Scaling a Declarative Cluster Manager Architecture with Query Optimization Techniques (Technical Report)

Kexin Rong\textsuperscript{1,2}, Mihai Budiu\textsuperscript{1}, Athinagoras Skiadopoulos\textsuperscript{3}, Lalith Suresh\textsuperscript{1}, Amy Tai\textsuperscript{4}
VMware Research\textsuperscript{1}, Georgia Institute of Technology\textsuperscript{2}, Stanford University\textsuperscript{3}, Google\textsuperscript{4}

\textbf{ABSTRACT}
Cluster managers play a crucial role in data centers by distributing workloads among infrastructure resources. Declarative Cluster Managers (DCM) is a new cluster management architecture that allows users to express placement policies declaratively using SQL-like queries. In this paper, we share our experiences scaling up this architecture from moderate-sized enterprise clusters (hundreds or thousands of nodes) to hyperscale clusters (tens of thousands of nodes) through the lens of query optimization. To do so, we first formally specify DCM’s declarative language, C-SQL, which introduces new semantics such as variable columns on top of standard SQL to express constraint optimization problems. Given cluster management logic specified as C-SQL programs, we explore and adapt techniques from classic query optimization, namely incremental view maintenance and predicate pushdown, to improve the execution efficiency of the relation and constraint components of C-SQL programs. We evaluate the effectiveness of our optimizations through a case study of building Kubernetes schedulers using C-SQL. Our optimizations demonstrated an almost 3000x speed up in database latency and reduced the size of optimization problems by as much as 1/300 of the original without affecting feasibility.

PVLDB Reference Format:
Kexin Rong\textsuperscript{1,2}, Mihai Budiu\textsuperscript{1}, Athinagoras Skiadopoulos\textsuperscript{3}, Lalith Suresh\textsuperscript{1}, Amy Tai\textsuperscript{4}, Scaling a Declarative Cluster Manager Architecture with Query Optimization Techniques (Technical Report). PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/vmware/declarative-cluster-management.

1 INTRODUCTION
Cluster managers like Kubernetes [9], OpenStack [4], and OpenShift [3] are important building blocks of today’s data centers. They dynamically assign workloads to the underlying infrastructure and configure them according to a variety of policies. Some policies represent \textit{hard constraints} (e.g., never assign two replicas of a storage service to the same node), which must always hold in any cluster management decision. Others are \textit{soft constraints} (e.g., spread these web servers across different geographies if possible), which represent preferences and decision quality. Cluster management logic is notoriously hard to develop [50], since it often involves NP-hard combinatorial optimization tasks that cannot be efficiently solved using best-effort heuristics, as is the norm today.

Declarative Cluster Managers (DCM) [50] is a radically different architecture that allows developers to specify what the cluster manager should achieve, not how it should do so. Using a declarative language we formalize as C-SQL in this paper, developers can specify constraints as queries over cluster states stored in a relational database. The DCM runtime encodes the latest cluster state from the database into an optimization problem as specified by the C-SQL queries, and solves the problem using off-the-shelf constraint solvers (Figure 1). DCM’s declarative approach significantly reduces the development and maintenance efforts for complex cluster management logic, compared to existing, ad-hoc designs that require developers to write large amounts of custom, imperative code from scratch. DCM also improves performance by generating efficient constraint solver encodings that scale to larger problem sizes compared to brittle, handcrafted heuristics.

Driven by the needs of some hyperscale operators who are interested in DCM, we have since embarked on a journey to scale DCM from moderate-sized enterprise clusters (hundreds or thousands of nodes) to hyperscale clusters (tens of thousands of nodes). Our key insight behind the scaling efforts is to leverage the incremental nature of cluster management problems: changes in cluster states are often a few magnitudes smaller than the size of the whole database, so it makes sense to keep the amount of work done proportional to the size of the changes instead of the size of the database. Achieving
this design goal is quite challenging and necessitates novel techniques at the intersection of constraint optimization and classic database query optimization, which we describe below.

First, we formally define DCM’s query language C-SQL, which allows users to declaratively specify constraint queries against relations. While C-SQL is closely related to SQL, it importantly introduces new semantics such as variable columns to support constraint optimization problems. The DCM runtime evaluates C-SQL programs in two phases: relation evaluation, wherein the runtime queries the latest cluster state required for a given decision; and constraint evaluation, wherein the runtime encodes the fetched relations into constraint formulae that can be passed to a solver. This new formalism allows us to reason about optimization opportunities through the lens of query optimization for C-SQL programs.

To facilitate relation evaluation, we integrate DCM with an incremental engine, Differential Datalog (DDlog) [12, 45], to enable incremental computation on the relations. Previously, DCM users used traditional in-memory embedded databases (e.g., H2 [6]) for managing cluster state and were forced to write less natural C-SQL queries in pursuit of performance. For example, to manually simulate incremental view maintenance, users had to carefully design the schema by splitting static and dynamic parts of the cluster states into separate tables. Instead, we leverage an incremental engine like DDlog to automatically incrementalize computations given queries. With DDlog, users are able to express their constraints using concise queries without worrying about performance, which matches the promise of the declarative programming model. Integrating developing systems like DDlog with DCM was a significant undertaking of more than 11K lines of code; we elaborate on the challenges faced and lessons learned in § 4.

To facilitate constraint evaluation, we introduce a new optimization called Feasibility-Preserving Predicate Pushdown (FP-pushdown) to automatically reduce the size of optimization problems generated from C-SQL programs without affecting the feasibility. Constraint solvers already simplify a given optimization problem in a “presolve” phase before they start searching for solutions, by eliminating unnecessary variables and restricting the domain of variables. Inspired by the predicate pushdown technique, FP-pushdown pushes down relevant constraints from the solver’s presolve phase to the relation-evaluation phase of the DCM processing pipeline, to filter out irrelevant parts of the search space in advance. This is done by statically analyzing C-SQL programs, symbolically inferring variable domains, and generating predicates that can be pushed down to the (now incremental) relation evaluation. Although FP-pushdown can not access live, runtime data about variable domains like the solver, there is still significant room for optimization. In an example, FP-pushdown trims a placement problem of assigning pods to a 50K-node Kubernetes cluster into one that only reasons about fewer than 1.4K nodes, reducing the 95th percentile scheduling latency from 8 seconds to 80-800 milliseconds.

In summary, as an information system architectures paper, this work makes the following contributions:

1. We report on experiences scaling up a new architecture for declarative cluster management. We discuss our central design goal of making the architecture’s processing pipeline incremental as well as the challenges of achieving this goal.
2. We provide a formal specification of DCM’s declarative constraint language C-SQL, highlighting its similarities and differences with respect to standard SQL.
3. We introduce two optimization techniques, based on incremental view maintenance and predicate pushdown, to improve the execution efficiency of C-SQL programs.
4. Through a case study of building Kubernetes schedulers using C-SQL, we demonstrate that our optimizations enabled a nearly 3000× improvement in latency for fetching and updating the cluster state database, and reduced the size of optimization problems down to 0.29–2.7% of the original without affecting feasibility.

2 C-SQL BACKGROUND AND DESIGN

In this section, we explain the context in which DCM was designed and formalize DCM’s declarative language C-SQL. We first present the use case of a modern cluster scheduler in Kubernetes, and the challenges with current designs based on hand-crafted heuristics (§ 2.1). We then explain how DCM and its C-SQL programming model significantly improve the programmability and scalability compared to the status-quo (§ 2.2).

2.1 Background: heuristic-based cluster management

We use the running example of building a Kubernetes scheduler [1] throughout this paper to aid the reader. Note that neither the problems nor the solutions presented are unique to scheduling in Kubernetes. As shown in our earlier work [50], similar challenges recur across several kinds of distributed systems, such as for policy-based configuration, data replication, and load-balancing across machines, all of which are combinatorial optimization problems.

Policies. The Kubernetes scheduler is tasked with assigning pods to nodes. A pod is the smallest unit of scheduling in Kubernetes and typically represents one or more containers each. Pods can be tagged with a set of policies that constrain which nodes they can be assigned to. The scheduler supports over 30 different hard and soft constraints. These include various flavors of capacity constraints; inter-pod/node affinities and anti-affinities that affect which nodes, regions or data-centers pods can be placed in; taints and tolerations that attract and repel pods from nodes based on operator goals; networking constraints like the availability of host ports; and myriad soft constraints that encode opportunities for performance gains, such as preferring nodes that already have the required container images for a pod locally. In addition, the scheduler uses a plugin framework that allows operators to add custom policies based on their specific deployments, which users leverage heavily.

Cluster state representation. The scheduler typically maintains a swathe of in-memory data structures that present a view of the relevant cluster states. For example, in Kubernetes [9], a centralized, persistent data store hosts all the cluster states, whereas services like the Kubernetes scheduler maintain an in-memory cache of that state, synchronized via a client library. The scheduler typically caches a view of the set of pods, nodes, volumes, and application abstractions such as groups of replicas. Policies often need to cross-reference multiple types of state to make decisions (e.g., reason...
We present details of C-SQL’s semantics and execution below.

## 2.2 Declarative Cluster Managers with C-SQL

To overcome the programmability and scalability challenges in existing designs, recent work introduced the idea of a *declarative* cluster manager (DCM) [50]. DCM replaces the imperative, filter-score pipeline of a heuristic-based cluster manager with a declarative one written in C-SQL, where developers specify what the cluster manager should achieve, not how. On the high level, DCM users design a cluster state representation using a relational model instead of ad-hoc data structures. They then describe policies as constraint queries and outputs a score for a given node. After processing all candidate nodes, the pod is assigned to the highest ranked node.

**Current designs and challenges.** Today’s cluster managers use the “filter-score” [1, 25, 55] architecture to implement policy-based optimization logic, such as scheduling, resource management, and placement. In this design, policies are organized as a processing pipeline that evaluates every node that is considered for scheduling (usually all nodes or a sample of all nodes) to place a given pod. For each pair of pod $p$ and node $n$, the pipeline first evaluates policies with hard constraints, each of which is implemented as a single function that decides whether or not $p$ can be placed on $n$. This is the filter phase. If $p$ survives all filtering policies, it proceeds to the scoring phase, where the pipeline evaluates each soft constraint and outputs a score for a given node. After processing all candidate nodes, the pod is assigned to the highest ranked node.

There are two key issues with existing designs:

- **Over time, the sprawl of ad-hoc data structures in the cluster state representation worsens maintainability.** A common issue is that the custom data structures used for precomputing and caching optimizations tend to be brittle in the face of evolving requirements. The result of this is that evolving the scheduler with new features and policies is hard, and cannot be done in isolation without significant refactoring effort.
- **At scale, it becomes challenging to implement highly needed but complex policies efficiently** [50]. Sampling is often used to keep scheduling performance tractable, where only a small subset of nodes are considered for each scheduling decision. However, given that each policy is self-contained imperative code with opaque semantics, sampling cannot be easily implemented in a policy-aware manner and can therefore miss feasible solutions.

### 2.2.1 C-SQL semantics

A constraint optimization problem describes a set of *variables* and *constraints* that restrict the values that can be assigned to each variable. C-SQL extends standard SQL to support specifying constraint optimization problems. We present about existing pod, node and volume arrangements all at once). In addition, many policies require the scheduler to incrementally materialize summaries of the cluster state that are not directly exposed by the centralized data store. For example, the scheduler has its own logic to maintain the total spare resource capacity of each node in the cluster. The scheduler also maintains ad-hoc auxiliary data structures to cache prior decisions [1, 25, 55] to make coherent future decisions without having to scan the cluster state repeatedly. These types of precomputing and caching optimizations are necessary to keep scheduler performance tractable.

**Program:**

```
Program ::= (ConstraintQuery | IntermediateView)*
ConstraintQuery ::= CREATE CONSTRAINT <identifier> AS
                    (CHECK (Expr | ConstraintStatement)) | (MAXIMIZE Expr)
FROM Relation
IntermediateView ::= CREATE VIEW <identifier> AS Relation
ConstraintStatement ::= : A1Different(Expr)
                      | ALLEqual(Expr)
                      | Increasing(Expr)
                      | CapacityConstraint(Expr, NonVarExpr, NonVarExpr, NonVarExpr)
-- Expressions involving VarColumn cannot have nullable columns.
-- Expressions not derived from VarColumns are denoted as NonVarExpr.
Expr ::= Expr Op Expr
        | UnOp (Expr)
        | Expr IN (SubQuery)
        | EXISTS (SubQuery)
        | NonVarExpr IS NULL
        | NonVarExpr IS NOT NULL
        | Aggregate // only valid in group-by context
        | Column
        | Literal
        | VarColumn
Op ::= AND | OR | = | != | >= | < | <= | + | - | * | / | %
UnOp ::= ~ | NOT
Aggregate ::= ANY(Expr)
        | ALL(Expr)
        | SUM(Expr)
        | COUNT(Expr)
        | MIN(Expr)
        | MAX(Expr)
Relation ::= SELECT (Expr)*
            FROM (Table | Join *)
            [WHERE NonVarExpr]
            [GROUP BY NonVarExpr, NonVarExpr]*
            [HAVING NonVarExpr]
Join ::= Relation JOIN Relation ON NonVarExpr
```

*Figure 2: Simplified constraint query grammar of C-SQL.

Each constraint query takes a relation and produces constraints from every row of that relation. DCM enforces the conjunction of all CHECK constraints and maximizes the sum of all MAXIMIZE expressions.*

The simplified grammar of C-SQL in Figure 2, and defer the formal specification to Appendix A of the technical report [43].

The key addition to C-SQL is the introduction of *variable columns* in tables (colored in red in Figure 2). Each cell in a variable column is a single named variable for the constraint solver, while traditional columns in the table represent constants in the optimization problems. Each base table can contain a combination of variable columns and constant columns.

A C-SQL program comprises a set of *constraint queries* that pose constraints over variable columns. A constraint query is a query with a CHECK or MAXIMIZE clause, which represents hard and soft constraints respectively, as well as a relation against which the clause should apply. CHECK clauses require a boolean expression and MAXIMIZE clauses accept a numeric expression. The final program enforces the conjunction of all CHECK clauses and maximizes the sum of all MAXIMIZE expressions. These expressions (Expr) represent symbolic formulae, where the constants in the formula are instantiated from the rows of the relation.

Relations are constructed via the rules of regular SQL using selects, joins, where clauses, group-bys and aggregates. Crucially, symbolic formulae can only appear in CHECK and MAXIMIZE clauses. All relation-related clauses, such as join criteria, where clauses, having clauses and group-bys, cannot contain variable columns, as doing so makes the size of the output relation indeterminate.
2.2.2 C-SQL evaluation. To evaluate a C-SQL program, the DCM runtime translates the database into an optimization problem and solves it with a constraint solver. Concretely, the runtime generates code, which, when invoked, queries all tables and views referenced in the C-SQL program, encodes the data into an optimization problem, and passes the encoding to an off-the-shelf constraint solver. The DCM compiler’s primary code generation target produces Java code and interfaces with Google’s CP/SAT solver [7]. The result of the evaluation is that the runtime assigns values to variable columns according to the constraints specified in C-SQL.

A C-SQL processing pipeline is therefore evaluated in two parts. The relation-side of the pipeline evaluates all the relations, typically tables and views stored in a database, against which constraints have been defined. The constraint-side of the pipeline encodes a given set of relations into an optimization problem and solves the problem using a constraint solver. The full pipeline is illustrated in Figure 1.

2.2.3 C-SQL by example. We now explain how DCM’s C-SQL-based programming model works, using the Kubernetes example.

Cluster state database. A DCM-powered Kubernetes scheduler represents the cluster state in a relational database. This database is an embedded, in-memory database that is merely a cache of the persistent cluster state exposed by Kubernetes’ API. Example information represented in this state includes the set of pods, nodes, volumes, replica sets, and myriad metadata associated with all these objects. Importantly, by annotating some columns as variable columns, the schema now serves as a declarative specification of the cluster state. For Kubernetes schedulers, an example variable column would be the node_name column in the pods table (Figure 3), which represents the variables in a scheduling decision. For discrete variables such as the node_name column, the variable domain can also be specified in the schema via a foreign key constraint.

As explained in §2.2, the cluster state database hosts tables and views representing all the inputs required to make cluster management decisions. It is non-trivial to compute these views efficiently at scale, since some involve joining of large base relations. For example, computing a view to filter out machines that do not represent the set of all pods, their resource demands, the set of all nodes and the node resource capacities. Given views rarely change significantly between successive invocations of model.solve(), recomputing from scratch each time is highly inefficient. Top-down query evaluation as with most databases is therefore a poor fit for

Policies as constraints. Policies are expressed in DCM as constraint queries against relations using C-SQL. Figure 4 shows example hard and soft constraints specified via a CHECK or MAXIMIZE clause against a table (’pods_to_assign’). DCM users express policies using a range of standard SQL features, including joins, group bys, aggregates, sub-queries, and correlated sub-queries.

Compiler and runtime processing pipeline. Figure 5 shows DCM’s programming model. Model.compile() takes in a C-SQL program, generates code, compiles and then loads the code into memory, wrapped as a Model object. At runtime, Model.solve() fetches the required input data from the database, solves the optimization problem, and returns a solution as the same tables, but with values assigned to variable columns. Model.solve() is invoked whenever a new cluster management decision needs to be made (e.g., schedule all pods that are pending assignment).

3 CHALLENGES WITH SCALING THE C-SQL PROCESSING PIPELINE

Cluster management logic is highly incremental in nature; the cluster state database changes often but most changes are small relative to the size of the overall database. For example, even in a datacenter with O(100K) pods, a typical scheduling decision might only involve O(100) pods at a time. These decisions often need to be made at sub-second timescales (e.g., at a cost of a few milliseconds of scheduling latency per pod).

The frequent interaction with the cluster state database in DCM’s C-SQL processing pipeline and the incremental nature of cluster management lead to the following design goal:

Design goal. Keep the amount of work done in the pipeline proportional to the size of the change, and not the size of the database.

In this section, we describe the opportunities and challenges of achieving this design goal in the relation-evaluation (§3.1) and constraint-evaluation (§3.2) parts of the C-SQL processing pipeline. We then given an overview of our proposed optimizations, which can be thought of as query optimization for C-SQL programs (§3.3).
Without DDlog, many views would be split. The relations in gray are re-used across several view calculations – with DDlog, we re-use the effort to update these views.

Split views. The first recurring pattern was to split the database schema into tables or views representing changing and fixed portions of the states. For example, we would maintain separate tables for the placed pods and yet-to-be-placed pods, as opposed to a single table that has to be scanned for the pending pods each time. Views that use pods as input would then be written to specifically refer to placed pods and/or pending pods, which further bifurcates the schema. Worse, it becomes the DCM user’s responsibility to maintain these separate views as the cluster state evolves, which counteracts our goals of having a declarative programming model.

As an example of this challenge, consider the inter pod anti-affinity matches view in Figure 6. This is a key view in the Kubernetes scheduler we built using DCM, which shows groups of pods that are mutually anti-affine, and are not to be placed on the same nodes. Only a small subset of the final view changes whenever new pods arrive (or a few existing pods leave). However, to keep performance tractable, several base tables and views in the dataflow, including PODS, POD LABELS, MATCH EXPRESSIONS, as well as intermediate views like MATCHING PODS had to be split according to the pattern described above. Many tables and intermediate views in Figure 6 (the gray boxes) are also consumed by multiple downstream relations that relate to other policies. For example, the MATCH EXPRESSIONS and MATCHING PODS relations are used in several scheduling policies that are configured using Kubernetes’ label-based matching DSL. Without result re-use and caching, these relations would be evaluated multiple times per scheduling decision, which further worsens performance. The only way to avoid redundant view evaluations is to maintain base tables that cache results via imperative code; again, a burden on the DCM user.

Figure 7 shows what performance would be without split views. The plot shows the cdf of latency for scheduling decisions in a setting dominated by the inter pod anti-affinity matches view shown in Figure 6. Even for small clusters of 50-100 nodes, the time per scheduling decision is dominated by the database’ latency to evaluate all views; the database takes several seconds whereas the rest of the pipeline contributes under 10-20 milliseconds in total.

Aggregates and triggers. The second pattern was to compute expensive aggregates by simulating materialized views with triggers, which again requires user code. For example, consider the view that maintains the spare capacity per node mentioned above: on every update to a “pods” table that assigned a pod to a node, we set up a trigger that in turn updates that node’s spare capacity. This requires the developer to maintain even more imperative code that needs to be evolved in tandem with the declarative specification of the schema and constraints via C-SQL.

To see the maintenance effort required, we describe our own experience extending a C-SQL specification to support custom resources in Kubernetes. Custom resources are specified by operators at runtime, typically used for configuring special hardware like GPUs or FPGAs. Previously, we had a fixed set of resource types and a column each for demands and capacities for CPU, memory, disk and other known resource types. However, with custom resources, the set of resource types itself is dynamic which prevents us from embedding resource type into the schema. Therefore, we need to split off information about resource demands and capacities from the PODS and NODES base tables into additional tables called POD_RESOURCE_DEMANDS and NODE_RESOURCE_CAPACITYs, each of which had a column each for resource types and resource demands/capacities. In doing so, the spare capacity view computation becomes a four-way join instead of the original two-way join between the PODS and NODES base tables. The schema change also immediately invalidates the previous trigger written using H2’s imperative and non-portable Java APIs.

3.2 Constraint-side challenge: large optimization problem sizes

We now move onto challenges with efficient constraint evaluation. Once DCM pulls in the input data it needs from the database, it encodes the data into an optimization problem to hand to the constraint solver. An optimization problem is represented as a graph of variables linked to one another by constraints.

As cluster sizes increase, constraint solvers become the main bottleneck for DCM’s scalability. This is expected as constraint solvers such as Google’s CP-SAT do not scale well to large problem sizes. To maintain good performance, it is therefore crucial to keep
the size of the optimization problem small. The solver attempts to simplify the optimization problem prior to finding solutions by applying complex rules during a “presolve” phase, including removing redundant variables and constraints and tightening variable bounds. We have noticed in our prior work [50] that this presolve phase is the primary contributor to increased latency at larger cluster sizes.

The fact that cluster management problems are highly incremental also leads to opportunities to improve the solver’s efficiency. Since changes are often much smaller than the size of the database, only a subset of cluster states should be affected by these changes. For example, consider placing a pod with well-defined affinity constraints in a 10K node cluster. If only 10 nodes can satisfy these affinity constraints, it is highly inefficient to burden the solver with reasoning about the remaining 9990 nodes in the optimization problem. Dealing with all nodes in the cluster requires larger tables to flow through the C-SQL pipeline (Figure 1), which is more expensive to encode and to solve. Ideally, we would filter out these 9990 nodes earlier in processing pipeline. However, this type of filtering is non-trivial when a mix of complex constraints are involved, given that each constraint is potentially a rich SQL-like query.

3.3 Overview of solutions

We introduce two query optimization-inspired techniques to improve the execution efficiency of C-SQL programs: automatic incremental view maintenance for the efficient relation evaluation (§4) and feasibility-preserving predicate pushdown (FP-pushdown) for efficient constraint evaluation (§5).

First, instead of requiring users to simulate incremental view maintenance (IVM) through imperative code, we refactor the database component of DCM to use an incremental engine that can automatically incrementalize the computation. We use Differential Datalog (DDlog) [12, 45] as the IVM engine. To bridge the differences in semantics and runtime between SQL and DDlog, we develop a SQL-to-DDlog compiler and a SQL-frontend for DDlog.

Second, we automatically simplify optimization problems generated from C-SQL programs by pushing down predicates from constraints to relations. Recall that in a C-SQL program, each constraint is associated with a relation that defines the context; the more we can restrict the relation, the smaller the encoded optimization problem will be. To do so, we statically analyze constraint queries and extract filtering conditions from the constraint side that can be moved to the relation side. The filtering conditions are specified as additional views on the base relation, which can also benefit from the incremental evaluation enabled by the IVM engine.

In the next sections, we describe both building blocks in detail.

4 RELATION EVALUATION VIA AUTOMATIC INCREMENTAL VIEW MAINTENANCE

As discussed in § 3.1, incremental view maintenance is key to scaling the relation evaluation component of the C-SQL pipeline. While IVM has been a long and active research area, few robust, battle-tested, and well-maintained IVM engines exist. We furthermore needed one that was in-memory and could be embedded in a program as described in § 2.2. For these reasons, we decided to build upon Differential Datalog (DDlog), given that it is developed in-house and already used in production.

DDlog [12, 45] is a programming language for expressing incremental computations. DDlog’s core incremental engine is based on Differential Dataflow [10, 37] and Timely Dataflow [40]. DDlog programs are written in a datalog-based language, but with extensions for rich types, arithmetic, strings, functions and procedural extensions with support for loops and variables, among other things.

A DDlog program instance represents a set of input relations and queries on those relations that result in output relations. DDlog programs operate on changes to inputs relations; the program accepts batches of changes (inserts, deletes, updates) to input relations and outputs a corresponding batch of changes to output relations.

A DDlog program is compiled down to a rust program that implements the dataflow expressed using Differential Dataflow [10]. The generated rust program can then be compiled, linked to and interacted with a range of programming languages, like C, C++, Rust, Java and Go. These client bindings allow programs to instantiate DDlog programs, update input relations and receive changes to output relations. All states in a DDlog program are maintained in memory, which makes it a good fit for representing a scheduler’s view of cluster state as with DCM.

Integrating DDlog was challenging due to differences in semantics and runtime between SQL and DDlog, which we describe below.

4.1 Bridging SQL and DDlog semantics

SQL and DDlog have important semantic differences that make it non-trivial to translate SQL schemas and views into the corresponding DDlog programs. For example, DDlog operates on sets, whereas SQL operates on bags. SQL also uses ternary logic when evaluating predicates (null, true, false), which have implications on whether tuples from different relations being joined appear in output relations. DDlog can generate incremental view maintenance plans for any query, but some operators do not offer efficient incremental evaluation. For example, some aggregates in SQL do not have efficient incremental implementations in space/time (e.g., array_agg(), where the ordinals matter).

We took on a significant undertaking to build a SQL-to-DDlog compiler in roughly 7K lines of code and 4K lines of tests in Java, using the Presto [13] parser as its frontend. The compiler supports standard SQL: select, project, join, windowing, groupby, aggregations, set operations, and most SQL expressions using SQL’s ternary logic semantics. Due to the challenges described above, the compiler is only able to support a subset of SQL. For example, all SELECT queries need to be SELECT DISTINCT queries, to preserve set semantics. Neither the SQL-to-DDlog compiler nor DDlog optimizes query plans; we therefore organize views such that filtering happens earlier in the dataflow, whereas aggregates that are difficult to incrementalize appear later in the pipeline on smaller inputs. This structure also allows us to easily handle some unsupported operations (like LIMIT and ORDER BY) outside DDLog in user code, between the database side and DCM.

4.2 Bridging different runtime interfaces

DCM’s programming model is based on SQL and interacts with standard interfaces like JDBC that are designed for ad-hoc queries. In a typical relational database, the schema can be freely manipulated at runtime via DDL queries. DDlog, however, only supports standing
queries, which means all tables have to be created at initialization and then compiled and dynamically loaded at runtime.

To bridge this gap, we implemented a SQL-frontend to DDlog, using frameworks like JOOQ [8] that DCM already uses heavily. Our frontend allows code to interact with a DDlog program over a JDBC interface as if it were an embedded database without persistence. The frontend requires the entire schema (DDL) to be specified upfront at initialization. It then translates the SQL schema to a corresponding DDlog program using the SQL-to-DDlog compiler, invokes the DDlog compilation toolchain, and loads the resulting DDlog program into memory.

As a result, our existing corpus of DCM-based projects could use DDlog and its incremental computation capabilities with minimal disruption. The JDBC code worked as it used to before, whereas some of the SQL schema and views had to be tweaked to support the slight change in SQL dialect (e.g., the requirement to use SELECT DISTINCT as mentioned above). We have open sourced the SQL-frontend and the SQL-to-DDlog compiler for future users who are interested in experimenting with DDlog [14].

4.3 Simplified programming model

With the compiler and the runtime in place, our C-SQL programs significantly simplified. Avoiding split views (§3.1) was a significant quality of life improvement, both in terms of keeping views simple and focused on their intent, and in terms of the performance engineering needed. For example, previously, when a single logical view was decomposed into several views, it was difficult to identify which queries were performance bottlenecks. It was also tedious to rewrite queries as requirements evolved. With incremental view maintenance however, we were able to write straightforward SQL that stayed true to our design goal of declarative programming: the developer should focus on the what, not the how.

We explain this simplification by revisiting the inter pod anti-affinity matches view in Figure 6. Without DDlog, we would have to split all input views into fixed and changing parts, and perform the resulting bookkeeping in imperative user code, which also impairs maintainability. In addition, all the gray views in Figure 6 would be either evaluated multiple times per scheduling decision, or simulated manually using triggers. With DDlog, all base tables and views are incrementally updated and materialized, eliminating the need for imperative user code for split views and triggers.

In terms of development effort, for the view in Figure 6, our earlier version had to split off and specialize the matching_pods logic for each policy (like the inter-pod anti affinity matches view). For the anti-affinity logic, the split view was carefully scoped to only consider matching rules specified for anti-affinity requirements, and was only partially incremental with respect to the pods table. And yet, this bloated the overall anti-affinity code to about 70 lines of SQL. Fully incrementalizing the computation by hand for all the views involved would have compounded the code size for each additional split view and worsened maintainability. In stark contrast, with DDlog, we wrote matching_pods once, in a reusable and policy-agnostic way, making it possible to write the anti-affinity view with only 20 lines of SQL. We highlight these differences with detailed code examples in Appendix B of the technical report [43].

5 CONSTRAINT EVALUATION VIA FEASIBILITY-PRESERVING PREDICATE PUSHDOWN

In this section, we present the design and implementation of the FP-pushdown optimization, which improves the efficiency of constraint evaluation in the C-SQL pipeline by automatically reducing the size of the optimization problem passed to the solvers. This optimization is directly inspired by the predicate pushdown technique, which moves filtering operators close to data sources to remove irrelevant records early in the processing. Similarly, our idea is to push down predicates from the constraint side of a C-SQL program to the relation side, therefore filtering out irrelevant inputs to the constraint solver early in the pipeline. The net effect of the optimization is similar to the solver’s presolve phase (explained in §3.2), during which variable domains are tightened.

FP-pushdown is different from classic query optimization in that it pushes down filtering conditions from constraint queries (CHECK and MAXIMIZE clauses) to the relations against which the constraints apply. This is done by inferring domain restricting conditions from constraints and expressing them as additional filtering predicates on the relations. Since there is less information about variable domains during the problem definition phase (at initialization, based on the schema alone) compared to the presolve phase (which uses live data at runtime), we statically analyze the constraints to infer such domain restricting conditions. Once these conditions are pushed to the relations, we can leverage classic query optimization techniques, including incremental computation enabled by DDlog, to efficiently execute the domain restricting queries at runtime.

5.1 Problem setup: domain restricting views

Many cluster management problems involve combinatorial optimization tasks (e.g., assigning application to machines), where the solution set is discrete. We therefore focus the discussion of FP-pushdown on discrete decision variables, whose domain can be defined via foreign key constraints as shown in Figure 3.

Suppose that var is a variable column in the base table. and that var is a foreign key that references column uid in the domain table D. Our goal is to automatically generate one or more views R, hereinafter referred to as domain restricting views, that tighten the domain of var without affecting the feasibility of the optimization problem. Each view R has the same schema as the domain table D, but only contains a subset of its records. The views are computed by identifying domain restricting conditions in constraint queries.

Hard and soft constraints may contain a domain restricting condition of the form \( \text{var} \text{ op } Q \). Here, Q is a query or an expression that determines the variable domain, and op is a SQL operator (e.g., IN, NOT IN, =, ≠). For each Q, we compute a domain restricting query \( Q' \) which specifies a variable domain inferred from Q. \( Q' \) might be a superset of Q that omits predicates; \( Q' \) could also be computed as the top K entries from Q, using a custom sort order on Q that encodes domain knowledge. Overall, the smaller the size of \( Q' \), the more we can tighten the domain of var based on Q.

Given a set of domain restricting queries \( Q's \), a domain restricting view R is computed as the union/difference of \( Q's \), depending on whether the \( Q' \) defines parts of the domain to include or to exclude. R now represents a tightened domain of var that we can
5.2 Inferring domain restricting queries (DRQ)

Domain restricting queries (DRQs) are SQL queries that define filtering conditions relevant to the variable’s domain. We consider three types of DRQs:

(1) \( Q_{inc} \): DRQs from hard constraints defining parts of the domain that contain feasible solutions.
(2) \( Q_{exc} \): DRQs from hard constraints defining parts of the domain that do not contain feasible solutions.
(3) \( Q_s \): DRQs based on domain knowledge defining parts of the domain that are likely to contain feasible solutions.

Given a set of DRQs, we can compute a single domain restricting view \( R \) for the variable column via \( R \equiv Q_{inc} \cup Q_s \setminus Q_{exc} \).

**Hard constraints.** Predicates and subqueries that only involve constants are a common type of domain restricting conditions found in hard constraints. For example, the `constraint_node_predicates` in Figure 4 is a hard constraint that contains a domain restricting condition of the form \((\text{var IN/NOT IN } Q_c)\), where \( Q_c \) is a subquery that does not involve variables. Because these conditions do not depend on values of other variables, we can extract and precompute such predicates and subqueries directly as DRQs.

Inclusion DRQs can be pushed down to relations regardless of whether there are additional predicates in the constraint query, since adding additional values to the domain does not affect the correctness of the solution. In comparison, exclusion DRQs \( Q_{exc} \) can not be pushed down safely if there exist any OR predicates in the same constraint that involve non-variable columns. For example, consider the `exclusion_constraint` in Figure 8. This constraint can be rearranged as `exclusion_rewrite_constraint` by moving the non-variable expression into the relation. In the rewritten form, it is clear that the constraint \((\text{var NOT IN } Q_c)\) only applies to a subset of the relation that satisfies the where clause. Therefore, \( Q_c \) can not be excluded from the shared domain of all variables. Appendix C of the extended report [43] discusses how to deal with such constraints that only apply to a subset of the relations.

**Top k with domain knowledge.** In addition to explicitly defined hard constraints, users often apply heuristics and implicit preferences to cluster management problems based on domain knowledge. We encode such knowledge via a custom sort order on the domain table \( D \). The sort order implies that, all else equal, records that are ranked high are more likely to contain feasible solutions compared to those that are ranked low. For example, one common heuristic for placing pods is to prioritize nodes with more available resources. This can be expressed via a sort order based on the spare_capacity column in the nodes table. Another heuristic could be to prioritize nodes without any anti-affinity constraints or topology constraints to those that have many. This can be encoded into the custom sort order by “discounting” the available resources of a node based on constraints. For example, we decrease the spare capacity of a node by a factor of \( 0 < \gamma < 1 \) each time the node appears in such constraints. Finally, the domain restricted query \( (Q_k) \) is computed by taking the top \( k \) entries from the sorted table.

Unlike hard constraints which do not affect correctness, over-restricting the variable domain with small values of \( k \) could incorrectly lead to infeasible solutions. Unfortunately, the solver can not distinguish whether infeasible solutions are because \( k \) is set too small or because the original optimization problem is indeed infeasible. Therefore, in case of infeasible solutions, we simply fall back to the default DCM solver logic which considers the entire variable domain. This avoids the worst-case scenario in which the solver repeatedly evaluates the feasibility of the optimization problem with increasing values of \( k \), only to find the problem itself to be infeasible at the very end. By default, we set \( k \) to be \( \alpha \) times the size of the variable column. Empirically, we observed that setting \( \alpha = 2 \) effectively reduces the problem size without affecting correctness in the workloads that we have experimented with (§6.4).

5.3 Efficient Implementation of DRQs

Finally, we describe the implementation details of extracting DRQs from constraints and using DDL to enforce them.

To identify constraints that affect the domain of a variable column \( \text{var} \), we implement an additional pass in DCM’s compiler that statically inspects the schema and constraints, and applies the rules...
in §5.2 to extract DRQs. After the schema is analyzed, we then construct for each variable column of a single domain restricting view \( R = Q_{inc} \cup Q_{s} \setminus Q_{exc} \).

For each table or view \( T \) referenced by a constraint, the compiler outputs DDL queries for the augmented tables \( T' \). \( T' \) only contains rows from \( T \) that match the domain specified in \( R \) (§5.1). Figure 9 shows an example of an inclusion DRQ inferred from our test workloads, which extracted filtering conditions from two hard constraints. The DRQs are expressed as joins since DDlog does not currently support subqueries. We include a detailed analyses of the contribution of different types of DRQs in §6.4.

The initially supplied schema along with the generated domain restricting views and augmented tables give us the augmented schema. This augmented schema is provided to DDlog at initialization time. In doing so, the FP-pushdown capability is transparent to our Kubernetes scheduler: the core user code that translates the Kubernetes API is mocked (i.e., the nodes themselves are simulated). We have tested with simulated cluster sizes of 500, 5000 and 50000 nodes. We report results from 5 runs for each configuration. The experiments were conducted on a server with 8 Intel Xeon Platinum 8259CL CPUs and 32GB memory.

**Workload.** We use the 2019 Azure public traces [38] to generate workloads. The workload comprises groups of pods that are created with CPU and memory resource reservations. We replay three variants of the workload, each with a different fraction \( F \) of replica groups configured with inter-pod anti-affinities and node affinities within the group. Both policy types are commonly used in practice — configuring inter pod anti-affinities is a recommended Kubernetes best practice, for example [50] — and a challenging constraint to solve as it involves reasoning across groups of pods. Node affinities are used to match pods to nodes that have workload-specific requirements such as custom hardware or to run workloads in pre-defined availability zones. The workload also uses a variety of hard and soft constraints found in Kubernetes scheduler, including capacity constraints and load balancing requirements.

**Baseline.** Our prior work has already shown that DCM outperforms the default Kubernetes scheduler by \( 2 \times \) in scalability [50]. Our experiments therefore focus on comparing against DCM directly. Specifically, we evaluate the following variants of DCM:

- **DCM:** The original implementation of DCM as described in prior work [50].
- **+ivm:** DCM with DDlog as the backend for incremental view maintenance.
- **+ivm+pushdown:** DCM with the DDlog backend and FP-pushdown optimization.

## 6 PERFORMANCE EVALUATION

Our evaluation seeks to answer the following questions:

- How does IVM improve relation evaluation in DCM (§6.2)?
- How does IVM and FP-pushdown combine to help DCM scale to 50K node sized clusters (§6.3)?
- Can FP-pushdown effectively reduce optimization problem sizes and offer robust performances (§6.4)?

Overall, we find that IVM is essential to both scheduling latency and throughput, allowing the system to efficiently compute and query summaries over the entire cluster state. Furthermore, combining IVM and FP-pushdown decreases scheduling latencies by two to three orders of magnitude, thereby enabling a DCM-based scheduler to scale to a cluster of 50K nodes. Lastly, FP-pushdown reduces optimization problem sizes by over 300x in the 50K node case and its performance is stable across a range of top-k parameters.

### 6.1 Experiment Setup

We model the evaluation after the original DCM paper to enable a direct comparison [50].

**Environmental.** Given the focus on scalability, we use DCM’s benchmarking harness to evaluate our Kubernetes scheduler in a setting where the Kubernetes API is mocked (i.e., the nodes themselves are simulated). We have tested with simulated cluster sizes of 500, 5000 and 50000 nodes. We report results from 5 runs for each configuration. The experiments were conducted on a server with 8 Intel Xeon Platinum 8259CL CPUs and 32GB memory.

In §5.2, we present an example of an inclusion DRQ inferred from our test workloads, which extracted filtering conditions from two hard constraints. The DRQs are expressed as joins since DDlog does not currently support subqueries. We include a detailed analyses of the contribution of different types of DRQs in §6.4.

The initially supplied schema along with the generated domain restricting views and augmented tables give us the augmented schema. This augmented schema is provided to DDlog at initialization time. In doing so, the FP-pushdown capability is transparent to our Kubernetes scheduler: the core user code that translates the Kubernetes API is mocked (i.e., the nodes themselves are simulated). We have tested with simulated cluster sizes of 500, 5000 and 50000 nodes. We report results from 5 runs for each configuration. The experiments were conducted on a server with 8 Intel Xeon Platinum 8259CL CPUs and 32GB memory.

**Workload.** We use the 2019 Azure public traces [38] to generate workloads. The workload comprises groups of pods that are created with CPU and memory resource reservations. We replay three variants of the workload, each with a different fraction \( F \) of replica groups configured with inter-pod anti-affinities and node affinities within the group. Both policy types are commonly used in practice — configuring inter pod anti-affinities is a recommended Kubernetes best practice, for example [50] — and a challenging constraint to solve as it involves reasoning across groups of pods. Node affinities are used to match pods to nodes that have workload-specific requirements such as custom hardware or to run workloads in pre-defined availability zones. The workload also uses a variety of hard and soft constraints found in Kubernetes scheduler, including capacity constraints and load balancing requirements.

**Baseline.** Our prior work has already shown that DCM outperforms the default Kubernetes scheduler by \( 2 \times \) in scalability [50]. Our experiments therefore focus on comparing against DCM directly. Specifically, we evaluate the following variants of DCM:

- **DCM:** The original implementation of DCM as described in prior work [50].
- **+ivm:** DCM with DDlog as the backend for incremental view maintenance.
- **+ivm+pushdown:** DCM with the DDlog backend and FP-pushdown optimization.

## 6.2 Performance with IVM

We first compare DCM with and without IVM in a 500-node setting, with both systems using the same cluster state schema. We do not consider simulating materialized views, given that this is not a maintainable trajectory for our codebase (§3.1).

Figure 10 reports the insert and scheduling throughput of DCM before and after adopting DDlog as the IVM backend. The number in the parenthesis indicates the factor by which the arrival rate of pods is scaled up. We find that while DCM without IVM can quickly insert new records into the database in response to the newly created pods (red dotted line), its scheduling rate (blue solid line) struggles...
6.3 Performance with IVM and FP-pushdown

Given IVM is indispensable for DCM’s performance, we conduct all remaining experiments using the DDlog backend.

Impact of FP-pushdown optimization. Figure 12 reports the scheduling latencies for DCM with and without the FP-pushdown optimization. The presented latencies are for a batch of pods, as the scheduler batches up to 50 decisions at a time.

As the cluster sizes increase, without FP-pushdown, scheduling latency increases significantly from a median of hundreds of milliseconds to a few seconds. In comparison, latency with FP-pushdown grows more slowly, staying within the hundreds of milliseconds median latency even at 50K node scale. This is because, with FP-pushdown, the size of the optimization problem is no longer a function of the size of the entire cluster, but a function of the size of changes (i.e., the number of new pods that need to be placed in the system and the nodes being updated as a result). DCM with FP-pushdown consistently improves performance over reasoning about all nodes, where even the p95 latency with 50K nodes is faster than the 5th percentile latency without FP-pushdown.

With $F = 0\%$, FP-pushdown only reasons about the top-K nodes in the cluster, making performance constant relative to cluster sizes, given that the batch size of pods per decision is a constant capped
we see larger problem sizes in cases with larger batches of pods per with the cluster size. With node affinities, a scheduling problem simplifies the optimization problem using complex heuristics. In baseline DCM, as the cluster size increases, the gap between the relative contributions of the database and the constraint solver to the overall latency widens. Specifically, the p95 latency of the database increases from 2.0ms at 500 nodes to 87ms at 50000 nodes, the p95 latency of orToolsTotal increases from 333ms to 706ms, and the p95 latency of modelCreation increases from 68ms to 7056ms. As explained earlier, constraint solvers are notoriously hard to scale to large problem sizes, which makes the solver increasingly a bottleneck at larger cluster sizes. Given the larger number of records involved, all phases in the pipeline, from database fetches, to model creation, presolving and the overall solution search experience significant latency increases. Model creation in particular experiences the worst latency degradation of all the phases, given that that phase involves several passes over the fetched input data to produce the optimization problem encoding.

In comparison, with FP-pushdown, the scheduling latency is relatively unaffected by the increasing cluster sizes. This is because we efficiently and automatically scope the problem so that the fetched data in each placement decision remains small even at large cluster sizes. This in turn leads to all subsequent phases involving the DCM runtime and solver use to speed up as well.

### 6.4 Detailed Analysis of FP-pushdown

#### Effect on optimization problem size

Table 1 reports the average domain size of key decision variables in the optimization problem. Recall that domain size directly affects the search space for the optimization problem. In our case, the decision variables’ domain size is a function of the number of nodes considered per scheduling decision. Without FP-pushdown, DCM reasons over decision variables that are proportional to the size of the cluster regardless of the value of $F$. With FP-pushdown, however, DCM reduces the number of decision variables to $30 \sim 84\%$ of the original in a 500-node cluster, $2.9 \sim 23\%$ in a 5000-node cluster and $0.29 \sim 2.7\%$ in a 50000-node cluster. Table 1 also breaks down the contribution from inclusion ($Q_{inc}$), exclusion ($Q_{exc}$) and topk ($Q_{k}$) DRQs. For our test workloads, $Q_{inc}$ extracts nodes with affinity constraints and taint toleration, and $Q_{exc}$ extracts nodes with anti-affinity constraints. At $F = 0\%$, all contribution comes from the topk DRQ ($Q_{k}$). At $F = 50\%$ and $F = 100\%$, the contribution from $Q_{k}$ shrinks, since variables in the top-k view overlap with those in the inclusion view. The exclusion view is small since we only exclude a particular node if it appears in the anti-affinity constraints of all pods in the scheduling batch. Nevertheless, we adjust the ranking of nodes (an average between 5 to 30) that appear in the anti-affinity constraints in the topk view according to the discounting mechanism described in § 5.2.
Impact of topk. We now measure the impact of $k$ from our top-k mechanism on scheduling latency. In addition to the nodes selected by DRQs, $k$ determines the number of nodes based on user-provided scheduling preferences. $k$ is typically selected based on the number of new pods considered per scheduling decision. We represent the ratio of $k$ to the max number of new pods as $\alpha$, and plot the effect of increasing $\alpha$ in Figure 14. We find that performance is consistent (low milliseconds to tens of milliseconds per batch) across all values of $F$ for low values of $\alpha$ (1-8). This represents a judicious band in the number of extra nodes considered per scheduling decision (50-800 nodes), thereby making $k$ convenient to pick. At higher values of $\alpha$ ($\geq 16$), latencies inflate for $F = 50\%$ and $F = 100\%$, as the model creation, presolve and solving times increase as per the trends shown in Figure 13.

7 RELATED WORK

Data-center management. Several works have tackled the in-flexibility of modern cluster management designs. DCM [50] tackles the recurring problem of combinatorial optimization tasks in datacenters via a SQL-based programming model which we formalize as C-SQL in this paper. DBOS [47] proposes a multi-node datacenter operating system, built around a distributed database to manage state, encouraging a programming model based on traditional SQL and stored procedures. C-SQL can be used to add constraint optimization based decision-making to a DBOS deployment.

A long body of works have explored the use of solvers to aid various facets of datacenter resource management [16, 19, 20, 22–24, 27, 29, 52, 53]. These systems are not based on a relational programming model like C-SQL, and instead use hand-crafted constraint solver encodings, which are comparatively difficult to extend and evolve over time.

Using custom solvers for specific problem domains can also yield significant rewards. For example, Shard Manager [34] uses domain knowledge to partition the problem and use local search to assign shards in distributed applications. Wrasse [41] uses a balls-and-bins based abstraction to model allocation problems and uses a GPU-based solver to find assignments. DCM and C-SQL allow different constraint solver backends to be plugged in for constraint evaluation, which allows such approaches to benefit from the simpler and more expressive relational programming model.

Lastly, DeltaPath [17] builds a network routing controller that is capable of incremental updates built on top of differential dataflow. Full-stack SDN [49] demonstrates the use of an incremental engine to build a software-defined networking stack. Similar to DCM and C-SQL, these papers highlight the opportunity of using incremental computation to build key control plane functionality.

Constraint query language. The idea of combining declarative database programming with constraint solving has been previously explored in the context of constraint databases [30, 31, 33, 44], albeit with a different focus on the representation and querying of spatial temporal data [18, 42, 51]. The main idea is that a tuple in the relational database model can be generalized to represent a conjunct of constraints over a small number of variables. These generalized tuples can therefore provide a finite representation of infinite sets, such as a spatial object which is an infinite point set. In comparison, C-SQL targets combinatorial optimization problems in cluster management applications. In terms of semantics, C-SQL supports additional forms of constraints in addition to conjunctions, such as objective functions defined via $\text{MAXIMIZE}$ clauses.

Predicate pushdown. Predicate pushdown [54] is a well-known query optimization technique for pushing filtering and projection operators down a query plan tree as far as possible, to reduce subsequent query processing costs. As a generalization of the pushdown technique, the idea of moving predicates around (e.g., up, down and sideways) in the query plan for performance reasons has been explored in the 90s [15, 26, 35]. Most query optimization techniques cannot push down predicates below a join operator; unless the predicates are on columns used in the join condition [21, 39, 46, 56]. Predicate pushdown has been implemented in data analytics systems and data warehouses such as Apache Spark [11], Snowflake [2] and BigQuery [5] as a crucial performance feature to eliminate unnecessary data partitions. More broadly, predicate pushdown style optimization has shown performance benefits in applications such as sensor networks [48], dataflow operators [28], video analytics systems [32] and machine learning inference queries [36]. FP-pushdown is another example of applying the predicate pushdown technique in a new domain of constraint solving. In particular, the optimization extracts predicates from the constraint side of C-SQL programs to push down to the relation side.

8 CONCLUSION AND OPEN QUESTIONS

In this paper, we report on our experience improving the scalability and programmability of a Declarative Cluster Manager architecture. Our experience highlights the power of the declarative programming model, which in our case, centers around DCM’s constraint language C-SQL. C-SQL extends SQL with novel semantics for expressing optimization problems. C-SQL allows programmers to specify what the cluster management should achieve and leaves the optimization and execution plans to the runtime – something that today’s heuristic-based cluster managers cannot do. In addition, C-SQL’s similarity to SQL allowed us to adapt decades of query optimization research to optimize the execution of C-SQL programs. Our query optimization inspired techniques, incremental view maintenance and FP-pushdown, significantly speed up the relation and constraint evaluation components of a C-SQL program. As a result, we are able to scale a DCM-powered Kubernetes scheduler past its original limits to hyperscale-sized cluster and do so with simpler schema and less imperative user code.

The inherently incremental nature of cluster management makes it an ideal setting for leveraging incremental computation within C-SQL. Incremental computation itself is an active area of work, with open questions around the classes of programs that can be automatically and efficiently incrementalized. In the context of C-SQL, extending the pipeline to also incrementally prepare and solve constraint programs is a clear avenue for future work. In addition, FP-pushdown simplifies optimization problems automatically, speeding up constraint evaluation latencies by one to two orders of magnitude. However, whether more solver optimizations performed by a constraint solver could be pushed down to the database layer is an open question. Part of the challenge is that the full set of solver optimizations remain highly domain- and input-specific, given an optimization problem with a mix of constraints.
REFERENCES


Appendices

A FORMAL SEMANTICS OF THE CONSTRAINT LANGUAGE

A.1 Types

We assume some base types \( BT \), including \( \mathbb{N} \) (integers), \( \mathbb{B} \) (Booleans), \( \mathbb{R} \) (reals), and \( S \) (strings).

A.2 Formulas

For each base type \( T \) we introduce a new type, \( F(T) \), which is the type of symbolic formulas with type \( T \). These formulas are syntactic objects, defined by the following grammar:

\[
\text{formula} ::= \text{Const} \\
\quad \text{| VariableName} \\
\quad \text{| formula binaryOperation formula} \\
\quad \text{| unaryOperation formula} \\
\quad \text{| aggregate} (\text{formulaList}) \\
\]

\[
\text{formulaList} ::= \text{formula} \\
\quad \text{| formulaList formula} \\
\]

\[
\text{binaryOperation} ::= \text{AND} \mid \text{OR} \mid = \mid != \mid > \mid >= \mid < \mid <= \mid + \mid - \mid \star \mid / \mid \%
\]

\[
\text{unaryOperation} ::= - \mid \text{NOT}
\]

\[
\text{aggregate} ::= \text{ANY} \mid \text{ALL} \mid \text{SUM} \mid \text{COUNT} \mid \text{MIN} \mid \text{MAX}
\]

A.3 Relations

We use the term "relation" to refer to database tables, views, and results produced by queries. As in SQL, all relations are statically typed, with types that can be inferred using a simple directed analysis. All values in a column have the same type, which is either specified (statically) by the database schema or inferred from the query or view structure. A row (of a relation) is a function that maps a column name to a value of the corresponding type. We use \( \{\text{col0} \mapsto 5, \text{col1} \mapsto 2\} \) to denote a row with value 5 in column \( \text{col0} \) and value 2 in column \( \text{col1} \). A relation is a set (or multiset) of rows.

The type of a column can be a base type \( T \) or a formula type \( F(T) \). For tables, all values in a column of type \( F(T) \) are required to be \( \text{VariableName} \). More general formulas of type \( F(T) \) can only appear in queries or views.

As in SQL, the semantics of a query or view is a function of a database instance \( DB \), which stores the contents of all base tables. In contrast to SQL, where the semantics of a relation is a (multiset) of tuples, in our model the semantics of a relation is a set of constraints involving symbolic variables. We define this semantics bottom-up. We start by defining the semantics of an expression. Expressions can in general appear in SELECT, WHERE, GROUP BY, JOIN, and HAVING statements.

A.4 Semantics of expressions

The grammar of expressions is given by:

\[
\text{expr} ::= \text{expr} \text{ binaryOperation } \text{expr} \\
\quad \text{| unaryOperation } \text{expr} \\
\quad \text{| Const} \\
\quad \text{| ColumnName} \\
\quad \text{| Constant}
\]

An expression is always evaluated in the context of a row. The type of an expression depends on the types of its arguments: when all arguments have base types, the expression itself evaluates to a base type, as in standard SQL. However, if any of the arguments is a symbolic formula, then the expression also evaluates to a symbolic formula. We show the semantics of an expression \( e \) when applied to a row \( r \) as \( \llbracket e \rrbracket (r) \).

\[
\llbracket \text{Const} \rrbracket (r) = \text{Const} \\
\llbracket \text{ColName} \rrbracket (r) = r(\text{ColName}) \\
\llbracket \text{expr1 binaryOp expr2} \rrbracket (r) = \llbracket \text{expr1} \rrbracket (r) \text{ binOp } \llbracket \text{expr2} \rrbracket (r) \\
\llbracket \text{unOp expr} \rrbracket (r) = \text{unOp } \llbracket \text{expr} \rrbracket (r)
\]

For example, consider the expression \( e = 1 + \text{Age} \), where \( \text{Age} \) is a column name. Let us evaluate it in two contexts:

- When \( r = \{\text{Age} \mapsto 10\} \), we have that \( \llbracket e \rrbracket (r) \) evaluates to an integer with value 11.
- When \( r = \{\text{Age} \mapsto x + 2\} \), where \( r(\text{Age}) \in F(\mathbb{N}) \) is an integer formula, we have that \( \llbracket e \rrbracket (r) \) evaluates to an integer formula, which is \( 1 + (x + 2) \).

A.5 Semantics of row expressions

Consider a SELECT statement such as SELECT \( e1 \) AS \( \text{col1} \), \( e2 \) AS \( \text{col2} \) FROM \( T \). This statement evaluates two expressions and specifies a column name for each of them (the column names must
be distinct). This statement evaluates a row expression. The grammar of a row expression is given by:

\[
\text{<simpleRowExpr>} ::= \text{<expr>} \ AS \ \text{colName}
\]

\[
\text{<RowExpr>} ::= \text{<simpleRowExpr>}
\]

The semantics of a \text{<simpleRowExpr>} is a row with a single column: \[\{e \ AS \ \text{colName}\}(r) = \{\text{colName} \mapsto e(r)\}.
\]

The semantics of a row expression is the disjoint union of all the components:

\[
\text{rowExpr}(r) = \bigcup_{e \in \text{rowExpr}} [e](r).
\]

A.6 Semantics of relations

Now that we understand the semantics of expressions, we can define the semantics of relations. First, we define the meaning of an expression in the context of a relation, the result is a (multi)set.

Evaluating an expression in the context of a relation \(R\) produces a set composed of the evaluation of the expression for each row of the relation:

\[
[e](R) = \bigcup_{e \in R} [e](r).
\]

Aggregate expressions can only be evaluated in the context of a collection of values; these values can be the values in a table, query, view or in a group produced by a group-by clause:

\[
\text{<aggregateExpr>} ::= \text{<aggregate>} (<\text{expr}>)
\]

An aggregate expression over a set of base type values produces, as in SQL, a result with a base type. An aggregate expression evaluated over a set of formulas produces a (single) corresponding formula involving the aggregate:

\[
\text{agg}(e)(R) = \text{agg}([e](R)).
\]

For example, consider the expression SUM(col) evaluated over a relation where col is a column of type \(F(N)\) with the following rows: \{col \mapsto 1 + \text{var1}, \text{col} \mapsto \text{var2} + \text{var1}\}. This expression evaluates to a formula with type \(F(N)\), which is SUM(1 + \text{var1}, \text{var1} + \text{var2}). (Recall that the grammar of symbolic formulas includes aggregation functions.)

Similarly, evaluating a row expression in the context of a relation \(R\) produces another relation, which is composed of the union of the results of the evaluation of the expression for all the rows:

\[
\text{rowExpr}(R) = \bigcup_{e \in \text{rowExpr}} [e](r).
\]

We now define the semantics of a relation inductively on its structure. For relations that operate on relations with only base types, we allow the full SQL language with no restrictions. The following simplified grammar show the operations available for relations that have at least one column that is a symbolic formula.

\[
\text{<relation>} ::= \text{SELECT} \ <\text{RowExpr}> \\
\text{ FROM (} \ <\text{tableRef}> \ | \ <\text{join}> \text{)} \\
\text{[ WHERE } \ <\text{baseExpr}> \text{] } \\
\text{[ GROUP BY } \ <\text{baseColumnList}> \text{] } \\
\text{[ HAVING } \ <\text{baseExpr}> \text{]}
\]

\[
\text{<join>} ::= \text{<relation>} \ JOIN \ <\text{relation}> \ ON \ <\text{baseExpr}>
\]

In this grammar, we denote any \text{<expr>} that evaluates to a base type by \text{<baseExpr>}, and a list of columns that all have base types by \text{<baseColumnList>}\footnote{These are, in fact, semantic checks enforced by the type-checker, and not by the grammar.}. Notice that we do not allow filtering, joining, or grouping by expressions that evaluate to symbolic formulas. This restriction ensures that the constraints generated are tractable.

The semantics of a \text{<relation>} is always evaluated in the context of a concrete database \(DB\). We now give a semantics for each of the possible productions in the grammar.

The semantics of a table is simply a (multi)set composed of the semantics of all its rows, each evaluated on itself: \(\bigcup_{e \in T} [e](r)\). Recall that a table can only contain base type values or variable names. Consider the following table with schema \(\{\text{col0} : \mathbb{N}, \text{col1} : F(N)\}:

\[
\{\{\text{col0} \mapsto 0, \text{col1} \mapsto \text{var1}\}, \{\text{col0} \mapsto 2, \text{col1} \mapsto \text{var2}\}\}
\]

The semantics of this table is a relation with two columns: \text{col0} of type \(\mathbb{N}\), and \text{col1} of symbolic formulas with type \(F(N)\).

The semantics of a view defined by a query is the semantics of the query itself:

\[
\text{CREATE VIEW V AS } Q(DB) = Q(DB).
\]

The semantics of filtering a relation \(R\) by a predicate \text{expr} (as occurring in WHERE and HAVING) is defined as in SQL; the fact that filtering expressions need to evaluate to \(\mathbb{B}\) (and cannot have type \(F(\mathbb{B})\)) ensures that filtering can be evaluated immediately, and does not need to produce constraints that are evaluated by the solver:

\[
\langle e \rangle(R) = \{ r \mid \text{expr}(r) = \text{true}, r \in R \}.
\]

Similarly, GROUP BY, requires that the columns grouped-on have base type values; its semantics is defined as in SQL: the result of a GROUP BY is a set of nested relations, where each column name is mapped to a relation containing all rows belonging to the group:

\[
\langle \text{GROUP BY } c_1 \cdots c_n \rangle(R) = \\
\bigcup_{g \in R} \{ c_1 \mapsto \{ r \mid g(c_1) \mapsto \{ r \mid r \in R \land g(c_1) = r[c_1]\} \}
\]

As in SQL, we require the result of each GROUP BY to be immediately used in an aggregation.

Finally, the JOIN is semantically equivalent to a Cartesian product followed by a filter and a selection; since the join condition is constrained to use only base types, the join filter is implemented as described above. So we only need to define the semantics of a Cartesian product, which is the same as in SQL, assuming that the column names of the joined relations are distinct:

\[
\langle R_1 \times R_2 \rangle(DB) = \{ r \mid r_1 \in R_1 \land r_2 \in R_2 \}.
\]

A.7 Generating constraints

Finally, our goal is to have a language that generates constraints that are solved automatically. So far we have reused SQL to create a language that can generate symbolic formulas. We now add one extra statement to SQL that creates constraints from formulas, as follows:

\[
\text{<constraint>} ::= \text{<constraint> [ , } \text{<constraint> ]}
\]

\[
\text{<constraint>} ::= \text{CREATE CONSTRAINT } \langle\text{identifier}\rangle \ AS \ \langle\text{checkOrOptimize}\rangle \ FROM \ \langle\text{relation}\rangle
\]
A CHECK statement generates a set of constraints. A MAXIMIZE statement generates an optimization function. An optimization problem can contain multiple CREATE CONSTRAINT statements. We now specify the semantics of these statements.

A CHECK statement must be followed by an expression of type $F(\mathcal{B})$. Applying the statement to a row evaluates the expression for the specified row and generates a Boolean symbolic formula, which is then interpreted as a constraint.

$$\text{CHECK } expr\rangle(r) = \text{expr}\rangle(r).$$

For example, the formula $\text{var}3 + 2 = 5$ generates the constraint $\text{var}3 + 2 = 5$, which has a unique solution of 3 for $\text{var}3$. Applying the statement to a relation generates the conjunction of the constraints for all rows:

$$\text{CHECK } expr\rangle(R) = \land_{r \in R}\text{CHECK } expr\rangle(r).$$

Similarly, $\text{AllDifferent}(\text{expr}\rangle(r))$ requires all formulas to evaluate to different values.

A MAXIMIZE statement takes an arbitrary expression of type $F(\mathcal{H})$ or $F(\mathcal{E})$ and generates an optimization function. This is a symbolic formula of a numeric type that depends on the symbolic variables.

In the end, each CREATE CONSTRAINT statement generates either a set of constraints to satisfy or an optimization function. We can model that as a pair containing a set of constraints and an optimization function:

$$\langle \text{constraint}\rangle(DB) \in (2^{F(\mathcal{B})}, F(\mathcal{H}))$$

, where the set of constraints is empty for a MAXIMIZE statement, or the formula to maximize is a constant for a CHECK statement.

The semantics of a problem, which is a sequence of CREATE CONSTRAINT statements, is then given by the following: the union of all sets and the sum of all functions:

$$\text{problem}(DB) = \text{combine}_{c \in \text{problem}}\text{constraint}(c)(DB)$$

where the combine function is defined as:

$$\text{combine}(\{(c_i, a_i)\}) = \bigcup_i \sum a_i$$

i.e., form the union all constraints and add up all optimization functions.

### B VIEW SIMPLIFICATION WITH DDLOG

We provide an example of the kind of simplification that emerges in our views when using DDlog.

Kubernetes’ design extensively leverages a mechanism to tag cluster entities (like pods, nodes, volumes etc.) with key-value labels. Control plane code can then use a DSL to specify search queries for entities based on these labels. For Kubernetes schedulers, the labeling mechanism is used to support policies like inter-pod anti-affinity. For example, an anti-affinity requirement can be configured for a pod $A$ with a match expression (app In [web-server]); doing so specifies that the pod $A$ must avoid nodes where a pod has the label app with value web-server.

Figure 15 shows a simplified version of the SQL we first wrote (without DDlog) to identify the set of pods that match an anti-affinity requirement. We have two types of label operators, In and Exists, which are similar except that Exists only checks if a label’s key matches and ignores the value of the label. Without DDlog, it was a challenge to keep performance tractable by having as few records be scanned per query. We had to carefully design the schema to have a table that only has match expressions specified as part of an anti-affinity requirement and scope the matching_pods view to only consider those pods that are yet to be assigned. With the databases we evaluated like H2, even with carefully designed indexes, doing so still does not guarantee that the pod_info and pod_labels...
we can compute domain restricting views to restrict variable do-

views thereby not only improves performance but also maintain-

views. Note the absence of any joins specific to the anti-affinity

<table>
<thead>
<tr>
<th>false</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>pod3</td>
<td>True</td>
<td>Guaranteed</td>
</tr>
<tr>
<td>pod2</td>
<td>false</td>
<td>BestEffort</td>
</tr>
<tr>
<td>pod1</td>
<td>false</td>
<td>Burstable</td>
</tr>
</tbody>
</table>

Figure 17: Base table pods_to_assign with one variable col-

cumn node_name and two non-variable columns has_port_req

and qos_class.

CREATE CONSTRAINT qos_constraint AS
SELECT * FROM pods_to_assign
CHECK (qos_class = 'BestEffort' OR
node_name IN (SELECT node_name FROM valid_nodes));

CREATE CONSTRAINT port_constraint AS
SELECT * FROM pods_to_assign JOIN
pods_with_port_requests_scheduled
ON pods_to_assign.uid = pods_with_port_requests_scheduled.uid
CHECK (has_port_req = false OR NOT(CONTAINS(
pods_with_port_requests_scheduled.node_matches,
pods_to_assign.node_name)));

Figure 18: Example constraints.

pod_labels tables will not be scanned by the query planner, and

the query is therefore not fully incremental. Worse, the same match

expression logic has to be repeated for each type of scheduling

policy that needs it, such as pod and node affinities, taints and tol-
erations, and more. Doing so bloats the schema artificially, making

schema evolution a challenge.

Figure 16 instead shows what incremental view maintenance
enables. With DDlog, we instead have a single, concise, and general
query to evaluate all match expressions, that are shared by different
views. Note the absence of any joins specific to the anti-affinity
policy or the set of pods under consideration. Instead, these joins
appear in a single corresponding downstream view per scheduling
policy (like the inter-pod affinity matches view in Figure 6). Be-
ing incrementally updated and shared across multiple downstream
views thereby not only improves performance but also maintain-

ability.

C GENERATING MULTIPLE DRQS

Depending on whether different cells in a variable column are

treated as a single decision variable with a shared domain or in-
dependent decision variables with different constraints, we can
generate one or more domain restricting views for each variable
column. Specifically, there are three levels of granularity at which
we can compute domain restricting views to restrict variable do-


(1) Column-level: generating a single domain restricting view

R for the variable column var.

(2) Group-level: generate a domain restricting view R_i for each

group g_i = \{c_1, c_2, ..., c_k\} (T) of cells, where c_1, c_2, ..., c_k

are a subset of input (non-variable) columns on the same table.

(3) Cell-level: generating one domain restricting for each cell

in the variable column.

Here, we discuss how to extend column-level domain restricting
views to the group-level. Note that cell-level views are a degenerate
case of group-level views, where each group contains one cell.

View generation at the group level is similar to the column-
level except it requires an additional preprocessing step that par-
titions the variable column and the constraint set by the input
(non-variable) columns c_1, c_2, ..., c_k. For the purpose of refining
DRQs, only the subset of non-variable columns that appear in the
constraints and group-by clauses need to be considered as grouping
columns. Given each partition of the variable column and constraint
set, one can use the process described in Section 5.2 to generate
the corresponding domain restricting views for the partition/group.

Compared to column-level view generation, the additional infor-
mation introduced by the input columns can be used to eliminate
or to extract additional DRQs for each group.

We illustrate this idea using the example constraint in Figure 17
and Figure 18. Here, both non-variable columns qos_class and
has_port_req are referenced in the constraints, so we have a total
of 6 row groups defined by their Cartesian product has_port_req =
\{false, true\} \times qos_class = \{BestEffort, Guaranteed, Burstable\}. For
constraints of the form predicate(non_var) AND/OR DRQ, we

can refine the DRQs by explicitly evaluating predicates involving
the non variable columns using the grouping information. For exam-
ple, in port_constraint, the 3 groups with (has_port_req=false)

can ignore the DRQ since the constraint is already satisfied; the
3 groups with (has_port_req=true) can safely push down the
DRQ (both inclusion and exclusion), since the domain restricting
condition must be satisfied by all members of the group. Similarly,
the two groups with (qos_class="BestEffort") can ignore the
qos_constraint. As a result, we are able to refine the domain re-
stricting conditions for different row groups compared to using a
single domain restricting view for the entire variable column.