simulation using the nested RDD mechanism.
- We modified the default Spark task-scheduling mechanism so that it can support many parallel light schedulers. We measured its performance against the default Spark scheduler, and found a speedup of up to 2.1× for computations using fine-grain tasks.

2 Spark Support for Nested Operations

Consider the example code shown in Fig. 2 that creates two RDDs from two HDFS files (lines 1–2) and performs a map operation on RDD `file1` (lines 3–7). The "mapper" function of the map operation performs a filter operation on RDD `file2` for every word in RDD `file1` to select all words of larger length. The calls to `collect()` are there to force the computation to take place and collect the results into an array, as otherwise RDDs would behave essentially like lazy futures. This is a simple example of nested operators.

By default, Spark does not support such nested RDD operations. This is mainly because the RDD metadata required to schedule new computations are stored only at the master node, with executor nodes simply running tasks assigned to them. This example could be easily encoded in SQL and resolved using a cartesian product. Our system, however, allows full recursion, where the map function could itself contain maps and be recursive. Moreover, using nesting we achieve better scheduling, and outperform standard Cartesian product by up to 7x, as shown in Section 4.

Handling nested RDD operators inside the user-defined functions of a map operation as shown in Fig. 2 (lines 4–6) requires the executor nodes that run the tasks of the outer RDD map operator (lines 3–7) to behave as the master node and schedule the "nested" filter job created in the mapper function (line 4). Adding such functionality to the executor nodes would greatly increase their complexity, as the RDD data would need to be replicated on executors, which would require maintaining RDD metadata consistent among all distributed copies of an RDD. Apart from being inefficient, this would undo the simplicity and efficiency of the MapReduce model. Thus, our design forwards the nested operators back to the master, to avoid a distributed scheduler setup.

In the example in Fig. 2, the outer collect method (line 7) will force the runtime to schedule the outer RDD computation. Since no shuffle operations are involved, the dependency DAG that Spark constructs to properly order consecutive RDD operations will consist of only one stage[3] that contains one transformation of the HadoopRDD `file1` to a MappedRDD returned by the `map` method. The Spark scheduler will try to submit this stage and since there are not waiting parent stages it will proceed with creating and submitting the missing tasks. Then the scheduler will create tasks that execute the mapper function (lines 4–6) for each word in the `file1` RDD. Specifically, the scheduler will serialize a closure of the mapper function and distribute it to the executors; then it will create a task for every partition of the `file1` RDD and send each task to an idle executor to run the mapper function on that partition and thus create the corresponding partition of the result RDD.

In the executor nodes, when the mapper function shown in Fig. 2 (lines 4–6) runs, it will try to invoke a filter operation on the `file2` RDD. We extended the executor functionality to capture this event and send a *CreateRDD* message to the scheduler node. The message contains an identifier of the RDD object referenced, the (reflective) name of the invoked operation, and a serialized version of the user-defined function that is applied.

We extended the Spark scheduler to receive such messages from the executors. Upon receiving such a forwarded RDD operation message, the scheduler looks up the RDD with the specified id and, using jvm reflection, invokes the specified operation. In the execution of the example code in Fig. 2 it will invoke the `map` method of the `file2` RDD, creating the desired RDD that describes the result. Note that, as RDDs are essentially futures in Spark, no computation will yet take

[3] In Spark, a stage is a set of consecutive operators that can be grouped and executed together, per partition.

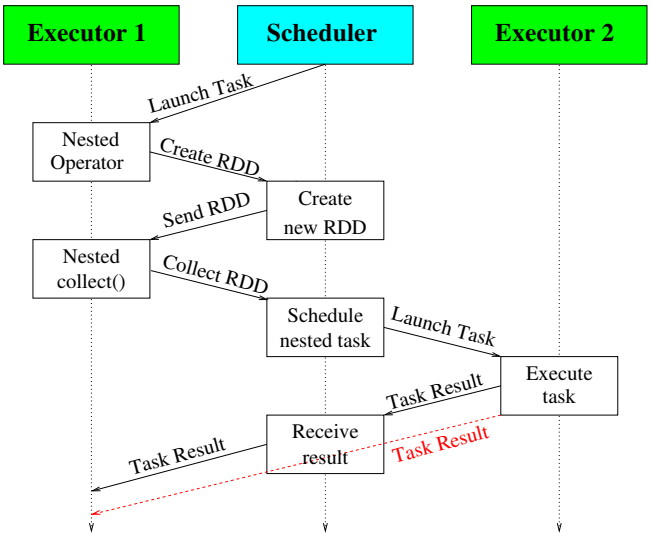


Fig. 3. Executor asks the scheduler to schedule a nested RDD operation

place at this point. The scheduler then will simply send back to the executor an identifier of the created RDD. The executor, upon receiving that message, will create a proxy of that RDD object based on the identifier received, and use it to continue the computation of the mapper function for the task of the outer map operation on `file1`. When that mapper function calls `collect()` (line 6), the executor will send a *CollectRDD* message to ask the scheduler to collect the new RDD, using its identifier, and send back the result. Fig. 3 shows the sequence of messages that will be sent for this example.

In the naïve master-executor protocol described above, the master schedules the nested job after receiving the CollectRDD message. When the nested job is done, it receives the result from all the executors where the job was scheduled, combines all partitions into a collected array, and sends the result to the executor that issued the nested operation. This would mean that there is an unnecessary transfer of data to the master node, and from there to the executor that issued the nested job. To avoid the transfer overhead that would also make the master node a centralized bottleneck for all nested computations, we modified the executor code to send the nested task result directly to the executor that issued the nested job (shown by the red dashed line) and also send an ack to the scheduler that the nested task finished, to free the executor resources. This way, the only transfer of data to the master is for `collect()` operations called at the master node in the top-level of the program.

Following the protocol described so far, the executor sends a CreateRDD message to the scheduler for every RDD operation that the mapper function of the `file1` map operation performs on `file2`. However, only the `collect()`

operation requires the master to actually schedule a nested computation. This is because the Spark RDD abstraction is lazy and the operations are not computed immediately. We took advantage of this property by grouping all the nested operators into a single message. RDDs are both lazy and immutable, which permits us to pack all the RDD operator arguments in a per-executor global data structure. Then, at the end of a task or when a nested `collect()` is triggered, the executor sends all the RDD transformations to the master node, to create all the RDDs described and schedule any required computations.

3 Scheduling

Recursive decomposition of data tends to create many small tasks. Moreover, simple computations like summing or counting an RDD often result in many lightweight tasks where scheduling overhead is comparable to the actual task computation. Although computations rarely constitute the whole of a Spark program, they are often found within larger computations as, for instance, a stage in an analytics pipeline, or "inner" jobs in a Spark-nested program as described in the previous section. The default Spark scheduling algorithm underperforms for jobs like that, because:

1. The scheduling path is sequential, which means that if a job consists of many tiny tasks, scheduling itself will take a lot of time in the critical path of the computation, while the processing time will be negligible.
2. After a worker has finished a task, it sends a request message to the scheduler, so that the driver sends a new task to the worker. That increases the total time by at least one RTT for every task and every worker, since the scheduler receives and handles these messages sequentially.

These issues may be exacerbated when a large number of executors cannot be properly managed by a single, centralized Spark scheduler. To address this, we designed and implemented a parallel version of the Spark scheduler. We modified the Spark scheduler to send multiple tasks to each executor and amortize the idle time between tasks over many requests. This decreases the time between when a worker finishes a task and sends a message to the scheduler and when the scheduler answers with the next task to run. Specifically, we inserted a local task queue per executor, and modified the centralized scheduler to keep track of these coalesced task sets. Every time a worker core finishes a task, it first tries scheduling one of the tasks in the local task queue, and only generates network traffic and a request to the centralized scheduler if the local queue is empty.

Moreover, we modified the central Spark scheduler to schedule task-sets in parallel. Specifically, instead of using a single scheduler-master, we deploy a set of schedulers organized hierarchically as a set of *ProxyScheduler* actors under the standard Spark master node. The standard Spark scheduler creates a few large task-sets per job and sends them to the proxy schedulers; each proxy scheduler is then responsible for sending smaller task-sets or individual tasks to the executors. This reduces congestion at the master scheduler, occuring either because tasks

are too small or because there is a large number of pending executor messages. We do not assign specific executor groups to the proxy schedulers, and instead allow all proxy schedulers to send work to all available executors. This works well in practice when the available work is much more than the executors, which is almost always the case in Spark analytics applications.

To schedule and track tasks to executors, each proxy scheduler keeps a copy of all the executor metadata that the standard Spark master normally maintains. This creates a consistency issue, as not all of these copies may be updated at the same time. We solve this by keeping all the "heartbeat" functionality Spark uses for tracking executor availability at the Spark master, and only forward information about executors from the master to the proxy schedulers. This means that at any given time the latest metadata about the state of one given executor's availability are at the master, and the metadata about all tasks in that executor's queue are distributed among all proxy schedulers that may have sent tasks to that executor.

To handle the case of executor state changes, the master scheduler sends a message to all proxy schedulers when the heartbeat process discovers that an executor has changed state. For example, when an executor is started, it sends a message to the master to inform that the executor is registered —as in the standard Spark scheduler. Then, the master broadcasts to all proxy schedulers the state of the newly registered worker. Eventually, all the schedulers will have the same view of the cluster state.

A similar problem of distributing copies of metadata occurs in tracking task completions. Specifically, the standard Spark scheduler uses *StatusUpdate* messages that contain information about whether a task has started, is executing, has finished, or has failed. In our distributed scheduler, these messages are sent from the workers to the proxy schedulers. The proxy schedulers eventually forward all *StatusUpdate* messages to the central Spark scheduler. We have not yet managed to recreate any cases where this creates a bottleneck; in that case we expect it would be straightforward to reduce the strain on the Spark scheduler by handling task completions and failures in the proxy schedulers without any forwarding of that information.

The standard Spark scheduler balances loads among executors by sending tasks only to the executors that have free cores. In avoiding the update messages by coalescing sets of tasks per executor and in allowing all proxy schedulers to send tasks to all executors, we have removed the load balancing guarantees of the standard Spark scheduler. However, we found that by tranferring some of the master functionality to the executors suffices in practice to give load-balanced executions.

Specifically, we use a best-effort approach for balancing task loads, where each executor locally schedules tasks from a queue to cores as they become available. The per executor local queue we inserted is visible by all executor threads. This means that in a case where an executor is loaded with some heavy and some light tasks, the threads executing the light tasks that will finish earlier, will dequeue and execute more tasks. Thus, when a job consists of some heavy tasks, even if

they are scheduled on the same executor it is highly improbable that they will be executed by the same core.

Note however that this solution is best effort. In most cases given enough executor cpus the load will be equally balanced. In an exetremely bad scenario where too many straggler tasks are scheduled into the same executor while the other executor takes all the lightweight tasks, the runtime will be highly affected. We tried to stress our best-effort solution by constructing benchmarks with highly-imbalanced tasks (Section 4, but were unable to create such a scenario in practice.)

4 Evaluation

We evaluated the performance of our scheduler using a set of micro-benchmarks and an operator from a real, large analytics application, that we were able to rewrite to use nested operators. The code for our scheduler and the micro-benchmarks is available at `https://github.com/p01K/spark-nested`.

We designed a set of micro-benchmarks to consist of non computationally demanding tasks, so that the scheduling overhead becomes a bottleneck. When such computations are used in analytics applications, it may not be feasible to create larger tasks, as the overhead of repartitioning is comparable to the scheduling overhead of fine-grain tasks. [4] The datasets contain integers or words, split into a defined number of partitions, and intentionally cached so that the tasks do not take extra time loading the data. We first invoke a count operation in all benchmarks, without counting it in the total run time, so that we ensure that the dataset is stored in memory. We ran each benchmark 15 times and measure the last 10 runs, so that the runtime is not affected by the JVM class loading, JIT compiling or other optimization techniques [6].

We used the following benchmarks:

- The filter benchmark generates a dataset of random numbers and returns those that are products of a defined number.
- The sum benchmark adds the dataset values using the reduce operator.
- The collect benchmark simply brings all the elements to the master node.
- The longtail benchmark simulates a taskset whose runtime follows a long tail distribution.
- The word count benchmark counts the references of each word.

We implemented our scheduler in Apache Spark 1.6.0. We ran all benchmarks on a cluster of 5 nodes, where each node has 4 Intel i5-3470 cores, 16GB memory, and is running Debian Linux and OpenJDK7. The nodes are connected through 1GBs network. We measured the average round-trip time between any two nodes to be on the order of 0.1 ms.

We compare our scheduling algorithm with the default Spark scheduling. To have a valid comparison, we tried to use equal resources for scheduling and for

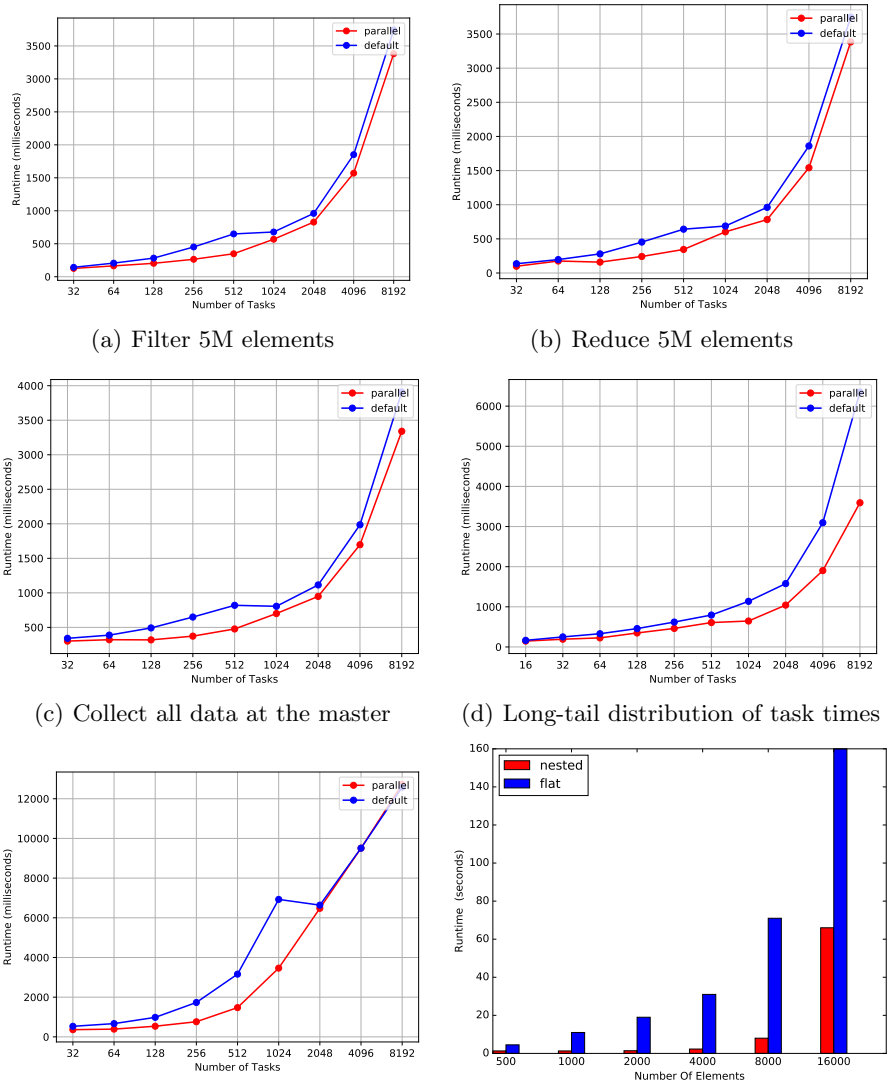
[4] We have encountered such small tasks in map and filter operations that operate on fine-grain partitions within larger workflows, in actual analytics applications.

task execution; the runs with default Spark use one node as a Spark master and 4 nodes as executors, while the runs with our distributed scheduler deploy all proxy schedulers together with the Spark master on one node, and use 4 nodes as executors. This way, both schedulers have exactly the same resources devoted to scheduling and to task execution.

We ran these benchmarks with a fixed number of elements (5M), and a variable number of partitions (64 to 8192) to measure how the number and granularity of tasks affects the runtime difference between the two schedulers. Fig. 4(a) presents average running time of a simple filter operation on 5 million elements. The number of partitions of the input RDD is equal to the number of tasks. The y-axis shows the runtime in milliseconds. Our scheduler consistently outperforms the default Spark scheduler by at least 1.11× and up to 1.86×. Much of that difference seems to be a constant factor, which we believe is due to the reduction of worker idle time while waiting for the next task. As task granularity becomes smaller, both schedulers perform worse. The consistent performance "knee" observed for the default Spark scheduler at 512 tasks is not correlated with idle time in the worker cores nor network traffic measured, and could be due to partition migrations.

Fig. 4(b) compares the default Spark scheduler to our distributed scheduler on a reduction that sums 5M random integers. The horizontal axis is the number of partitions that the dataset is distributed into. Reducing the number of messages and parallelization of scheduling gains a constant factor over the default Spark scheduler, resulting in 1.12× to 1.87× better performance. Fig. 4(c) compares the two schedulers on simply collecting all the elements of a partitioned RDD to the master node. We observe the same behavior even when the task execution time is zero in this case, again due to the reduction in scheduling overhead and message latencies. To evaluate how well our best effort load balancing heuristic performs compared to the load balancing guarantees provided by the default Spark scheduler, we ran a microbenchmark that simulates tasks with highly different running times, following a long-tailed distribution. Fig. 4(d) presents a comparison of the two schedulers on the long-tail benchmark. Again, the distributed scheduler achieves a speedup between 1.13× and 1.77× over the default Spark scheduler. This result is consistent accross executions with negligible variance; we expect that for executors with more than 4 cores it is highly unlikely that straggler tasks will cause imbalance and large latency in the total job execution time. Finally, Fig. 4(e) presents the comparison on a standard word count benchmark. Again, the distributed scheduler outperforms the default Spark scheduler by up to 2.15×.

We used the Cartesian product as a benchmark to compare the performance of nested queries versus flat queries. For the flat, non-nested version we used the cartesian RDD operator. Fig. 4(f) compares the total running times between the two versions. We found that writing a cartesian product as a two-level nested RDD operation parallelizes it into smaller but parallel jobs and achieves a total speedup of up to 8×, mainly due to the parallel scheduling of the work.



(a) Filter 5M elements

(b) Reduce 5M elements

(c) Collect all data at the master

(d) Long-tail distribution of task times

(e) Word count

(f) Comparison between flat and nested operators in Cartesian product

Fig. 4. Comparison with original Spark scheduler

To demonstrate the programming expressiveness of using nested RDD operations we implemented the Barnes-Hut n-body gravity simulation algorithm using nested operators, and evaluated it for various numbers of data points, up to 8000 bodies. Note that there is no comparison against the default Spark scheduler as Barnes-Hut is recursive and thus not directly portable to flat MapReduce, with-

Dataset	Size	Nested	Flat
1 hour	46.45K	33s	39s
1 day	450MB	4m	4.4m
1 month	12.6G	57m	OOM
1 month	17.3G	1.5h	OOM
2 months	30G	3.8h	OOM

Fig. 5. Flat vs Nested query

points	time (sec)
32k	3
64K	3.7
128K	7
256K	10
512K	20
1M	60

Fig. 6. N-Body

out completely restructuring the algorithm to use explicit iterations and simulate a stack. Fig. 6 presents the results.

To further evaluate the effect of nested queries, we used a (closed source, proprietary) analytics application used by a telecommunications provider to perform user characterization and classification from CDR data. We extracted a part of the full workflow that used nested loops to iteratively perform multple map reductions, and rewrote it using nested map operators. Fig. 5 presents a comparison on synthetic datasets of various sizes that closely match actual data[5]. The data sets used correspond to one hour, one day, one low-usage month and one high-usage month of data, while the last line uses a two-month dataset produced by concatenation of the two one-month datasets. We observed a large difference in scalability between the two versions. When the size of the dataset is relatively small, both versions execute in similar time, with the nested version having a small speedup. But for larger data sizes, the nested version executes successfully, while the flat terminates with `OutOfMemory` exception.

5 Related Work

Many analytics query execution engines use dataflow models and languages to express computations. Similar to Spark, Naiad [9] is a distributed data analytics engine for cyclic dataflow programs, Stratosphere [1] (now Flink), is a distributed data analytics engine aimed at stream analytics, and Hadoop is a map-reduce framework for large batch computations. Although all engines allow non-pure User Defined Functions in certain cases, none allow these computations to recursively include other queries(manually tried to use a MapFunction function inside a MapFunction function in Flink 1.1.2, which failed with a runtime error).

Spark supports the execution of SQL queries, which can be nested. Nesting of SQL statements, however, does not correspond to actual recursion in the computed queries; nested SQL statements amount to simply sequenced computations. The same limitation applies to REX [8], which introduces a new programming model similar to SQL, called RQL, that uses the notion of deltas (or small updates). Similarly, Datalog execution engines [14, 15] can express queries on recursive relations in Spark. Although Datalog relations can be recursively

[5] Actual data was not available for analysis due to privacy constraints

defined and may correspond to fixpoint computations, they are closer to iterative fixpoints and do not amount to fully recursive computations; for instance it is not straightforward to express the Barnes-Hut n-body simulation in datalog, where the mapper and reducer functions apply themselves recursively in nested map-reductions.

The Spark default scheduler uses Delay scheduling [17] to send a task where its data are stored, before any available worker. Ousterhout et al. [11] propose Sparrow, a decentralized scheduling algorithm. The scheduling of a job is assigned to a random scheduler, that sends each task probe to 2 random workers. When the worker dequeues a task probe, it asks for the task binary from the scheduler and the corresponding scheduler sends the task to the worker that asks first. In comparison, our batching of small tasks simplifies the scheduling overhead and may have a probabilistically worse worst-case-scenario, although we did not observe this issue in parctice. Moreover, solutions to straggler occurrence proposed by Ousterhout et al. [10] produce a lot of fine-grain tasks, decreasing that probability further.

Malte Schwarzkopf et al. [12] present Omega, a distributed scheduling mechanism where each scheduler has full access to the cluster. Each scheduler is given a private, local, frequently-updated copy of the cluster state for making scheduling decisions. We chose not to replicate scheduling state between master and proxy schedulers, to avoid the complication of maintaining all copies coherent, thus introducing additional fail points in the scheduling algorithm. SkewTune [7] is a Hadoop extension that tries to eliminte skew in map reduce jobs. When SkewTune identifies a straggler it repartitions the remaining data, to increase parallelism. Yadwadkar et al. [16] describe a way to reduce straggler mitigation to multi-task learning. Their models can predict if a task will be a straggler, creating a separate model for each cluster node.

6 Conclusions

We present an extension of the default Spark scheduler that supports nested RDD operations and allows the programmer to express recursive computations intuitively. We demonstrate this by using it to implement the Barnes-Hut n-body simulation in Spark. We found that our extension of the RDD abstraction creates many small jobs, so we extended the default Spark scheduler with distributed scheduling to reduce scheduling overhead. We evaluated our system and found it outperforms the standard Spark scheduler by up to 2.15×.

Acknowledgements

This work was supported in part by the 7th Framework Programme of the European Community for Research, Technological Development and Demonstration Activities (FP7) project ASAP (grant agreement 619706); and by the Horizon 2020 Framework Programme for Research and Innovation project ExaNeSt (grant agreement 671553).

References

1. A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, Dec. 2014.
2. Apache Software Foundation. Hadoop.
3. J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324:446–449, Dec. 1986.
4. Databricks. Spark survey results, 2015.
5. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.
6. A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Object-oriented Programming, Systems, Languages, and Applications*, 2007.
7. Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: Mitigating skew in mapreduce applications. In *ACM SIGMOD International Conference on Management of Data*, 2012.
8. S. R. Mihaylov, Z. G. Ives, and S. Guha. Rex: Recursive, delta-based data-centric computation. *Proc. VLDB Endow.*, 5(11):1280–1291, July 2012.
9. D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Symposium on Operating Systems Principles*, 2013.
10. K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The case for tiny tasks in compute clusters. In *Hot Topics in Operating Systems*, pages 14–14, 2013.
11. K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Symposium on Operating Systems Principles*, 2013.
12. M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems*, 2013.
13. J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proc. VLDB Endow.*, 8(13):2110–2121, Sept. 2015.
14. A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo. Big data analytics with datalog queries on spark. In *ACM SIGMOD International Conference on Management of Data*, 2016.
15. J. Wang, M. Balazinska, and D. Halperin. Asynchronous and Fault-Tolerant Recursive Datalog Evaluation in Shared-Nothing Engines. *PVLDB*, 8(12):1542–1553, 2015.
16. N. J. Yadwadkar, B. Hariharan, J. Gonzalez, and R. H. Katz. Faster jobs in distributed data processing using multi-task learning. In *SDM*, pages 532–540. SIAM, 2015.
17. M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *European Conference on Computer Systems*, 2010.
18. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Hot Topics in Cloud Computing*, 2010.