Building Scalable and Flexible Cluster Managers Using Declarative Programming

Lalith Suresh, Joao Loff[1], Faria Kalim[2], Sangeetha Abdu Jyothi[3], Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, Michael Gasch
VMware, [1]IST (ULisboa) / INESC-ID, [2]UIUC, [3]UC Irvine and VMware

Abstract

Cluster managers like Kubernetes and OpenStack are notoriously hard to develop, given that they routinely grapple with hard combinatorial optimization problems like load balancing, placement, scheduling, and configuration. Today, cluster manager developers tackle these problems by developing system-specific best effort heuristics, which achieve scalability by significantly sacrificing the cluster manager's decision quality, feature set, and extensibility over time. This is proving untenable, as solutions for cluster management problems are routinely developed from scratch in the industry to solve largely similar problems across different settings.

We propose DCM, a radically different architecture where developers specify the cluster manager's behavior declaratively, using SQL queries over cluster state stored in a relational database. From the SQL specification, the DCM compiler synthesizes a program that, at runtime, can be invoked to compute policy-compliant cluster management decisions given the latest cluster state. Under the covers, the generated program efficiently encodes the cluster state as an optimization problem that can be solved using off-the-shelf solvers, freeing developers from having to design ad-hoc heuristics.

We show that DCM significantly lowers the barrier to building scalable and extensible cluster managers. We validate our claim by powering three production-grade systems with it: a Kubernetes scheduler, a virtual machine management solution, and a distributed transactional datastore.

1 Introduction

Today's data centers are powered by a variety of cluster managers like Kubernetes [10], DRS [47], Openstack [15], and OpenShift [14]. These systems configure large-scale clusters and allocate resources to jobs. Whether juggling containers, virtual machines, micro-services, virtual network appliances, or serverless functions, these systems must enforce numerous cluster management *policies*. Some policies represent *hard constraints*, which must hold in any valid system configuration; e.g., "each container must obtain its minimal requested amount of disk space". Others are *soft constraints*, which reflect preferences and quality metrics; e.g., "prefer to scatter replicas across as many racks as possible". A cluster manager therefore solves a challenging combinatorial optimization problem of finding configurations that satisfy hard constraints while minimizing violations of soft constraints.

Despite the complexity of the largely similar algorithmic problems involved, different cluster managers in various contexts tackle the configuration problem using custom, system-specific best-effort heuristics—*an approach that often leads to a software engineering dead-end* (§2). As new types of policies are introduced, developers are overwhelmed by having to write code to solve arbitrary combinations of increasingly complex constraints. This is unsurprising given that most cluster management problems involve *NP-hard combinatorial optimization* that cannot be efficiently solved via naive heuristics. Besides the algorithmic complexity, the lack of separation between the cluster state, the constraints, and the constraint solving algorithm leads to high code complexity and maintainability challenges, and hinders re-use of cluster manager code across different settings (§2). Anecdotally, developers at a large software vendor mention that even policy-level feature additions to cluster managers *take months* to develop.

Our contribution This paper presents *Declarative Cluster Managers* (DCM), a radically different approach to building cluster managers, wherein the implementation to compute policy-compliant configurations is *synthesized* by a compiler from a high-level specification.

Specifically, developers using DCM maintain cluster state in a relational database, and *declaratively* specify the constraints that the cluster manager should enforce using SQL. Given this specification, DCM's compiler synthesizes a program that, at runtime, can be invoked to pull the latest cluster state from the database and compute a set of policy-compliant changes to make to that state (e.g., compute optimal placement decisions for newly launched virtual machines). The generated program – an *encoder* – encodes the cluster state and constraints into an optimization model that is then solved using a constraint solver.

In doing so, DCM significantly lowers the barrier to building cluster managers that achieve all three of *scalability*, *high decision quality*, and *extensibility* to add new features and policies. In contrast, today's cluster managers use custom heuristics that heavily sacrifice both decision quality and extensibility to meet scalability goals (§2).

For scalability, our compiler generates implementations that construct highly efficient constraint solver encodings that scale to problem sizes in large clusters (e.g., 53% improved p99 placement latency in a 500 node cluster over the heavily optimized Kubernetes scheduler, §6.1).

For high decision quality, the use of constraint solvers under-the-covers guarantees optimal solutions for the specified problems, with the freedom to relax the solution quality if needed (e.g., 4× better load balancing in a commercial virtual machine resource manager, §6.2).

For extensibility, DCM enforces a strict separation between the a) cluster state, b) the modular and concise representation of constraints in SQL, and c) the solving logic. This makes it easy to add new constraints and non-trivial features (e.g., making a Kubernetes scheduler place both pods and virtual machines in a custom Kubernetes distribution, §6.3).

Several research systems [46, 53, 57, 78, 92] propose to use constraint solvers for cluster management tasks. These systems all involve a significant amount of effort from optimization experts to handcraft an encoder for specific problems with simple, well-defined constraints – let alone encode the full complexity and feature sets of production-grade cluster managers (e.g., Kubernetes has 30 policies for driving initial placement alone). Even simple encoders are challenging to scale to large problem sizes and are not extensible even when they do scale (§8). In fact, for these reasons, *constraint solvers remain rarely used* within production-grade cluster managers in the industry-at-large: none of the open-source cluster managers use solvers and, anecdotally, nor do widely used enterprise offerings in this space.

Instead, with DCM, developers need not handcraft heuristics nor solver encodings to tackle challenging cluster management problems.

Providing a capability like DCM is fraught with challenges. First, cluster managers operate in a variety of modes and timescales: from incrementally placing new workloads at millisecond timescales, to periodically performing global reconfiguration (like load balancing or descheduling); we design a programming model that is flexible enough to accommodate these various use cases within a single system (§3). Second, constraint solvers are not a panacea and are notoriously hard to scale to large problem sizes. DCM's compiler uses carefully designed optimization passes that bridge the wide chasm between a high-level SQL specification of a cluster management problem and an efficient, low-level representation of an optimization model – doing so leverages the strengths of the constraint solver while avoiding its weaknesses (§4).

Summary of results We report in-depth about our experience building and extending a Kubernetes Scheduler using DCM. We implement existing policies in Kubernetes in under 20 lines of SQL each. On a 500 node Kubernetes cluster on AWS, DCM improves 95th percentile pod placement latencies by up to 2×, is 10× more likely to find feasible placements in constrained scenarios, and correctly preempts pods 2× faster than the baseline scheduler. We also report simulation results with up to 10K node clusters and experiment with non-trivial extensions to the scheduler, like placing both pods and VMs within a custom Kubernetes distribution. We also use DCM to power a commercial virtual machine management solution where we improved load balancing quality by 4×. Lastly, we briefly discuss a distributed transactional datastore where we implemented several features with a few lines of SQL.

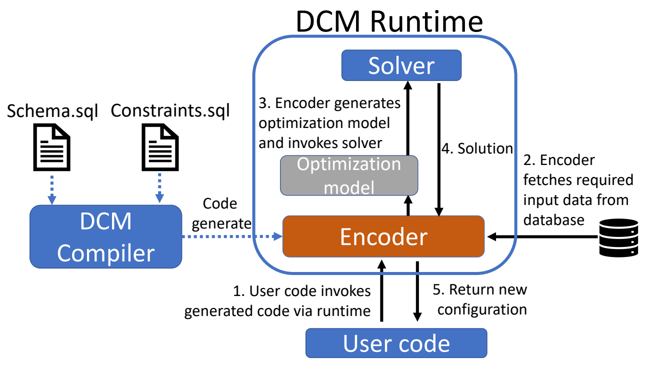

DCM Runtime
Solver
Schema.sql
Constraints.sql
3. Encoder generates optimization model and invokes solver
4. Solution
Optimization model
2. Encoder fetches required input data from database
DCM Compiler
Code generate
Encoder
1. User code invokes generated code via runtime
5. Return new configuration
User code

Figure 1: DCM architecture. Dotted lines show the compilation flow. Solid lines show runtime interactions between the DCM runtime, user code and the cluster state DB.

2 Motivation

Our motivating concern is that ad-hoc solutions for cluster management problems are regularly built from scratch in the industry, due to the wide range of specialized data-center environments and workloads that organizations have, for which off-the-shelf solutions do not suffice. Even beyond dedicated cluster managers like Kubernetes [10], OpenStack [15], and Nomad [50], similar capabilities are routinely embedded within enterprise-grade distributed systems like databases and storage systems: e.g., for policy-based configuration, data replication, or load-balancing across machines, all of which are combinatorial optimization problems.

Today, developers handcraft heuristics to solve these cluster management problems that incur significant engineering overhead. First, the heuristics are hard to scale to clusters with hundreds to thousands of nodes; they often require purpose-built and inflexible pre-computing and caching optimizations to remain tractable [40,95]. Even then, the heuristics are challenging to get right as developers have to account for arbitrary combinations of constraints. Second, the heuristics sacrifice decision quality to scale (e.g., load balancing quality), which is not surprising given that combinatorial optimization problems cannot be solved efficiently via naive heuristics. Third, they lead to complex code that makes it hard to extend and evolve the cluster manager over time; it is not uncommon for policy-level feature additions to take multiple months' worth of effort to deliver.

We illustrate the above challenges using Kubernetes as a representative example.

Kubernetes example The Kubernetes Scheduler is responsible for assigning groups of containers, called *pods*, to cluster

Policy	Description
H1-4	Avoid nodes with resource overload, unavailability or errors
H5	Resource capacity constraints: pods scheduled on a node must not exceed node's CPU, memory, and disk capacity
H6	Ensure network ports on host machine requested by pod are available
H7	Respect requests by a pod for specific nodes
H8	If pod is already assigned to a node, do not reassign
H9	Ensure pod is in the same zone as its requested volumes
H10-11	If a node has a 'taint' label, ensure pods on that node are configured to tolerate those taints
H12-13	Node affinity/anti-affinity: pods are affine/anti-affine to nodes according to configured labels
H14	Inter-pod affinity/anti-affinity: pods are affine/anti-affine to each other according to configured labels
H15	Pods of the same service must be in the same failure-domain
H16-20	Volume constraints specific to GCE, AWS, Azure.
S1	Spread pods from the same group across nodes
S2-5	Load balance pods according to CPU/Memory load on nodes
S6	Prefer nodes that have matching labels
S7	Inter-pod affinity/anti-affinity by labels
S8	Prefer to not exceed node resource limits
S9	Prefer nodes where container images are already available

Figure 2: Policies from the baseline Kubernetes scheduler, showing both hard (H) constraints and soft (S) constraints.

nodes (physical or virtual machines). Each pod has a number of user-supplied attributes describing its resource demand (CPU, memory, storage, and custom resources) and placement preferences (the pod's affinity or anti-affinity to other pods or nodes). These attributes represent hard constraints that must be satisfied for the pod to be placed on a node (H1–H20 in Table 2). Kubernetes also supports soft versions of placement constraints, with a violation cost assigned to each constraint (S1–S9 in Table 2). Like other task-by-task schedulers [15, 94, 95], the Kubernetes default scheduler uses a greedy, best-effort heuristic to place one task (pod) at a time, drawn from a queue. For each pod, the scheduler tries to find feasible nodes according to the hard constraints, score them according to the soft constraints, and pick the best-scored node. Feasibility and scoring are parallelized for speed.

Decision quality: not guaranteed to find feasible, let alone optimal, placements Pod scheduling is a variant of the multidimensional bin packing problem [18, 21], which is NP-hard and cannot, in the general case, be solved efficiently with greedy algorithms. This is especially the case when the scheduling problem is *tight* due to workload consolidation and users increasingly relying on affinity constraints for performance and availability.

To remain performant, the Kubernetes scheduler only considers a random subset of nodes when scheduling a pod, which might miss feasible nodes [93]. Furthermore, the scheduler may commit to placing a pod and deny feasible choices from pods that are already pending in the scheduler's queue (a common weakness among task-by-task schedulers [40]).

Feature limitations: best-effort scheduling does not support global reconfiguration Many scenarios require the scheduler to simultaneously reconfigure arbitrary groups of pods

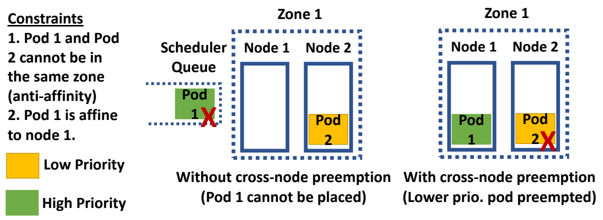

Constraints
1. Pod 1 and Pod 2 cannot be in the same zone (anti-affinity)
2. Pod 1 is affine to node 1.
Low Priority
High Priority
Scheduler Queue
Pod 1
Zone 1
Node 1
Node 2
Pod 2
Without cross-node preemption (Pod 1 cannot be placed)
Zone 1
Node 1
Node 2
Pod 1
Pod 2
With cross-node preemption (Lower prio. pod preempted)

Figure 3: An anti-affinity constraint prevents Pod 1 and Pod 2 from being in the same zone, pod 1 is affine to node 1, and pod 2 has a lower priority than pod 1. Placing pod 1 on node 1 requires evicting pod 2 on node 2.

and nodes. For instance, Figure 3 shows a scenario where a high priority pod (pod 1) can only be placed on node 1, but to do so, the scheduler has to preempt a lower priority pod on node 2. Computing this rearrangement requires simultaneous reasoning about resource and affinity constraints spanning multiple pods and nodes, which cannot be achieved in the current architecture. Thus, although such global reconfiguration is in high demand among users, it is unsupported in Kubernetes [60, 64].

Extensibility: Best-effort scheduling leads to complex code Similar to Borg [40, 95], Kubernetes needs careful engineering to keep scheduling tractable at scale. Several policies like inter-pod affinity (Table 2-H14) and service affinities (Table 2-H15) are compute intensive because they require reasoning over groups of pods. These policies are kept tractable using carefully designed caching and pre-computing optimizations that are fragile in the face of evolving requirements. For example, it is hard to extend inter-pod affinity policies to specify the number of pods per node [58, 59, 61–63], and there are discussions among developers to restrict these policies to make the code efficient [60]. For similar reasons, there are discussions among developers to remove the service affinity policy due to accumulating technical debt around its pre-computing optimizations [69]. Such complexity accumulates to make entire classes of policies requested by users difficult to implement in the scheduler [60, 64, 73].

Beyond policy-level extensions, the tight coupling between the cluster state representation in the scheduler's datastructures and the scheduling logic makes it near impossible to introduce changes to the underlying abstractions (e.g., extending the scheduler to also place tasks other than pods, like virtual machines [71]) without a complete rewrite [66].

3 Declarative Programming with DCM

Our position is that developers should specify cluster management policies using a high-level declarative language, and let an automated tool like DCM generate the logic that efficiently computes policy-compliant decisions. Architecturally, it allows the behavior of the cluster manager to be described and evolved independently of the implementation details.

We use SQL as the declarative language for specifying policies for multiple reasons. First, it allows us to consistently describe and manipulate both the cluster state and the constraints on that state. Second, it is a battle-tested and widely known language, which aids adoption. Third, it is sufficiently expressive that we have not felt the need for designing yet another DSL (§4.1.1).

We now demonstrate DCM's capabilities and programming model with a simplified guide to building a Kubernetes scheduler with it. Our scheduler operates as a drop-in replacement for the default scheduler (§2), supporting all of its capabilities and adding new ones.

The workflow in a DCM-powered scheduler consists of three steps (Figure 1). First, the scheduler stores the cluster state in an SQL database based on an SQL schema designed by the developer. Second, the developer extends the schema with scheduling constraints, also written in SQL. The compiler generates an encoder based on the constraints and schema. Third, at runtime, the scheduler invokes the generated encoder via the DCM library as new pods are added to the system. The generated encoder pulls the required cluster state from the database, produces and solves an optimization model that is parameterized by that state, and outputs the required scheduling decisions.

Cluster state database Kubernetes stores all state (of nodes and pods) in an etcd [36] cluster. The default scheduler maintains a cache of relevant parts of this state locally using in-memory data structures. In DCM, we replace this cache with an in-memory embedded SQL database (H2 [4]) and specify an SQL schema (tables and views) to represent the cluster state. Currently, the schema uses 18 tables and 12 views to describe the set of pods, nodes, volumes, container images, pod labels, node labels, and other cluster state. The developer annotates some columns in the schema as *decision variables*, i.e., variables to be assigned automatically by DCM. For example, a placement decision of a pod on a node is represented by the table in Figure 4 with decision variables (`node_name`) annotated as `@variable_columns` and other *input variables* supplied by the database.

Constraints Next, the developer specifies constraints against the cluster state as a collection of SQL views. DCM supports both hard and soft constraints, encompassing *all* the cluster management *policies* that the system must enforce.

Hard constraints are specified as SQL views with the annotation `@hard_constraint`. For example, consider the constraint in Figure 5, which states that pod P can be scheduled on node N if N has not been marked unschedulable by the operator, is not under resource pressure, and reports as being ready to accept new pods. We implement this by declaring a view, `constraint_node_predicates`, with a `check` clause

```
-- @variable_columns (node_name)
create table pods_to_assign
(
  pod_name varchar(100) not null primary key,
  status varchar(10) not null,
  namespace varchar(100) not null,
  node_name varchar(100),
  ... -- more columns
);
```

Figure 4: A table describing pods waiting to be scheduled. The `@variable_columns` annotation indicates that the `node_name` column should be treated as a set of decision variables. Other columns are input variables, whose values are supplied by the database.

```
create view valid_nodes as
select node_name from node_info
where unschedulable = false and memory_pressure = false
  and out_of_disk = false and disk_pressure = false
  and pid_pressure = false and network_unavailable = false
  and ready = true;

-- @hard_constraint
create view constraint_node_predicates as
select * from pods_to_assign
check (node_name in (select node_name from valid_nodes));
```

Figure 5: A hard constraint to ensure pods that are pending placement are never assigned to nodes that are marked unschedulable by the operator, are under resource pressure, or do not self-report as being ready to accept new pod requests.

that asserts that all pods must be assigned to nodes from the `valid_nodes` view computed in the database.

Soft constraints are also specified as SQL views with annotation `@soft_constraint` and contain a single record storing an integer value. DCM ensures that the computed solution *maximizes* the sum of all soft constraints. For example, consider CPU utilization load balancing policy across nodes in a cluster (Figure 6). We first write a convenience view (`spare_capacity_per_node`) that computes the spare CPU capacity after pod placement. We then describe a soft constraint view (`constraint_load_balance_cpu`) on the minimum spare capacity in the cluster. This forces DCM to compute solutions that maximize the minimum CPU utilization of nodes, thereby spreading pods across the cluster.

Compiler and runtime The DCM interface for programmers is shown in (Figure 8). The first step is invoking the DCM compiler using the schema and constraints as input. This generates a program (e.g., a Java program – §4.1.2) that pulls the required tables from the database, constructs an optimization model, and solves it using a constraint solver. The generated program is compiled using the relevant toolchain (e.g., `javac` – §4.1.2) and loaded into memory. The compiler returns a `Model` object that wraps the loaded program.

When pods are added to the system, the scheduler updates the relevant tables (like `pods_to_assign`). The scheduler

```
create view spare_capacity_per_node as
select (node.available_cpu_capacity
        - sum(pods_to_assign.cpu_request)) as cpu_spare
from node join pods_to_assign
  on pods_to_assign.node_name = node.name
group by node.name;

-- @soft_constraint
create view constraint_load_balance_cpu as
select min(cpu_spare) from spare_capacity_per_node;
```

Figure 6: A soft constraint that maximizes the minimum spare CPU capacity in the cluster for load balancing.

```
-- @hard_constraint
create view constraint_node_affinity as
select * from pods_to_assign
check (pods_to_assign.has_requested_node_affinity = false
       or pods_to_assign.node_name in
         (select node_name
          from candidate_nodes_for_pods
          where pods_to_assign.pod_name =
                candidate_nodes_for_pods.pod_name));
```

Figure 7: A membership constraint to describe node affinity (the pod must only be assigned to nodes it is affine to, as computed in another view `candidate_nodes_for_pods`)

then invokes `model.solve()` (Figure 8) to find an optimal placement for these pods by assigning values to `pods_to_assign.node_name` according to the specified constraints. The call returns a copy of the `pods_to_assign` table with the `node_name` column reflecting the computed optimal placement. The scheduler then uses this data to issue placement commands for each pod via the Kubernetes scheduling API (the same API used by the default scheduler).

Note that DCM treats the state database as the input to every call to `model.solve()`. It does not (and cannot) assume the cluster configuration changes based on the computed solution, because the caller may choose not to apply the solution, there may be errors during reconfiguration or numerous other external events. This is in sharp contrast to prior art that uses handcrafted solver encodings for specific cluster management tasks, where all the cluster state is duplicated within the solver's memory [40, 53, 57, 92] (§8).

Supporting diverse cluster management tasks and tunable search scopes DCM enables developers to arbitrarily tune the search space of a problem by controlling the data within the input tables and views. For example, consider the the set of pods in the `pods_to_assign` table. For fast incremental initial placement, we can populate the table with a fixed size batch of newly created pods only, and compute placement decisions for the entire batch at a time. For a pod preemption model (a kind of global reconfiguration), we populate the same table with previously placed pods and specify additional constraints that assign a bounded number of pods to a "null node" (representing preemption). Similarly, the

Operation	Description
`model = DCM.compile(schema)`	Invoke DCM compiler to synthesize an encoder from the SQL schema and constraints
`model.connect(db)`	Establish JDBC connection to the state DB
`model.solve(timeout)`	Solve constraints and return a set of tables

Figure 8: DCM interface

scheduler may dynamically sample the subset of nodes to be considered for placement in a given iteration.

In this manner, DCM allows developers to easily instantiate models that solve different sub-problems within a cluster manager. We implemented three models in our Kubernetes scheduler: one for fast initial placement, a slower timescale pre-emption model, and an admin-triggered tool for de-scheduling [68] which can be used to recover capacity from highly utilized nodes by terminating pods.

4 DCM Design

As we show in §3, DCM enables programmers to specify cluster management policies using a high-level declarative language familiar to most programmers, and code generate the logic that efficiently computes policy-compliant decisions. However, we have to address several key challenges to realize such a capability.

First, given the amount of expertise that is typically required to handcraft efficient optimization models and encodings, it is non-trivial to bridge the gap between the SQL representation of a problem and the generation of a corresponding encoder that interacts with solvers via their respective low-level APIs. We discuss how the DCM compiler synthesizes efficient encoders in §4.1.

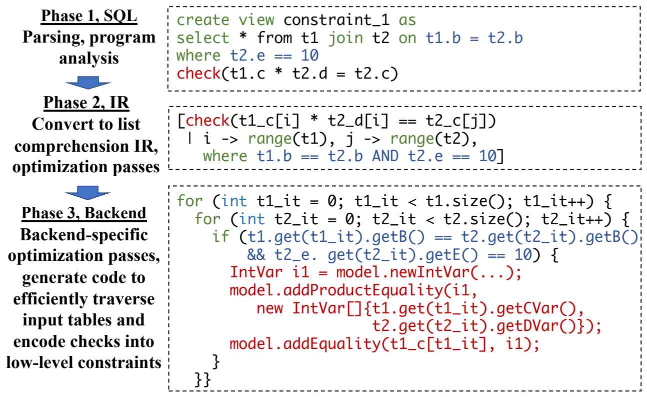
Second, given that programmers need systematic ways to test and debug the policies that they write, we describe how we leverage a common solver capability of finding *unsatisfiable cores* to aid in debugging (§4.2).

4.1 DCM compiler

The DCM compiler generates an encoder that produces optimization models according to the database schema and the constraints specified by the developer.

4.1.1 Syntax and expressiveness

The compiler accepts input SQL tables with variable columns of type integer, boolean, and string (floating point is supported if the backend solver supports it, like Gecode [2]). The compiler supports a subset of the SQL query language for constraint specification, including most commonly used constructs (inner join, anti-join, group by, aggregate queries, sub-queries, correlated sub-queries as seen in Figures 5 and 6, `ARRAY` columns), arithmetic and logical expressions (standard Boolean operators, linear arithmetic, comparisons over integers, and equality checks over strings), standard SQL aggre-


Phase 1, SQL
Parsing, program analysis
Phase 2, IR
Convert to list comprehension IR, optimization passes
Phase 3, Backend
Backend-specific optimization passes, generate code to efficiently traverse input tables and encode checks into low-level constraints

```
create view constraint_1 as
select * from t1 join t2 on t1.b = t2.b
where t2.e == 10
check(t1.c * t2.d = t2.c)
```

```
[check(t1_c[i] * t2_d[i] == t2_c[j])
 | i -> range(t1), j -> range(t2),
   where t1.b == t2.b AND t2.e == 10]
```

```
for (int t1_it = 0; t1_it < t1.size(); t1_it++) {
  for (int t2_it = 0; t2_it < t2.size(); t2_it++) {
    if (t1.get(t1_it).getB() == t2.get(t2_it).getB()
        && t2_e. get(t2_it).getE() == 10) {
      IntVar i1 = model.newIntVar(...);
      model.addProductEquality(i1,
         new IntVar[]{t1.get(t1_it).getCVar(),
                      t2.get(t2_it).getDVar()});
      model.addEquality(t1_c[t1_it], i1);
    }
  }}
```

Figure 9: System compiler workflow, showing a simple SQL query, its representation in the IR as a list comprehension, and a simplified version of the corresponding generated code by our or-tools backend.

gate functions (`sum`, `min`, `max`, `count`), and additional aggregates that help in constraint modeling (e.g., the `all_different` aggregate which enforces pair-wise inequalities among a set of variables (§4.1.3)). The compiler is extensible, allowing user-defined aggregates to be added with a few lines of code. We also take special care in dealing with SQL nulls, depending on the backend.

Both SQL tables and views computed in the database can be used as input relations for hard and soft constraints (Figures 5 and 6). This allows developers to efficiently construct inputs for a DCM `model` by exploiting the full extent of the SQL syntax and capabilities supported by the database. For example, the database can efficiently process joins to compute the required inputs for a model.

Using this syntax, we were able to compactly specify all hard and soft constraints encountered in our case studies, including resource capacity, affinity, anti-affinity, and load balancing constraints along various axes. SQL is significantly more concise than the low-level interfaces supported by solvers, with an SQL view compiling down to many low-level constraints.

4.1.2 Compiler workflow

Figure 9 shows the compiler's workflow in generating an encoder from the high-level SQL. The example shown is simplified Java generated by our OR-tools backend.

Phase 1, SQL Internally, the SQL parser first extracts all table and view definitions from the supplied database schema, and produces syntax trees for all the hard and soft constraints. It then performs a series of passes over the queries to infer whether parts of the constraints can be evaluated in the database. For example, 1) simple sub-queries that do not involve variables are better evaluated in the database rather than in the generated encoder or the solver, 2) some backends (like Minizinc) cannot efficiently compute the groups produced by an SQL `group by` because they cannot represent tuples – we can compute these groups in the database as well.

Phase 2, Intermediate representation Next, the compiler converts the query to an intermediate representation (IR) based on list-comprehension syntax [37]. In Figure 9, the SQL query is simplified in the IR as nested for loops and a filtering condition (instead of tables and predicates *across* various clauses of the SQL query). In general, the list comprehension syntax makes it easy to apply standard query optimization algorithms (like unnesting subqueries). It has been well-studied in the database community [23, 24, 37, 45, 55, 56].

Phase 3, Backend The compiler backend generates a program that produces an optimization model by interacting with the interfaces exposed by specific solver toolkits, e.g., setting up linear inequalities for an ILP solver. The IR facilitates support for multiple backends, allowing systems to benefit from different types of solvers. Regardless of the backend, the generated encoders prepare optimization models where variables and constraints are parameterized by the content of the cluster state database. At runtime, the encoder binds these variables to values extracted from the database before dispatching the optimization model to the solver.

OR-tools CP-SAT: Our 'flagship' backend generates encoders for the Google OR-tools CP-SAT solver [3]. It generates Java code to pull cluster state from the database at runtime and iterate over the state to encode constraints efficiently using the CP-SAT solver APIs. It translates joins into hash-table based accesses when feasible (e.g., equality-based joins using primary keys, unique columns specified via the use of SQL `distinct`), or into nested for loops otherwise (Figure 9). We generate several utility classes to aid the encoding (like type-safe tuple classes to refer to records from different tables). The backend performs common sub-expression elimination within the generated code fragments (e.g., when computing a complex expression within `if` or `for` blocks).

MiniZinc: Our initial DCM prototype interacted with the MiniZinc toolkit [80], which exposes a high-level constraint modeling language, and thereby supports integration with a variety of solvers. However, despite our best efforts, we could not scale it to clusters larger than 50-100 nodes due to the limited control we have over Minizinc encodings. However, we use it to interface with tools for debugging models §4.2.

4.1.3 Generating scalable encodings

We now discuss details of our backend for the Google OR-tools CP-SAT solver [3]. A CP-solver encoding specifies different constraints over input variables. For example, to encode a simple intermediate expression $a = (b < 10)$, we need to introduce a constraint $b < 10$, and link the truth value

```
// Unfixed arity
sum([1 | i -> range(t1)
    where t1.c1[i] = 10])
```

```
// Fixed arity
sum([(t1.c1[i] = 10)
    | i -> range(t1)])
```

Figure 10: Two equivalent IR expressions to compute the sum over a subset of a variable column (`t1.c1`).

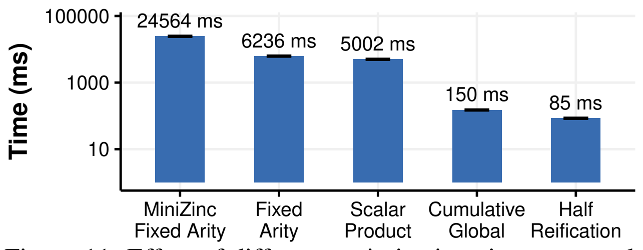


Figure 11: Effect of different optimizations in our or-tools backend on the runtime of a benchmark model (Figure 6). Not shown are option types, where runtimes exceed a minute.

of that constraint to a by introducing two more constraints $a \rightarrow (b < 10)$ and $\neg a \rightarrow (b \geq 10)$ (a process called *full reification*). At a high-level, CP-solvers find feasible solutions via a combination of search and *propagation*: at each node of the search tree, the solver iteratively changes the domain of a variable, causing constraints linked to that variable to update other variables they are linked to, and so on until a fixed point is reached. Therefore, every additional constraint and auxiliary variable we introduce impacts solver performance.

In short, like any constraint solver, the CP-SAT solver's performance is highly sensitive to the encoding (two equivalent encodings often result in vastly different solver runtimes). Hence, we employ various optimizations in DCM compiler based on structural information extracted from the SQL program and IR, to generate encoders that produce efficient models.

The key takeaway is that a literal translation of SQL queries into an encoding is *not scalable*, we therefore have to find smarter encodings. We do so by using re-writing rules that mimic what an optimization expert would otherwise handcraft into an encoder. We describe the impact of these rules using a benchmark that is bottlenecked by the `spare_capacity_per_node` view in Figure 6.

Re-writing to use fixed arity Consider the two expressions in our IR shown in Figure 10. Both these statements correspond to a high-level SQL operation to compute the number of elements in a variable column with values equal to 10. In the first case, we cannot statically determine the size of the filtered list in our encoder, because the values of the variables (and therefore the arity of the list to sum) are not *fixed yet*. The general way to encode such an expression is to use *option types* [75], where a variable might be 'absent'. However, several auxiliary variables and constraints need to be introduced to encode such an expression using option types. This problem is exacerbated by the join in `spare_capacity_per_node`, and where we do not know the arity of the produced *table*. An option type encoding for 1000 nodes and 50 pods for this view produces roughly *200K auxiliary variables and 400K constraints*, and makes our benchmark take more than a minute to complete.

Another approach is to avoid option types, and instead, compute a *sum of predicates* (Figure 10), which achieves the same result because the predicates evaluate to 0 or 1. Doing so keeps the number of auxiliary variables proportional to the number of predicates to evaluate, and brings the benchmark's runtime from minutes to 6.2s (Figure 11). The same encoding in MiniZinc takes 24s to solve, representing the inefficiencies introduced by high-level modeling frameworks.

Re-writing to use scalar products Re-writing to use fixed arity introduces $O(|pods| * |nodes|)$ reified boolean variables and constraints to encode whether a particular combination of rows from both tables appear in the final result set, which is then used to compute the sums required to specify hard constraints (like capacity bounds) and soft constraints (like load balancing requirements). Rather than naively iterate row by row to create several auxiliary variables that are summed to compute the load, our compiler infers that we are effectively computing a scalar product between two vectors, which can be expressed more efficiently with fewer auxiliary variables, which improves runtime by 20% (Figure 11). However, this does not still eliminate the $O(|pods| * |nodes|)$ boolean variables and is still prohibitively slow for large clusters.

Re-writing to use global constraints Global constraints are constraints over *groups* of variables for which solvers implement specialized and efficient propagator algorithms that dramatically reduce the search space of the problem. Encoding a problem using global constraints is key to scaling optimization models to large problem sizes because they typically avoid the need to generate too many auxiliary variables and constraints that burden the solver.

The join discussed above can be further optimized by using global constraints. Observe that we compute the load so that we can specify a capacity constraint on it (the load should be less than a constant). We can therefore detect this possibility and generate code to encode the join and the capacity constraint together as an *interval packing problem*: given a set of interval variables $I_1, I_2 \dots I_n$ of lengths $S_1, S_2 \dots S_n$, with demands $D_1, D_2 ... D_n$ pack them onto a timeline represented by the intervals, such that the total demand at any given instant of time is less than a capacity C (here, each interval variable represents a cell on the variable column, whereas the timeline represents rows on the column being joined to). This allows us to use a well known global constraint, *cumulative*. Crucially, this non-trivial encoding only introduces $O(|pods| + |nodes|)$ auxiliary variables (instead of $O(|pods| * |nodes|)$), that allows us to scale that SQL query to large cluster sizes, leading to a *96%* reduction in runtime over the previous optimization in our benchmark (Figure 11).

Another example of a global constraint is the `all_different` constraint that ensures a group of variables take different values (a common pattern in various anti-affinity policies). We also leverage membership global constraints where possible to encode the SQL `IN` operator.

As we mention in §4.1.1, we provide a suite of custom aggregate functions that developers may use in their models: these functions take advantage of the above global constraints under the covers.

Full- versus half-reification We already mentioned full reification (e.g., $a \leftrightarrow (b < 10)$). In several cases, it is both correct and more efficient to only introduce *half reified* constraints of the form $a \rightarrow (b < 10)$, because it introduces only half the constraints. Doing so is particularly effective to encode soft constraints for placement preferences in the following form: $b \rightarrow (var = A \vee var = B \ldots)$ (the solver tries to maximize b, which coaxes var to draw from a pool of preferred values). Again, using structural information from the IR, we can statically determine when we only need half-reified expressions. In our benchmark, this further yields a 43% improvement in performance (Figure 11).

4.2 Testing and debugging models

An important concern in using DCM is understanding *why* the cluster manager failed in finding a valid solution (e.g., a pod placement decision). Did the cluster run out of resources? Did the user mistakenly specify mutually contradicting constraints, e.g., affinity and anti-affinity over the same group of pods? Or was there a bug in the developer's constraint specification? DCM improves debuggability by taking advantage of a common solver capability: identifying *unsatisfiable cores* [72]. An unsatisfiable core is a minimal subset of model constraints and inputs that suffices to make the overall problem unsatisfiable (e.g., a single input variable that cannot simultaneously satisfy two contradicting constraints).

We leverage this capability by providing a translation layer that extracts an unsatisfiable core from the solver and identifies corresponding SQL constraints and records in the tables that lead to a contradiction. We found this invaluable when debugging complex scenarios involving affinity and anti-affinity constraints. For example, a common pattern we experienced when adding and testing new affinity policies was that the new policy *tightened the problem*, and therefore triggered a violation of some other constraint in the system (such as resource capacity constraints). Rather than suspect a bug in our specification of the new policy, the unsatisfiable core rightfully points us to the contradiction between the capacity constraint and the affinity requirement.

4.3 Implementation

Our DCM implementation is 6.1K LoC in Java for the library and an additional 2.7K LoC for tests. Of this, the OR-tools and MiniZinc backends take up roughly 2K and 1K LOC, respectively. We use the JOOQ library [8] to conveniently interface with different SQL databases. All our experiments use the OR-Tools backend, given that MiniZinc simply does not scale to large cluster sizes. Our implementation is available as an open-source project [17].

5 Experience using DCM

We now describe the three case studies we applied DCM to, and qualitatively assess the development effort to do so. The case studies are presented in the reverse chronological order in which we applied DCM; we found that the overall design and SQL-based programming model were stable throughout the process, even though we did harden DCM along the way. DCM's ability to compute unsatisfiable cores (§4.2) were invaluable in all these efforts.

Kubernetes scheduler This use case is both our most recent and most comprehensive application of DCM. Like the Kubernetes default scheduler, our scheduler also runs as a pod within Kubernetes and uses the same REST APIs and hooks to consume and actuate upon the Kubernetes' cluster state. Our scheduler consumes the same cluster metadata (information about nodes, pods, labels, and other input information) as the default scheduler to make scheduling decisions. Our scheduler uses an embedded in-memory SQL database (H2) as a cache of the cluster state, which is used to serve inputs to `model.solve()` (§3). It computes scheduling decisions for a batch of pods at a time.

Our scheduler is implemented in ~1.5K lines of Java code, with 1.8K lines of code for tests. Roughly half the scheduler's code is boilerplate to subscribe to Kubernetes' state and store it in an in-memory SQL database (H2 [4]). All the tables, views, and policies amount to ~550 lines of SQL. We implemented all policies in Table 2, except H16-20, as they were specific to volume management on GCE, AWS, and Azure.

Of the 550 lines of SQL, roughly two-thirds describe input tables and views that are executed entirely in the database, whereas the rest were used to describe constraints. We were able to handle heavy and complex joins in the database (e.g. computing groups of pods that repel each other due to inter pod anti-affinity). Support for SQL `ARRAY` columns was critical to bounding the size of inputs to DCM.

We spent the vast majority of our effort trying to understand Kubernetes' semantics. Particularly, Kubernetes' *match expression* logic, a DSL used to filter objects based on *labels*, was widely used within several policies supported by the scheduler (taints, tolerations, node/pod (anti) affinities, etc.). Its semantics differed arbitrarily across policies (multiple affinity requirements for nodes were treated as a logical OR, whereas the equivalent for pods was treated as a logical AND [11, 12, 90]). Once we understood the semantics, translating the requirements into SQL was straightforward: it took a few hours per policy to design and code (i) the schema and constraints, (ii) the logic to write Kubernetes' state into the database, and (iii) the required unit and integration tests.

Most of the time and effort in performance engineering the scheduler went into ensuring that the views executed by the database used the right indexes. While these views could all be expressed concisely in SQL (<20 LOC), the most concise SQL was sometimes not the most performant. For example, an `OR` in some join predicates caused H2 to not use indexes and revert to scans. We had to split these queries into two and `UNION` the results together to meet our performance needs [6]. In another case, we used triggers to simulate materialized views [5], to incrementally update resource reservation counters on each node as pods were placed (rather than compute these statistics each time using a join during pod placement).

VM load balancing utility We built a tool for suggesting VM migrations in a commercial data-center management solution. The system has a resource manager that makes VM placement decisions at various timescales. At slower timescales (e.g., every five minutes), the system introspects the state of the entire cluster and identifies a series of VM migrations to make, with the objective of reducing the overall standard deviation of node resource utilization (along multiple resource dimensions). We apply DCM to improve load balancing in §6.2. The load balancing and capacity constraints we introduced were structurally similar to the ones we specified in our Kubernetes use case.

Distributed transactional datastore We implemented a management plane from scratch for a distributed transactional data platform used in a commercial product. Nodes in the system assume one or more 'roles', such as being a serializer (as in Megastore [20], Omid [25]), a backup serializer, data nodes (that host data shards and replicas), or management nodes. We apply DCM to the management node logic, supporting several requirements provided by engineers. We replicated existing failure handling policies and added new capabilities like distributing roles across nodes, and rack-aware placement of data shards. All policies used <10 lines of SQL, as the system was simple compared to our other use cases.

6 Evaluation

The premise of our work is that DCM is a viable approach to building cluster managers that can (Q1) scale, (Q2) compute high-quality decisions, and (Q3) be easily extended. We answer these questions as follows:

Q1: For scalability: we study the Kubernetes scheduler we built using DCM and evaluate it on a 500 node cluster on Amazon EC2 using workload characteristics from Azure [77]. We use simulations to study scalability up to 10K nodes.

Q2: For decision quality: we study scenarios involving our Kubernetes scheduler as well as the load balancer we built for the commercial virtual machine management platform.

Q3: For extensibility: we were able to express all cluster management policies in our three use cases using SQL. In addition, we discuss a non-trivial extension we built for our Kubernetes scheduler using DCM.

6.1 Q1: Scalability evaluation

We set up a 500 node Kubernetes cluster running on AWS.[1] We use t3.2xlarge instances (8 vCPUs, 32 GB RAM) for the Kubernetes master node and t3.small instances for worker nodes, given that in these experiments, the focus is on the schedulers running on the master node.

Workload We use a publicly available trace from Azure [77], that describes a month's worth of workload information for two million VMs in 2019[2]. It gives us a trace of replicas that were launched, with their corresponding CPU and memory reservations. We replay 14 hours worth of traces and speed them up by 20$\times$ to achieve arrival rates seen in Borg clusters at Google [95] (median/peak of 100/500 pod creations per second). We then replay three variants of the workload, each with a fraction **F** of replica groups configured with inter-pod anti affinities within the group; $\mathbf{F} = 100\%$ is a Kubernetes best practice for availability reasons [1], and users even run automated tools like `kube-score` [9] to prevent pods from being deployed without anti-affinities configured. The anti-affinity policy is a challenging constraint because it requires reasoning across groups of pods and uses several handcrafted performance optimizations in the Kubernetes default scheduler (§2). In addition, the workload also exercises most hard and soft constraints shown in Table 2.

End-to-end latency results Figure 12 shows the end-to-end latency for bringing up pods on the AWS cluster, for different values of **F**. The latency is measured from when the workload generator issues a pod creation command to when the pod first changes its status to `Running`. The default scheduler's end-to-end latency degrades as more pods are configured with anti-affinity constraints, with its 95th percentile latency degrading from 4.14s at $F = 0$ to 12.45s at $F = 100$. On the other hand, DCM incurs a higher latency than the default scheduler at $F = 0$ due to the added latency in its critical path from the database and solver (p95 of 5.33s). However, DCM's end-to-end latency characteristics are insensitive to the fraction of pods configured with constraints (ECDFs for DCM are identical across all F, Figure 12). At $F = 100$, DCM improves the 95th percentile end-to-end latency over the default scheduler by 53% (5.9s vs 12.45s).

Note that our scheduler evaluates all nodes per scheduling decision, whereas the default scheduler only evaluates half the nodes in the cluster for scalability reasons (the number of

[1]Kubernetes requires careful configuration and tuning to scale beyond 500 nodes. Even at this size, we had to overcome several issues around networking, kubelet failures, and API throttling to stabilize the cluster [27,70]. We defer to simulations to stress DCM beyond 500 nodes.

[2]Our results are qualitatively similar when using the 2017 trace.

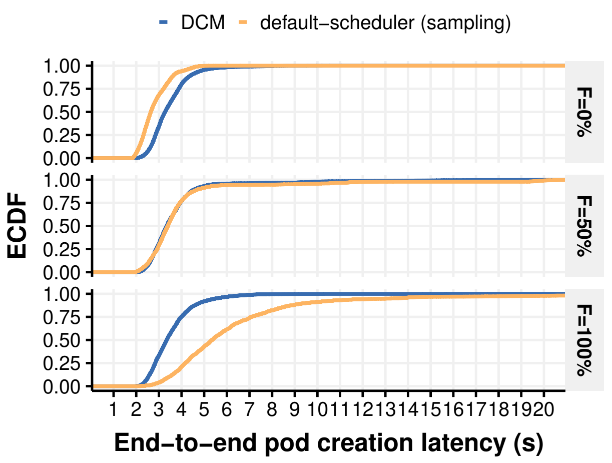


Figure 12: End-to-end pod creation latency on a 500 node AWS cluster, using workload characteristics from the Azure 2019 trace [77] (trace sped up by 20×). **F** is the percentage of pods configured with anti-affinity constraints.

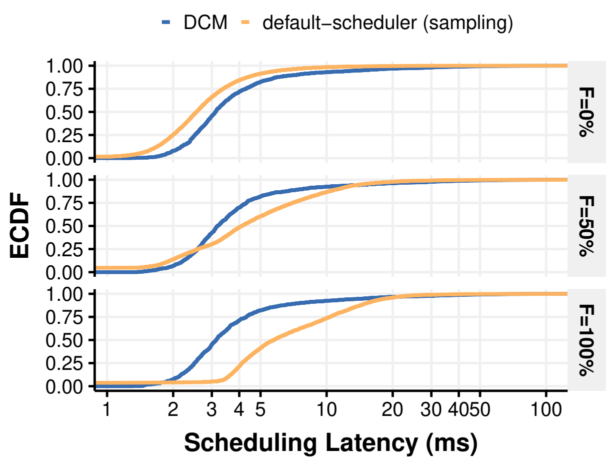


Figure 13: Per-pod scheduling latency at 20× trace speed up (log-scale). Despite our use of a full-featured SQL database and a constraint solver in the critical path, DCM improves per-pod scheduling latency at higher values of F.

#Nodes	N=500	N=5K	N=10K
#Variables	5524	45983	91010

Table 1: Average number of model variables before the OR-tools presolve phase.

nodes sampled depends on the total cluster size [93]). When we configured the Kubernetes scheduler to evaluate all nodes, its average latency **doubled** over DCM. Furthermore, the default scheduler incurs a significant amount of engineering complexity in the form of caching and pre-computing optimizations for the sole purpose of speeding up anti-affinity predicates. In contrast, DCM only required a simple SQL specification of the same constraints using 4 SQL views.

Per-pod scheduling latency Figure 13 shows the per-pod scheduling latency for DCM versus the baseline. For DCM, we measure the amortized latency over a batch of pods (maximum batch size of 50 pods), which includes the time taken for querying data from the database, creating a model based on the input data, running the solver, and returning results to the calling code. For Kubernetes, to be conservative, we only measure the time taken to execute all predicates and priorities for a pod once that pod is pulled from the scheduler's work queue. DCM's scheduling latency is competitive with the baseline at $F = 0$: DCM experiences a median (p95) pod scheduling latency of 3.11ms (14.46ms) versus 2.55ms (6.19ms) for the default scheduler. However, DCM's per-pod scheduling latency is similar across all tested values of F, whereas the default scheduler's latency increases significantly with F. At $F = 100$, DCM improves average latency by 1.6× (5.13ms versus 8.04ms). The p95 latency for DCM to schedule a batch of pods stayed under 250ms in all cases. DCM's ability to schedule a batch of pods at a time is what leads to larger absolute savings in end-to-end latency (Figure 12) versus the per-pod scheduling latency.

DCM scheduling latency breakdown Figure 14 breaks down the scheduling latency in DCM by its various phases: the time to fetch inputs from the database (`database`), to encode the inputs into an optimization model (`modelCreation`), and to run the solver (`orToolsTotal`). We also plot the time spent within the or-tools *presolve* phase (`presolve`), where the solver applies several complex optimizations to simplify the supplied encoding. `dcmSolve` subsumes `modelCreation` and `orToolsTotal`. The sum of `dcmSolve` and `database` equals the total scheduling latency.

At a cluster size of 500 and $F = 0$, fetching the required inputs from the database is inexpensive (mean 0.58ms per-pod) compared to invoking the solver (2.8ms per-pod) (Figure 14). DCM's generated code is highly efficient at model creation, contributing an average latency of 450μs per-pod. Similarly, presolve times are also low, staying around 2.5ms for 95% of cases. As we increase F, the database's latency gradually increases (mean 0.88ms) due to the views computed in the database for finding groups of pods that repel each other, but the increase remains small relative to the overall latency. Importantly, solver latency is largely unaffected by the more complex constraints.

Effect of increased cluster sizes To study the impact of cluster size and scale on DCM, we turn to simulations. This is straightforward to do in our scheduler implementation: we simply mock the Kubernetes API, mimicking cluster sizes of 500, 5000, and 10000 nodes. It allows us to replay the same Azure traces against an identical DCM scheduler, but subject the system to a variety of scales and loads. To stress DCM, we speed up the trace by 100× and set $F = 100\%$.

We observe the scheduling latency breakdowns again in Figure 15. At 5K node scale, the per-pod scheduling latency

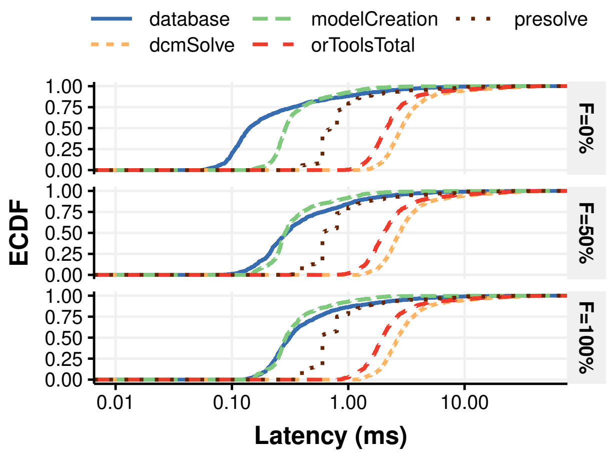


Figure 14: Scheduling latency breakdown between time spent fetching input data from the database (`database`), our generated code creating an optimization model (`modelCreation`), the 'pre-solve' phase in or-tools (`presolve`), and the total solve time in or-tools (`orToolsTotal`).

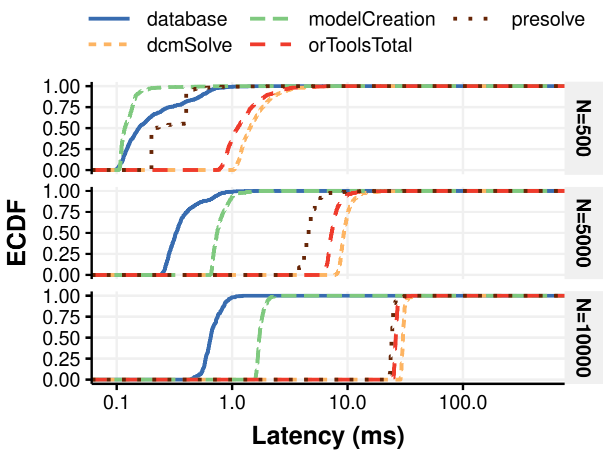


Figure 15: Simulation: scheduling latency breakdown at different cluster sizes. Per-pod scheduling latencies stay under 30ms even at 10K node scale. At 5K and 10K node scales, the presolve phase of the solver dominates.

is under 13ms (and 690ms per batch) 99% of the time. At 10K node scale, however, the p99 per-pod scheduling latency is 30ms and 1.6s per batch, which is high. To dig deeper, note that the relative contributions of the database and the constraint solver to the overall latency widen with increasing cluster size. As we mentioned before, our scheduler considers **all** nodes when placing pods, which shifts more of the burden to the solver as cluster sizes increase. We note that the `presolve` phase is the primary contributor to the overall latency. This is because of an API limitation in OR-tools around creating interval variables (§4.1.3).

In particular, there are steps within the solver's presolve pass that we could perform efficiently during model creation, but the OR-tools API is not rich enough to permit. This forces our generated code to construct models with redundant variables (proportional to the number of nodes) that the solver internally tidies up into a more compact encoding (specifically, when encoding our capacity constraints). Table 1 shows the average model sizes generated by our encoder – the presolve phase trims these models down by an order of magnitude. The added cost of repeatedly performing this step on every scheduling decision is acceptable at cluster sizes of up to 5000 nodes ($<$ 1ms at $N = 500$ and $<$ 8ms at $N = 5K$, Figure 15). We are reaching out to the or-tools developers to see if the API can be augmented to avoid this cost.

6.2 Q2: Decision quality evaluation

Kubernetes packing efficiency for higher consolidation We now evaluate a common enterprise data center scenario, where cluster sizes are typically small (under fifty nodes), but consolidation rates need to be kept high. In such scenarios, it is imperative for schedulers to find feasible and dense packings. We use a common pod affinity/anti-affinity pattern seen in production workloads [67], where nginx [13] servers in a web application need to be co-located on the same machine as an in-memory Redis cache [84]. We create 30 such applications, each with 10 pods, with pod CPU and memory requirements following an exponential distribution. We generate 35 such workloads, which leads to a different arrival sequence of resource demands per experiment.

We find that DCM places 100% of pods in 29 out of 35 experiments, and in the worst case, places at least 93% of pods across all runs. In contrast, the baseline scheduler packs all pods only in 3 out of 35 instances. This highlights DCM's effectiveness at placing *groups* of pods. Instead, the baseline myopically places one pod at a time, causing it to make decisions that prevent future pods from being placed. Note, if the pods appear well spaced apart in time, or DCM uses smaller batching sizes, its performance will approach that of the baseline.

Kubernetes placement convergence time for preemptions We now test DCM's effectiveness in making global reconfiguration decisions. We replay a workload used to test Kubernetes' preemption logic [65], that creates 3 sets of pods with different priorities. The resource demands are set to accommodate *only* the highest priority pods on the cluster, and the lower priority pods should either not be placed or be preempted. The default scheduler invokes its preemption logic on a pod-by-pod basis and uses a set of heuristics to determine when to retry pods it could not place (e.g according to a backoff policy, and retrying when nodes report status updates). In doing so, the baseline scheduler takes almost 100 seconds to place all high priority pods. Instead, DCM initially places low priority pods and systematically replaces them

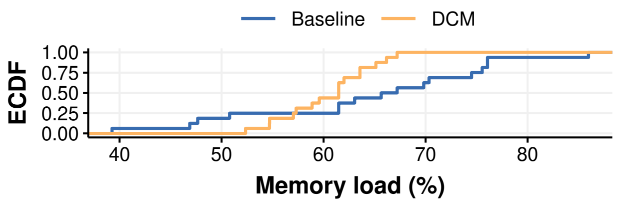


Figure 16: VM load balancing use case: memory load distribution across hosts with and without DCM.

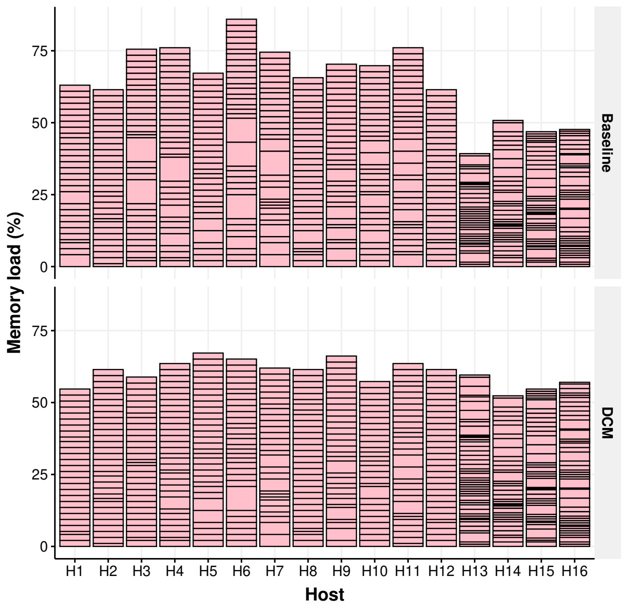


Figure 17: Load distribution before and after using DCM. The X-axis represents hosts, the Y-axis represents the memory utilization per host. Hosts have different capacities, and boxes in each bar represent VM sizes, scaled to the host's capacity.

with higher priority pods in phases as new pod requests arrive, by invoking its preemption model to look at the state of all nodes (§3). In doing so, the scheduler converges to placing all high priority pods in just under 50 seconds, twice as fast as the baseline scheduler.

VM load balancing quality evaluation We test our VM load balancing tool (§4.3) using a trace from a bug report submitted by a customer. This production cluster has 16 hosts with heterogeneous CPU and memory capacities, and 524 VMs with a range of CPU and memory sizes. The baseline system's heuristic-based load balancer could not identify VM migrations to improve the load distribution of the cluster, which led to the bug report. Figure 16 (baseline) shows the memory utilization of every host as per the trace (we only show memory utilization because there were no CPU resource reservations by the VMs). Figure 17 shows the VM sizes, scaled according to each host's capacity.

With DCM, we specified the necessary hard constraints (capacity and affinity requirements) and a soft constraint that minimizes the load difference between the most and least utilized node. We then asked the tool to identify twenty VM migrations, which significantly improved the load distribution (Figure 16). With the baseline, the most loaded and least loaded nodes were at 85% and 39% utilization, whereas DCM found moves to spread utilization between 52% and 67%. DCM took a second to make its decision, whereas the baseline heuristic takes roughly five seconds.

6.3 Q3: Extensibility

We validate our hypothesis that DCM enables building *extensible* cluster managers. §5 discusses the ease with which we added policies to all three case studies, taking only a few hours per policy to design, implement, and test. In this section, we focus on a more challenging test of extensibility by discussing a non-trivial modification to our Kubernetes scheduler.

Our case study involves a custom Kubernetes distribution where the Kubernetes control plane deploys both pods and VMs (the nodes run both pods and VMs). A challenge here is that the default scheduler's implementation is intricately coupled to the Kubernetes data-structures that represent pods (for example, every predicate and priority implementation expects a pod object). VMs, as a Kubernetes object that can also be placed on nodes, is beyond the Kubernetes scheduler's resource management model. This scheduling inflexibility is a known pain point in the Kubernetes community [66] The only option today is for the Kubernetes scheduler to coordinate with another scheduler that can deploy VMs. This is, therefore, a good case study to validate DCM's extensibility goal.

We extend the Kubernetes scheduler we built using DCM to jointly reason about pods and VMs. From a placement standpoint, pods and VMs are simply *tasks* that need to be assigned to nodes, and only represent a slightly different set of constraints (for example, VMs can be migrated but pods cannot, because most of the Kubernetes ecosystem does not assume pods can be migrated).

Most of our effort went into the Java code and boilerplate required to subscribe to the Kubernetes API to learn about new VM creations (specifically, a Kubernetes Custom Resource [85]) and writing these obtained objects into our database. On the SQL side, however, these capabilities only involved minimal changes to the DCM-based Kubernetes scheduler: we added four constraints in total and made a cosmetic change to the SQL schema for readability (replacing instances of `pods_to_assign` with `tasks_to_assign`). The minimal effort here was possible only because DCM enforces a declarative approach to specifying the cluster state and the constraints on it.

7 Discussion and future work opportunities

Solver Scalability Cluster sizes in enterprises are typically modest and well within Kubernetes' scalability targets. This is a scale that we are confident DCM excels at (§6.1).

Pushing DCM to hyperscale needs, however, is an interesting area for future work. While the techniques described in §4.1.3 are required for scalability, there will inevitably be a point where it is better to *partition a problem*, something that DCM cannot perform automatically for the developer. For example, in a cluster with 100K nodes, it is likely overkill to evaluate every single node in the cluster for an optimal placement decision. Instead, the 100K nodes can be partitioned into 10 or 20 groups that are somewhat similar in composition (analogous to Borg cells [95] or sub-clusters in Hydra [29]). A DCM model (or a custom heuristic) could pick a group to place a workload in, followed by another model that places the workload within the selected group (the second model, in this case, would use as input, a table/view with only nodes from the selected group). Another approach would be to evaluate multiple such groups in parallel and pick the result with the best objective function. We explicitly designed DCM's programming model for such flexibility.

There are several further opportunities for improving performance that we have not yet explored. For example, solvers can be configured to return good-enough (as opposed to optimal) results, when the current best solution is within a certain bound, to improve performance.

Database scalability In-memory, incremental view maintenance is key to scaling the database side. For now, we had to simulate materialized views using triggers in H2 (§5). H2's simplicity also meant that its optimizer did not perform several natural query transformations that more mature engines do, which required us to write more complex SQL than was required (§5). This is additional work that would not be required with an incremental engine. We are currently integrating DCM with the Differential Datalog (ddlog) engine [86].

Expressiveness of SQL So far, across all use cases (§5), we are yet to find a policy we could not express using this model. We are confident of SQL's expressive power for several reasons.

SQL shines at concisely cross-referencing state across different tables, a capability that has been useful in a broad range of contexts (e.g., SQCK [48]). We simply leverage that strength of SQL to both represent complex cluster state and concisely specify constraints spanning several tables; the actual check clauses and objective function expressions within these constraint queries are typically comparable to what is shown in Figures 5, 6, and 7.

The more complex SQL we have written are for views executed in the database, which become inputs for the generated code (§4.1.1, §5). There is a lot of expressive power here; for example, developers may use a database's user-defined functions for specific input transformations if required, but we have not yet needed to do so (even for handling Kubernetes's match expression DSL, §5).

At the same time, DCM does require expressing policies in terms of *intent*, rather than the exact steps of an algorithm. We anticipate that this will pose a learning curve for some developers.

Generality of optimizations DCM cannot prevent users from writing SQL that generates inefficient code, a common challenge for declarative programming models (SQL databases provide tools for users to inspect query plans for this reason, like the `EXPLAIN` query). For now, our compiler warns developers when it emits inefficient code (e.g., cross products across tables without indexes), and exposes detailed diagnostics to understand runtime performance (e.g., Figure 14 and Table 1).

We provide a suite of aggregate functions (like `all_different`) that we encourage developers to use because it leads to clearer policies and makes it straightforward to generate efficient code that uses global constraints (§4.1.1). So far, we only added functions if they were useful across several use cases. It is similar with rewrite rules: we only add ones that have broad utility (e.g., we find the fixed-arity rule applying to most uses of `SUM/COUNT`).

8 Related work

Use of solvers for resource management A large body of work has used solvers for resource management, including CP solvers [51] to pack and migrate VMs; flow network solvers [40, 53] and MIPs [38, 42–44, 91, 92] for job scheduling, and ILPs for traffic engineering [30]. These systems use handcrafted encoders written by optimization experts for specific problems. Even in the industry, we find that the few organizations that use solvers for such tasks typically have dedicated teams of optimization experts. In contrast, we generate scalable encoders from a declarative specification written using SQL. Our programming model significantly lowers the barrier to powering systems with constraint solvers – developers express policies directly against the cluster state, without having to translate them into the low-level mathematical formalisms of solver encodings.

We sketched out the initial idea for DCM in a workshop paper [89]. In this paper, we extend this preliminary work with a detailed design and implementation, and a comprehensive evaluation using three case studies.

Quincy [53] and Firmament [40] use flow network solvers for scheduling, which yield quick solve times (sub-second, even for topologies with thousands of nodes), but cannot model many classes of constraints, including inter-task constraints like affinity/anti-affinity [41]. Compared to DCM, they involve a high degree of modeling complexity: developers need to map scheduling constraints to flow network constructs like vertices, arcs, and flows, which make it challenging to apply to general systems like Kubernetes. For example, the experimental Poseidon Kubernetes scheduler [88] is based on Firmament, but the developers found constraints like inter-pod affinity both hard to implement [87] and scale [16] (up to 3× slower than the default scheduler *evaluating all nodes*).

Wrasse [82] uses a DSL based on a balls and bins abstraction to specify resource allocation constraints and a GPU-based solver to find solutions. Its low-level modeling language makes it hard to express complex constraints; it supports resource capacity constraints, but not other important classes of constraints like affinities and load balancing.

Some production systems use meta-heuristic search for resource management. VMware DRS [47] uses a greedy hill-climbing search, and Service Fabric [76] uses simulated annealing. Several systems employ a variety of heuristics for resource management [26, 28, 29, 32, 33, 39, 54]. These works neither use declarative programming techniques nor benefit from solver-based optimal solutions to enforce policies.

Simplifying systems using relational languages Several works have used the strengths of relational languages to simplify systems programming. Boom Analytics [19] uses the Overlog language to build an HDFS/Hadoop clone with comparable performance. P2 [74] also uses Overlog, but to declaratively specify peer-to-peer overlays. Ravel [96] is an SDN controller that uses SQL databases to abstract and manipulate network state. SQCK [48] simplifies filesystem checker implementations by using declarative queries to validate complex filesystem images instead of writing low-level C code. DCM builds on the above ideas and not only uses a relational database to store and manipulate cluster state but also code generates logic to search for new configurations based on constraints written in SQL.

Network configuration synthesis Network configuration synthesis from high-level specification [22, 34, 35] for BGP and OSPF is orthogonal to dynamic cluster management with constraint specification by DCM. ConfigAssure [79] and Alloy [78] use model finding to identify configurations that satisfy a specification given by an administrator (or detect errors in existing ones). Alloy uses a DSL for specification, whereas ConfigAssure uses the Prolog language. DCM, on the other hand, works on top of standard SQL databases and is capable of supporting optimization goals as well. The tested use cases for ConfigAssure and Alloy are well within scope for DCM.

DSLs for infrastructure automation Many configuration management tools use custom DSLs. Hewson et al. [52] propose an object-oriented DSL to specify a configuration for a data-center, which is enforced by a constraint solver. PoDIM [31] does not use a solver but uses an SQL-like DSL to specify requirements for a configuration. Configuration management tools like Puppet [81], Ansible [83], Terraform [49], and Helm [7] all use custom DSLs to configure and deploy infrastructure repeatably. These systems target a different use case than DCM: they are not designed to solve optimization tasks within a dynamic distributed system at short timescales but instead target infrastructure deployment, which runs at much slower timescales.

9 Conclusion

Cluster management logic is notoriously hard to develop, given that they routinely involve combinatorial optimization tasks that cannot be efficiently solved using best-effort heuristics. With DCM, we propose building cluster managers where the implementation to compute policy-compliant decisions is synthesized by a compiler from a high-level specification. DCM significantly lowers the barrier to building cluster managers that scale, compute high-quality decisions, and are easy to evolve with new features over time. We validate our thesis by applying DCM to three production use cases: we built a Kubernetes scheduler that is faster and more flexible than the heavily optimized default scheduler, improved load balancing quality in a virtual machine management solution, and easily added features to a distributed transactional data store.

Acknowledgements

We thank our shepherd Lidong Zhou and the anonymous reviewers for their valuable feedback. We are grateful to Mihai Budiu, Jon Howell, Sujata Banerjee, and Jacques Chester for their valuable inputs that helped shape this project.

References

[1] 10 most common mistakes using Kubernetes. https://blog.pipetail.io/posts/2020-05-04-most-common-mistakes-k8s/.

[2] Gecode. https://www.gecode.org/.

[3] Google OR-Tools. https://developers.google.com/optimization/.

[4] H2 Database. https://github.com/h2database/h2database/.

[5] H2 Database Features: Triggers. https://h2database.com/html/features.html#triggers.

[6] H2 Database: Performance. https://h2database.com/html/performance.html.

[7] Helm. https://helm.sh/.

[8] JOOQ. https://github.com/jOOQ/jOOQ.

[9] Kube Score. https://github.com/zegl/kube-score.

[10] Kubernetes. http://github.com/kubernetes/kubernetes.

[11] Kubernetes Issue 52141: Multiple matchExpressions in nodeSelectorTerms works unexpectedly. https://github.com/kubernetes/kubernetes/issues/52141.

[12] Kubernetes Issue 70394: s/ORed/ANDed/ nodeSelectorTerms matchExpressions. https://github.com/kubernetes/kubernetes/pull/70394#issuecomment-434127780.

[13] Nginx. http://nginx.org/en/docs/http/load_balancing.html.

[14] Openshift. https://www.openshift.com/.

[15] Openstack. https://www.openstack.org/.

[16] Poseidon benchmarks. https://github.com/kubernetes-sigs/poseidon/blob/master/docs/benchmark/README.md.

[17] Declarative Cluster Management Github Repository. https://github.com/vmware/declarative-cluster-management/, 2019.

[18] Susanne Albers and Michael Mitzenmacher. Average-case analyses of first fit and random fit bin packing. *Random Structures & Algorithms*, 16(3):240–259, 2000.

[19] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. Boom analytics: Exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 223–236, New York, NY, USA, 2010. Association for Computing Machinery.

[20] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[21] Nikhil Bansal, Alberto Caprara, and Maxim Sviridenko. Improved approximation algorithms for multidimensional bin packing problems. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 697–708. IEEE, 2006.

[22] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Network configuration synthesis with abstract topologies. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 437–451, New York, NY, USA, 2017. ACM.

[23] Kevin Beyer, Don Chambérlin, Latha S. Colby, Fatma Özcan, Hamid Pirahesh, and Yu Xu. Extending XQuery for Analytics. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 503–514, New York, NY, USA, 2005. ACM.

[24] Scott Boag, Don Chamberlin, Mary F Fernández, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu. XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery, 2002. Retrieved March 2019.

[25] Edward Bortnikov, Eshcar Hillel, Idit Keidar, Ivan Kelly, Matthieu Morel, Sameer Paranjpye, Francisco Perez-Sorrosal, and Ohad Shacham. Omid, reloaded: Scalable and highly-available transaction processing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 167–180, Santa Clara, CA, 2017. USENIX Association.

[26] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 285–300, Broomfield, CO, October 2014. USENIX Association.

[27] Architecting Kubernetes clusters. https://learnk8s.io/kubernetes-node-size.

[28] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based scheduling: If you're late don't blame us! In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, SOCC '14, pages 2:1–2:14, New York, NY, USA, 2014. ACM.

[29] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan. Hydra: A federated resource manager for data-center scale analytics. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 177–192, Boston, MA, February 2019. USENIX Association.

[30] Emilie Danna, Subhasree Mandal, and Arjun Singh. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. In *2012 Proceedings IEEE INFOCOM*, pages 846–854. IEEE, 2012.

[31] Thomas Delaet and Wouter Joosen. Podim: A language for high-level configuration management. In *LISA*, volume 7, pages 1–13, 2007.

[32] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 77–88, New York, NY, USA, 2013. ACM.

[33] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. *SIGARCH Comput. Archit. News*, 42(1):127–144, February 2014.

[34] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Network-wide configuration synthesis. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 261–281, Cham, 2017. Springer International Publishing.

[35] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. NetComplete: Practical network-wide configuration synthesis with autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 579–594, Renton, WA, 2018. USENIX Association.

[36] etcd. etcd. https://github.com/coreos/etcd, 2014.

[37] Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25(4):457–516, December 2000.

[38] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: Scheduling of long running applications in shared production clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[39] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 365–378, New York, NY, USA, 2013. Association for Computing Machinery.

[40] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 99–115, Savannah, GA, 2016. USENIX Association.

[41] Ionel Corneliu Gog. Flexible and efficient computation in large data centres, 2018.

[42] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 455–466, New York, NY, USA, 2014. Association for Computing Machinery.

[43] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 65–80, USA, 2016. USENIX Association.

[44] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 81–97, USA, 2016. USENIX Association.

[45] Torsten Grust. Monoid Comprehensions as a Target for the Translation of OQL. In *Workshop on performance enhancement in object bases, Schloss Dagstuhl*, 1996.

[46] B. Guenter, N. Jain, and C. Williams. Managing cost, performance, and reliability tradeoffs for energy-aware server provisioning. In *2011 Proceedings IEEE INFOCOM*, pages 1332–1340, April 2011.

[47] Ajay Gulati and Xiaoyun Zhu. VMware distributed resource management: design, implementation, and lessons learned. *VMware Technical Journal*, 1(1):45–64, 2012.

[48] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SQCK: A declarative file system checker. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 131–146, USA, 2008. USENIX Association.

[49] HashiCorp. Terraform. https://www.terraform.io/, 2014.

[50] HashiCorp. Nomad. https://www.nomadproject.io/docs/internals/scheduling.html, 2015.

[51] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Entropy: A consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 41–50. ACM, 2009.

[52] John A Hewson, Paul Anderson, and Andrew D Gordon. A declarative approach to automated configuration. In *LISA*, volume 12, pages 51–66, 2012.

[53] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *ACM Symposium on Operating systems principles (SOSP)*, pages 261–276. ACM, 2009.

[54] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards automated SLOs for enterprise clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 117–134, USA, 2016. USENIX Association.

[55] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. Fast queries over heterogeneous data through engine customization. *Proc. VLDB Endow.*, 9(12):972–983, August 2016.

[56] Manos Karpathiotakis, Ioannis Alagiannis, Thomas Heinis, Miguel Branco, and Anastasia Ailamaki. Just-in-time data virtualization: Lightweight data management with vida. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research (CIDR)*, number EPFL-CONF-203677, 2015.

[57] Arie MCA Koster, Manuel Kutschka, and Christian Raack. Towards robust network design using integer linear programming techniques. In *Next Generation Internet (NGI), 2010 6th EURO-NF Conference on*, pages 1–8. IEEE, 2010.

[58] Kubernetes. Add a new predicate: max replicas limit per node. https://github.com/kubernetes/kubernetes/pull/71930, 2018.

[59] Kubernetes. Add max number of replicas per node/topologyKey to pod anti-affinity. https://github.com/kubernetes/kubernetes/issues/40358, 2018.

[60] Kubernetes. Affinity/Anti-Affinity Optimization of Pod Being Scheduled #67788. https://github.com/kubernetes/kubernetes/pull/67788, 2018.

[61] Kubernetes. Allow Minimum (or Maximum) Pods per failure zone. https://github.com/kubernetes/kubernetes/issues/66533, 2018.

[62] Kubernetes. Maximum of N per topology value. https://github.com/kubernetes/kubernetes/pull/41718, 2018.

[63] Kubernetes. MaxPodsPerNode - be able to set hard and soft limits for deployments / replicasets. https://github.com/kubernetes/kubernetes/issues/63560, 2018.

[64] Kubernetes. Pod priorities and preemption. https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/, 2018.

[65] Kubernetes. Scheduler sometimes preempts unnecessary pods. https://github.com/kubernetes/kubernetes/issues/70622, 2018.

[66] Kubernetes. Add custom resource scheduling. https://github.com/kubernetes/kubernetes/issues/82118, 2019.

[67] Kubernetes. Assigning Pods to Nodes. https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#more-practical-use-cases, 2019.

[68] Kubernetes. Kubernetes Descheduler. https://github.com/kubernetes-sigs/descheduler, 2020.

[69] Kubernetes mailing list. Let's remove ServiceAffinity . https://groups.google.com/forum/#!topic/kubernetes-sig-scheduling/ewz4TYJgL0M, 2018.

[70] Kubernetes Master Tier For 1000 Nodes Scale. https://tinyurl.com/y97ysbrd.

[71] KubeVirt. https://kubevirt.io/.

[72] Kevin Leo and Guido Tack. Debugging unsatisfiable constraint models. In Domenico Salvagnin and Michele Lombardi, editors, *Integration of AI and OR Techniques in Constraint Programming*, pages 77–93, Cham, 2017. Springer International Publishing.

[73] Kubernetes Topology Manager Limitations. https://kubernetes.io/docs/tasks/administer-cluster/topology-manager/#known-limitations.

[74] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 75–90, New York, NY, USA, 2005. Association for Computing Machinery.

[75] Christopher Mears, Andreas Schutt, Peter J. Stuckey, Guido Tack, Kim Marriott, and Mark Wallace. Modelling with option types in MiniZinc. In Helmut Simonis, editor, *Integration of AI and OR Techniques in Constraint Programming*, pages 88–103, Cham, 2014. Springer International Publishing.

[76] Microsoft. Service Fabric. https://tinyurl.com/y728dctp, 2016.

[77] Microsoft. Azure Public Dataset. https://github.com/Azure/AzurePublicDataset, 2017.

[78] Sanjai Narain et al. Network configuration management via model finding. In *LISA*, volume 5, pages 15–15, 2005.

[79] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. Declarative infrastructure configuration synthesis and debugging. *J. Network Syst. Manage.*, 16:235–258, 09 2008.

[80] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[81] Puppet Labs. Puppet. https://puppet.com/, 2005.

[82] Anshul Rai, Ranjita Bhagwan, and Saikat Guha. Generalized resource allocation for the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 15:1–15:12, New York, NY, USA, 2012. ACM.

[83] Red Hat. Ansible. https://www.ansible.com/, 2012.

[84] Redis. Redis. http://redis.io/, 2009.

[85] Kubernetes Custom Resources. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/.

[86] Leonid Ryzhyk and Mihai Budiu. Differential datalog. In *Datalog 2.0*, Philadelphia, PA, June 4-5 2019.

[87] SIG Scheduling. Affinity/Anti-Affinity Update . https://groups.google.com/forum/#!msg/kubernetes-sig-scheduling/nHWb9zCMOyo/tkbtFf8lBgAJ, 2018.

[88] SIG Scheduling. Poseidon . http://github.com/kubernetes-sigs/poseidon, 2018.

[89] Lalith Suresh, João Loff, Nina Narodytska, Leonid Ryzhyk, Mooly Sagiv, and Brian Oki. Synthesizing cluster management code for distributed systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 45–50, New York, NY, USA, 2019. Association for Computing Machinery.

[90] Assigning Pods to Nodes: affinity and anti affinity. https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#affinity-and-anti-affinity.

[91] Alexey Tumanov, James Cipar, Gregory R. Ganger, and Michael A. Kozuch. Alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[92] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, EuroSys '16, pages 35:1–35:16, New York, NY, USA, 2016. ACM.

[93] Kubernetes Scheduler Performance Tuning. https://kubernetes.io/docs/concepts/scheduling-eviction/scheduler-perf-tuning/.

[94] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[95] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 18:1–18:17, Bordeaux, France, 2015. ACM.

[96] Anduo Wang, Xueyuan Mei, Jason Croft, Matthew Caesar, and Brighten Godfrey. Ravel: A database-defined network. In *Proceedings of the Symposium on SDN Research*, SOSR '16, New York, NY, USA, 2016. Association for Computing Machinery.