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

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

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 define DCM's declarative language, C-SQL, which is based on SQL but importantly introduces new semantics to support constraint optimization problems. To improve the execution efficiency of C-SQL programs, we explore and adapt techniques from classic query optimization, namely incremental view maintenance and predicate pushdown, to accelerate the evaluation of the relation and constraint parts of C-SQL programs respectively. We evaluate the effectiveness of our optimizations through a case study of building Kubernetes schedulers using C-SQL. Our optimizations demonstrated an almost 3000× speed up in database latency and reduced the size of the optimization problem as much as 1/300 of the original without affecting the feasibility.

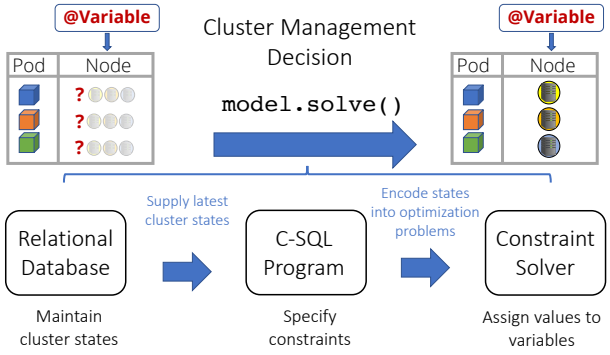

Figure 1: DCM processing pipeline. On each cluster management decision (e.g., assigning new pods to nodes in Kubernetes), DCM pulls in the latest state (tables and views) from the cluster state representation, encodes the state and constraints into an optimization problem, solves the problem using the constraint solver, and returns the tables with values assigned to variable columns.

PVLDB Reference Format:
Kexin Rong[1,2], Mihai Budiu[1], Athinagoras Skiadopoulos[3], Lalith Suresh[1], Amy Tai[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 [8], OpenStack [3], and OpenShift [2] 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 *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 *soft constraints* (e.g., spread these web servers across different geographies if possible), which represent the quality of a decision (or the objective function). Cluster management logic is notoriously hard to develop [46], since it routinely involves combinatorial optimization tasks that cannot be efficiently solved using best-effort heuristics as is the norm today.

Declarative Cluster Managers (DCM) [46] 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. 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

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

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) [11, 41], to enable incremental computation on the relations. Previously, DCM users used traditional in-memory embedded databases (e.g., H2 [5]) 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 changing 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 the optimization problems generated from a C-SQL program without affecting the feasibility. Constraint solvers already attempt to simplify a given problem in a "presolve" phase before they start searching for solutions, by eliminating unnecessary variables and restricting the domain of variables. Analogous to predicate pushdown, FP-pushdown pushes down relevant predicates from the solver's presolve phase to the relation-evaluation phase of the 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 all the way 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 formally present DCM's declarative constraint language C-SQL, highlighting its similarities and differences with respect to regular 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).

We use the running example of building a Kubernetes scheduler 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 [46], 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.

2.1 Background: heuristic-based cluster management

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, 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

```
Program ::= (ConstraintQuery | IntermediateView)*
ConstraintQuery ::=
    CREATE CONSTRAINT <identifier> AS
    (CHECK (Expr | ConstraintStatement)) | (MAXIMIZE Expr)
    FROM Relation
IntermediateView ::= CREATE VIEW <identifier> AS Relation
ConstraintStatement ::= AllDifferent(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 (, Expr)*
          FROM ( Table | Join )
          [Where NonVarExpr]
          [GroupBy 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.

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, so that the scheduler can make coherent decisions without having to scan the cluster state repeatedly. These types of precomputing and caching optimizations are necessary to keep scheduler performance tractable.

Current designs and challenges. Today's cluster managers use the "filter-score" 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) in order 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 each soft constraint is evaluated to output a score for a given node. After all candidate nodes go through the pipeline, 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. 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 can not be easily implemented in a policy-aware manner and can therefore miss feasible solutions.

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 [46]). 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 constraints (hard and soft) that the cluster manager should enforce on the cluster state database, instead of writing hand-crafted heuristics. We present C-SQL's semantics and execution below.

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 the simplified grammar of C-SQL in Figure 2, and defer the formal specification to Appendix A of the technical report [39].

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.

```
-- @VARIABLE_COLUMNS (node_name)
CREATE TABLE pods
(
 pod_uid CHAR(14) NOT NULL PRIMARY KEY,
 status VARCHAR(10) NOT NULL,
 namespace VARCHAR(100) NOT NULL,
 node_name VARCHAR(100),
 ... -- more columns
 FOREIGN KEY (node_name) REFERENCES nodes(name)
);
```

Figure 3: Schema `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 CONSTRAINT constraint_node_predicates AS
CHECK (node_name IN (SELECT node_name FROM valid_nodes)) FROM
FROM pods_to_assign;

CREATE CONSTRAINT constraint_node_preferences AS
MAXIMIZE (node_name IN (SELECT node_name FROM least_loaded))
FROM pods_to_assign;
```

Figure 4: Example hard and soft constraints (`CHECK` and `MAXIMIZE` clauses) that constraints or prefers that the variable column `node_name` is assigned to nodes described in some views (`valid_nodes` and `least_loaded`).

2.2.2 ***C-SQL evaluation.*** To evaluate a C-SQL program, the DCM runtime translates the current database into an optimization problem and solves it with a constraint solver. In doing so, the runtime assigns values to variable columns according to the constraints specified in C-SQL. Concretely, the runtime generates code, which when invoked, queries the set of 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 [6].

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. 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` column in the `pods` table (Figure 3), which represents the variables in a scheduling decision. For discrete variables such as the `node` column, the variable domain can also be specified in the schema via a foreign key constraint.

Operation	Description
`model = Model.compile(CSQLProgram)`	Invoke DCM compiler to generate a solving strategy from the C-SQL program
`model.solve(conn, timeout)`	Pull data from JDBC connection, solve constraints and return solution as tables

Figure 5: DCM's programming model.

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).

3.1 Relation-side challenge: simulating materialized views

We first discuss the challenges with efficient relation evaluation. 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 have any spare capacity requires a join between four tables, representing the set of all pods, their resource demands, the set of all nodes and the node resource capacities. Given views often change little between successive invocations of `model.solve()`, recomputing from scratch each time is highly inefficient. Top-down query evaluation as with

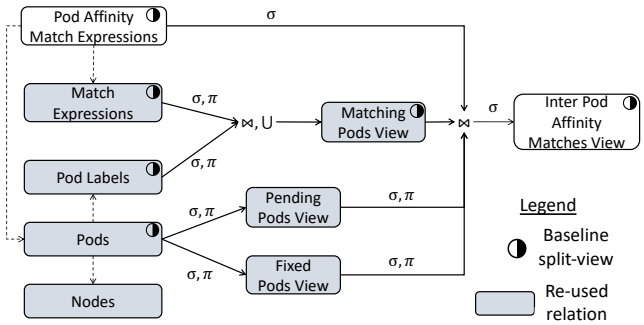

Figure 6: Simplified computation graph for the "inter-pod anti-affinity matches" view. Solid lines indicate input/output relationships, dashed lines indicate foreign key relationships. Without DDlog, many views would be split. The relations in gray are re-used across several view calculations – with DDlog, we re-use the effort to update these views.

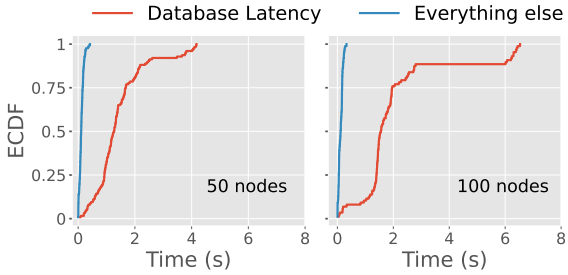

Figure 7: Without split views or an incremental engine, the latency for each scheduling decision is bottlenecked by the database' view evaluation latency, even in small clusters.

most databases is therefore a poor fit for such workloads. To improve efficiency, DCM users had to manually simulate materialized views in various ways. We describe two common approaches below.

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 oending 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 the 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; it shows groups of pods that are mutually anti-affine to one another, 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 the 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 iterative 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 20 milliseconds in total.

Aggregates and triggers. The second pattern was to compute expensive aggregates by simulating materialized views with triggers, which again requires imperative 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_CAPACITIES, 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 not surprising, as constraint solvers like Google's CP-SAT are notoriously hard to scale to large

problem size. Therefore, it is important to keep the size of the optimization problem small. To simply the optimization problem before searching for solutions, the solver already applies complex rules during a "presolve" phase, including eliminating redundant variables and constraints and tightening variable bounds. As observed in our prior work [46], this presolve phase is the primary contributor to increased latency at larger cluster sizes.

The fact that cluster management problems are incremental in nature also leads to opportunities to improve the efficiency of the solver. 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 into an optimization problem, and more expensive 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) [11, 41] 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 problem generated from C-SQL programs by pushing down predicates from constraints to relations. Recall that in the C-SQL specification, each constraint is associated with a relation that defines the context; the more we can restrict the relation, the smaller the optimization problem will be. To do so, we statically analyze constraint queries and extract filtering conditions from the constraint side to be moved to the relation side. The filtering conditions are specified as additional views on the base relation, which can then benefit from the incremental evaluation enabled by the IVM engine.

In the next sections, we describe both of these building blocks.

4 RELATION EVALUATION VIA AUTOMATIC INCREMENTAL VIEW MAINTENANCE

As explained earlier, incremental view maintenance is key to scaling the evaluation of relations in 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.

Differential Datalog (DDlog) [11, 41] is a programming language for expressing incremental computations. It builds upon the differential dataflow [9] library. DDlog programs are written in a datalog-based language, but with extensions for expressing 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 [9]. 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 [12] 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 [7] 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 [13].

4.3 Simplified programming model

With the compiler and the runtime in place, our C-SQL programs *significantly* simplified. Avoiding split views (§3) 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. Worse, it was 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 [39].

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, 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 variables, whose domain can be defined in schema via foreign key constraints such as in Figure 3.

Suppose that *var* is a variable column in the base table. Suppose that *var* is a foreign key that references column *uid* in 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 correctness of the optimization problem. Each view R has exactly one column *uid*, and contains a subset of the records in the domain table D. The views are computed by identifying domain restricting conditions in constraint queries.

Hard and soft constraints may contain a domain restricting expression of the form `(var 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.

```
CREATE CONSTRAINT exclusion_constraint AS
    CHECK ((NonVarExpr) OR (var NOT IN Qc))
    FROM Relation

CREATE CONSTRAINT exclusion_rewrite_constraint AS
    CHECK (var NOT IN Qc)
    FROM Relation WHERE NonVarExpr == false
```

Figure 8: Example of an exclusion condition that can not be safely pushed into the relations due to a NonVarExpr in the OR predicate of the constraint query.

```
(SELECT DISTINCT name,resource,capacity
 FROM spare_capacity_per_node AS t1
 JOIN pod_node_selector_matches AS t2
 ON ARRAY_CONTAINS(t2.node_matches, t1.name))
UNION
(SELECT DISTINCT name, resource,capacity
 FROM spare_capacity_per_node AS t1
 JOIN pods_that_tolerate_node_taints AS t2
 ON t1.name = t2.node_name)
```

Figure 9: Example inclusion domain restricting queries in which `spare_capacity_per_node` is the domain table.

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 incorporate into the DCM model definition. Let tables/views being accessed in Q be $REL(Q)$. For each table/view T in $REL(Q)$, we can compute an augmented table $T' \equiv (T \bowtie_{T.uid=R.uid} R)$. At runtime, the DCM model reads from T' instead of T.

The above problem formulation generates a single domain restricting view R for the entire variable column var, which we refer to as column-level views. Alternatively, we can treat each cell in the variable column as an independent decision variable with potentially different domains and generate a separate domain restricting view for each cell. The column-level view is essentially the union of the domains of variables in each cell. For the purpose of reducing optimization problem sizes, the two options are comparable, since the number of variables in the optimization problem is determined by the size of the unioned domains. Therefore, the discussions in this section assume column-level views and we defer extensions of our technique to generating multiple domain restricting views in Appendix C of the technical report [39].

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 set membership query of the form `(var IN/NOT IN Qc)`, 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}s) 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 `(var NOT IN Qc)` 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 [39] 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 ones 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 capacity 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_s) 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 ks, only to find the problem itself to be infeasible at the very end. By default, we set k to be α 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

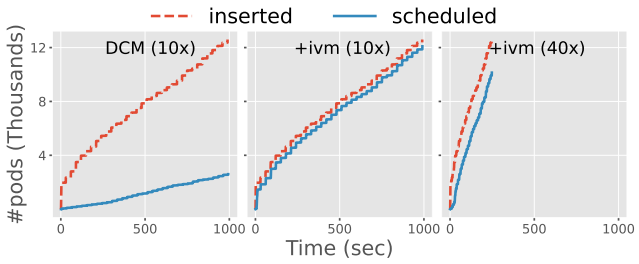

Figure 10: Insert and scheduling rate in a 500-node cluster with and without incremental view maintenance (DDlog).

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

To inspect constraints that affect the domain of a variable column *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 *var*, 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 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 decisions made by the DCM model back to the Kubernetes API is oblivious to FP-pushdown being used. Using DDlog, the additional views imposed by the augmented schema are also evaluated incrementally on every modification to the cluster state. We evaluate the end-to-end implications of the scheme in §6.

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 300× in the 50K node case and its performance is stable across a range of top-k parameters.

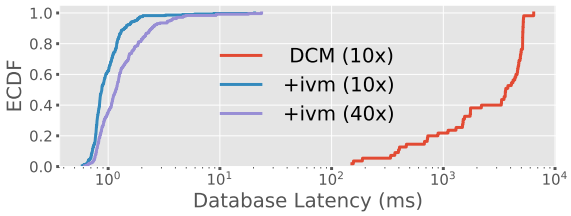

Figure 11: Distribution of database latency (fetch and update records) before and after incremental view maintenance.

6.1 Experiment Setup

We model the evaluation after the original DCM paper to enable a direct comparison [46].

Environment. 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.

Workload. We use the 2019 Azure public traces [35] 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 [46] — 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× in scalability [46]. 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 [46].
- +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

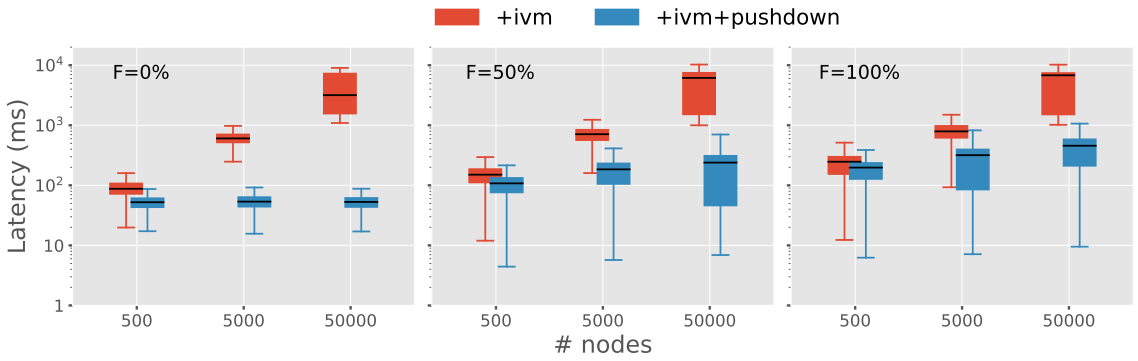

Figure 12: Scheduling latency for a batch of pods (max 50) at 10× trace speed up (log-scale) and different cluster sizes.

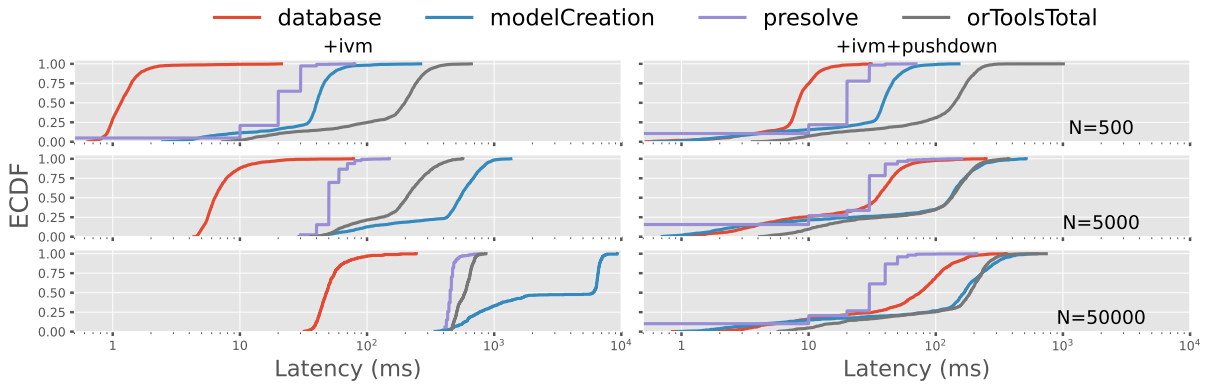

Figure 13: Scheduling latency breakdown per batch of pods (max 50). Breakdown includes time spent fetching updating data from the database (database), creating an optimization model from generated code (modelCreation), the presolve phase in or-tools (presolve), and the total solve time in or-tools (orToolsTotal) at $F = 100\%$ and different cluster sizes.

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 to keep up. This is because, in each scheduling iteration, DCM's queries to fetch the latest states are bottlenecked by top-down query evaluation, which performs redundant work even when only small changes happen to the cluster state between iterations. This cluster state database bottleneck is apparent in Figure 11, which reports the total time to fetch and update records from the database. Concretely, the 95 percentile database latency at 10× trace speed up decreases from over 5 seconds without IVM to 1.7ms with IVM, which is a near 3000× speed up.

With DDlog, however, scheduling throughput can keep up with the request arrival rate and the database latencies remain tractable (95 percentile latency around 3.5ms) even at 40× and higher trace speedups. Without additional optimizations like FP-pushdown, factors like the solver's performance become a bottleneck at a roughly 40x speedup of the overall trace (Figure 10). Figure 13 shows the relative contribution of the database to the overall scheduling latency compared to other steps with IVM. In the 500 node case, the p95 latency of fetching all the required cluster states from the database is under 2ms. In short, cluster state management via IVM using DDlog is not a bottleneck even with high arrival rates.

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

	# nodes = 500			# nodes = 5000			# nodes = 50000		
	F=0%	F=50%	F=100%	F=0%	F=50%	F=100%	F=0%	F=50%	F=100%
DCM	2000.00	2000.00	2000.00	20000.00	20000.00	20000.00	200000.00	200000.00	200000.00
+ FP-pushdown	599.90	1451.01	1693.32	596.92	2720.88	4640.93	583.86	2761.16	5573.45
Q_{inc}	0	1426.97	1678.65	0	2626.61	4601.21	0	2360.24	5308.06
Q_{exc}	0	.0037	0	0	.0092	.0061	0	.0060	.0081
Q_s	599.90	24.05	14.67	596.92	94.27	39.71	583.86	400.92	265.39

Table 1: The top half of the table shows the average number of rows in variable columns with and without the FP-pushdown optimization; the bottom half of the table shows the detailed breakdown of the row counts in the variable column contributed from inclusion (Q_{inc}), exclusion (Q_{exc}) and topk (Q_s) DRQs.

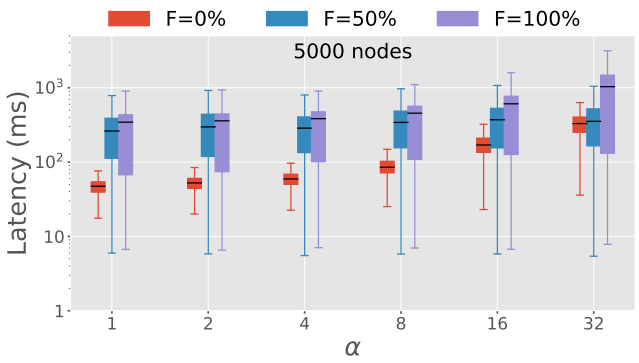

Figure 14: Impact of increasing k to the overall batch scheduling latency for 5000 nodes ($\alpha = k/\#NewPods$). Performance is stable for low values of α. At higher values of α, performance approaches the baseline version that considers all nodes per scheduling decision, thereby increasing scheduling latencies.

at 50. With $F = 50\%$ and $F = 100\%$, given the complex mix of constraints configured, we find that scheduling performance varies with the cluster size. With node affinities, a scheduling problem sometimes becomes straightforward with a small search space when the number of pods per decision is small. At the higher percentiles, we see larger problem sizes in cases with larger batches of pods per decision, each with a different group of nodes it is affine to.

Finally, as was our design goal, FP-pushdown did not affect the feasibility of the solver in any of the above test cases.

Latency breakdown. Figure 13 reports the detailed breakdown of scheduling latency at $F = 100\%$ and cluster sizes of 500, 5000 and 50000 nodes before and after the FP-pushdown optimization. Specifically, the breakdown includes the time to fetch data from the database (`database`), to encode the fetched data into an optimization problem (`modelCreation`), and the total time spent in the or-tools solver (`orToolsTotal`). We also highlight the time spent in the presolve phase in or-tools (`presolve`), where the solver 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 $30-84\%$ of the original in a 500-node cluster, $2.9-23\%$ in a 5000-node cluster and $0.29-2.7\%$ in a 50000-node cluster. Table 1 also breaks down the contribution from inclusion (Q_{inc}), exclusion (Q_{exc}) and topk (Q_s) 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_s). At $F = 50\%$ and $F = 100\%$, the contribution from Q_s 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 α, and plot the effect of increasing α 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 α (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 α (>=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 inflexibility of modern cluster manager designs. DCM [46] 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 [43] 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 [15, 18, 19, 21–23, 25, 27, 48, 49]. 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 [32] uses domain knowledge to partition the problem and use local search to assign shards in distributed applications. Wrasse [37] uses a balls-and-bins based abstraction to model allocation problems and uses a GPU-based solver to find assignments. DCM and C-SQL allows 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 [16] builds a network routing controller that is capable of incremental updates built on top of differential dataflow. Full-stack SDN [45] 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 [28, 29, 31, 40], albeit with a different focus on the representation and querying of spatial temporal data [17, 38, 47]. 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 `MAXIMIZE` clauses.

Predicate pushdown. Predicate pushdown [50] 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 [14, 24, 33]. Most query optimization techniques can not push down predicates below a join operator, unless the predicates are on columns used in the join condition [20, 36, 42, 51]. Predicate pushdown has been implemented in data analytics systems and data warehouses such as Apache Spark [10], Snowflake [1] and BigQuery [4] as a crucial performance feature to eliminate unneeded data partitions. More broadly, predicate pushdown style optimization has shown performance benefits in applications such as sensor networks [44], dataflow operators [26], video analytics systems [30] and machine learning inference queries [34]. 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

[1] [n.d.]. Micro-partitions and Data Clustering. https://docs.snowflake.com/en/user-guide/tables-clustering-micropartitions.html. Last accessed: September 28, 2022.

[2] [n.d.]. OpenShift. https://www.openshift.com/. Last accessed: April 2019.

[3] [n.d.]. OpenStack. https://www.openstack.org/. Last accessed: May 2016.

[4] [n.d.]. Query partitioned tables. https://cloud.google.com/bigquery/docs/querying-partitioned-tables. Last accessed: September 28, 2022.

[5] 2007. H2 Database. https://github.com/h2database/h2database/. Last accessed: Apr 30, 2020.

[6] 2010. Google OR-Tools. https://developers.google.com/optimization/. Last accessed: Apr 23, 2019.

[7] 2011. JOOQ. https://github.com/jOOQ/jOOQ. Last accessed: Apr 23, 2019.

[8] 2014. Kubernetes. http://github.com/kubernetes/kubernetes. Last accessed: Apr 5, 2016.

[9] 2015. Differential Dataflow. https://github.com/TimelyDataflow/differential-dataflow.

[10] 2019. Dynamic Partition Pruning in Apache Spark. https://www.databricks.com/session_eu19/dynamic-partition-pruning-in-apache-spark. Last accessed: September 28, 2022.

[11] 2021. Differential Datalog. github.com/vmware/differential-datalog.

[12] 2021. PresoDB. https://prestodb.io/.

[13] 2022. DDlog's SQL frontend and SQL-to-DDlog compiler. https://github.com/vmware/differential-datalog/tree/master/sql.

[14] Surajit Chaudhuri and Kyuseok Shim. 1999. Optimization of queries with user-defined predicates. *ACM Transactions on Database Systems (TODS)* 24, 2 (1999), 177–228.

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

[16] Desislava Dimitrova, John Liagouris, Sebastian Wicki, Moritz Hoffmann, Vasiliki Kalavri, and Timothy Roscoe. 2018. DeltaPath: dataflow-based high-performance incremental routing. https://doi.org/10.48550/ARXIV.1808.06893

[17] Martin Erwig, Markus Schneider, Michalis Vazirgiannis, et al. 1999. Spatio-temporal data types: An approach to modeling and querying moving objects in databases. *GeoInformatica* 3, 3 (1999), 269–296.

[18] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. 2018. Medea: Scheduling of Long Running Applications in Shared Production Clusters. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 4, 13 pages. https://doi.org/10.1145/3190508.3190549

[19] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. 2016. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 99–115. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gog

[20] Goetz Graefe. 1995. The cascades framework for query optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[21] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-Resource Packing for Cluster Schedulers. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) *(SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 455–466. https://doi.org/10.1145/2619239.2626334

[22] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic Scheduling in Multi-Resource Clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 65–80.

[23] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. Graphene: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 81–97.

[24] Joseph M Hellerstein and Michael Stonebraker. 1993. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*. 267–276.

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

[26] Fabian Hueske, Mathias Peters, Matthias J Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. 2012. Opening the Black Boxes in Data Flow Optimization. *Proceedings of the VLDB Endowment* 5, 11 (2012).

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

[28] Paris C Kanellakis, Gabriel M Kuper, and Peter Z Revesz. 1990. Constraint query languages (preliminary report). In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. 299–313.

[29] Paris C Kanellakis, Gabriel M Kuper, and Peter Z Revesz. 1995. Constraint query languages. *J. Comput. System Sci.* 51, 1 (1995), 26–52.

[30] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: optimizing neural network queries over video at scale. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1586–1597.

[31] Gabriel Kuper, Leonid Libkin, and Jan Paredaens. 2013. *Constraint databases*. Springer Science & Business Media.

[32] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. 2021. Shard Manager: A Generic Shard Management Framework for Geo-Distributed Applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 553–569. https://doi.org/10.1145/3477132.3483546

[33] Alon Y Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. 1994. Query optimization by predicate move-around. In *VLDB*. 96–107.

[34] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. 2018. Accelerating machine learning inference with probabilistic predicates. In *Proceedings of the 2018 International Conference on Management of Data*. 1493–1508.

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

[36] Inderpal Singh Mumick and Hamid Pirahesh. 1994. Implementation of magic-sets in a relational database system. *ACM SIGMOD Record* 23, 2 (1994), 103–114.

[37] Anshul Rai, Ranjita Bhagwan, and Saikat Guha. 2012. Generalized Resource Allocation for the Cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing* (San Jose, California) *(SoCC '12)*. ACM, New York, NY, USA, Article 15, 12 pages. https://doi.org/10.1145/2391229.2391244

[38] Philippe Rigaux, Michel Scholl, Luc Segoufin, and Stéphane Grumbach. 2003. Building a constraint-based spatial database system: model, languages, and implementation. *Information Systems* 28, 6 (2003), 563–595.

[39] Kexin Rong, Mihai Budiu, Athinagoras Skiadopoulos, Lalith Suresh, and Amy Tai. 2022. Scaling a Declarative Cluster Manager Architecture with Query Optimization Techniques (Technical Report). https://github.com/vmware/declarative-cluster-management/blob/vldb23/docs/tr.pdf.

[40] Francesca Rossi, Peter Van Beek, and Toby Walsh. 2006. *Handbook of constraint programming*. Elsevier.

[41] Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. In *Datalog 2.0*. Philadelphia, PA. http://budiu.info/work/ddlog.pdf

[42] Praveen Seshadri, Joseph M Hellerstein, Hamid Pirahesh, TY Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J Stuckey, and S Sudarshan. 1996. Cost-based optimization for magic: Algebra and implementation. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. 435–446.

[43] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael Cafarella, Vijay Gadepally, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. 2022. DBOS: A DBMS-Oriented Operating System. *Proc. VLDB Endow.* 15, 1 (jan 2022), 21–30. https://doi.org/10.14778/3485450.3485454

[44] Utkarsh Srivastava, Kamesh Munagala, and Jennifer Widom. 2005. Operator placement for in-network stream query processing. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 250–258.

[45] Debnil Sur, Ben Pfaff, Leonid Ryzhyk, and Mihai Budiu. 2022. Full-Stack SDN. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks* (Austin, Texas) *(HotNets '22)*. Association for Computing Machinery, New York, NY, USA, 130–137. https://doi.org/10.1145/3563766.3564101

[46] Lalith Suresh, João Loff, Faria Kalim, Sangeetha Abdu Jyothi, Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, and Michael Gasch. 2020. Building Scalable and Flexible Cluster Managers Using Declarative Programming. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 827–844.

[47] David Toman and Jan Chomicki. 1998. Datalog with integer periodicity constraints. *The Journal of Logic Programming* 35, 3 (1998), 263–290.

[48] Alexey Tumanov, James Cipar, Gregory R. Ganger, and Michael A. Kozuch. 2012. Alsched: Algebraic Scheduling of Mixed Workloads in Heterogeneous Clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing* (San Jose, California) *(SoCC '12)*. Association for Computing Machinery, New York, NY, USA, Article 25, 7 pages. https://doi.org/10.1145/2391229.2391254

[49] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. 2016. TetriSched: Global Rescheduling with Adaptive Plan-ahead in Dynamic Heterogeneous Clusters. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (London, United Kingdom) *(EuroSys '16)*. ACM, New York, NY, USA, Article 35, 16 pages. https://doi.org/10.1145/2901318.2901355

[50] Jeffrey D. Ullman. 1989. *Principles of Database and Knowledge-Base Systems, Volume II.* Computer Science Press.
[51] Brett Walenz, Sudeepa Roy, and Jun Yang. 2017. Optimizing iceberg queries with complex joins. In *Proceedings of the 2017 ACM International Conference on Management of Data.* 1243–1258.

Appendices

A 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:

```
<formula> ::= Constant
| VariableName
| <formula> <binaryOperation> <formula>
| <unaryOperation> <formula>
| <aggregate> ( <formulaList> )

<formulaList> ::= <formula>
| <formula> , <formulaList>

<binaryOperation> ::=  AND | OR  | = | !=  | >
 | >= | < | <= | + | - | * | / | %
<unaryOperation> ::= - | NOT
<aggregate> ::= ANY | ALL | SUM | COUNT | MIN | MAX
```

`<formula>`s are typed, with typing rules following standard mathematical expressions (we omit them from this presentation). A `Constant` with a base type T, when used in a formula has type $F(T)$. For example, 5 has type $\mathbb{N}$, but when used within a formula has type $F(\mathbb{N})$. Each `VariableName` also has a type $F(T)$ for some base type, which is derived from the context, as we explain below. Each binary or unary operation requires arguments of some appropriate types, e.g., addition requires operands of type $F(\mathbb{N})$ or $F(\mathbb{R})$ and produces a result of the same type as both operands.

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 $\{\texttt{col0} \mapsto 5, \texttt{col1} \mapsto 2\}$ to denote a row with value 5 in column `col0` and value 2 in column `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 `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 constrast to SQL, where the semantics of a relation is a (multi)set 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:

```
<expr> ::= <expr> <binaryOperation> <expr>
        | <unaryOperation> (<expr>)
        | ColumnName
        | 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 $[\![e]\!](r)$.

$$[\![\texttt{Const}]\!](r) = \texttt{Const}$$
$$[\![\texttt{ColName}]\!](r) = r(\texttt{ColName})$$
$$[\![expr1\ binOp\ expr2]\!](r) = [\![expr1]\!](r)\ binOp\ [\![expr2]\!](r)$$
$$[\![unOp\ expr]\!](r) = unOp\ [\![expr]\!](r)$$

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

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

A.5 Semantics of row expressions

Consider a SELECT statement such as `SELECT e1 AS col1, e2 AS 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:

```
<simpleRowExpr> ::= <expr> AS colName
<rowExpr> ::= <simpleRowExpr>
        | <rowExpr>, <simpleRowExpr>
```

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

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

$$[\![rowExpr]\!](r) = \uplus_{e \in rowExpr} [\![e]\!](r).$$

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:

$$[\![e]\!](R) = \cup_{r \in R}\{[\![e]\!](r)\}.$$

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:

```
<aggregateExpr> ::= <aggregate> ( <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:

$$[\![agg(e)]\!](R) = agg([\![e]\!](R)).$$

For example, consider the expression `SUM(col)` evaluated over a relation where `col` is a column of type $F(\mathbb{N})$ with the following rows: $\{\texttt{col} \mapsto 1 + \texttt{var1}, \texttt{col} \mapsto \texttt{var2} + \texttt{var1}\}$. This expression evaluates to a formula with type $F(\mathbb{N})$, which is `SUM(1 + var1, var1 + 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:

$$[\![rowExp]\!](R) = \cup_{r \in R}[\![rowExp]\!](r).$$

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.

```
<relation> ::=  SELECT <rowExpr>
                FROM ( <tableRef> | <join> )
                [ WHERE <baseExpr> ]
                [ GROUP BY baseColumnList ]
                [ HAVING <baseExpr> ]

<join> ::= <relation> JOIN <relation> ON <baseExpr>
```

In this grammar, we denote any `<expr>` that evaluates to a base type by `<baseExpr>` , and a list of columns that all have base types by `<baseColumnList>` [1]. 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 `<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: $[\![T]\!](DB) = \cup_{r \in T}\{[\![r]\!](r)\}$. Recall that a table can only contain base type values or variable names. Consider the following table with schema ($\texttt{col0} : \mathbb{N}, \texttt{col1} : F(\mathbb{N})$):

$$\{\{\texttt{col0} \mapsto 0, \texttt{col1} \mapsto var1\},$$
$$\{\texttt{col0} \mapsto 2, \texttt{col1} \mapsto var2\}\}$$

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

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

$$[\![\texttt{CREATE VIEW V AS } Q]\!](DB) = [\![Q]\!](DB).$$

The semantics of filtering a relation R by a predicate $expr$ (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:

$$[\![\sigma_{expr}]\!](R) = \{r \mid [\![expr]\!](r) = true, r \in R\}.$$

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:

$$[\![\texttt{GROUP BY } col]\!](R) =$$
$$\cup_{g \in R}\{g[col] \mapsto \{r \mid r \in R \wedge g[col] = r[col]\}\}$$

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:

$$[\![R1 \times R2]\!](DB) = \{r1 \uplus r2 \mid r1 \in [\![R1]\!](R1), r2 \in [\![R2]\!](R2)\}.$$

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:

```
<problem> ::= <constraint> [ , <constraint> ]*
<constraint> ::= CREATE CONSTRAINT <identifier> AS
<checkOrOptimize> FROM <relation>

<checkOrOptimize> ::= CHECK <expr>
| CHECK <setConstraint>
| MAXIMIZE <expr>

<setConstraint> ::= AllDifferent ( <expr> )
 | AllEqual ( <expr> )
 | Increasing ( <expr> )
```

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.

[1] These are, in fact, semantic checks enforced by the type-checker, and not by the grammar.

A CHECK statement must be followed by an expression of type $F(\mathbb{B})$. 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*.

$$[\![\texttt{CHECK } expr]\!](r) = [\![expr]\!](r).$$

For example, the formula `var3 + 2 = 5` generates the constraint `var3 + 2 = 5`, which has a unique solution of 3 for `var3`. Applying the statement to a relation generates the *conjunction* of the constraints for all rows:

$$[\![\texttt{CHECK } expr]\!](R) = \wedge_{r \in R}[\![\texttt{CHECK } expr]\!](r).$$

$[\![\texttt{CHECK AllEqual}(expr)]\!](R) = AllEqual\{[\![expr]\!](r) | r \in R\}$, where $AllEqual(S) = \{s_i = s_j | s_i \in S, s_j \in S\}$.

Similarly, `AllDifferent` requires all formulas to evaluate to different values.

A `MAXIMIZE` statement takes an arbitrary expression of type $F(\mathbb{N})$ or $F(\mathbb{R})$ 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:

$$[\![constraint]\!](DB) \in (2^{F(\mathbb{B})}, F(\mathbb{N}))$$

, 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:

$$[\![problem]\!](DB) = combine_{(}c \in problem[\![c]\!](DB))$$

where the *combine* function is defined as:

$$combine(\{(c_i, o_i)\}) = (\cup_i c_i, \sum_i o_i)$$

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` 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 tolerations, 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

```
CREATE VIEW inter_pod_aa_matching_pods AS
(SELECT * FROM pods_to_assign
  JOIN pod_aa_match_expressions on pods_to_assign.pod_name =
       pod_aa_match_expressions.pod_name
  JOIN pod_labels
    ON   pod_aa_match_expressions.label_operator = 'Exists'
     AND pod_aa_match_expressions.label_key = pod_labels.label_key
  JOIN pod_info on pod_labels.pod_name = pod_info.pod_name
  WHERE pods_to_assign.has_pod_aa_requirements = true)
UNION
(SELECT * FROM pods_to_assign
  JOIN pod_aa_match_expressions
       ON pods_to_assign.pod_name = pod_aa_match_expressions.pod_name
  JOIN pod_labels
    ON pod_aa_match_expressions.label_operator = 'IN'
       AND pod_aa_match_expressions.label_key = pod_labels.label_key
       AND pod_labels.label_value = pod_aa_match_expressions.label_value
  JOIN pod_info on pod_labels.pod_name = pod_info.pod_name
  WHERE pods_to_assign.has_pod_aa_requirements = true)
```

Figure 15: Evaluating anti-affinity matches without DDlog. This matching logic has to be repeated for every policy that uses Kubernetes' label-based matching rules, preventing reuse. It is also not fully incremental as tables such as `pod_labels` and `pod_info` gets scanned depending on their contents, even with carefully designed indexes.

```
CREATE VIEW matching_pods AS
(SELECT DISTINCT expr_id, pod_uid
 FROM (SELECT DISTINCT * FROM match_expressions
       WHERE match_expressions.label_operator = 'In') me
 JOIN pod_labels
    ON me.label_key = pod_labels.label_key
       AND me.label_value = pod_labels.label_value)
UNION
(SELECT DISTINCT expr_id, pod_uid
FROM (SELECT DISTINCT * FROM match_expressions
      WHERE match_expressions.label_operator = 'Exists') me
JOIN pod_labels
    ON me.label_key = pod_labels.label_key)
```

Figure 16: Generic pod label matching view with DDlog, used by several views such as the anti-affinity matches view. Note the absence of joins specific to the anti-affinity logic or the set of pods to assign.

```
-- @VARIABLE_COLUMNS (node_name)
uid  | has_port_req | qos_class  | node_name
pod1 | false        | BestEffort | ?
pod2 | true         | Guaranteed | ?
pod3 | false        | Burstable  | ?
```

Figure 17: Base table `pods_to_assign` with one variable column `node_name` and two non-variable columns `has_port_req` and `qos_class`.

appear in a single corresponding downstream view per scheduling policy (like the inter-pod affinity matches view in Figure 6). Being incrementally updated and shared across multiple downstream views thereby not only improves performance but also maintainability.

C GROUP-LEVEL DRQS

```
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.

Depending on whether different rows in the variable column are treated as a single decision variable with a shared domain or independent decision variables with different constraints, we can generate one or more domain restricting views for each table. Specifically, there are three levels of granularity at which we can compute domain restricting views to restrict variable domains:

(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 row group $g_i = \gamma_{c_1,...,c_k}(T)$ of rows, where $c_1, c_2, ..., c_k$ are defined by a set of input (non-variable) columns on the same table.
(3) Row-level: generating one domain restricting view per row in the variable column.

In this section, we discuss how to extend column-level domain restricting views to the group-level. Note that row-level views are a degenerate case of group-level views, where each row defines a different group.

View generation at the group level is similar to the column-level except it requires an additional preprocessing step that partitions the variable column and the constraint set by the input (non-variable) columns $c1, c2, ..., c_k$. Given each partition of the variable column and constraint set, one can use the process described above to generate the corresponding domain restricting views for each row group.

The main idea for group-level DRQ is that the additional grouping information introduced by these non-variable columns can be used to eliminate or to extract additional DRQs for each group. For the purpose of refining DRQs, only the subset of all non-variable columns that appear in the constraints and group-by clauses need to be considered.

We illustrate this idea using our running example in Figure 17 and Figure 18. Here, both non-variable columns `qos_class` and `has_port_req` are relevant for refining, 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 further refine the DRQs by explicitly evaluating predicates involving the non variable columns using the grouping information. Specifically in `port_constraint`, the 3 groups defined by `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 restricting conditions for different row groups.