Resource Elasticity for Large-Scale Machine Learning

Botong Huang[2]*, Matthias Boehm[1], Yuanyuan Tian[1], Berthold Reinwald[1], Shirish Tatikonda[1], Frederick R. Reiss[1]

[1] IBM Research – Almaden; San Jose, CA, USA
[2] Duke University; Durham, NC, USA

ABSTRACT

Declarative large-scale machine learning (ML) aims at flexible specification of ML algorithms and automatic generation of hybrid runtime plans ranging from single node, in-memory computations to distributed computations on MapReduce (MR) or similar frameworks. State-of-the-art compilers in this context are very sensitive to memory constraints of the master process and MR cluster configuration. Different memory configurations can lead to significant performance differences. Interestingly, resource negotiation frameworks like YARN allow us to explicitly request preferred resources including memory. This capability enables automatic resource elasticity, which is not just important for performance but also removes the need for a static cluster configuration, which is always a compromise in multi-tenancy environments. In this paper, we introduce a simple and robust approach to automatic resource elasticity for large-scale ML. This includes (1) a resource optimizer to find near-optimal memory configurations for a given ML program, and (2) dynamic plan migration to adapt memory configurations during runtime. These techniques adapt resources according to data, program, and cluster characteristics. Our experiments demonstrate significant improvements up to 21x without unnecessary over-provisioning and low optimization overhead.

1. INTRODUCTION

Enterprise data management evolves to a diverse mix of transactional and analytical tools and languages [1, 32]. Increasing data sizes, the need for advanced analytics [10, 32], and very specific workload characteristics [11, 46] led to specialized systems for large-scale machine learning (ML). Application scenarios are ubiquitous and range from traditional statistical tests for correlation and sentiment analysis to cutting-edge ML algorithms including customer classifications, regression analysis, and product recommendations.

The state-of-the-art on large-scale ML aims at *declarative* ML with high-level languages based on linear algebra and statistical functions [6, 7, 12, 13, 20, 24, 31, 32, 46, 48]. In contrast to tailor-made ML algorithms or ML libraries, this declarative approach allows (1) full flexibility to specify new ML algorithms or customizations, (2) physical data independence of the underlying data representation (dense/sparse matrices, matrix blocking, etc), and (3) both efficiency and scalability via automatic cost-based optimization of hybrid in-memory and distributed runtime plans. ML systems often exploit MapReduce [15] or more general data-parallel distributed computing frameworks like Spark [49] in order to share cluster resources with other specialized systems.

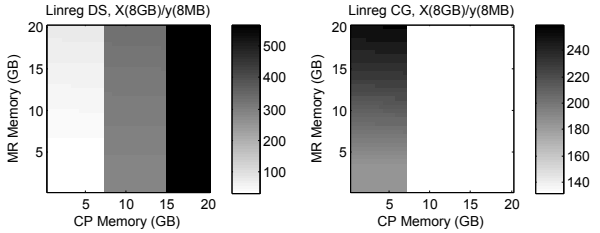


Figure 1: Estimated runtime [s] of two ML scripts for linear regression with different control program (CP) and MapReduce (MR) memory configurations.

Problem of Memory-Sensitive Plans: Compiling hybrid runtime plans is important for declarative ML to enable users to write their ML algorithms once but ensure efficiency for small and medium problems (e.g., median job sizes of analytics clusters were reported as <14 GB [39]), and good scalability for larger problems. However, a key observation is that plan compilation naturally becomes very sensitive to memory configurations of master and map/reduce task processes. Different cluster configurations quickly lead to very different performance characteristics. This sensitivity is problematic for two reasons. First, the user needs to reason about plans and cluster configurations to achieve best performance, which contradicts the goal of declarative machine learning. Second, finding a good static cluster configuration is a hard problem because of the variety of ML algorithms and completely different workload characteristics.

Example Cost Comparison: Figure 1 shows the costs of two linear regression algorithms with very different memory preferences. Both algorithms—a closed form direct solve (DS) on the left, and an iterative conjugate gradient (CG) on the right—solve an ordinary least square problem $\mathbf{y} = \mathbf{X}\beta$. The diagram shows estimated costs for different memory configurations of the master process, the so-called control program (CP), and MapReduce tasks (MR). For the case of 1,000 features, DS is compute-intensive and hence prefers a massively parallel, distributed runtime plan with small CP

*Work done during an internship at IBM Research – Almaden.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

http://dx.doi.org/10.1145/2723372.2749432.

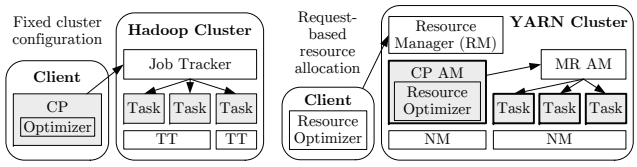
memory. In contrast, the iterative CG is IO bound and thus benefits from a large CP memory, where we read the data once and repeatedly compute matrix-vector multiplications in-memory. Given the estimated costs, we would use 2 GB CP/MR processes for DS, but a 10 GB CP process for CG.

Resource Negotiation Frameworks: A static cluster configuration is always a compromise that misses opportunities, especially in multi-tenancy scenarios where the same cluster is shared by different specialized systems for ML, graph analysis, ETL, streaming, and query processing. These multi-tenancy scenarios are addressed by next generation resource negotiation frameworks like YARN [47], Mesos [23], Omega [41], or Fuxi [53], which provide the flexibility of sharing resources across specialized systems that no longer need to rely on the same distributed runtime framework. Most importantly for declarative ML, these frameworks allow us to tackle the aforementioned problem of memory-sensitive plans in a very principled way.

Challenges: Resource elasticity for large-scale ML is a challenging problem. First, there is a variety of ML use cases with very different performance characteristics, which makes cost estimation crucial. Second, declarative ML requires very good robustness because users rely on automatic optimization. Robustness includes simplicity and maintainability with regard to internal optimization phases, effectiveness to find near-optimal plans, and efficiency in terms of low overhead even for complex ML programs. Third, automatic resource configuration itself is challenging. Decisions on CP and MR memory budgets are inter-related, and intermediate result sizes might be unknown during initial compilation due to conditional control flow and data dependent operations.

Contributions: The primary contribution of this paper is a systematic approach to automatic resource elasticity for large-scale ML. Our basic idea is to leverage YARN's resource requests and to optimize resource configurations for a given ML program via *online what-if analysis* according to program, data and cluster characteristics. We build on existing ideas from physical design tuning and introduce a novel architecture for automatic resource optimization of large-scale ML programs. This also includes novel program-aware grid enumeration, pruning, and re-optimization techniques. The major strengths of our approach are simplicity and robustness. We achieved this by (1) generating and costing runtime plans for enumerated resource configurations, and (2) resource re-optimization during dynamic recompilation. In detail, we make the following technical contributions:

- *Problem Formulation:* After a brief background description of SystemML and resource negotiation frameworks like YARN, we conceptually formulate the resource optimization problem in Section 2.
- *Resource Optimizer:* We introduce our cost-based resource optimizer in Section 3. This includes the cost model, a search space characterization, as well as program-aware enumeration and pruning techniques.
- *Runtime Plan Adaptation:* In addition, we explain how to use this resource optimizer for runtime plan adaptation in Section 4. This includes extended cost estimation techniques and runtime migration strategies.
- *Experiments:* Finally, we integrate the resource optimization framework into SystemML. We describe end-to-end results as well as optimization overhead using a variety of use cases in Section 5.



(a) MR1 Integration (b) YARN Integration

Figure 2: SystemML's YARN Integration.

2. BACKGROUND AND OVERVIEW

As necessary background, we briefly describe SystemML as a representative example of a declarative ML system as well as YARN as a request-based resource negotiation framework. Based on those preliminaries, we then formulate the *ML program resource allocation problem* and present an overview of our resource optimizer to solve this problem.

2.1 SystemML

SystemML [5, 20] enables declarative, large-scale ML. Users write ML algorithms in a high-level ML scripting language with R-like syntax, called DML. This language includes linear algebra, statistical functions, and control flow constructs like loops and branches. These scripts are compiled into a hierarchy of program blocks as defined by the control structure. Each program block is compiled into DAGs of a high-level operators (HOP), low-level operators (LOP), and eventually executable instructions. At HOP level, we apply optimizations like common subexpression elimination, algebraic simplification rewrites, and compute operation memory estimates. These estimates reflect our in-memory runtime that pins inputs and outputs into memory in order to prevent repeated deserialization in operations that access the same data multiple times. SystemML compiles hybrid runtime plans of in-memory CP operations and large-scale MR operations. This decision is part of operator selection, where we decide upon physical operators (LOPs), apply additional rewrites, and pack MR operators of a DAG into a minimal number of MR jobs. Finally, in case of unknown sizes of intermediate results, we dynamically recompile DAGs for runtime plan adaptation.

SystemML and similar systems are sensitive to memory. First, operator selection of CP/MR operations is based on a simple yet very effective heuristic. We assume that in-memory CP operations require less time than their distributed MR counterparts and hence choose a CP operator if its memory estimate fits in the CP memory budget. Second, for many important operations like matrix multiplication, there are specific map-only MR operators that load one input into the mappers. This is similar to broadcast joins in Jaql or Hive (memory/map join) [27]. These operations are chosen if one input fits in the MR memory budget. Appendix A and B describe an example ML program and further details on ML program compilation, including an overview of memory-sensitive operators and compilation steps.

Figure 2(a) shows SystemML's integration with Apache MapReduce v1. The control program (CP)—that drives the overall program flow and executes in-memory operations—runs as a Hadoop client. This client also submits MR jobs to the job tracker if necessary. Intermediate data is exchanged through HDFS. Default client size and map/reduce task sizes are predefined in a static cluster configuration, and the jobtracker is unaware of resources consumed by the client.

2.2 YARN

YARN (Yet Another Resource Negotiator) [34, 47] aims at improved scalability and programming flexibility for shared clusters by providing basic resource management. Important daemons are (1) a per-cluster Resource Manager (RM) that monitors resource usage and node liveness and schedules application resource requests, as well as (2) per-node Node Managers (NM) that monitor local resources and start/kill application processes. Applications provide custom Application Masters (AM) that are responsible for resource negotiation with the RM and per-application scheduling of distributed tasks. Clients submit their applications along with a resource request (memory, number of cores) for the AM container to the RM. After the AM is started by the NM on a particular node, the AM can make additional resource container requests for distributed tasks.

As the context for this paper, Figure 2(b) shows SystemML's YARN integration. The control program (CP) runs inside a custom AM. Similar to the MR1 integration, the CP submits MR jobs if necessary. In YARN, this transparently spawns a per-job MR AM. The client uses our proposed resource optimizer to decide on the initial resource request for the CP AM and MR tasks. Furthermore, the CP AM also uses the resource optimizer for dynamic plan adaptations. In contrast to MR1, this allows us to decide on the size of CP and MR tasks as needed by an ML program.

2.3 Problem Formulation

We model an ML program P and its resource configuration $\mathcal{R}_P$ as a set of program blocks $B = (B_1, \ldots, B_n)$ and a set of resources $\mathcal{R}_P = (r_P^c, r_P^1, \ldots, r_P^n)$, where r_P^c denotes the resources of the master process (CP) and r_P^i denotes the resources of distributed computations in program block B_i. Let cc denote a cluster configuration that contains information about available resources and minimum/maximum allocation constraints min_{cc} and max_{cc}. Furthermore, let $C(P, \mathcal{R}_P, cc)$ denote a cost function that estimates the costs of the runtime plan created for a given program (including inputs), resources, and cluster configuration. We formulate the related optimization problem as follows:

DEFINITION 1. *(**ML Program Resource Allocation Problem**). Find the optimal resource configuration $\mathcal{R}_P^*$ given program P and cluster configuration cc with*

$$\mathcal{R}_P^* = \min \left(\underset{\mathcal{R}_P \in [min_{cc}, max_{cc}]}{\arg\min} C\left(P, \mathcal{R}_P, cc\right) \right). \quad (1)$$

The goal is to find the resource configuration (within min/max constraints of the cluster) that minimizes costs for the given program. In case of multiple resource configurations $\mathcal{R}_P$ with minimal costs, $\mathcal{R}_P^*$ is defined as the minimum resources in order to prevent unnecessary over-provisioning. For comparing resource vectors, we define that $\mathcal{R}_P^x \leq \mathcal{R}_P^y$ if and only if $sum(P, \mathcal{R}_P^x, cc) \leq sum(P, \mathcal{R}_P^y, cc)$, where $sum()$ is the time-weighted sum of used resources.

Problem Instantiations: There are many instantiations to this problem. For request-based resource allocation as used in YARN, we use memory m_P^c as resource r_P^c and directly request the optimal resources $\mathcal{R}_P^*$. For offer-based resource allocation as used in Mesos, we are also interested in the optimal resource allocation $\mathcal{R}_P^*$ but have additional optimization decisions in case of non-matching offers.

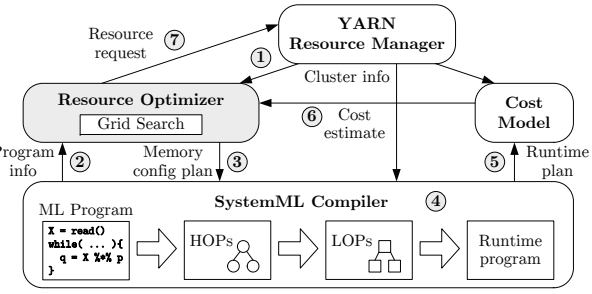


Figure 3: SystemML Resource Optimizer.

2.4 Resource Optimizer Overview

At a high level, we use an *online what-if analysis* to compute near-optimal resource configurations for a given ML program. Our design decision of an independent resource optimizer aims at robustness. This design ensures simplicity and maintainability at the cost of additional optimization overhead and missing optimality guarantee. However, as we shall show, even lightweight compiler interactions allow for program-aware resource optimization, which achieves both robustness and near-optimal plans with low overhead.

Resource Optimizer Architecture: Figure 3 shows the general architecture of our resource optimizer. In the first step, the optimizer obtains the cluster information cc from the RM. This includes min/max memory constraints, number of nodes/cores, and more specific configurations like HDFS block size. In the second step, we let the compiler generate the HOP representation of the input program P. We obtain basic program information such as the program structure B and memory estimates from this representation. The resource optimizer then uses grid enumeration and pruning strategies to find a near-optimal resource configuration (step 3-6). For each enumerated resource configuration, we modify the memory budgets of the compiler, generate the LOP DAG and runtime program, and use the cost model to compute a time estimate. Doing cost estimation on runtime programs is important for robustness because it automatically takes all optimization phases of the entire compilation chain into account. This enumeration process is repeated until we have systematically evaluated the search space or a budget of optimization time is exceeded. Finally (step 7), we request the AM memory that resulted in minimal costs.

Resource Optimizer Use Cases: Our resource optimizer has two major purposes (see Figure 2(b)). First, we employ this resource optimizer for *initial resource optimization* before the client submits the application to the RM. Second, we use the same optimizer for *runtime resource optimization* in terms of runtime plan adaptation if sizes of intermediates are initially unknown or changing. In the following two sections, we describe the details of the core resource optimizer and extensions for runtime plan adaptation.

3. INITIAL RESOURCE OPTIMIZATION

We now describe the core resource optimizer as used for initial resource optimization. This includes the cost model and search space, as well as grid enumeration and pruning.

3.1 Cost Model

As a fundamental building block for cost-based resource optimization, we need a cost model and accurate estimates. Our basic idea is to use a white-box cost model based on

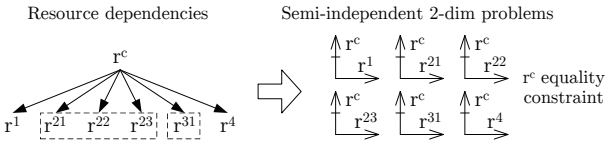


Figure 4: Search Space Characterization.

generated runtime plans, because the runtime plan automatically reflects all independent optimization phases. A more detailed discussion with examples can be found in [4]. We define the costs $C(P, \mathcal{R}_P, cc)$ as estimated execution time of the runtime plan generated for P given the resources $\mathcal{R}_P$ and cluster configuration cc. This time-based model allows us to reflect IO, compute, and latency at the same time. In contrast to related work of MR job tuning [18, 21, 43] and MR progress indicators [33], this approach gives us an *analytical cost model* for entire ML programs. This model is able to cost alternative physical runtime plans without the need for black-box sample runs or in-progress plan execution.

Time Estimates: In detail, time estimation works as follows. As essential foundation, we use input and intermediate result sizes as inferred and compiled into the runtime plan (if known during compilation). We scan the runtime plan in execution order and track sizes and states of live variables. An in-memory operation changes the state of its inputs and output to in-memory. The time estimate of a CP instruction consists of IO and compute time. We estimate IO time based on variable state, size, format, and default format-specific read/write bandwidths, while we estimate compute time based on size, operation-specific number of floating point operations, and default peak performance. The time estimate of an MR-job instruction is more complex: it consists of job and task latency, in-memory variable export, map read, map compute, map write, shuffle, reduce read, reduce compute, and reduce write times, and it can refer to multiple map/reduce instructions that are packed into the same job. Both IO and compute of map/reduce tasks are divided by the map/reduce degree of parallelism as inferred for this particular MR-job instruction, where we take the CP/MR resources into account when computing the resulting degree of parallelism. Finally, we aggregate time estimates along the program structure. For conditional branches, the aggregate is a weighted sum of time aggregates. For loops, we scale the time aggregate by the number of iterations; if the number of iterations is unknown we use a constant which at least reflects that the body is executed multiple times.

3.2 Search Space

As a basis for our discussion of enumeration strategies, we characterize essential properties of the underlying search space. We use the notion of a *resource dependency* for interrelated cost influences of resources. Furthermore, we make a simplifying assumption of *semi-independent resources*, i.e., that MR resources of different program blocks are independent. This is a reasonable assumption because the dependencies are indeed very weak: MR resources affect the choice of physical operators and hence the execution location, which might influence the number of output files and thus affects data-dependent program blocks. However, this effect is usually negligible and not considered by the cost model.

Key Properties: Figure 4 shows the resulting resource dependencies between CP resources r_P^c and the set of MR resources $(r_P^1, \ldots, r_P^n)$. For SystemML, this dependency is very strong because the CP memory determines which operations are scheduled to MR and thus can change the scope and impact of r_P^i. This particular dependency structure leads to the following two key properties:

- *Monotonic Dependency Elimination:* The CP memory r_P^c exhibits monotonic impact on dependencies. If the compiler decided for r_P^c to execute all operations of block B_i in CP, this eliminates the dependency to r_P^i. Increasing r_P^c does not reintroduce this dependency.
- *Semi-Independent 2-Dim Problems:* For a given CP resource r_P^c, the resource configurations of r_P^i and r_P^{i+1} are independent. This leads to a partitioning of $\mathcal{R}_P = (r_P^c, r_P^1, \ldots, r_P^n)$ into n semi-independent 2-dimensional problems $((r_P^c, r_P^1), \ldots, (r_P^c, r_P^n))$, which exhibit a global equality constraint on r^c.

We will exploit these properties for pruning and parallelization. The property of *semi-independent problems* also leads to a linear optimization time in the number of program blocks, which is important for large ML programs.

Algorithm 1 optimizeResourceConfig

Require: program P, cluster info cc, enum $type1$, $type2$
1: $S_{r^c} \leftarrow$ enumGridPoints$(P, cc, type1, \text{asc})$
2: $S_{r^m} \leftarrow$ enumGridPoints$(P, cc, type2, \text{asc})$ *// 3.3.2*
3: $\mathcal{R}_P^* \leftarrow \emptyset$ *// init best resource vector*
4: $C^* \leftarrow \infty$ *// init best program cost*
5: **for all** $r^c \in S_{r^c}$ **do** *// for each CP memory r^c*
6: $B \leftarrow$ compileProgram$(P, \mathcal{R}_P = (r^c, min_{cc}))$
7: $B' \leftarrow$ pruneProgramBlocks(B) *// 3.4*
8: memo $\leftarrow (min_{cc}, C(B))$
9: **for all** $B'_i \in B'$ **do** *// **in parallel**, Appx C*
10: **for all** $r^i \in S_{r^m}$ **do** *// for each MR memory r^i*
11: $B_i \leftarrow$ recompile$(B_i, \mathcal{R}_P = (r^c, r^i))$
12: **if** $C(B_i) <$ memo$[i, 2]$ **then** *// found better r^i*
13: memo$[i,] \leftarrow (r^i, C(B_i))$
14: $P \leftarrow$ recompile$(P, \mathcal{R}_P = (r^c,$ memo$[, 1]))$
15: **if** $C(P) < C^*$ **then** *// found better $\mathcal{R}_P^*$*
16: $\mathcal{R}_P^* \leftarrow (r^c,$ memo$[, 1])$
17: $C^* \leftarrow C(P)$
18: **return** $\mathcal{R}_P^*$ *// resource vector w/ best costs*

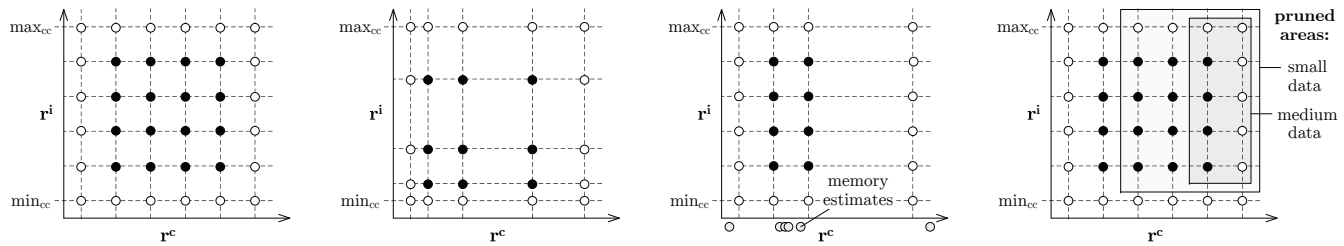
3.3 Grid Enumeration Strategies

Due to the goal of simplicity and robustness, we aim to keep our core enumeration algorithm independent of specific rewrites and optimization techniques. For this reason, we discretize the continuous search space and use basic grid enumeration strategies. In the following, we introduce the overall enumeration algorithm and different techniques for generating grid points via systematic and directed search.

3.3.1 Overall Grid Enumeration Algorithm

The input to our algorithm is the ML program P (including input data characteristics), the cluster configuration cc, and the types of grid point generators for the two dimensions of resources we are interested to decide upon. In general, the algorithm aims at solving the *ML program resource allocation problem* as defined in Definition 1. Finally, the output is the minimal resource configuration $\mathcal{R}_P^*$ with minimal cost.

Algorithm 1 shows our overall enumeration algorithm. We first materialize ascending grid points per dimension with the chosen grid point generators (lines 1-2). In addition,



(a) Equi-Spaced Grid (b) Exp-Spaced Grid (c) Memory-Based Grid (d) Grid Pruning

Figure 5: Basic Grid Point Generation and Pruning Strategies.

we initialize $\mathcal{R}_P^*$ as the best resource vector and C^* as the globally best program cost (lines 3-4). Due to the property of *semi-independent problems*, the outer loop iterates over all enumerated grid points for CP memory r^c (line 5). In each iteration, we compile the program P according to the resources (r^c, min_{cc}). This compilation serves as baseline compilation for the given r^c value and unrolls the program P into the set of program blocks B (line 6). Based on this compilation, we can access basic information—such as the number of MR jobs—per program block according to the given r^c. This information now allows us to prune all program blocks whose costs are guaranteed to be independent of MR resources (line 7, see Subsection 3.4). Thus, pruning prevents unnecessary enumeration, compilation and costing for the r^i dimension. For all remaining program blocks B', we evaluate the second dimension according to the current r^c memory budget (lines 8-10). This evaluation employs a simple memoization structure *memo*, which is an $n \times 2$ matrix, where the i^{th} row refers to program block B_i, the first column represents resources r^i, and the second column related costs $C(B_i, r^i)$. For each triple of (r^c, B_i, r^i), we recompile the program block, cost the generated partial runtime plan and update the *memo* structure if we found a better r^i configuration (lines 11-13). After enumerating all points of the second dimension, we have found the best r^i for each program block B_i given a fixed r^c value. We then compile the entire program according to the memoized resource vector (line-14) in order to take the overall control structure like loops into account when estimating the total costs. This cost estimate is used to compare and update the globally best resources and costs (lines 15-17). Finally, we return $\mathcal{R}_P^*$, as the resource configuration with minimal costs (line 18).

3.3.2 Grid Point Generators

In this algorithm, the chosen strategy for enumerating grid points per dimension is important for finding near-optimal plans with low overhead. We now discuss alternative strategies, all of which are bounded by min_{cc} and max_{cc}.

Equi-Spaced Grid: As a first strategy, we use an equi-spaced grid with fixed-size gaps g between points per dimension as shown in Figure 5(a). In case of a fixed number of points m per dimension, we compute g by $g = (max_{cc} - min_{cc})/(m-1)$; otherwise, we use $g = min_{cc}$. Equi-spaced enumeration allows for a systematic evaluation of the search space without fundamental assumptions and without danger of missing large areas of resource configurations. The finer the grid granularity, the more likely we find the optimal resource configuration but we cannot give an optimality guarantee because the actual optimum might be in between points. However, in practice even 15 to 45 points per dimension are enough to find near-optimal plans.

Exp-Spaced Grid: Our second strategy aims to reduce the number of points without affecting the allocation quality. A common observation is that for small resource configurations, we see more frequent plan changes than for large configurations. Our exp-space grid exploits this observation by exponentially increasing gaps g between points as shown in Figure 5(b). The i^{th} gap is computed by $g_i = w^{i-1} \cdot min_{cc}$, where by default we use $w = 2$. This exp-spaced grid only requires a logarithmic number of points, while the equi-spaced grid requires a linear number of points. Especially in cluster environments with large max_{cc} constraint, this significantly reduces the number of generated points. However, this strategy can be too coarse-grained for extending the mapper memory without hurting the degree of parallelism.

Memory-Based Grid: While both the equi-spaced and exp-spaced grid are independent of the ML program at hand, our third strategy additionally exploits program-specific characteristics. A key observation is that plan changes often occur at resource configurations equal to the memory estimates of operations. Our memory-based grid strategy accordingly leverages the memory estimates of the compiler as shown in Figure 5(c). We still want to discretize the search space and hence combine this with the equi-spaced grid. In detail, whenever there exists a memory estimate between two grid points, we enumerate both left and right points. For memory estimates outside the min_{cc} and max_{cc} constraints, we fall back to these extreme values. In the worst-case (where memory estimates are spread across the entire search space), this strategy is equivalent to the equi-spaced grid. However, this strategy is now aware of the input program and its data. Different data leads to different memory estimates and thus different enumerated grid points. This approach allows a more directed search. There are, however, also counterexamples where enumeration solely based on memory estimates will not produce the best plan. For example, consider two large matrix multiplications $\mathbf{X}\,\mathbf{v}$ and $\mathbf{X}\,\mathbf{w}$; in order to pack them into one MR job for scan sharing of $\mathbf{X}$, both $\mathbf{v}$ and $\mathbf{w}$ together need to fit in the mapper memory. Decisions like this are not captured by the individual operation memory estimates. We currently use the memory estimates of all program blocks to materialize a single grid for all program blocks (see Algorithm 1, line 2).

Composite Grids: These basic grid generators can now be used to create composite grid generators. First, we can select different generators for each dimension. The resulting grid is defined by the cross product of points per dimension. Second, we can compose one-dimensional generators with a simple union. For example, our default *hybrid* grid enumeration strategy uses for both dimensions an overlay of (1) the memory-based grid and (2) the exp-spaced grid in order to combine both directed and systematic search.

Discussion on Gradient-Based Methods: We could view the two dimensional problems as general optimization problems and apply gradient-based methods. However, these methods do not apply here because it is not a convex optimization problem. Different resource configurations can result in the same plan and thus the same costs. These cost plateaus would lead to gradients of 0. In addition, the high impact of resource configurations motivated us to rather rely on a systematic, compiler-assisted search space exploration.

3.4 Pruning Techniques

Independent of the chosen grid enumeration strategy, we apply simple yet very effective pruning techniques.

Pruning Blocks of Small Operations: We can exploit the search space property of *monotonic dependency elimination* for additional pruning. As a precondition for pruning, for each r^c value, we compile the entire program with min_{cc} as baseline. Whenever, a program block does not contain any MR jobs, the second dimension of MR memory is irrelevant and can be pruned. Furthermore, if a program block does not contain any MR jobs for r_1^c, it will not reintroduce them for a larger CP memory $r_2^c > r_1^c$. This fact leads to pruning of entire areas as shown in Figure 5(d). An important resulting property of this grid pruning is that the number of pruned points depends on the data size. The smaller the data, the earlier every operation fits in memory and we can prune the second dimension. This property is important for small relative optimization overhead. For small data (where execution time is likely small), we prune more points and thus have very small overhead; for larger data, we prune fewer points, but the higher optimization overhead is justified by higher expected execution time.

Pruning Blocks of Unknowns: Pruning blocks of small operations works very well if sizes of intermediates can be inferred. However, if inference is not entirely possible, even a single unknown operation per block prevents the pruning of the second dimension of this block. One key observation is that if we are not able to infer intermediate sizes, often entire blocks become unknown. Clearly, if sizes and hence memory estimates are unknown, we will not find different plans with different costs. We exploit this property by pruning all blocks where all MR operations have unknown dimensions.

4. RUNTIME RESOURCE ADAPTATION

Optimizing resource configurations via online what-if analysis fundamentally depends on size inference over the entire ML program. Sizes of intermediate results—in terms of dimensions and sparsity—are important for memory estimation, plan generation, and thus cost estimation. Even with advanced strategies for size propagation and memory estimation, there are fundamental challenges that ultimately lead to unknowns and hence the need for runtime plan adaptation. Examples are (1) conditional size and sparsity changes, (2) user-defined functions with unknown output sizes, and (3) data dependent operations. For instance, the following contingency-table/sequence statement—as used for multinomial logistic regression—takes the multi-valued $n \times 1$ label vector $\mathbf{y}$ and creates a boolean $n \times k$ indicator matrix $\mathbf{Y}$, where k is the number of categories (classes).

```
1: Y = table( seq(1,nrow(X)), y );
```

The size of this intermediate $\mathbf{Y}$ determines the size of the gradient and thus affects operations on $\mathbf{X}$. Without looking at the data of $\mathbf{y}$, memory estimates are extremely overestimated. Based on all these unknowns, the initial resource optimization might not have created a near-optimal configuration. However, as we execute the ML program, unknowns like data sizes and sparsity become known and can be exploited for runtime plan adaptation. This shares common goals with adaptive query processing [16]. From a runtime perspective, adjusting the memory configuration of stateless jobs or reducing the CP AM memory are trivial, while increasing this memory requires obtaining a new container and hence needs state migration, potentially to a container running on a different node. In this section, we describe our simple and robust approach for runtime adaptation. We give an overview of resource adaptation, as well as details on runtime migration support, and optimizer extensions for when and how to reoptimize resource configurations.

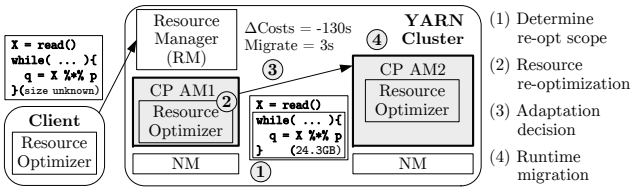


Figure 6: Overview Runtime Resource Adaptation.

4.1 Overview and Runtime Migration

Our basic approach for runtime adaptation is integrated with our default dynamic recompilation [5] at the granularity of last-level program blocks. If a block is subject to dynamic recompilation and if after dynamic recompilation at least one MR job exits, we trigger our runtime resource optimization. The intuition is as follows: The common case of adaptation is to extend the AM memory, because operations do not fit in the current memory budget, which leads to spawning unnecessary MR jobs. Furthermore, the latency of an MR job covers potential optimization overheads.

Resource Reoptimization: As shown in Figure 6, runtime adaptation has four steps. First, we determine the program scope of re-optimization. Second, we leverage the previously described resource optimizer to find the optimal resource configuration, where we always update MR configurations. Third, we decide on runtime adaptation if the cost benefit of the new configuration amortizes the estimated migration costs. Fourth, if beneficial, we migrate the current AM runtime and resume execution in the new container.

AM Runtime Migration: Efficient runtime migration is a necessary precondition for any related optimization decisions. At a high level, our runtime migration relies on *AM process chaining*, which allows for transparent runtime resource adaptation, invisible to the client. Assume we decided to migrate the current CP runtime. The current AM first requests a new container according to the new CP resources. Then, we materialize the current runtime state. This state comprises the current position of execution as well as the stack of live variables. We currently store the state on HDFS, but in-memory transfer is possible as well. In detail, we write all dirty variables, i.e., all variables whose state differs from its HDFS representation. Furthermore, we also write the current position and the new resource configuration. Once the new runtime container is started, we compile the original script according to the new configuration, restore the stack of live variables, jump to the current po-

sition and resume execution. Note that export and restore of variables happens transparently through our buffer pool. This simple approach is safe because all our operators and UDFs are stateless and—due to migration at program block boundaries—all intermediates are bound to logical variable names. Finally, as soon as the program finishes, the chain of AM processes is rolled-in, in reverse order.

Discussion on Robustness: At this point, another advantage of our what-if resource optimizer becomes obvious. We do not need to serialize execution plans but can pass the original script. Any recompilation of the original script according to the new resource configuration leads to exactly the same plan (after recompilation) as the resource optimizer has found during optimization. Furthermore, keeping the entire chain of AM processes alive might seem like large unnecessary resource consumption. However, in practice this works very well because the chain usually contains just two processes and the initial process is often small.

4.2 Re-Optimization Scope and Decisions

Resource re-optimization is part of dynamic recompilation. Despite low optimization overhead, it is prohibitive to execute resource optimization for every program block, because this might add relatively large overhead and requires a larger scope to be effective. Hence, we apply resource optimization only if dynamic recompilation compiled MR jobs.

Re-Optimization Scope: Determining a large program scope P' for resource reoptimization aims to prevent local decisions where the migration costs might not be amortized or might lead to multiple runtime migrations. Recall that recompilation works on last-level program blocks. Starting there, our heuristic is to expand the scope from the current position to the outer loop or top level in the current call context (main or current function) to the end of this context. The intuition is that this covers most iterative scripts, where—at least within loops—dimensions do not change.

Example Re-Optimization Scope: In our example of multinomial logistic regression, the number of classes k determines the size of intermediates. For example, assume a $10^7 \times 10^2$ input matrix $\mathbf{X}$. On dense data, the memory requirement of below matrix multiplications is 8 GB for $k = 2$ classes, but 24 GB for $k = 200$ classes. Dynamic recompilation is triggered on last-level program blocks, i.e., lines 3 and 5. We extend the re-optimization scope to the entire outer `while` loop in order to (1) account for repeated execution, and (2) prevent unnecessary repeated runtime migrations.

```
1: while( !outer_converge ) { ...
2:   while( !inner_converge ) { ...
3:     Q = P[, 1:k] * X %*% V;
4:   }
5:   grad = t(X) %*% (P[, 1:k] - Y[, 1:k]);
```

Extended Resource Optimization: Resource optimization then reuses our core resource optimizer with one extension. Instead of returning the optimal resource configuration, we return both the globally optimal resources $\mathcal{R}_P^*$ and the locally optimal resource $\mathcal{R}_P^*|r^c$, given the current r^c configuration. The latter is used for cost comparison and as resource configuration if we decide against migration.

Adaptation Decision: We extend our cost model to compute the benefit of migration $\Delta C = C(P', \mathcal{R}_P^*) - C(P', \mathcal{R}_P^*|r^c)$ (i.e., $\Delta C \leq 0$) and the migration costs C_M. Migration costs are defined as the sum of IO costs for live variables and latency for allocating a new container. If the migration costs C_M are amortized by ΔC, we migrate as described in Subsection 4.1; otherwise we use $\mathcal{R}_P^*|r^c$ as the new configuration but continue in the current AM container.

5. EXPERIMENTS

Our experiments study the behavior of automatic resource optimization for a variety of representative use cases. We aim to understand end-to-end optimization potential as well as to quantify the efficiency and effectiveness of enumeration strategies. To summarize, the major findings are:

End-To-End Improvements: Our resource optimizer found in most cases near-optimal configurations. The improvements are significant due to reduced MR-job latency for mid-sized data and more scalable plans for large data. Avoided over-provisioning also led to good throughput. Sources of suboptimality are buffer pool evictions (only partially considered by our cost model) and unknown data sizes.

Optimization Overheads: On the tested variety of ML programs and data sizes, the optimization overhead was very low. However, both grid enumeration and pruning strategies are crucial to accomplish that, especially for larger ML programs. Parallel optimization (see Appx C) achieved additional improvements but is currently of no crucial necessity.

Runtime Resource Adaptation: Re-optimization and runtime migration are crucial for ML programs with initial unknowns. The end-to-end improvements are large, while the overhead of runtime migration is low. On the tested ML programs, only up to two migrations were necessary.

5.1 Experimental Setting

Cluster Setup: We ran all experiments on an 1+6 node cluster, i.e., one head node of 2x4 Intel E5530 @ 2.40 GHz-2.66 GHz with hyper-threading enabled and 64 GB RAM, as well as 6 nodes of 2x6 Intel E5-2440 @ 2.40 GHz-2.90 GHz with hyper-threading enabled, 96 GB RAM, 12x2 TB disk storage, 10Gb Ethernet, and Red Hat Enterprise Linux Server 6.5. We used Hadoop 2.2.0 and IBM JDK 1.6.0 64bit SR12. YARN's RM was configured with min/max allocation constraints of 512 MB and 80 GB, respectively; the NMs were also configured with resources of 80 GB. Whenever we configure a specific heap size, we set equivalent max/initial heap sizes and we request memory of 1.5x the max heap size in order to account for additional JVM requirements. Our HDFS capacity was 107 TB (11 disks per node), and we used an HDFS block size of 128 MB. Finally, we used SystemML (as of 10/2014) with defaults of 12 reducers (2x number of nodes) and a memory budget of 70% of the max heap size.

ML Programs and Data: In order to study the behavior of resource elasticity in various scenarios, we use five ML programs and generated both dense and sparse data of different dimensions. Table 1 gives an overview of the characteristics and script-level parameters of these ML programs. Note that we use full-fledged scripts that also compute various additional statistics such as residual bias etc. The number of lines and program blocks are indicators of the program size, which affects the optimization overhead. Two scripts exhibit unknown dimensions during initial compilation ('?'). The other attributes refer to script-level configurations: *icp* is a switch for computing model intercept and scale/normalize the base data, λ refers to a regularization constant, ϵ to the convergence tolerance, and *maxi* to the max number of iterations. This table also shows basic characteristics: Linreg DS (direct solve) is non-iterative, Linreg CG (conjugate

Table 1: Overview ML Program Characteristics.

Prog.	#Lines	#Blocks	?	Icp.	λ	ϵ	Maxi.
Linreg DS	209	22	N	0	0.01	N/A	N/A
Linreg CG	273	31	N	0	0.01	10^{-9}	5
L2SVM	119	20	N	0	0.01	10^{-9}	$5/\infty$
MLogreg	351	54	Y	0	0.01	10^{-9}	5/5
GLM	1,149	377	Y	0	0.01	10^{-9}	5/5

gradient) has a single loop, while the remaining scripts—L2SVM (L2-regularized support vector machine), MLogreg (multinomial logistic regression), and GLM (generalized linear model, poisson/log)—use nested loops for overall convergence and finding the optimal update per iteration. As test data, we use scenarios of XS (10^7 cells), S (10^8 cells), M (10^9 cells), L (10^{10} cells), and XL (10^{11} cells) data size with different number of columns (1,000/100) and sparsity (1.0, 0.01). The number of rows is #cells/#cols. All experiments use binary input data. For dense data, these scenarios correspond to 80 MB, 800 MB, 8 GB, 80 GB, and 800 GB.

Baseline Comparisons: To the best of our knowledge, SystemML is the first declarative ML system that automatically optimizes resource configurations. Hence, we compare against baselines with *static* resource configurations, which use exactly the same YARN runtime integration as our resource optimizer. In detail, we use max CP/MR heap size configurations of 512 MB/512 MB (`B-SS`), 53.3 GB/512 MB (`B-LS`), 512 MB/4.4 GB (`B-SL`), and 53.3 GB/4.4 GB (`B-LL`). The reasoning is that 512 MB is a minimum container request, 53.3 GB a maximum container request ($53.3\,\text{GB} \cdot 1.5 \approx 80\,\text{GB}$), and 4.4 GB is the maximum task container size that allows to exploit all 12 physical cores and disks per node ($12 \cdot 4.4\,\text{GB} \cdot 1.5 \approx 80\,\text{GB}$). Appx D also provides a runtime-level comparisons with SystemML's runtime on Spark.

5.2 End-to-End Baseline Comparison

In a first series of experiments, we investigate the end-to-end runtime improvement with optimized resource configurations compared to the described baselines. The purpose is to study the effectiveness of initial resource optimization, and hence we disable runtime resource adaptation for the moment. We report end-to-end client elapsed time, which already includes optimization overhead and read/write from/to HDFS. Since we aim for generality, we show how our optimizer behaves over a variety of scenarios.

Figure 7 shows the overall baseline comparisons of initial resource optimization (`Opt`) on Linreg DS. A summary of the found resource configurations is shown in Table 2, where we report CP and max MR sizes. Overall we see that `Opt` achieves in all scenarios an execution time close to the best baseline; in some cases even better performance. It is important to note that on different scenarios, different baselines perform best. There are several interesting observations:

First, Figure 7(a) shows the results for XS-L on dense1000 which is a compute-intensive use case. For scenario S (800 MB), we see that small CP sizes perform worse because the MR job latency is not amortized. However, for

Table 2: Opt Resource Config, Linreg DS, [GB].

Scenario	B-LL	(a)	(b)	(c)	(d)
XS	53.3/4.4	0.5/2	0.5/2	0.5/2	0.5/2
S	53.3/4.4	8/2	0.5/2	8/2	0.5/2
M	53.3/4.4	0.5/2	1/2	0.5/2	4/2
L	53.3/4.4	0.5/2	8/2	8/2	26.9/2
XL	53.3/4.4	8/2	8/2	53.4/12.8	22.7/12.8

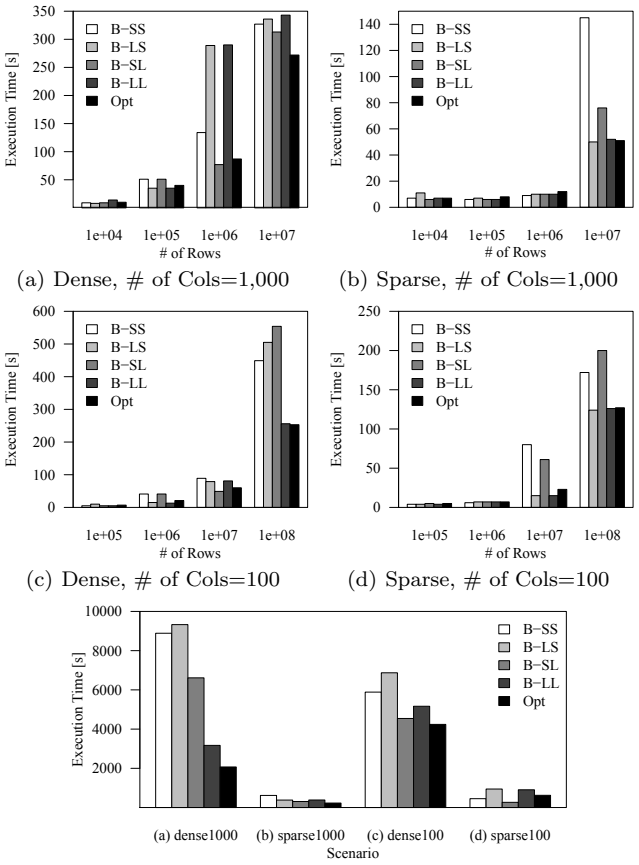
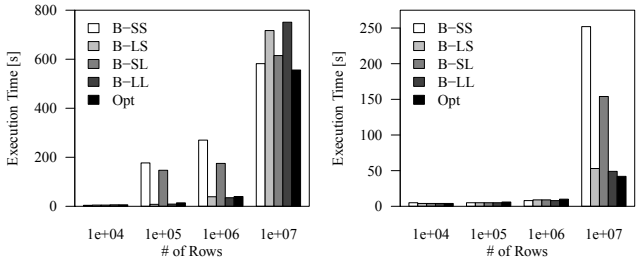


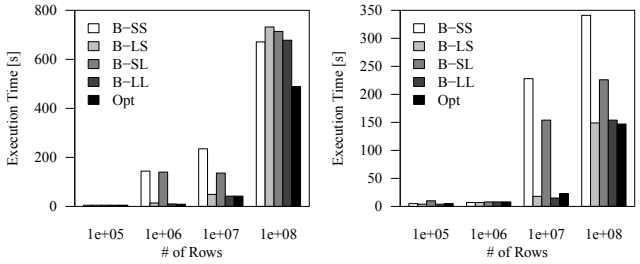
Figure 7: Experiments Linreg DS: Scenarios XS-XL.

scenario M (8 GB), small CP sizes perform 4x better than pure single node computations because the small CP sizes forced distributed runtime plans. `B-SS` is slower than `B-SL` because the small MR tasks caused too many concurrent tasks and hence cache trashing. `Opt` found very good plans in those scenarios. On scenario L (80 GB), we see that all baselines perform worse than `Opt` because they either use too few or too many tasks while our optimizer adjusts the number of tasks via a minimum task size based on the number of available virtual cores. Second, Figures 7(b) and 7(d) show the results of the sparse1000 and sparse100 scenarios where usually in-memory operations perform best. `Opt` also finds that plan with slightly worse performance due to more buffer pool evictions because of the smaller heap size. Third, Figure 7(c) shows the dense100 case which, in contrast to dense1000, is less compute-intensive and hence there are smaller differences. However, on L (80 GB) both large CP (for several vector operations) and large task sizes are important. `Opt` found these configurations and performed slightly better than `B-LL` because of no unnecessary over-provisioning which also limits the degree of parallelism.

Figure 7(e) shows the results of Linreg DS on scenario XL, which is 800 GB in dense and hence twice as large as aggregate cluster memory. For dense1000, we see again substantial improvements because, despite the good IO subsystem, the right plan (without large shuffling) really matters. `Opt` finds the best plan and is faster than `B-LL` due to a higher degree of parallelism. On dense100, `Opt` selects a plan with 13 GB tasks which cannot be chosen by the baselines. How-

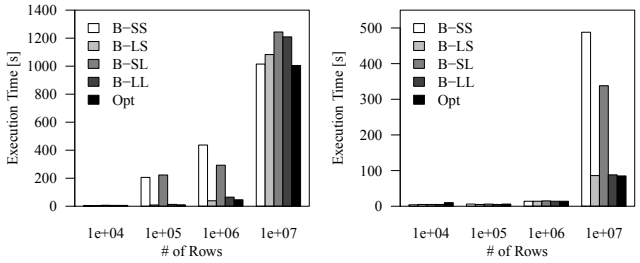


(a) Dense, # of Cols=1,000 (b) Sparse, # of Cols=1,000

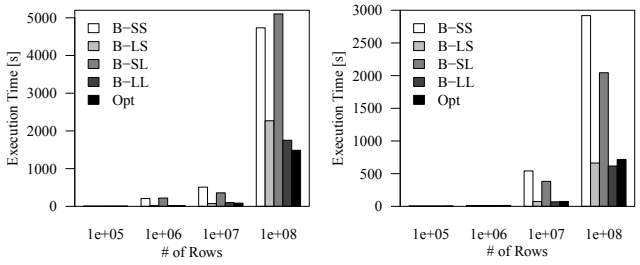


(c) Dense, # of Cols=100 (d) Sparse, # of Cols=100

Figure 8: Experiments Linreg CG: Scenarios XS-L.

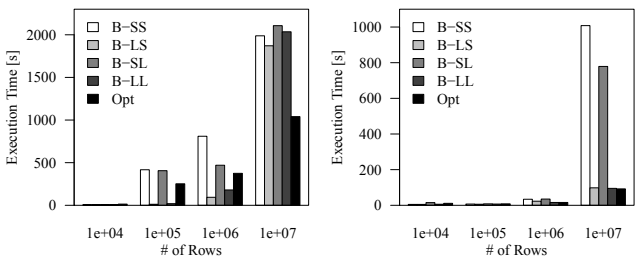


(a) Dense, # of Cols=1,000 (b) Sparse, # of Cols=1,000

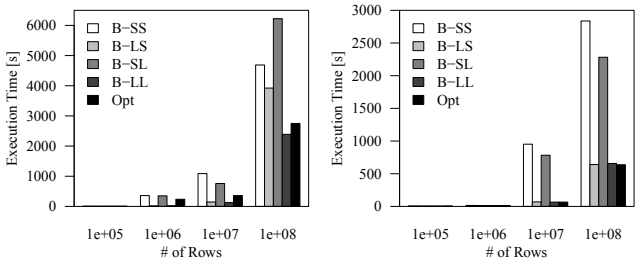


(c) Dense, # of Cols=100 (d) Sparse, # of Cols=100

Figure 9: Experiments L2SVM: Scenarios XS-L.

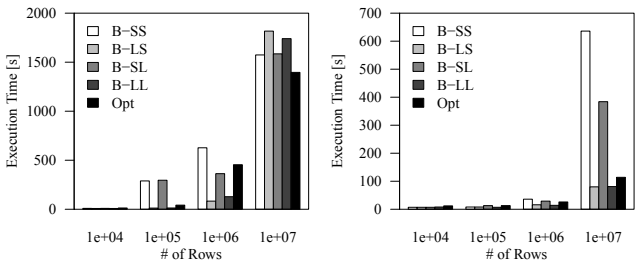


(a) Dense, # of Cols=1,000 (b) Sparse, # of Cols=1,000

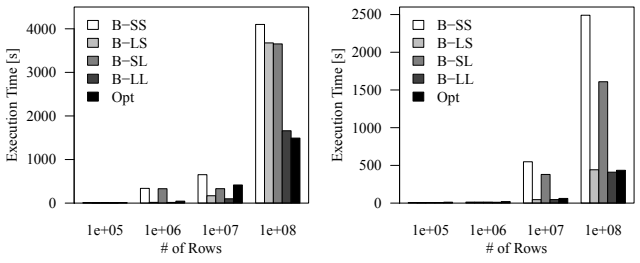


(c) Dense, # of Cols=100 (d) Sparse, # of Cols=100

Figure 10: Experiments MLogreg: Scenarios XS-L.



(a) Dense, # of Cols=1,000 (b) Sparse, # of Cols=1,000



(c) Dense, # of Cols=100 (d) Sparse, # of Cols=100

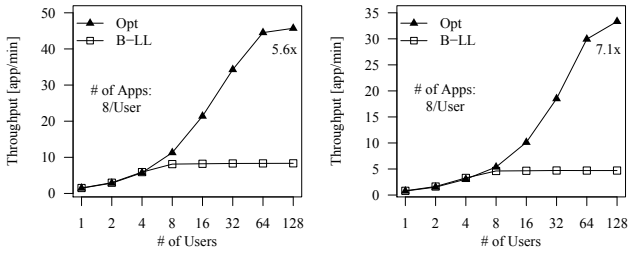
Figure 11: Experiments GLM: Scenarios XS-L.

ever, these large tasks reduce the degree of parallelism and thus are only moderately faster. The sparse cases are significantly faster but again `Opt` found a near-optimal plan. In all these scenarios, the optimization overhead was negligible because even on XL, after pruning only 3 blocks remained.

Figures 8-11 show the results for the remaining four ML programs on scenarios XS-L. There are three takeaways. First, on scenarios S and M, a larger CP memory usually leads to significant improvements, where `Opt` automatically finds these configurations. More iterations would lead to even higher improvements because the initial read is better amortized. Second, for scenario L, both CP and MR memory budgets matter, where `Opt` found again near-optimal configurations. Third, unknowns are a major problem. This applies to MLogreg and GLM. For all dense scenarios M, `Opt` was not able to find the right configuration here due to unknowns in the core loops. Sometimes, few known operations act as guards to find good plans. However, we address this problem in a principled way with CP migration.

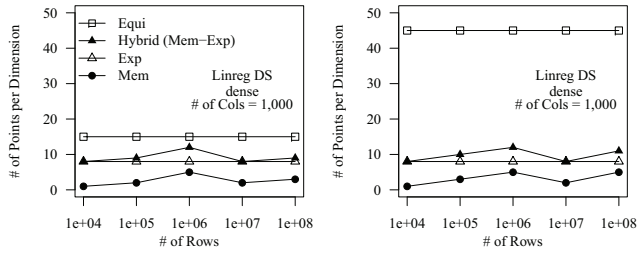
5.3 Throughput

In another series of experiments, we investigate our secondary optimization objective, namely preventing unnecessary over-provisioning. We aim to quantify the effect of over-provisioning via throughput experiments because the allocated resources per application directly affect the maximum number of parallel applications (users). We compare `Opt` with `B-LL` which was the most competitive baseline from an end-to-end runtime perspective. To measure throughput, we created a multi-threaded driver that spawns client processes at the head node. Each driver thread represents a user and each client process an application. We varied the number of users $|U| \in [1, 128]$, where each user is running 8 applications in order to account for setup and tear down.



(a) Linreg DS, S dense1000 (b) L2SVM, M, sparse100

Figure 12: End-to-End Throughput Comparison.



(a) Base Grid m=15 (b) Base Grid m=45

Figure 13: Comparison of Grid Generators.

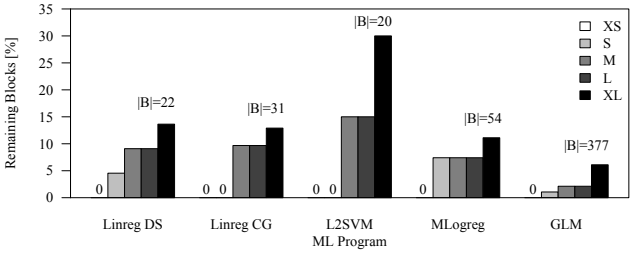


Figure 14: Effect of Block Pruning Strategies.

The reported throughput is measured as total number of applications divided by total driver execution time.

Figures 12(a) and 12(b) show the results for Linreg DS, scenario S, dense1000 and L2SVM, scenario M, sparse100, respectively. Our resource optimizer decided for 8 GB/2 GB and 4 GB/2 GB memory configurations, while `B-LL` always used 53.3 GB/4.4 GB. Given the described cluster allocation constraints, this led to a maximum application parallelism of $6 \cdot \lfloor 80\,\text{GB}/(1.5 \cdot 8\,\text{GB}) \rfloor = 36$ and 78 for `Opt` as well as 6 for `B-LL`. For this reason, there are no differences up to 4 users; but from 8 users onwards, we see increasing improvements. For `B-LL`, the throughput saturates at 6, while for `Opt`, it saturates according to the found memory configurations at 36 and 78 users. Similar but less significant effects can also be observed for larger scenarios including MR jobs.

5.4 Optimization Overhead

In a third series of experiments, we investigate the efficiency of our resource optimizer in terms of optimization overhead. Since grid enumeration and pruning techniques are largely orthogonal, we examine them individually and subsequently report the overall optimization times.

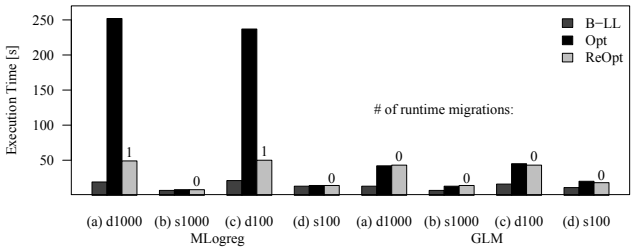
Grid Enumeration Strategies: Our overall algorithm materializes grid points per dimension and enumerates the cross product of those points. Since grid generators can be arbitrarily combined, we compare the different strategies with regard to one dimension. Figure 13 shows the number of generated points for the scenarios XS-XL of Linreg DS, dense1000. The cluster constraints were min_{cc}=512 MB and max_{cc} = 53.3 GB. We compare the equi-spaced grid (`Equi`), exponentially-spaced grid (`Exp`), memory-based grid (`Mem`), and our composite hybrid grid (`Hybrid`), where both `Equi` and `Mem` rely on a basic grid. First, Figure 13(a) compares those strategies on a grid of m=15 points. Obviously, `Equi` and `Exp` are independent of the input data size, where `Exp` requires only 8 instead of 15 points. `Mem` (and thus also `Hybrid`) depends on the input data and often requires only few points. For small data, all memory estimates are below min_{cc} and hence min_{cc} constitutes the only point. With increasing data size, we get more points (5 at 8 GB); if the data size exceeds max_{cc}, some estimates are truncated at max_{cc} and hence usually we get fewer points. Second, Figure 13(b) shows the results for m=45 points which leads to a larger difference between `Equi` and `Exp`. In addition, we see slightly more points for `Mem` due to smaller bin sizes. To summarize, our default strategy `Hybrid` naturally adapts to the problem, has good coverage but requires only few points which is important due to the resulting squared grid.

Pruning Strategies: Pruning of program blocks is of crucial importance to our algorithm because it significantly reduces the search space by eliminating the entire second dimension of each pruned block. Figure 14 shows the percentage of remaining blocks after pruning for all ML programs on dense, #Cols=1,000 data. We see that pruning blocks of small operations is highly beneficial. However, without pruning unknowns, both MLogreg and GLM would have a constant offset of 14 and 64, regardless of the actual data size which would really hurt for small scenarios.

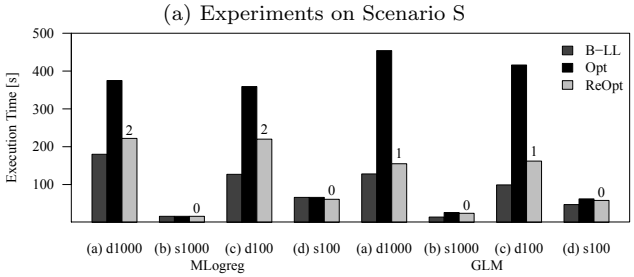
Overall Optimization Overhead: To summarize, we now report the optimization times for all used ML programs on use case dense1000. Table 3 shows for the different programs and scenarios, the number of block recompilations, the number of cost model invocations, the optimization time, and the relative overhead regarding total execution time. Note that costing the entire program is counted as a single cost model invocation, and we used `Hybrid`, m=15 for both dimensions as well as sequential enumeration as used for our previous baseline comparisons. Appendix C reports further results on parallel resource optimization. Most algorithms require a very low optimization time, except GLM which is a very complex ML program with $|B| = 377$ program blocks.

Table 3: Optimization Details Dense1000.

Prog.	Scen.	# Comp.	# Cost.	Opt. Time	%
Linreg DS	XS	352	8	**0.35s**	3.5
	S	417	30	**0.41s**	1.0
	M	678	168	**0.71s**	0.8
	L	448	104	**0.56s**	0.2
	XL	536	149	**0.73s**	<0.1
Linreg CG	XS	496	8	**0.43s**	7.2
	S	558	9	**0.47s**	3.3
	M	924	192	**0.88s**	2.2
	L	640	152	**0.66s**	0.1
L2SVM	XS	320	8	**0.36s**	6.0
	S	360	9	**0.41s**	4.1
	M	660	192	**0.85s**	1.8
	L	464	152	**0.71s**	<0.1
MLogreg	XS	1,188	11	**0.99s**	7.1
	S	1,476	299	**1.91s**	0.8
	M	2,148	650	**2.89s**	0.8
	L	1,540	363	**2.07s**	0.2
GLM	XS	8,294	11	**4.56s**	35.1
	S	8,518	235	**5.78s**	13.8
	M	11,576	1,034	**11.17s**	2.4
	L	8,998	715	**9.24s**	0.7



(a) Experiments on Scenario S



(b) Experiments on Scenario M

Figure 15: End-to-End Baseline Comparisons with Runtime Plan Adaptation for MLogreg and GLM.

5.5 Runtime Plan Adaptation

Unknown size and sparsity of intermediate results is a major issue, where initial resource optimization systematically leads to suboptimal choices. This problem of unknowns applies to MLogreg and GLM. We now study the benefits of runtime resource adaptation as introduced in Section 4.

Figure 15 shows the end-to-end comparison results of the ML programs MLogreg and GLM with enabled runtime resource adaptation. We compare again total elapsed time of baseline `B-LL`, our resource optimizer without adaptation (`Opt`), and with adaptation (`ReOpt`). We also annotated the number of runtime migrations. First, Figure 15(a) shows the results on scenario S (800 MB in dense). For this small case, we see high benefits of adaptation because eliminating the unnecessary MR job latency has high impact due to the small data size. Even a single runtime migration was sufficient to come close to the best baseline. Furthermore, there was no negative impact on use cases where no adaptation was required. It is noteworthy that GLM did not require any adaptation on this scenario because for example on dense1000, costs of a few known operations led to an initial CP size of 2 GB while for MLogreg the initial CP size remained at the minimum of 512 MB. Second, Figure 15(b) shows the results on scenario M (8 GB in dense). We see that now both GLM and MLogreg require runtime adaptation, which has again significant impact but not as high as for scenario S because the relative effect of MR job latency becomes less important. Again even one or two adaptations were sufficient to achieve near-optimal performance.

Finally, we explain the reasons for the moderately worse performance compared to the best baseline. For example consider MLogreg, scenario S, dense1000. Even before computing the `table()` expression that is responsible for most unknowns, the original AM spawned an unnecessary job on `rowSums(X^2)` for initializing the trust region. The actual migration then took less than $3s$ out of which we spend only $381ms$ for writing six intermediates to HDFS. Note that we did not write the largest input $\mathbf{X}$ because it was unmodified. In general, additional overheads stem from more buffer pool evictions and sometimes reading the input data again.

6. DISCUSSION

Optimizing resource configurations for a given ML program via online what-if analysis significantly improved the performance of SystemML on top of MapReduce as a stateless computing framework. In this section, we discuss questions related to stateful frameworks like Spark, cluster-utilization based adaptation, additional resources, and simplicity as the motivation for what-if analysis.

Resource Elasticity for Stateful Frameworks like Spark: Spark [49] on YARN uses standing containers for its executors that hold in-memory data partitions and need to accommodate intermediate results of all tasks. Despite sharing opportunities for broadcast data, this implies that similar resource-aware what-if analysis techniques could be used to automatically size executors and decide on the degree of parallelism. Spark keeps its statically configured executor resources for the lifetime of an application because YARN does currently not allow to resize containers and a reallocation would lose in-memory data. Existing work on recommending Spark resources [37] relies on blockmanager traces of previous runs. In contrast, resource optimization could help to reduce unnecessary over-provisioning to increase cluster throughput for *unseen* ML programs and data. Appendix D provides an initial potential analysis. However, resource elasticity for frameworks like Spark also faces additional challenges like the integration with lazy evaluation.

Cluster-Utilization-Based Adaptation: Our runtime resource adaptation addresses unknown sizes of intermediates. However, there are also use cases for resource adaptation to changing cluster utilization. For example, consider scenarios where we decided to use distributed plans in order to exploit full cluster parallelism but the cluster is heavily loaded. In those situations, a fallback to single node in-memory computation might be beneficial. This would require extended strategies for *when* to trigger resource re-optimization depending on cluster utilization, which can be incorporated into the presented what-if analysis framework. There is already related work on the impact of varying storage access costs on plan choices in DBMS [38]. This is an interesting direction for future work.

Additional Resources Beyond Memory: We formulated the ML program resource allocation problem general enough to cover arbitrary resources. So far, however, we mainly focused on the most important resource instantiation of memory because currently (1) SystemML's in-memory runtime is single-threaded and (2) YARN schedulers use by default a so-called `DefaultResourceCalculator` which only considers memory. Integrating additional resources like the number of cores would essentially increase the dimensionality of the search space. There are interesting side-effects to consider: e.g., usually the degree of parallelism affects memory requirements [6]. Hence, it fits into our what-if analysis framework but additional pruning strategies are necessary. We leave this as another item for future work.

Simplicity and Robustness: Finally, we like to stress the importance of simple and robust resource optimization. Consider agile environments with very short release cycles. Our resource optimizer relies on online what-if analysis to prevent compiler changes and costing runtime plans takes all compilation steps into account. At the same time, we exploit information from the compiler for enumeration and pruning. This design led to a resource optimizer that automatically adapts to compiler or runtime changes which is invaluable.

7. RELATED WORK

What-if optimizer interfaces and parameter tuning have been studied intensively. More recent work also addresses declarative ML and resource elasticity. However, our resource optimizer is the first work on automatic optimization of resource configurations for declarative ML programs.

ML Frameworks on YARN: Any large-scale ML system on top of MapReduce—like Mahout [3], SystemML [20], or Cumulon [24]—works, due to binary compatibility, out-of-the box on YARN but still with static cluster configurations. Other frameworks like Spark [49], REEF [9] or Flink [32]—which also provide ML capabilities—explicitly exploit YARN for stateful distributed caching but again rely on static configurations. This requires the user to reason about cluster configurations and potentially leads to over-provisioning. However, there do exist interesting early ideas on *reactive* resource-elastic machine learning algorithms for stateful frameworks like REEF [35]. This work specifically aims at algorithm-specific extensions for graceful scale-down in case of task preemption. Instead of restarting the entire ML program, they approximate the loss function of lost partitions and continue the overall optimization. In contrast, we focus on the general case of declarative ML and *proactive* online what-if analysis for resource configurations.

In-Memory ML and Data Analytics: For declarative ML with hybrid runtime plans, memory is often used as constraint [5, 6]. This inherently leads to memory-sensitive plans. The problem is further intensified by advanced optimization techniques, out of which many directly affect memory requirements. Examples are task-parallel ML programs where the degree of parallelism affects the number of intermediates [6], operator selection involving broadcast joins [5, 40, 48], NUMA-aware data replication [51], materialization and subsampling for model and data reuse [50], sparsity-aware matrix multiplication chain optimization [5, 28], as well as reducing the number of intermediates in arithmetic expressions [5, 45, 52]. However, none of these existing works automatically optimize resource configurations.

MR-Job Parameter Tuning: There is also existing work on understanding and tuning MR job configurations. PerfXplain [29] allows performance queries on the impact of different job configurations. Furthermore, the Starfish project explored profiling and cost-based parameter optimization for MR jobs via what-if analysis [21], where they used black-box gridding and recursive random search as search strategies. PStorM [18] extended this work to profile matching and re-occurring MR job optimization. MRTuner [43] further improved this line of work with an analytic producer-transporter-consumer model. In contrast to our work, these approaches target the general case of arbitrary MR jobs with UDFs but (1) focus on one job at-a-time, (2) rely on logs of previous executions or sampling to obtain execution costs, and (3) are not applicable for resource-aware optimization due to missing interactions with the compiler.

Cloud-Aware ML Workflow Tuning: Cloud settings offer an elastic tradeoff between execution time and monetary cost [26]. Elastisizer [22] (part of the Starfish project) automatically solves cluster sizing problems for general-purpose data-intensive analytics. The authors optimized for execution time given a provided configuration search space. This area is relevant because node types constitute resource configurations as well. Cumulon [24]—as a declarative ML system—also specifically addressed this cloud setting by optimizing the monetary cost of an ML program under time constraints. Recently, Cumulon also considers the challenging auction-based market of computing resources such as Amazon's spot instances [25], where they respect additional risk tolerance constraints. In contrast to our resource optimizer, these works optimize for a fixed program or physical plan due to missing interactions with the compiler.

Elasticity in DBMS and Array Stores: Existing work on elasticity in distributed DBMS and array stores mostly target skewed OLTP workloads and expanding distributed array databases. This includes techniques for live partition migration in shared storage [14] and shared nothing [19] architectures. Accordion [42] addresses the problem of when and which partitions to migrate, via an optimization algorithm based on linear programming. The algorithm is invoked periodically, and minimizes the amount of data moved under various constraints like max load capacity and max memory capacity per node. In contrast, elasticity for array databases [17] specifically targets growing databases. In this context, they propose strategies for deciding when to expand and for incremental partitioning, also under the objective of minimizing data movement. In contrast, our approach does not focus on stateful databases but plan-aware what-if analysis for unseen ML programs and input data.

Physical Design Tuning: There is a long history of physical design tuning via what-if analysis, which is typically done offline with advisors for indexes, partitioning, and materialized views [2], as well as multi-dimensional clustering [54] or compression [30]. This area is relevant because it deals with repeated optimization regarding different configurations. The INdex Usage Model (INUM) [36] reuses internal query plans that are independent of data access operators and a cache of reusable plans for more complex scenarios. Configuration-parametric optimization [8] generates a partial memo structure with extension points (so-called access-path-request operators) and reuses this structure for arbitrary configurations. This requires optimizer extensions to consider all relevant plans and prevent too eager branch-and-bound pruning. In contrast, we rely on very lightweight compiler interactions for enumeration and pruning.

8. CONCLUSIONS

We introduced a systematic approach to resource elasticity for large-scale ML in order to address the problem of memory-sensitive plans. Our simple and robust resource optimizer finds—via online what-if analysis—near-optimal resource configurations with low optimization overhead. In conclusion, automatic resource optimization is another step towards declarative large-scale machine learning. Automatic resource optimization frees the user from deciding on static cluster configurations, which is hard because it requires an understanding of generated runtime plans and it is always a compromise between a variety of ML programs and specialized systems in a multi-tenancy environment. In those environments, resource elasticity also allows to scale-up ML systems as required and still share cluster resources due to the integration with global resource management. Our results are also broadly applicable and can be transferred to any declarative ML system that supports both scale-up and scale-out according to cluster configurations. Future work includes task-parallel ML programs and resource elasticity for stateful runtime frameworks, which both require extended cost estimation and runtime migration strategies.

Acknowledgement: We thank Alexandre Evfimievski and Prithviraj Sen for creating the used DML scripts and Umar Farooq Minhas and our reviewers for valuable comments.

9. REFERENCES

[1] D. Abadi et al. The Beckman Report on Database Research. *SIGMOD Record*, 43(3), 2014.

[2] S. Agrawal, E. Chu, and V. R. Narasayya. Automatic Physical Design Tuning: Workload as a Sequence. In *SIGMOD*, 2006.

[3] Apache. *Mahout: Scalable machine learning and data mining*. `https://mahout.apache.org/`.

[4] M. Boehm. Costing Generated Runtime Execution Plans for Large-Scale Machine Learning Programs. *ArXiv e-prints*, 2015. arXiv:1503.06384 [cs.DC].

[5] M. Boehm, D. R. Burdick, A. V. Evfimievski, B. Reinwald, F. R. Reiss, P. Sen, S. Tatikonda, and Y. Tian. SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. *IEEE Data Eng. Bull.*, 37(3), 2014.

[6] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. Burdick, and S. Vaithyanathan. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *PVLDB*, 7(7), 2014.

[7] V. R. Borkar, Y. Bu, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Declarative Systems for Large-Scale Machine Learning. *IEEE Data Eng. Bull.*, 35(2).

[8] N. Bruno and R. V. Nehme. Configuration-Parametric Query Optimization for Physical Design Tuning. In *SIGMOD*, 2008.

[9] B. Chun, T. Condie, C. Curino, R. Ramakrishnan, R. Sears, and M. Weimer. REEF: Retainable Evaluator Execution Framework. *PVLDB*, 6(12), 2013.

[10] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD Skills: New Analysis Practices for Big Data. *PVLDB*, 2(2), 2009.

[11] T. Condie, P. Mineiro, N. Polyzotis, and M. Weimer. Machine Learning for Big Data. In *SIGMOD*, 2013.

[12] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Cetintemel, and S. Zdoni. Tupleware: "Big" Data, Big Analytics, Small Clusters. In *CIDR*, 2015.

[13] A. Crotty, A. Galakatos, and T. Kraska. Tupleware: Distributed Machine Learning on Small Clusters. *IEEE Data Eng. Bull.*, 37(3), 2014.

[14] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration. *PVLDB*, 4(8), 2011.

[15] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[16] A. Deshpande, Z. G. Ives, and V. Raman. Adaptive Query Processing. *Foundations and Trends in Databases*, 1(1), 2007.

[17] J. Duggan and M. Stonebraker. Incremental Elasticity for Array Databases. In *SIGMOD*, 2014.

[18] M. Ead, H. Herodotou, A. Aboulnaga, and S. Babu. PStorM: Profile Storage and Matching for Feedback-Based Tuning of MapReduce Jobs. In *EDBT*, 2014.

[19] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *SIGMOD*, 2011.

[20] A. Ghoting, R. Krishnamurthy, E. P. D. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative Machine Learning on MapReduce. In *ICDE*, 2011.

[21] H. Herodotou and S. Babu. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. *PVLDB*, 4(11), 2011.

[22] H. Herodotou, F. Dong, and S. Babu. No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics. In *SOCC*, 2011.

[23] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[24] B. Huang, S. Babu, and J. Yang. Cumulon: Optimizing Statistical Data Analysis in the Cloud. In *SIGMOD*, 2013.

[25] B. Huang, N. W. D. Jarrett, S. Babu, S. Mukherjee, and J. Yang. Cumulon: Cloud-Based Statistical Analysis from Users Perspective. *IEEE Data Eng. Bull.*, 37(3), 2014.

[26] K. Kambatla, A. Pathak, and H. Pucha. Towards Optimizing Hadoop Provisioning in the Cloud. In *HotCloud*, 2009.

[27] K. Karanasos, A. Balmin, M. Kutsch, F. Ozcan, V. Ercegovac, C. Xia, and J. Jackson. Dynamically Optimizing Queries over Large Scale Data Platforms. In *SIGMOD*, 2014.

[28] D. Kernert, F. Koehler, and W. Lehner. SpMachO - Optimizing Sparse Linear Algebra Expressions with Probabilistic Density Estimation. In *EDBT*, 2015.

[29] N. Khoussainova, M. Balazinska, and D. Suciu. PerfXplain: Debugging MapReduce Job Performance. *PVLDB*, 5(7), 2012.

[30] H. Kimura, V. R. Narasayya, and M. Syamala. Compression Aware Physical Database Design. *PVLDB*, 4(10), 2011.

[31] D. Lyubimov. *Mahout Scala Bindings and Mahout Spark Bindings for Linear Algebra Subroutines*. Apache, 2014. `http://mahout.apache.org/users/sparkbindings/ScalaSparkBindings.pdf`.

[32] V. Markl. Breaking the Chains: On Declarative Data Analysis and Data Independence in the Big Data Era. *PVLDB*, 7(13), 2014.

[33] K. Morton, M. Balazinska, and D. Grossman. ParaTimer: A Progress Indicator for MapReduce DAGs. In *SIGMOD*, 2010.

[34] A. C. Murthy, V. K. Vavilapalli, D. Eadline, J. Niemiec, and J. Markham. *Apache Hadoop YARN: Moving Beyond Mapreduce and Batch Processing with Apache Hadoop 2*. Pearson Education, 2013.

[35] S. Narayanamurthy, M. Weimer, D. Mahajan, T. Condie, S. Sellamanickam, and K. Selvaraj. Towards Resource-Elastic Machine Learning. In *NIPS workshop Biglearn*, 2013.

[36] S. Papadomanolakis, D. Dash, and A. Ailamaki. Efficient Use of the Query Optimizer for Automated Database Design. In *VLDB*, 2007.

[37] C. Reiss and R. H. Katz. Recommending Just Enough Memory for Analytics. In *SOCC*, 2013.

[38] F. Reiss and T. Kanungo. A Characterization of the Sensitivity of Query Optimization to Storage Access Cost Parameters. In *SIGMOD*, 2003.

[39] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. Nobody Ever Got Fired for Using Hadoop on a Cluster. In *HotCDP*, 2012.

[40] S. Schelter, C. Boden, M. Schenck, A. Alexandrov, and V. Markl. Distributed Matrix Factorization with MapReduce using a series of Broadcast-Joins. In *RecSys*, 2013.

[41] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *EuroSys*, 2013.

[42] M. Serafini, E. Mansour, A. Aboulnaga, K. Salem, T. Rafiq, and U. F. Minhas. Accordion: Elastic Scalability for Database Systems Supporting Distributed Transactions. *PVLDB*, 7(12), 2014.

[43] J. Shi, J. Zou, J. Lu, Z. Cao, S. Li, and C. Wang. MRTuner: A Toolkit to Enable Holistic Optimization for MapReduce Jobs. *PVLDB*, 7(13), 2014.

[44] M. A. Soliman, L. Antova, V. Raghavan, A. El-Helw, Z. Gu, E. Shen, G. C. Caragea, C. Garcia-Alvarado, F. Rahman, M. Petropoulos, F. Waas, S. Narayanan, K. Krikellas, and R. Baldwin. Orca: A Modular Query Optimizer Architecture for Big Data. In *SIGMOD*, 2014.

[45] S. Sridharan and J. M. Patel. Profiling R on a Contemporary Processor. *PVLDB*, 8(2), 2014.

[46] M. Stonebraker, P. Brown, D. Zhang, and J. Becla. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science and Engineering*, 15(3), 2013.

[47] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SOCC*, 2013.

[48] L. Yu, Y. Shao, and B. Cui. Exploiting Matrix Dependency for Efficient Distributed Matrix Computation. In *SIGMOD*, 2015.

[49] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.

[50] C. Zhang, A. Kumar, and C. Ré. Materialization Optimizations for Feature Selection Workloads. In *SIGMOD*, 2014.

[51] C. Zhang and C. Re. DimmWitted: A Study of Main-Memory Statistical Analytics. *PVLDB*, 7(12), 2014.

[52] Y. Zhang, H. Herodotou, and J. Yang. RIOT: I/O-Efficient Numerical Computing without SQL. In *CIDR*, 2009.

[53] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu. Fuxi: a Fault-Tolerant Resource Management and Job Scheduling System at Internet Scale. *PVLDB*, 7(13), 2014.

[54] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. J. Storm, C. Garcia-Arellano, and S. Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB*, 2004.

APPENDIX

A. EXAMPLE ML PROGRAM

The following example gives an idea of how exactly high-level ML scripts with an R-like syntax usually look like. In detail, we use a simplified version of L2SVM (support vector machine) as introduced in Subsection 5.1. This formulation solves the primal SVM optimization problem. Note that this script specifies the precise algorithm semantics but does not include any details of runtime plans or data representations.

```
X = read($X);
Y = read($Y);
intercept = $icpt; epsilon = $tol;
lambda = $reg; maxiterations = $maxiter;
num_samples = nrow(X); dimensions = ncol(X);
num_rows_in_w = dimensions;
if (intercept == 1) {
   ones  = matrix(1, rows=num_samples, cols=1);
   X = append(X, ones);
   num_rows_in_w = num_rows_in_w + 1;
}
w = matrix(0, rows=num_rows_in_w, cols=1);
g_old = t(X) %*% Y;
s = g_old; iter = 0;
Xw = matrix(0, rows=nrow(X), cols=1);
continue = TRUE;
while( continue & iter < maxiterations ) {
   # minimizing primal obj along direction s
   step_sz = 0;
   Xd = X %*% s;
   wd = lambda * sum(w * s);
   dd = lambda * sum(s * s);
   continue1 = TRUE;
   while( continue1 ) {
      tmp_Xw = Xw + step_sz*Xd;
      out = 1 - Y * tmp_Xw;
      sv = ppred(out, 0, ">");
      out = out * sv;
      g = wd + step_sz*dd - sum(out * Y * Xd);
      h = dd + sum(Xd * sv * Xd);
      step_sz = step_sz - g/h;
      if( g*g/h < 0.0000000001 ) {
         continue1 = FALSE;
      }
   }
   w = w + step_sz*s; #update weights
   Xw = Xw + step_sz*Xd;
   out = 1 - Y * Xw;
   sv = ppred(out, 0, ">");
   out = sv * out;
   obj = 0.5 * sum(out*out) + lambda/2 * sum(w*w);
   print("ITER " + iter + ": OBJ=" + obj);
   g_new = t(X) %*% (out * Y) - lambda * w;
   tmp = sum(s * g_old);
   if( step_sz*tmp < epsilon*obj ) {
      continue = FALSE;
   }
   #non-linear CG step
   be = sum(g_new*g_new)/sum(g_old*g_old);
   s = be * s + g_new;
   g_old = g_new; iter = iter + 1;
}
write(w, $model)
```

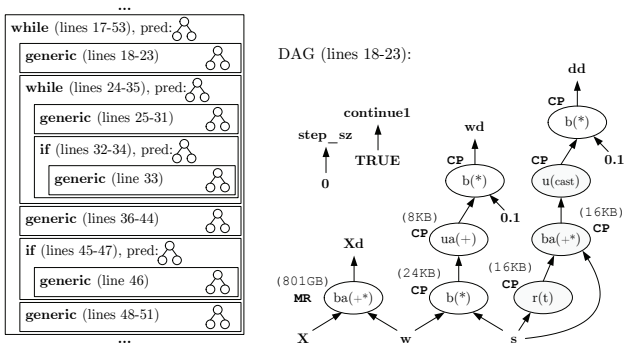
B. ML PROGRAM COMPILATION

In addition to the overview in Subsection 2.1, we now discuss the compilation process of SystemML in more detail. We use L2SVM from Appendix A as a running example.

Language-Level: In a first step, we parse the given ML script into a hierarchy of statement blocks and statements, using a generated parser. This statement block structure is naturally given by control flow constructs of the script as shown in Figure 16(a). We also do a semantic validation and read basic meta data like input data types, dimensions, and number of non-zeros (sparsity). Depending on the input format this meta data is either given in the header of the input (e.g., matrix market), or by a JSON meta data file.

HOP-Level: In a second step, we construct HOP DAGs for each statement block as partially shown in Figure 16(a). At HOP level, we apply optimizations like matrix multiplication chain optimization, common subexpression elimination, constant folding, algebraic simplifications, and other program rewrites. As an example, consider the statement block for lines 18-23. First, a static rewrite (independent of size) transforms $sum(\mathbf{s} \cdot \mathbf{s})$ into $sum(\mathbf{s}^2)$ since unary operations are easier to parallelize. Second, since $\mathbf{s}$ is a column vector, a dynamic rewrite, transforms $sum(\mathbf{s}^2)$ into $\mathbf{s}^\top\mathbf{s}$ to prevent the intermediate result of $\mathbf{s}^2$. Figure 16(b) shows the resulting HOP DAG. Other program rewrites include, for instance, branch removal. Consider the if branch on lines 7-11 and an input parameter $\$icpt = 0$. After constant folding, the predicate is known to be false and hence, we remove the branch which allows unconditional size propagation. Subsequently, we do intra-/inter-procedural analysis for size propagation and compute memory estimates per HOP operator based on these propagated sizes. These memory estimates are later used for operator selection in order to decide between in-memory (CP) and large-scale (MR) computation but also to decide upon alternative physical operators, some of which are constrained by available MR task memory.

LOP-Level: In a third, step, we construct a LOP DAG for each HOP DAG. First, we decide the execution type with a simple yet very effective heuristic. Whenever an operation fits into the memory budget of CP (given by a ratio of the max JVM size), we assume that in-memory computation is more efficient and decide for execution type CP. Second, for each HOP, we construct a sub-DAG of LOPs, where we choose from alternative LOP operators. For example, we provide six MR and three CP matrix multiplication operators. There are also physical operators for special HOP operator patterns like tertiary-aggregate operators for $sum(\mathbf{v1} \cdot \mathbf{v2} \cdot \mathbf{v3})$ (e.g., see lines 29/30). Third, during LOP



(a) Statement Blocks (b) HOP DAG (after rewrites)

Figure 16: Example ML Program Compilation.

Table 4: Memory-Sensitive Compilation Steps.

Compilation Step	Examples
Operator Selection (Exec Type)	CP vs MR (all HOP operators with at least one matrix input or output)
Operator Selection (Physical Ops)	(a) MapMM (map matrix mult) (b) MapMMChain (MapMM chain) (c) PMM (permutation matrix mult) (d) Map* (elementwise matrix-vector)
HOP-LOP Rewrites	(a) Avoid large transpose by transpose-mm rewrite: $\mathbf{X}^\top\mathbf{v} \rightarrow (\mathbf{v}^\top\mathbf{X})^\top$. (b) CP vs MR data partitioning for all map-side binary operators.
Piggybacking (MR jobs)	Bin packing constrained by sum of memory requirements of physical operators.

DAG construction, we apply specific HOP-LOP rewrites. Examples are decisions on data partitioning, unnecessary aggregation, and specific rewrites like $\mathbf{X}^\top\mathbf{v} \rightarrow (\mathbf{v}^\top\mathbf{X})^\top$ that are applied depending on the chosen LOP operator. Fourth, we generate the execution plan, by creating a runtime program consisting of program blocks and executable instructions. We create a program block per statement block and use a piggybacking algorithm to generate instructions for the entire DAG. Piggybacking tries to pack the given LOP DAG into a minimal number of MR jobs under several constraints like job types, execution location of instructions (e.g., map, aggregation, or reduce), and memory constraints.

Runtime-Level: In a fourth step, we also allow for dynamic recompilation during runtime if the initial size propagation was not able to resolve all unknowns. In this case, operator selection also marks DAGs for recompilation. Program blocks have references to the original HOP DAGs in order to regenerate runtime plans at any time during program execution. The recompilation hooks are given by the natural program structure or by artificially created cuts of program blocks. The primary target of recompilation are last-level program blocks (generic) or predicates. During recompilation, we exploit the size information of intermediates and propagate them through the DAG before re-applying dynamic rewrites and runtime plan generation.

Summary Memory-Sensitive Compilation Steps: Many of the mentioned compilation steps include decisions related to available memory of CP or MR tasks. Table 4 gives examples of those decisions. First, the memory estimate of each HOP operator—with at least one input/output matrix—directly affects its execution type. This allows us, for example, to schedule the entire inner loop of L2SVM into CP memory. Second, efficient physical operators for matrix multiplication and binary elementwise operations are crucial. For example, consider the matrix-vector multiplications in lines 13, 20, and 43, which are the only operations on the large $\mathbf{X}$. A `mapmm` operator (that broadcasts the vector) allows us to compute this multiplication in the mappers without the need for data shuffling of $\mathbf{X}$ but requires that the vector fits into the MR task memory. Third, rewrites like $\mathbf{X}^\top\mathbf{v} \rightarrow (\mathbf{v}^\top\mathbf{X})^\top$ are only applied if the newly introduced operators can be executed in CP memory. Fourth, also piggybacking of operators into MR jobs is constrained by memory. In addition, there are also memory-related cost factors during runtime. The buffer pool size is relative to the memory budget which affects evictions. In conclusion, there are many memory-sensitive compilation decisions across the entire compilation chain. Online what-if analysis automatically takes all of them and there inter-dependencies into account by generating and costing runtime plans.

C. PARALLEL RESOURCE OPTIMIZER

In order to reduce optimization overhead for large ML programs, we exploit the property of *semi-independent problems* for parallelization. We aim for a scalable algorithm without global synchronization barriers, with small serial fraction, and without overhead for small or unknown problems.

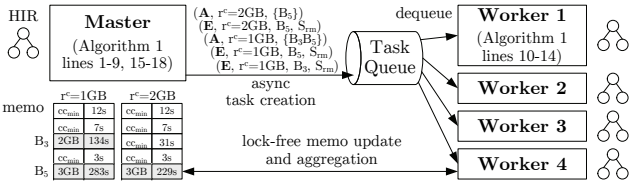
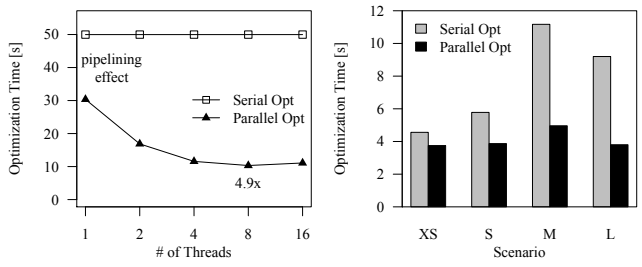


Figure 17: Parallel Resource Optimizer.

Figure 17 shows the architecture of our multi-threaded resource optimizer. We use a master and k parallel workers, where each worker creates its deep copy of the program and HOPs for concurrent recompilation. The basic idea is that the master enumerates CP memory budgets r^c, and does baseline compilation and pruning, while workers process optimization tasks of enumeration (`Enum_Srm`, (r^c, B'_i, S_{r^m})), and aggregation (`Agg_rc`, (r^c, B')). For each r^c, we create (1) for each $B'_i \in B'$ an `Enum_Srm` task, and (2) a single `Agg_rc` task. We use a central task queue for load balancing, where workers dequeue and execute tasks. This task-based approach is similar to parallel query optimization in Orca [44] but due to a fixed dependency structure does not require a specialized scheduler. We extend our *memo* structure to an $n \times 2 \cdot |S_{r^c}|$ matrix that replicates the *memo* structure for each r^c value, for which storage overhead is negligible. For `Enum_Srm` tasks, workers enumerate the second dimension and finally do a *lock-free* memo update of locally optimal resources and costs. For `Agg_rc` tasks, workers probe the memo structure until all blocks for a given r^c value have been updated and determine the aggregated program costs. After all tasks have been executed, the master picks the resources with minimal aggregated costs. For small problems, we do not instantiate parallel workers if the initial baseline compilation resulted in $B' = \emptyset$. This architecture enables (1) task-parallel optimization of semi-independent problems, and (2) pipelining of baseline compilation and enumeration.

Experiments Optimization Overhead: Figure 18 shows a comparison of serial and parallel optimization on GLM, dense, #Cols=1,000; ran on the head node (2x4 cores) of our cluster. Figure 18(a) shows `Equi` with a base grid of m=45. Even for one worker thread, there is an improvement due to pipeline parallelism. Figure 18(b) shows our default `Hybrid` and all scenarios. On scenarios M and L the benefit increases due to more points and less pruned blocks.



(a) `Equi`, m=45, dense1000 L (b) `Hybrid`, All Scenarios

Figure 18: Parallel Optimization for GLM.

Table 5: Spark Comparison: Different Data Sizes.

Scenario (⋆ CP only)	SystemML on MR w/ `Opt`	SystemML on Spark Plan 1 (Hyb.)	SystemML on Spark Plan 2 (Full)
XS (80 MB)	**6 s**⋆	25 s	59 s
S (800 MB)	**12 s**⋆	31 s	126 s
M (8 GB)	**40 s**⋆	43 s	184 s
L (80 GB)	836 s	**167 s**	347 s
XL (800 GB)	12,376 s	**10,119 s**	13,661 s

Table 6: Spark Comparison: Throughput #Users.

# of Users (Scenario S)	SystemML on MR w/ `Opt`	SystemML on Spark Plan 2 (Full)
1	**5.1 app/min**⋆	0.48 app/min
8	**35.6 app/min**⋆ (7.0x)	0.84 app/min (1.8x)
32	**69.8 app/min**⋆ (13.7x)	0.83 app/min (1.7x)

D. SPARK COMPARISON EXPERIMENTS

We also aim to analyze the potential of resource optimization for stateful frameworks like Spark [49].

Experimental Setting: The setup is the same as described in Subsection 5.1. We used Spark 1.2 (12/2014) and ran it in *yarn-cluster* mode, with a static resource configuration of 6 executors, 20 GB driver memory, 55 GB executor memory, and 24 cores per executor. Note that Spark (1) relies on static default configurations or per application parameters, and (2) uses more aggressive ratios for JVM overheads, which enabled to run all 6 executors besides the driver. SystemML uses the introduced resource optimizer and the application use case is L2SVM from Appendix A.

Baseline Comparison: In the interest of a fair comparison, we ported our SystemML runtime operations on Spark's JavaPairRDDs and compared (a) SystemML w/ resource optimizer on MR (w/o jvm reuse) with (b) SystemML runtime on Spark. Since Spark is on a lower abstraction level, we had to hand-code execution plans. We literally translated the L2SVM script and created two plans: (1) Hybrid, where only operations on $\mathbf{X}$ are RDD operations (lines 13, 20, and 43), while all others are CP-like operations, and (2) Full, where all matrix operations are RDD operations. Table 5 shows the results. First, single-node CP operations are important for small data sets (e.g., see scenarios XS-M, Spark Plan 1 and 2). Up to scenario M, Spark Plan 1 does not fully utilize all executor cores. Second, Spark has a sweet spot, due to RDD caching, where data does not fit in a single node but fits in aggregated memory (L). Third, for large data (>2x aggregated memory) there are no significant differences due to similar disk IO and deserialization costs.

Throughput Comparison: We also compare the resulting throughput of SystemML with our resource optimizer against SystemML on Spark, Plan 2 (Full). The use case is L2SVM, scenario S (800 MB) and we varied the number of users $|U|$. We reduced Spark's driver memory to 512 MB in order to prevent cluster deadlocks without constraining `maximum-am-resource-percent`. Table 6 shows the results. Moderate resource requirements of SystemML ($8GB$ CP memory, 1 core, no MR jobs) enabled a throughput improvement of 13.7x at 32 users. The speedup is suboptimal due to IO bandwidth saturation (disk/memory). In contrast, due to over-provisioning, even a single Spark application already occupied the entire cluster. The slight throughput increase with more users is due to latency hiding of driver setup and few overlapping applications each with fewer executors.

In conclusion, resource optimization is also important for frameworks like Spark to achieve high performance without over-provisioning, while maintaining application isolation.