Scaling a Declarative Cluster Manager Architecture with Query Optimization Techniques

Kexin Rong
Georgia Institute of Technology
VMware Research
krong@gatech.edu

Mihai Budiu
Feldera*
mbudiu@feldera.com

Athinagoras Skiadopoulos
Stanford University
askiad@stanford.edu

Lalith Suresh
Feldera*
lalith@feldera.com

Amy Tai
Google
amy.tai.2009@gmail.com

ABSTRACT

Cluster managers play a crucial role in data centers by distributing workloads among infrastructure resources. Declarative Cluster Management (DCM) is a new cluster management architecture that enables users to express placement policies declaratively using SQL-like queries. This paper presents our experiences in scaling this architecture from moderate-sized enterprise clusters ($10^2 - 10^3$ nodes) to hyperscale clusters (10^4 nodes) via query optimization techniques. First, we formally specify the syntax and semantics of DCM's declarative language, C-SQL, a SQL variant used to express constraint optimization problems. We showcase how constraints on the desired state of the cluster system can be succinctly represented as C-SQL programs, and how query optimization techniques like incremental view maintenance and predicate pushdown can enhance the execution of C-SQL programs. We evaluate the effectiveness of our optimizations through a case study of building Kubernetes schedulers using C-SQL. Our optimizations demonstrated an almost 3000× speed up in database latency and reduced the size of optimization problems by as much as 1/300 of the original, without affecting the quality of the scheduling solutions.

PVLDB Reference Format:
Kexin Rong, Mihai Budiu, Athinagoras Skiadopoulos, Lalith Suresh, and Amy Tai. Scaling a Declarative Cluster Manager Architecture with Query Optimization Techniques. PVLDB, 16(10): 2618-2631, 2023.
doi:10.14778/3603581.3603599

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/vmware/declarative-cluster-management/releases/tag/vldb23.

1 INTRODUCTION

Cluster managers like Kubernetes [8], OpenStack [2], and OpenShift [3] are important building blocks of today's data centers. They dynamically assign workloads to the underlying infrastructure and configure them according to a variety of policies. Some policies represent *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 preferences and decision quality. Cluster management logic is notoriously hard to develop [72], since it often involves NP-hard combinatorial optimization tasks that cannot be efficiently solved using best-effort heuristics, as is the norm today.

Declarative Cluster Managers (DCM) [72] is a novel architecture that allows developers to specify *what* the cluster manager should achieve, not *how* it should do so. DCM users can declaratively specify constraints over cluster states stored in a relational database. Using current cluster state and constraints, the DCM runtime constructs an optimization problem that can be solved using off-the-shelf constraint solvers (Figure 1). Compared to existing solutions that rely on ad-hoc, imperative code, DCM's declarative approach significantly reduces the development and maintenance efforts for complex cluster management logic. DCM also improves scheduling performance by leveraging constraint solvers that scale to larger problem sizes compared to brittle, handcrafted heuristics.

Driven by the needs of hyperscale operators who are interested in DCM, we have embarked on a journey to scale DCM from moderate-sized enterprise clusters ($10^2 – 10^3$ nodes) to hyperscale clusters (10^4 nodes). Our key insight for scalability is to leverage the incremental cluster state evolution: changes in cluster state are often much smaller than the size of the whole database; doing work proportional

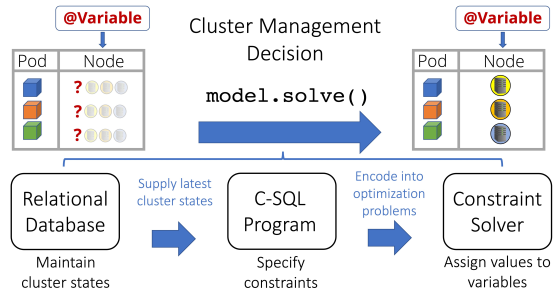

Figure 1: During cluster management decisions (e.g., assigning pods to nodes in Kubernetes), DCM retrieves the latest cluster state, encodes the state and constraints into an optimization problem, solves it using a constraint solver, and returns the tables with values assigned to variable columns.

*Work done while at VMware Research.

Proceedings of the VLDB Endowment, Vol. 16, No. 10 ISSN 2150-8097.
doi:10.14778/3603581.3603599

to the size of the changes instead of the size of the database can provide substantial benefits. Achieving this design goal turns out to be challenging and necessitates novel techniques at the intersection of constraint optimization and database query optimization.

First, we formally define DCM's query language C-SQL, which allows users to declaratively specify constraint queries against relations. While C-SQL is syntactically similar to SQL, it semantically represents optimization problems. C-SQL programs are evaluated in two phases: (1) *relation evaluation*, wherein the DCM runtime retrieves cluster state required for the optimization problem; and (2) *constraint evaluation*, wherein the runtime encodes the cluster state into constraint formulae to be solved by a standard solver. This new formalism allows us to explore optimization opportunities through the lens of *query optimization* for C-SQL programs.

To speed up relation evaluation, we integrate DCM with an incremental query engine, Differential Datalog (DDlog) [10, 67]. Previously, DCM relied on a traditional in-memory embedded database (H2 [4]), which forced users to write unnatural, manually incrementalized C-SQL queries for performance. For example, to *manually simulate* incremental view maintenance, users had to carefully design the schema by splitting static and dynamic parts of the cluster states into separate tables. Instead, we leverage the incremental engine of DDlog to automatically incrementalize computations. With DDlog, users can express constraint queries without worrying about performance, embodying a truly declarative programming model. Integrating DDlog with DCM was a significant engineering undertaking, requiring 11K lines of code; we elaborate on the challenges faced and lessons learned in §5.

To speed-up constraint evaluation, we introduce Feasibility-Preserving Predicate Pushdown (FP-pushdown) for automatically reducing optimization problem size in C-SQL programs without altering feasible solutions. Constraint solvers simplify a given optimization problem in a "presolve" phase before they start searching for solutions, by eliminating unnecessary variables and restricting variable domains. Inspired by the predicate pushdown technique, FP-pushdown pushes down relevant constraints from the solver's presolve phase to the relation-evaluation phase of the DCM processing pipeline, restricting the search space in advance. This is achieved by statically analyzing C-SQL programs, symbolically inferring variable domains, and generating predicates that can be pushed down to the (incremental) relation evaluation. Although FP-pushdown does not interact with runtime cluster state, it offers substantial performance benefits. For instance, the optimization reduced a 50K-node Kubernetes cluster placement problem to an equivalent one with fewer than 1.4K nodes, cutting 95th percentile scheduling latency from 8 seconds to 80-800 milliseconds.

In summary, this work makes the following contributions:

(1) We report on experiences scaling up a new architecture for declarative cluster management. We discuss our central design goal of making the architecture's processing pipeline incremental as well as the challenges of achieving this goal.

(2) We provide a formal specification of DCM's declarative constraint language C-SQL, highlighting its similarities and differences with respect to standard SQL.

(3) We introduce two optimization techniques, based on incremental view maintenance and predicate pushdown, to improve the execution efficiency of C-SQL programs.

(4) Through a case study of building Kubernetes schedulers using C-SQL, we demonstrate that our optimizations enabled a nearly 3000× improvement in latency for fetching and updating the cluster state database, and reduced the size of optimization problems down to 0.29-2.7% of the original without affecting feasibility.

2 CLUSTER MANAGEMENT BACKGROUND

In this section, we explain the context in which DCM was designed. 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 describe how DCM's declarative approach significantly improves the programmability and scalability compared to the status-quo (§2.2).

2.1 Heuristic-based Cluster Management

We use the example of building a Kubernetes scheduler [13] throughout this paper. Note that neither the problems nor the solutions presented are unique to scheduling in Kubernetes. Similar challenges recur across various distributed systems, such as for policy-based configuration, data replication, and load-balancing across machines, all of which can be reduced to combinatorial optimization problems.

Policies. The Kubernetes scheduler assigns *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 types of 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. 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 represent a view of the relevant cluster state. For example, in Kubernetes [8], a centralized, persistent data store hosts all 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., to reason about existing pod, node and volume arrangements simultaneously). 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 keep track of the spare resource capacity of each node in the cluster. The scheduler also maintains ad-hoc auxiliary data structures to cache prior decisions [13, 32, 79] to make coherent *future* decisions without having to scan the cluster state repeatedly. These types of precomputing and caching optimizations are necessary to keep scheduler performance tractable.

Challenges with the filter-score architecture. Today's cluster managers such as Kubernetes [13], Borg [74, 79], Twine [73], Protean [32] and vSphere [29] 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) to place a given pod. For each pair of pod p and node n, the pipeline first evaluates policies with hard constraints, each of which is implemented as a single function that decides whether or not p can be placed on n. This is the filter phase. If p survives all filtering policies, it proceeds to the scoring phase, where the pipeline evaluates each soft constraint and outputs a score for a given node. After processing all candidate nodes, the pod is assigned to the highest ranked node. There are two key issues with existing designs:

- Over time, the sprawl of ad-hoc data structures in the cluster state representation degrades maintainability. A common issue is that the custom data structures used for precomputing and caching optimizations are brittle when requirements evolve [42, 43, 45, 47, 48, 51]. Adding new features and policies thereby becomes hard and requires significant refactoring.
- Implementing complex policies efficiently at scale is challenging [44, 49, 55]. To maintain tractable scheduling performance, sampling is often employed, where a small subset of nodes are considered for each scheduling decision. However, as each policy is written in imperative code, sampling cannot be easily implemented in a policy-aware manner and can therefore miss feasible solutions.

2.2 Declarative Cluster Managers with DCM

To overcome the programmability and scalability challenges in existing designs, recent work introduced the idea of a *declarative* cluster manager (DCM [72]). DCM replaces the imperative, filter-score pipeline of a heuristic-based cluster manager with a declarative one using SQL-like syntax, where developers specify *what* the cluster manager should achieve, not *how*.

A DCM-powered scheduler uses a relational database model, instead of ad-hoc data structures, to represent cluster states. It uses a declarative query language, instead of hand-crafted heuristics, to specify policies for the cluster manager. The policies are defined using variables and constraints. Variables represent cluster properties that need to be decided (e.g., machine allocated for each service), and constraints express the space of legal solutions (e.g., no two services should execute on the same machine). A compiler transforms the constraints and the database schema to generate an *encoder program*. At runtime, the encoder is invoked when cluster state changes and generates an optimization problem by combining the constraints and the current state. A constraint solver produces a solution for the optimization problem, which is then converted into decisions by DCM (e.g., where to schedule each process). The workflow is illustrated in Figure 1.

3 C-SQL DESIGN AND EXECUTION

We formalize the declarative language of DCM, C-SQL, for the first time in this paper. This section discusses C-SQL's semantics (§3.1), compilation and evaluation (§3.2), and how it is used in cluster management problems (§3.3).

Overview. C-SQL is an extension of standard SQL for specifying constraint optimization problems using variables and constraints. In the data model, besides standard SQL column types, C-SQL tables can also have *columns of variables*. Each cell in a column of variables contains a single variable name. When translating C-SQL queries to optimization problems, variables will translate into decision variables for the constraint solver, while traditional table columns will translate to constants in the optimization formulas. Base tables can contain both variable and constant columns.

A C-SQL program comprises *constraint queries* that express constraints over variable columns. A constraint query is a query with a `CHECK` or `MAXIMIZE` clause, syntactically similar to `SELECT` clauses. `CHECK` represents hard constraints, while `MAXIMIZE` represents an optimization function that we wish to maximize. `CHECK` clauses require a boolean expression and `MAXIMIZE` clauses accept a numeric expression[1]. The C-SQL program composed of multiple such queries enforces the conjunction of all `CHECK` clauses and maximizes the *sum* of all `MAXIMIZE` expressions[2]. Queries are constructed using regular SQL syntax: `SELECT`, `JOIN`, `WHERE`, `GROUP BY`, and aggregates. A query evaluates to a symbolic formula.

3.1 C-SQL Syntax and Semantics

The syntax of C-SQL is given in Figure 2 and Figure 3. The language semantics is given in Figure 4. C-SQL extends SQL to express *symbolic formulae* over variables (e.g. ,"x + 2 < 5", where "x" is a variable). These formulae become inputs to constraint solvers, which assign values to variables to satisfy constraints. For ease of use, C-SQL builds on SQL's syntax and semantics with respect to types, relations and expression evaluation.

Types and Formulas. We assume the standard SQL base types, including $\mathbb{N}$ (integers), $\mathbb{B}$ (Booleans), $\mathbb{R}$ (reals). For each base type T we introduce a new type, $F(T)$, which is the type of *symbolic formulas* with type T. Formulas are syntactic objects defined by the grammar in Figure 2 (left). All formulas are statically typed. A `Constant` with a base type T has type $F(T)$ when used in a formula. 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 T. For example, if `x` is a `VariableName` with type $F(\mathbb{N})$, "`x + 5 < 2`" is a formula with type $F(\mathbb{B})$. Each binary or unary operation requires arguments of some appropriate types, e.g., addition requires operands of type $F(\mathbb{N})$ and produces a result $F(\mathbb{N})$.

Relations. We use the term "relation" to refer to database tables, views, and queries. A database maps identifiers (table and view names) to relations (table and view contents). We denote the contents of table T in the database as $DB[T]$. As in SQL, all relations are statically typed, with types that can be inferred by a simple syntax-directed analysis. All values in a column have the same type, which is either specified by the database schema or inferred from the query or view definition. A relation is a multiset of rows. A row of a relation is a function that maps each column name to a value of the corresponding type. For example, $\{$`col0` $\mapsto 5\}$ is a row with a single column named `col0` and a value of 5 for this column. A column in a relation can either be a base type T or a formula type

[1] Booleans are coerced into integers 0 and 1 when used in numeric expressions.
[2] `MINIMIZE` clauses are syntactic sugar for maximizing the negation of a formula.

```
<formula> ::= Constant | VariableName
          | <formula> <binOp> <formula>
          | <unOp> <formula>
          | <aggregate> ( <formulaList> )
<formulaList> ::= <formula>
          | <formula>,  <formulaList>
<binOp> ::=  AND|OR|=|!=|>|>=|<|<=|+|-|*|/|%
<unOp> ::= - | NOT
<aggregate> ::= ANY|ALL|SUM|COUNT|MIN|MAX
```

```
<expr> ::= <expr> <binOp> <expr>
  | <unOp> ( <expr> )
  | ColumnName
  | Constant
  | <expr> IS NULL
  | <expr> IS NOT NULL
<simpleRowExpr> ::= <expr> AS colName
<rowExpr> ::= <simpleRowExpr>
        | <rowExpr>, <simpleRowExpr>
<aggregateExpr> ::= <aggregate> ( <expr> )
```

```
  <view> ::= CREATE VIEW identifie
             AS relation
  <idOrRel> ::= identifier | <relation>
  <relation> ::=
    SELECT [DISTINCT] <rowExpr>
    FROM ( <join> | <idOrRel> )
    [ WHERE <baseExpr> ]
    [ GROUP BY baseColumnList ]
    [ HAVING <baseExpr> ]
<join> ::=
    <idOrRel> JOIN <idOrRel>>
    ON <baseExpr>
```

Figure 2: C-SQL syntax: formulas, expressions, queries. Angle brackets denote non-terminals.

```
<problem> ::= <constraint> [ , <constraint> ]*
<constraint> ::= CREATE CONSTRAINT identifier AS
                 <checkOrOptimize> FROM <relation>
<checkOrOptimize> ::= CHECK <expr>
                    | CHECK <setConstraint>
                    | MAXIMIZE <expr>
<setConstraint> ::= AllDifferent ( <expr> ) | AllEqual ( <expr> )
```

Figure 3: C-SQL syntax: constraints.

$F(T)$. For tables, all values in a column of type $F(T)$ are required to hold `VariableName` values; more complex expressions are not allowed in tables. For example, the table in Figure 5 has one column `node_name` that contains variables; the type of the column is $F($`varchar(100)`$)$. 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. Similarly, the semantics of a relation is a multiset of tuples, with the extension that some columns describe symbolic formulae. The semantics of the `CREATE VIEW` statement, shown in Figure 4(*view*), is to extend the database DB by adding a new view. We do not allow recursive view definitions, so this formula is well-defined.

Expressions. As in SQL, expressions can appear in `SELECT`, `WHERE`, `GROUP BY`, `JOIN`, and `HAVING` statements. The grammar of our core language of expressions, row expressions, and aggregate expressions is shown in Figure 2 (middle). Note that some non-terminals that appear in this grammar are the same as the non-terminals in the grammar of formulas (e.g, `<binOp>`, `<aggregate>`). This is not an accident: formulas will be built from expressions that appear in C-SQL queries. As in SQL, an expression e can only be evaluated in the context of a row r; we denote " the semantics of expression e evaluated for the row r" as $[\![e]\!](r)$. An expression in a context evaluates to a symbolic formula if any of its arguments is a formula; otherwise it evaluates to a constant. Figure 4 describes the semantics of expressions.

Example. Consider the expression "1 + `Age`", where `Age` is a column name. When applied to a row with an integer column `Age`, e.g., $r = \{\texttt{Age} \mapsto 10\}$, this expression evaluates to an integer with value 11. However, when applied to a row where `Age` is a symbolic integer formula, e.g. ,$r = \{\texttt{Age} \mapsto \texttt{x} + 2\}$, the expression evaluates to the integer formula $1 + (\texttt{x} + 2)$.

Expressions over relations and aggregates. As in SQL, evaluating an expression in the context of a relation R evaluates the expression for each row of the relation and takes the union of the results (equations (*eR*) and (*rowRel*) from Figure 4). Also similar to SQL, aggregate expressions can only be evaluated in the context of a collection of values, such as those in a table, query, view, or a group produced by a group-by clause. Similar to SQL, the semantics of an aggregate is a corresponding formula involving the aggregate, as shown in Equation (*agg*).

Semantics of Relations. Recall that relations can be tables, queries, or views (defined by queries). We define the semantics of a relation inductively on the defining query structure. We assume that syntactic sugar has been eliminated using standard query plan optimization techniques [16]: converting `IN` queries and correlated subqueries into joins, converting joins into Cartesian products followed by filtering, and converting `HAVING` into queries followed by `WHERE`. Queries that operate on relations containing base types can use the full SQL language with no restrictions. However, queries that operate on formulas are restricted, in the sense that some columns are not allowed to contain formulas; the simplified grammar in Figure 2 (right) describes valid queries when some input relations have at least one column that is a symbolic formula.

In this grammar, we denote any `<expr>` that evaluates to a base type (instead of a formula type) by `<baseExpr>`, and a list of columns that all have base types by `<baseColumnList>` [3].

Notice that we do **not** allow filtering, joining, or grouping by expressions whose type is a symbolic formula: all these clauses require a `<baseExpr>` as argument. This restriction ensures that the constraints generated are tractable. Despite these constraints, the resulting language is powerful enough to express all the constraints required by our application domain.

The semantics of a `<relation>` is always evaluated in the context of a database DB. The semantics of tables, views, filters, group-by and joins is exactly the same as in SQL; reusing the SQL syntax makes the language familiar to users. Formula types are not permitted for filter, group by or join expressions, which allows them to be directly evaluated by the database.

Generating constraints. Recall that our goal is to use C-SQL to generate constraints. So far we have reused SQL to create a language with symbolic formulas. We now extend SQL syntax to create constraints from formulas, as shown in Figure 3.

The semantics of a constraint is composed by a *pair* of symbolic formulas with type $\langle F(\mathbb{B}), F(\mathbb{R})\rangle$: the first element is a Boolean formula representing a hard constraint that must be satisfied; the second element is an optimization function whose value must be maximized. The two formulas generally share variables.

[3]These are semantic checks enforced by the type-checker, not by the grammar. The semantics of <baseExpr> is given by the semantics rules for <expr>.

$$
\begin{aligned}
\llbracket \texttt{Const} \rrbracket(r) &= \texttt{Const} && (const)\\
\llbracket \texttt{ColName} \rrbracket(r) &= r(\texttt{ColName}) && (col)\\
\llbracket e1 \,\texttt{<binOp>}\, e2 \rrbracket(r) &= \llbracket e1 \rrbracket(r) \,\texttt{<binOp>}\, \llbracket e2 \rrbracket(r) && (binop)\\
\llbracket \texttt{<unOp>}\, expr \rrbracket(r) &= \texttt{<unOp>}\, \llbracket expr \rrbracket(r) && (unop)\\
\llbracket e \texttt{ IS NULL} \rrbracket(r) &= \llbracket expr \rrbracket(r) = \texttt{NULL} &&\\
\llbracket e \texttt{ IS NOT NULL} \rrbracket(r) &= \llbracket expr \rrbracket(r) \neq \texttt{NULL} &&\\
\llbracket e \texttt{ AS colName} \rrbracket(r) &= \{\texttt{colName} \mapsto \llbracket e \rrbracket(r)\} && (name)\\
\llbracket rowExpr \rrbracket(r) &= concat_{e \in rowExpr}(\llbracket e \rrbracket(r) && (row)\\
\llbracket e \rrbracket(R) &= \cup_{r \in R}\{\llbracket e \rrbracket(r)\} && (eR)\\
\llbracket \texttt{<agg>}(e) \rrbracket(R) &= \texttt{<agg>}(\llbracket e \rrbracket(R)) && (agg)\\
\llbracket rowExp \rrbracket(R) &= \cup_{r \in R}\llbracket rowExp \rrbracket(r) && (rowRel)
\end{aligned}
$$

$$
\begin{aligned}
\llbracket \texttt{identifier} \rrbracket(DB) &= \cup_{r \in DB[\texttt{identifier}]}\{\llbracket r \rrbracket(r)\} && (identifier)\\
\llbracket \texttt{CREATE VIEW V AS } Q \rrbracket(DB) &= DB := DB \uplus \{\texttt{V} \mapsto \llbracket Q \rrbracket(DB)\} && (view)\\
\llbracket \texttt{SELECT } r \texttt{ FROM } R \rrbracket(DB) &= \llbracket r \rrbracket(\llbracket R \rrbracket(DB)) && (select)\\
\llbracket \texttt{WHERE } e \rrbracket(R) &= \{r \mid \llbracket e \rrbracket(r) = true, r \in R\} && (filter)\\
\llbracket \texttt{GROUP BY } col \rrbracket(R) &= \cup_{g \in R}\{g[col] \mapsto \{r \mid r \in R \wedge g[col] = r[col]\}\} && (group)\\
\llbracket R1 \times R2 \rrbracket(DB) &= \{concat(s,t) \mid s \in \llbracket R1 \rrbracket(DB), t \in \llbracket R2 \rrbracket(DB)\} && (join)\\
\llbracket \texttt{CHECK } e \rrbracket(r) &= \langle \llbracket e \rrbracket(r), 0 \rangle && (checkrow)\\
\llbracket \texttt{CHECK AllEqual}(e) \rrbracket(R) &= \langle AllEq\{\llbracket e \rrbracket(r) | r \in R\}, 0 \rangle && (alleq)\\
AllEq(S) &= \{\wedge_{i,j} s_i = s_j | s_i \in S, s_j \in S\} &&\\
\llbracket \texttt{CHECK AllDifferent}(e) \rrbracket(R) &= \langle AllDiff\{\llbracket e \rrbracket(r) | r \in R\}, 0 \rangle && (alldiff)\\
AllDiff(S) &= (\{\wedge_{i \neq j} s_i \neq s_j | s_i \in S, s_j \in S\} &&\\
\llbracket \texttt{MAXIMIZE}(e) \rrbracket(R) &= \langle true, \textstyle\sum_{r \in R} \llbracket e \rrbracket(r) \rangle && (maximize)\\
\llbracket \texttt{CHECK } expr \rrbracket(R) &= \langle \wedge_{r \in R} \llbracket expr \rrbracket(r), 0 \rangle && (check)\\
\llbracket problem \rrbracket(DB) &= combine(\{\llbracket c \rrbracket(DB), c \in problem\}) && (problem)\\
combine(\{\langle c_i, o_i \rangle\}_i) &= \langle \wedge_i c_i, \textstyle\sum_i o_i \rangle &&
\end{aligned}
$$

Figure 4: Semantics of C-SQL. Left: semantics of expressions. Right: semantics of relations. r is a row, R is a relation.

Check. 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. The semantics is given in Equation (*checkrow*): it generates a Boolean symbolic formula and a 0 value for the optimization function. Solving the constraint will provide values for all involved symbolic variables. For example, the formula `CHECK var+2=5` generates a constraint that has a unique solution `var=3`.

Applying the CHECK statement to a relation generates the *conjunction* of the constraints for all rows, as given in Equation (*check*).

Maximize. A MAXIMIZE statement generates an optimization function f, denoted as the pair (true, f), which consists of the satisfied Boolean constraint "true" and the optimization function f. The statement takes an arbitrary expression of type $F(\mathbb{N})$ or $F(\mathbb{R})$ and reinterprets this formula as an *optimization function* whose value must be maximized. A single formula is generated for the entire program, and multiple formulas can be combined using weights to signify their importance, in line with common constraint solver practices [6]. It is important to note that maximizing the sum of MAXIMIZE clauses does not guarantee the maximization of each individual clause, as certain clauses may not be maximizable simultaneously (e.g., `MAXIMIZE x` and `-x`). However, when clauses contain independent decision variables, maximizing the sum effectively maximizes each individual clause.

problem. Finally, a `<problem>` is a sequence of `CREATE CONSTRAINT` statements. Its semantics is given in Equation (*problem*): it combines all Boolean constraints using conjunction, and combines all optimization functions using addition. We look for an assignment to variables that satisfies all constraints and maximizes the function.

3.2 Evaluating C-SQL programs

Evaluating a C-SQL program given a state of the database involves two stages, as illustrated by the two arrows in Figure 1. First, in ***relation evaluation***, the DCM runtime generates code, which queries all the relations (typically tables and views stored in a database) referenced in the C-SQL program when invoked. Following that, the ***constraint evaluation*** stage encodes the given set of relations into an optimization problem and solves it using a standard constraint solver. As a result of the evaluation, we get an assignment of values to variables according to the constraints specified in C-SQL.

The DCM compiler generates an *encoder* from the C-SQL program specified by the developer. The frontend is based on Calcite [15], which parses C-SQL programs and generates, analyzes, and optimizes the resulting query. The compiler supports multiple backends to interface with different solvers. The flagship backend produces a Java-based encoder that interfaces with Google's CP/SAT solver [5] to solve constraints. The generated encoder fetches the relations' contents from the database and translates the result into constraints as defined by C-SQL's semantics. For each constraint query c, the compiler generates code to efficiently iterate over the relations referenced by the query and produce the corresponding constraint formulae using the solver-specific APIs (Figure 7). Relations are treated as vectors of records and iterated over using "for" loops and indices constructed by the encoder. Further compiler details are available in the DCM paper [72].

C-SQL cannot prevent the user from writing inefficient programs, a known challenge for declarative programming models. To help users, the compiler issues warnings when it produces inefficient code (e.g., nested "for" loops without indexes). At runtime, the encoder offers rich diagnostics, such as the number of variables and constraints used in each invocation of the solver, to help users understand performance and identify inefficiencies in their programs.

3.3 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. The 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, a variable column would be the `node_name` in the `pods` table (Figure 5), which represents the variables in a scheduling decision. For discrete variables such as the `node_name` column, the variable domain can be specified using a foreign key constraint.

Relations are updated by user code. In case of the scheduler, the DCM user receives notifications about changes to the cluster state (e.g., new pods created) with a Kubernetes client library and reflects these changes into the relations with DML statements.

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

Figure 5: `node_name` column represent decision variables. Other columns contain values supplied by the database.

```
CREATE CONSTRAINT node_predicates AS
        CHECK (node_name IN (SELECT node_name FROM valid_nodes))
        FROM pods_to_assign;
CREATE CONSTRAINT node_preferences AS
        MAXIMIZE (node_name IN (SELECT node_name FROM least_loaded))
        FROM pods_to_assign;

CREATE CONSTRAINT preemption_objective AS
        MAXIMIZE (priority * (node_name != 'NULL_NODE'))
        FROM pods_to_assign;
CREATE CONSTRAINT pod_affinity AS
        CHECK (pods_to_assign.has_pod_affinity_requirements = false) OR
                   (pods_to_assign.node_name NOT IN
                        (SELECT t1.node_name FROM pods_to_assign AS t1
                         JOIN inter_pod_affinity_matches AS t2
                         ON  t1.uid = t2.pod_uid
                         AND CONTAINS(t2.pod_matches,  t1.uid)))
        FROM pods_to_assign;
```

Figure 6: Example hard and soft constraints.

```
 // Hard constraint: node_predicates (pta = pods_to_assign)
 for (int pta_it = 0; pta_it < pta_it.size(); pta_it++) {
   model.varInDomain(pods_to_assign.getCol("node_name").get(pta_it),
                     valid_nodes.getCol("node_name"));
 }
 // Objective function: node_preferences
 for (int pta_it = 0; pta_it < pta_it.size(); pta_it++) {
   // Auxiliary variable to encode the truth value
   // of the varInDomain constraint
   BoolVar tmp_1 = model.varInDomainAsBoolVar(
                    pods_to_assign.getCol("node_name").get(pta_it),
                    least_loaded.getCol("node_name"));
   model.addToObjectiveFunction(bool_to_int(tmp_1));
 }
```

Figure 7: Simplified view of generated encoder code for top two C-SQL queries in Figure 6.

Policies as constraints. Policies are expressed in DCM as constraint queries against relations using C-SQL. Figure 6 shows example hard and soft constraints specified via `CHECK` and `MAXIMIZE` clauses against a table (`pods_to_assign`). The top two constraints require that variables from the column `node_name` are assigned a value present in the `valid_nodes` view (hard constraint), preferably one that also appears in the `least_loaded` view (soft constraint). The bottom two constraints showcase slightly more complex objectives: `preemption_objective` maximizes the priority of pending pods, and `pod_affinity` checks whether the assigned node satisfies the pod affinity requirements. Figure 7 shows a simplified view of the corresponding encoder the compiler produces, which sets up constraints for the constraint solver.

Compiler and runtime processing pipeline. Figure 8 shows DCM's Java APIs. `Model.compile()` takes in a C-SQL program, generates code, compiles and loads the code into memory, wrapped

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 8: DCM's programming model.

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 tables where variable columns are replaced by values that satisfy the constraints. `Model.solve()` is invoked whenever a new cluster management decision needs to be made (e.g., schedule all pods that are pending assignment).

4 CHALLENGES SCALING C-SQL

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, triggered by job arrivals or completions. These decisions often need to be made at sub-second timescales, such as 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. *The work performed by the scheduling pipeline should be proportional to the size of the change, and not the database size.*

In this section we describe the challenges of achieving this goal in the relation-evaluation (§4.1) and constraint-evaluation (§4.2) parts of the C-SQL processing pipeline.

4.1 Simulating IVM

We first discuss the challenges with efficient relation evaluation. As explained in §3.3, 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. Since tables have relatively small changes between successive invocations of `model.solve()`, recomputing the views from scratch each time is inefficient. Top-down query evaluation as with most databases is therefore a poor fit for such workloads. To improve efficiency, DCM users had to manually simulate incremental maintenance of materialized views (IVM) prior to this work. We describe two common approaches below.

Split views. The first approach splits the database schema into tables or views representing *changing* and *fixed* portions of the state. For example, we have separate tables for the placed pods and yet-to-be-placed pods, as opposed to a single table. Views that use pods as input would then be written to specifically refer to placed pods and/or pending pods, which further complicates the data model. Moreover, it is the DCM user's responsibility to maintain these separate views as the cluster state evolves, which defeats the benefits of a declarative programming model.

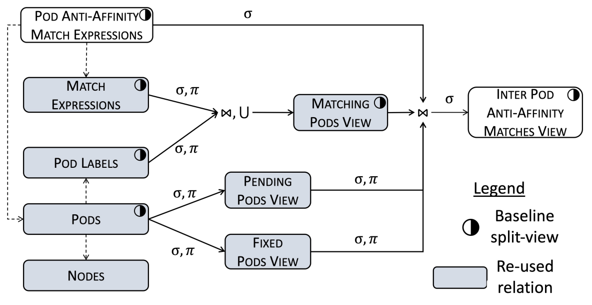

Figure 9: Simplified computation graph for the "inter-pod anti-affinity matches" view. Solid lines indicate input/output relationships, dashed lines indicate foreign key relationships. The relations in gray are re-used across several view calculations. With IVM, no view splitting is needed.

To illustrate split view, consider the INTER POD ANTI-AFFINITY MATCHES view in Figure 9. This is a key view in our DCM-powered Kubernetes scheduler, which shows groups of pods that are mutually anti-affine and should not be placed on the same nodes. Only a few rows of the view change whenever new pods arrive (or a few existing pods leave). However, to keep performance tractable, several base tables and views in the dataflow, including PODS, POD LABELS, MATCH EXPRESSIONS, as well as intermediate views like MATCHING PODS had to be split according to the pattern described above. Many tables and intermediate views in Figure 9 (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. To avoid redundant view evaluations, we need to maintain base tables that cache results via imperative code which is again, a burden on the DCM user.

Without split views, the scheduling latency is dominated by the database's latency to evaluate all views. For example, on a small cluster with 100 nodes, the database computation (H2 configured, running in-memory) takes several seconds, whereas the rest of the pipeline contributes under 10-20 milliseconds in total.

Aggregates and triggers. The second pattern is to compute expensive aggregates by simulating materialized views using triggers. Consider the view maintaining the spare capacity per node: on every update to a "pods" table that assigned a pod to a node, we set up a trigger that updates that node's spare capacity. This requires the developer to write imperative code and keep it in sync with the declarative specification of the schema and constraints of C-SQL.

To illustrate the large development efforts 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. A standard scheduler handles a fixed set of resource types and a column each for demands and capacities for: CPU, memory, disk, and other standard resource types. However, when using custom resources, the set of resource types itself is dynamic which prevents us from using a separate column for each resource type. 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 has 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 requires rewriting the triggers written using H2's Java APIs.

4.2 Handling Large Optimization Problems

Next, we describe challenges with the evaluation of large constraint optimization problems. The cluster state from the database is encoded into an optimization problem solved by the constraint solver. The constraint size is proportional to the database size, making solvers the main bottleneck for DCM's scalability as clusters grow. For example, we observed that increasing the cluster size from 500 to 50000 nodes increases the 95th percentile of constraint evaluation latency by two orders of magnitude to 7 seconds, vastly exceeding the goal of millisecond timescale scheduling.

To maintain good performance, it is crucial to keep the size of the optimization problem small. The solver attempts to simplify the optimization problem prior to finding solutions by applying complex rules in a "presolve" phase to remove redundant variables and constraints and tighten variable domains. We have previously noticed that this presolve phase is the primary contributor to increased latency at larger cluster sizes [72].

The fact that cluster management problems are highly incremental also leads to opportunities for improving the solver's efficiency. Since changes are often much smaller than the size of the database, only a subset of cluster states should be affected by these changes. For example, consider placing a pod with well-defined affinity constraints in a 10K node cluster. If only 10 nodes can satisfy these affinity constraints, it is highly inefficient to pass to the solver a formula involving the remaining 9990 nodes. Ideally, we would like to filter out these 9990 nodes early in the processing pipeline. However, this type of filtering is non-trivial when a mix of complex constraints expressed by C-SQL programs are involved.

5 IVM FOR THE CLUSTER STATE

As discussed in §4.1, incremental view maintenance (IVM) is key to scaling the relation evaluation component of the C-SQL pipeline. Instead of requiring users to simulate IVM through imperative code, we replace the query engine of DCM's database with an IVM engine that can automatically incrementalize the computation.

C-SQL's design allows for the use of any SQL-based IVM engine. Our queries use joins, aggregates, `GROUP BY`, `HAVING`, window operator `OVER`, and `UNION`. Differential Datalog (DDlog [10, 67]) support all above SQL features and has the added benefits of being developed in-house and having been deployed in production. It can also be embedded in-memory within a program, similar to our prior use of H2 for DCM. We therefore decided to build upon DDlog.

DDlog is a programming language for expressing incremental computations. DDlog's core incremental engine is based on Differential Dataflow [9, 58]. DDlog programs are written in Datalog, with extensions for rich types, arithmetic, strings, functions and a procedural language for writing user-defined functions.

A DDlog program instance represents a set of input relations as well as queries on those relations that result in output relations; output relations are equivalent to database views. DDlog programs compute on *changes* to inputs relations; a program accepts a batch of changes (inserts, deletes, updates) to all input relations and outputs a corresponding batch of changes to all output relations. Crucially, the computation of output changes is done directly, without evaluating the queries for the full database.

A DDlog program is compiled down to a Rust program that links to the Differential Dataflow library [9]. Client programs instantiate DDlog programs, update input relations and receive changes to output relations. DDlog programs keep state in memory, which makes them a good fit for representing a scheduler's view of cluster state. We implemented a compiler that translates SQL views into DDlog programs; we describe it in §5.2.

5.1 Simplified Programming Model

The use of an IVM engine can lead to significant simplification of C-SQL programs. By avoiding split views (§4.1), views can be kept simple while providing high performance. For example, previously, when a single logical view was decomposed into several views, it was difficult to identify which queries were performance bottlenecks. It was also tedious to rewrite queries as the database schema evolved to reflect new features. 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 9. This Figure shows the implementation before DDlog, which requires multiple views per table as well as imperative user code invoked by triggers. In addition, all gray views in Figure 9 were either evaluated multiple times per scheduling decision, or incrementalized manually using triggers. With DDlog, all base tables and views are incrementally updated and materialized, eliminating the need for imperative user code, split views, and triggers.

Case Study. We highlight the code simplification enabled by DDlog via a case study. Kubernetes extensively uses key-value labels to tag cluster entities (e.g., pods, nodes), enabling control plane code to employ a DSL to specify search queries for entities based on these labels. For instance, an anti-affinity requirement for pod *A* with a *match expression* (`app In [web-server]`) mandates that pod *A* avoids nodes with a pod labeled `app` and value `web-server`.

Figure 10 (top) shows a simplified version of the view we wrote before using the IVM engine to identify the set of pods that match an anti-affinity requirement. The full version required about 70 lines of SQL. Without DDlog, we had to carefully design the schema to reduce the number of records scanned. For example, a separate table `pod_aa_match_expressions` was created to isolate match expressions specified in the anti-affinity requirement and the view was scoped to only consider pods that are yet to be assigned (`pods_to_assign`). Even with carefully designed indexes, we can not guarantee that the `pod_info` and `pod_labels` tables will not be scanned by the query planner in databases like H2. The query is therefore not fully incremental. Worse, *the same match expression logic has to be duplicated for each type of scheduling policy that needs it*, such as

```
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 = 'Exist'
    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)

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 = 'Exist') me
 JOIN pod_labels ON me.label_key = pod_labels.label_key)
```

Figure 10: Example of view simplification enabled by the IVM engine. Note that the bottom view does not have joins specific to the anti-affinity logic or the set of pods to assign.

pod and node affinities [46], taints and tolerations [50] and many more. Doing so bloats the schema artificially, making schema evolution a challenge.

Figure 10 (bottom) shows the simplified code when using an IVM engine. This concise query, which evaluates all match expressions, can be shared by different downstream views. Notably, there are no joins specific to the anti-affinity policy or the set of pods under consideration. Instead, these joins appear in a single corresponding downstream view per scheduling policy (e.g., the inter-pod affinity matches view in Figure 9). By being incrementally updated and shared across multiple downstream views, the simplified code not only enhances performance but also promotes maintainability.

Most policies we implemented for scheduling involved match expressions, like the affinity and anti-affinity versions for inter-pod and node constraints, as well as taints and tolerations. For these policies, we see an average of 2.8× reduction in code size from ~185 lines per policy to ~64 lines by using DDlog. Policies where we simulated IVM using triggers such as the spare capacity policy saw no change in the SQL, but no longer needed triggers (roughly 80 lines of Java on average). Similarly, several views that were not incremental originally (like ones to extract the set of pending pods) became incremental with no code changes with the use of DDlog.

5.2 SQL-to-DDlog Compiler Implementation

We built a SQL-to-DDlog compiler in roughly 7K lines of code and 4K lines of tests in Java, using the Presto [11] 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 semantics differences between SQL and DDlog, the compiler only supports

SELECT DISTINCT queries. This particular restriction did not cause any issues during the migration, as we only worked with data organized as sets previously.

We also implemented a JDBC API to DDlog using JOOQ [7]. Our frontend interacts 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 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 where it acts as a pipeline maintaining the views continuously. As a result, our pre-existing corpus of DCM-based code could be switched to use DDlog and its incremental capabilities with minimal disruption. The code interfacing through JDBC was unchanged, whereas some SQL schema and views had to be changed to adapt to differences in the SQL dialect between H2 and Presto. The SQL-frontend and the SQL-to-DDlog compilers are open-source projects [12].

6 FEASIBILITY-PRESERVING PUSHDOWN

In §4.2, we discussed how constraint solvers are a bottleneck for DCM's scalability at larger cluster sizes. To address this challenge, we introduce the FP-pushdown optimization. FP-pushdown automatically simplifies optimization problems generated from C-SQL programs by pushing down predicates from constraints to relations. In C-SQL programs, constraints are generated from relations. By shrinking a relation, we can make the encoded optimization problem smaller. We achieve this by statically analyzing constraint queries and extracting filtering conditions from the constraint side that can be moved to the relation side. The filtering conditions are specified as additional views on the base relation, and can benefit from the incremental evaluation enabled by the IVM engine. The net effect of the optimization is similar to the solver's presolve phase, which reduces the number of variables and variable domains.

6.1 Problem Setup: Domain Restricting Views

Many cluster management problems involve combinatorial optimization tasks (e.g., assigning applications to machines), where the solution set is discrete. We therefore focus the discussion of FP-pushdown on discrete decision variables, whose domain can be defined via foreign key constraints as shown in Figure 5.

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

Hard and soft constraints may contain a domain restricting condition of the form `(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 (DRQ) Q', which is a SQL query that defines 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 and (3) Q_s: DRQs based on domain knowledge defining parts of the domain that are *likely* to contain feasible solutions.

Algorithm 1 Generating Domain Restricting Views

```
 1: function genDRQ($D, conditions$)
 2:     for $< tbl, col >$ in $conditions$ do
 3:         $queries$.add("SELECT * FROM D
 4:                 WHERE D.uid IN (SELECT col FROM tbl)")
 5:     return $queries$
 6: function DomainRestrictingView($D$)
 7:     $cond_{inc}, cond_{exc}$ = ExtractHardContraints($allViews$)
 8:     q_inc = String.join("UNION", genDRQ($D, cond_{inc}$))
 9:     q_exc = String.join("INTERSECT", genDRQ($D, cond_{exc}$))
10:     q_s = getTopK(discount($D, cond_{exc}, \gamma$ ))
11:     return "(q_s) UNION (q_inc) EXCEPT (q_exc)"
```

Given a set of DRQs, we can compute a single domain restricting view R for the variable column via $R \equiv Q_{inc} \cup Q_s \setminus Q_{exc}$ (line 6-11 in Algorithm 1). R now represents a narrowed 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 replaces T with T' in all queries.

6.2 Inferring Domain Restricting Queries (DRQ)

In this section, we describe how to generate DRQs from hard constraints and soft constraints.

Hard constraints. Predicates and subqueries that only involve constants are a common type of domain restricting conditions found in hard constraints. For example, the `node_predicates` constraint in Figure 6 is a hard constraint that contains a domain restricting condition 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 including additional values to the domain does not affect the correctness of the solution. In comparison, exclusion DRQs (Q_{exc}) can not be pushed down safely if there exist *any* OR predicates in the same constraint that involve non-variable columns. For example, consider the constraint `CHECK (NonVarExpr) OR (var NOT IN Qc)FROM Relation`. This constraint can be rewritten by moving the non-variable expression into the relation: `CHECK (var NOT IN Qc)FROM Relation WHERE NonVarExpr == false`. After rewriting, 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.

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 prefers nodes without any anti-affinity constraints or topology constraints instead of nodes with many constraints. This can be encoded into the custom sort order by "discounting" the available resources of a node based on constraints. For example, we decrease the spare capacity of a node by a factor of $0 < \gamma < 1$ each time the node appears in such constraints. Finally, the domain restricted query (Q_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 transform a solvable problem into one without solutions. At runtime, if the solver finds no solutions, we simply fall back to the default DCM solver logic which considers the entire variable domain.

6.3 Efficient Implementation of DRQs

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

To identify 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 §6.2 to extract domain restricting conditions (line 7 in Algorithm 1). The inclusion DRQ is generated by taking the union of the inclusion conditions (line 8), while the exclusion DRQ is generated by the intersection of the exclusion conditions (line 9). The exclusion conditions are also used to adjust the custom sort order on the domain table D before we take the top k (line 10). We include a detailed analyses of the contribution of different types of DRQs to the decision variable's domain size in the technical report [65].

Give the set of DRQs, we construct for each domain table a single domain restricting view $R = Q_{inc} \cup Q_s \setminus Q_{exc}$ (line 11). 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 (§6.1).

The initial schema, along with the generated domain restricting views and, the augmented tables are provided to DDlog together with the user queries. 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 does not depend on the use of FP-pushdown. The additional views of 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 §7.3.

7 PERFORMANCE EVALUATION

In this section, we evaluate the performance of DCM with the optimized C-SQL processing pipeline. Results show that:

- IVM is essential to both scheduling latency and throughput, allowing the system to efficiently compute and query summaries over the entire cluster state (§7.2)
- 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 (§7.3).

7.1 Experiment Setup

We model the evaluation after the original DCM paper to enable a direct comparison [72].

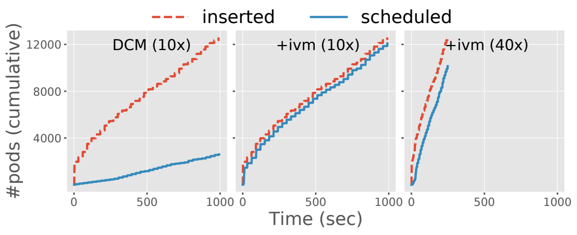

Figure 11: Number of pods inserted and scheduled over time in a 500-node cluster with and without IVM (DDlog). The trace's arrival rates are sped up by 10×/40×.

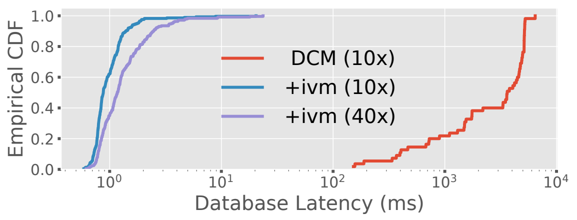

Figure 12: Empirical cumulative distribution function (ECDF) of database latency (fetch and update records) before and after incremental view maintenance.

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. Our evaluation utilizes the 2019 Azure public trace [59], which contains workload information of replica pod groups that were launched. Each replica pod group consists of one or more identical pods that were launched at the same time and have the same CPU and memory requirements. 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. Inter-pod anti-affinity constraints ensure that pods in the same replica group are not placed on the same node, while node affinity constraints match pods to a subset of available nodes that have workload-specific requirements such as custom hardware or to run workloads in pre-defined availability zones. For example, setting F=50% subjects half of the pod groups to these constraints. The larger F is, the harder it is to schedule the workload since these constraints involve reasoning over groups of pods. The workload also uses various hard and soft constraints 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 [72]. Our experiments therefore focus on comparing against DCM directly. Specifically, we evaluate the following variants of DCM:

- DCM: The original implementation as described in prior work.
- +ivm: DCM with DDlog as the IVM backend.
- +ivm+pushdown: DCM with DDlog backend and FP-pushdown.

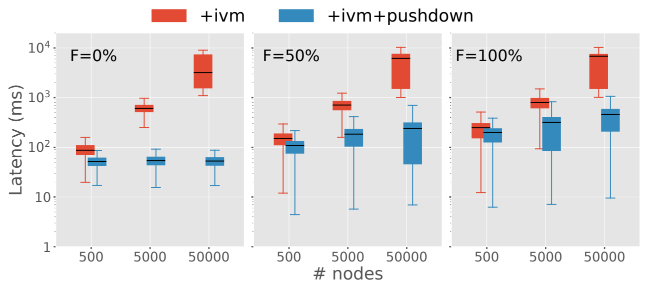

Figure 13: Scheduling latency for a batch of pods (max 50) at 10× trace speed up (log-scale) and different cluster sizes.

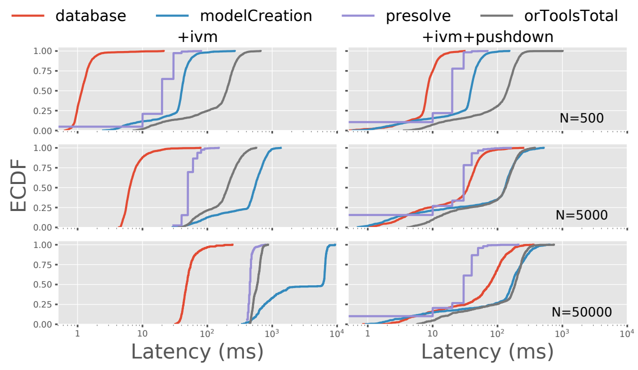

Figure 14: Scheduling latency breakdown per batch of pods (max 50) at $F = 100\%$ and different cluster sizes.

7.2 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 use split views, since it makes the code not maintainable (§4.1).

Figure 11 reports the insert and scheduling throughput of DCM, measured as the total number of pods inserted and scheduled over time, without and with an IVM backend. We speed up the pod arrival rate in the trace by 10× and 40×. 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 performs redundant work even when only small changes happen to the cluster state between iterations. This cluster state database bottleneck is apparent in Figure 12, which reports the empirical cumulative distribution function (ECDF) of database latency that includes the total time to fetch and update records from the database. Concretely, the 95th percentile (p95) database latency at 10× trace speed up decreases from over 5 seconds without IVM to 1.7ms with IVM (read from intersections between latency curves and a horizontal line at $y = 0.95$), which is a near 3000× speed up.

In comparison, using IVM, scheduling throughput can keep up with the request arrival rate and the database latencies remain tractable (p95 latency around 3.5ms) even at 40× and higher trace speedups. Figure 14 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. The low database latency shows that cluster state management using IVM via DDlog is not a bottleneck even with high arrival rates.

7.3 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 13 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.

Without FP-pushdown, scheduling latency increases significantly from a median of hundreds of milliseconds to a few seconds as the cluster sizes increase. 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 cluster size 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 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 has affinity to. We have included additional results in the appendix of the technical report [65] showing that 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.

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 14 reports the detailed breakdown of scheduling latency at $F = 100\%$ and cluster sizes of 500, 5000 and 50000 nodes with and without 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, the constraint solver's scalability is 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.

8 RELATED WORK

Declarative data-center management. Several works have tackled the inflexibility of modern cluster manager designs. DCM [72] tackles recurring combinatorial optimization tasks in data-centers via a SQL-based programming model, which we formalize as C-SQL in this paper. DBOS [69] 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. C-SQL is sufficiently expressive to deal with a broad range of cluster management tasks for two reasons. First, the relational model shines at representing complex system state, as seen by several systems that used relational languages to simplify systems programming [14, 30, 56, 62, 81]. Second, cluster management problems are often constraint optimization problems [19, 22, 23, 25–27, 34, 37, 76, 77], and C-SQL extends SQL to support constraint specification.

Constraint solvers for resource management. Many works have explored the use of solvers to aid various facets of datacenter resource management [19, 22, 23, 25–27, 34, 37, 76, 77]. 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 over time. Neither can these systems leverage query optimization techniques such as IVM without the relational model. Using custom solvers for specific problem domains can yield significant rewards. For example, Shard Manager [53] uses domain knowledge to partition the problem and use local search to assign shards in distributed applications. Wrasse [63] uses a balls-and-bins based abstraction to model allocation problems and uses a GPU-based solver to find assignments. DCM and C-SQL allow different constraint solver backends to be plugged in for constraint evaluation, which allows such approaches to leverage the simpler and more expressive relational programming model.

Constraint query language. The idea of combining declarative database programming with constraint solving has been previously explored in the context of constraint databases [38, 39, 52, 66], albeit with a different focus on the representation and querying of spatial temporal data [21, 64, 75]. 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 [78] 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, researchers have explored the idea of moving predicates around (e.g., up, down and sideways) in the query plan for performance reasons [17, 33, 54]. Most query optimization techniques can not push down predicates below a join operator, unless the predicates are on columns used in the join condition [24, 61, 68, 80]. More broadly, predicate pushdown style optimization has shown performance benefits in applications such as sensor networks [70], dataflow operators [35], video analytics systems [40] and machine learning inference queries [57]. FP-pushdown is another example of applying the predicate pushdown technique in a new domain of constraint solving. FP-pushdown differs from traditional predicate pushdown techniques since it pushes down predicates from the constraint side of C-SQL programs to the relation side, instead of just moving predicates around within relations.

Incremental View Maintenance. IVM has been extensively studied in the database community [18, 28, 31, 36, 60, 82, 83]. In this work, we build on decades of research in IVM and use an existing IVM engine as a black-box. We do not claim to make any contributions to IVM techniques. Our contribution is to show that we can essentially use a SQL-like language to express constraints and still apply standard IVM techniques for this new use case which produces constraints instead of query plans. Although any IVM engine could be used to speed up C-SQL 's relation evaluation, we chose to use DDlog due to its feature set and our operational expertise with it in production. For example, DBToaster [41], which provides an open-source IVM engine [1], could also be used but it does not currently support operators such as `UNION`, `HAVING`, and `WINDOW` used in DCM. Outside cluster management, incremental computation is also useful in enabling key control plane functionality, such as shown in DeltaPath [20] and Full-stack SDN [71].

9 CONCLUSION AND FUTURE WORK

This paper describes how we improved the scalability and programmability of a Declarative Cluster Manager architecture using C-SQL, a constraint language that extends SQL. The declarative programming model allows us to specify the goals of cluster management while leaving optimization and execution plans to the runtime, which is not possible with today's heuristic-based cluster managers. Our C-SQL language is similar to SQL, allowing us to apply decades of query optimization research to optimize C-SQL programs. As a result, we can now scale a DCM-powered Kubernetes scheduler beyond its original limits to hyperscale-sized clusters with simpler schema and less imperative user code. We believe that C-SQL is a versatile language for specifying constraint optimization problems using a SQL-style syntax. We plan to explore applications of C-SQL beyond cluster management in future work.

REFERENCES

[1] [n.d.]. DBToaster SQL Reference. https://dbtoaster.github.io/docs_sql.html. Last accessed: June 2023.

[2] [n.d.]. OpenStack. https://www.openstack.org/. Last accessed: June 2023.

[3] [n.d.]. Red Hat OpenShift. https://www.redhat.com/en/technologies/cloud-computing/openshift. Last accessed: June 2023.

[4] 2007. H2 Database. https://github.com/h2database/h2database/. Last accessed: June 2023.

[5] 2010. Google OR-Tools. https://developers.google.com/optimization/. Last accessed: February 2023.

[6] 2010. Google OR-Tools Documentation. https://github.com/google/or-tools/blob/stable/ortools/sat/docs/README.md. Last accessed: April 2023.

[7] 2011. JOOQ. https://github.com/jOOQ/jOOQ. Last accessed: June 2023.

[8] 2014. Kubernetes (K8s) Github. http://github.com/kubernetes/kubernetes. Last accessed: June 2023.

[9] 2015. Differential Dataflow. https://github.com/TimelyDataflow/differential-dataflow. Last accessed: June 2023.

[10] 2021. Differential Datalog. github.com/vmware/differential-datalog. Last accessed: June 2023.

[11] 2021. PresoDB. https://prestodb.io/. Last accessed: August 2022.

[12] 2022. DDlog's SQL frontend and SQL-to-DDlog compiler. https://github.com/vmware/differential-datalog/tree/master/sql. Last accessed: November 2022.

[13] 2022. Kubernetes. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/. Last accessed: June 2023.

[14] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M Hellerstein, and Russell Sears. 2010. Boom analytics: exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European conference on Computer systems*. ACM, 223–236.

[15] Apache. 2014. Apache Calcite. https://calcite.apache.org/. Last accessed: Feb 2023.

[16] Stefano Ceri and Georg Gottlob. 1985. Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. *IEEE Trans. Softw. Eng.* 11, 4 (apr 1985), 324–345. https://doi.org/10.1109/TSE.1985.232223

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

[18] Latha S Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. 1996. Algorithms for deferred view maintenance. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. 469–480.

[19] 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.

[20] 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

[21] 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.

[22] 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

[23] 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

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

[25] 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 (SIGCOMM '14)*. 455–466. https://doi.org/10.1145/2619239.2626334

[26] 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 (OSDI'16)*. 65–80.

[27] 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.

[28] Timothy Griffin and Leonid Libkin. 1995. Incremental maintenance of views with duplicates. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*. 328–339.

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

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

[31] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. 1993. Maintaining views incrementally. *ACM SIGMOD Record* 22, 2 (1993), 157–166.

[32] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. 2020. Protean: VM Allocation Service at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 845–861. https://www.usenix.org/conference/osdi20/presentation/hadary

[33] 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.

[34] 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.

[35] 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. *PVLDB* 5, 11 (2012).

[36] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2020. General dynamic Yannakakis: conjunctive queries with theta joins under updates. *The VLDB Journal* 29, 2-3 (2020), 619–653.

[37] 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.

[38] 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.

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

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

[41] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. 2014. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.* 23, 2 (2014), 253–278. https://doi.org/10.1007/s00778-013-0348-4

[42] Kubernetes. 2018. Add a new predicate: max replicas limit per node. https://github.com/kubernetes/kubernetes/pull/71930. Last accessed: Feb 2023.

[43] Kubernetes. 2018. Add max number of replicas per node/topology key to pod anti-affinity. https://github.com/kubernetes/kubernetes/issues/40358. Last accessed: Feb 2023.

[44] Kubernetes. 2018. Affinity/Anti-Affinity Optimization of Pod Being Scheduled #67788. https://github.com/kubernetes/kubernetes/pull/67788. Last accessed: Feb 2023.

[45] Kubernetes. 2018. Allow Minimum (or Maximum) Pods per failure zone. https://github.com/kubernetes/kubernetes/issues/66533. Last accessed: Jan 2019.

[46] Kubernetes. 2018. Assign Pods to Nodes using Node Affinity. https://kubernetes.io/docs/tasks/configure-pod-container/assign-pods-nodes-using-node-affinity/. Last accessed: Feb 2023.

[47] Kubernetes. 2018. Maximum of N per topology value. https://github.com/kubernetes/kubernetes/pull/41718. Last accessed: Feb 2023.

[48] Kubernetes. 2018. MaxPodsPerNode - be able to set hard and soft limits for deployments / replicasets. https://github.com/kubernetes/kubernetes/issues/63560. Last accessed: Feb 2023.

[49] Kubernetes. 2018. Pod priorities and preemption. https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/. Last accessed: June 2023.

[50] Kubernetes. 2018. Taints and Tolerations. https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/. Last accessed: Feb 2023.

[51] Kubernetes mailing list. 2018. Let's remove ServiceAffinity . https://groups.google.com/forum/#!topic/kubernetes-sig-scheduling/ewz4TYJgL0M. Last accessed: Feb 2023.

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

[53] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, et al. 2021. Shard Manager: A Generic Shard Management Framework for Geo-Distributed Applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. 553–569.

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

[55] Kubernetes Topology Manager Limitations. [n.d.]. https://kubernetes.io/docs/tasks/administer-cluster/topology-manager/#known-limitations. Last accessed: June 2023.

[56] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. 2005. Implementing Declarative Overlays. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*. 75–90.

[57] 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.

[58] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *CIDR*.

[59] Microsoft. 2017. Azure Public Dataset. https://github.com/Azure/AzurePublicDataset. Last accessed: June 2023.

[60] Hoshi Mistry, Prasan Roy, S Sudarshan, and Krithi Ramamritham. 2001. Materialized view selection and maintenance using multi-query optimization. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. 307–318.

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

[62] Ben Pfaff and Bruce Davie. 2013. RFC 7047: The Open vSwitch Database Management Protocol. https://datatracker.ietf.org/doc/html/rfc7047. Last accessed: June 2023.

[63] Anshul Rai, Ranjita Bhagwan, and Saikat Guha. 2012. Generalized Resource Allocation for the Cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC '12)*. Article 15, 12 pages.

[64] 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.

[65] 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.

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

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

[68] 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.

[69] 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. *PVLDB* 15, 1 (2022), 21–30. https://doi.org/10.14778/3485450.3485454

[70] Utkarsh Srivastava, Kamesh Munagala, and Jennifer Widom. 2005. Operator placement for in-network stream query processing. In *Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 250–258.

[71] 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

[72] 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)*. 827–844.

[73] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, et al. 2020. Twine: A Unified Cluster Management System for Shared Infrastructure. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. 787–803.

[74] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: The next Generation. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys '20)*. Article 30, 14 pages.

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

[76] 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 (SoCC '12)*. Article 25, 7 pages. https://doi.org/10.1145/2391229.2391254

[77] 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

[78] Jeffrey D. Ullman. 1989. *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press.

[79] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (Bordeaux, France). 1–17. http://doi.acm.org/10.1145/2741948.2741964

[80] 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.

[81] Anduo Wang, Xueyuan Mei, Jason Croft, Matthew Caesar, and Brighten Godfrey. 2016. Ravel: A Database-Defined Network. In *Proceedings of the Symposium on SDN Research* (Santa Clara, CA, USA) *(SOSR '16)*. Association for Computing Machinery, New York, NY, USA, Article 5, 7 pages. https://doi.org/10.1145/2890955.2890970

[82] Jingren Zhou, Per-Ake Larson, and Hicham G Elmongui. 2007. Lazy maintenance of materialized views. In *Proceedings of the 33rd international conference on Very large data bases*. 231–242.

[83] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. 1995. View maintenance in a warehousing environment. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*. 316–327.