Combinatorial Problem Solving over Relational Databases: View Synthesis through Constraint-Based Local Search

Toni Mancini
Sapienza University
Rome,Italy
tmancini@di.uniroma1.it

Pierre Flener
Uppsala University
Sweden
Pierre.Flener@it.uu.se

Justin K. Pearson
Uppsala University
Sweden
Justin.Pearson@it.uu.se

ABSTRACT

Solving combinatorial problems is increasingly crucial in business applications, in order to cope with hard problems of practical relevance. In these settings, data typically reside on centralised information systems, in form of possibly large relational databases, serving multiple concurrent transactions run by different applications. We argue that the use of current solvers in these scenarios may not be a viable option, and study the applicability of extending information systems (in particular database management systems) to offer combinatorial problem solving facilities. In particular we present a declarative language based on SQL for modelling combinatorial problems as second-order views of the data and study the applicability of constraint-based local search for computing such views, presenting novel techniques for local search algorithms explicitly designed to work directly on relational databases, also addressing the different cost model of querying data in the new framework. We also describe and experiment with a proof-of-concept implementation.

1. INTRODUCTION

Solving combinatorial problems is increasingly crucial in many business scenarios, in order to cope with hard problems of practical relevance, like scheduling, resource and employee allocation, security, and enterprise asset management. The current approaches (beyond developing ad-hoc algorithms) model and solve such problems with constraint programming (CP), mathematical programming (MP), SAT, or answer set programming (ASP). Unfortunately, in typical business scenarios, data reside on centralised information systems, in form of possibly large relational databases (DB) with complex integrity constraints, deployed in data centres with a highly distributed or replicated architecture. Furthermore, these information systems serve *multiple* concurrent transactions run by different applications. In these scenarios, the current approach of loading the combinatorial problem instance data into a solver and running it externally to the database management system (DBMS) may not be an option: data integrity is under threat, as the other business transactions could need write-access to the data during the (potentially very long) solving process. Furthermore, in some scenarios, the size of the portion of the data relevant to the problem may be too large for it to be easily represented in central memory, which is a requirement for current solvers. Finally, the complex structure of the data (or lack of structure, as in presence of, e.g., textual or geo-spatial information) may not permit an easy modelling into the languages offered by current solvers without extremely expensive problem-specific preprocessing and encoding steps.

In [2] we argue that to address these issues, which may hinder a wider applicability of declarative combinatorial problem solving technologies in business contexts, information systems and in particular DBMSs could be extended with means to support the specification of *second-order views* (i.e., views whose definition may specify NP-hard problems). Although the interest of second-order query languages has been mostly theoretical so far, in [2] we show that adding non-determinism to SQL (the DBMS language most widely used today) is a viable means to offer combinatorial problem modelling facilities to the DB user in practice.

In this paper we improve (Sec. 2) the SQL-based combinatorial problem modelling language of [2] for the definition of second-order views, with the aim of easing the modelling experience for the *average* DB user, who usually has no skills in combinatorial problem solving. In particular, we show that a *single* general-purpose new construct enables nondeterminism, which is the key mechanism to enhance the expressive power of SQL. After a short formalisation of the new language in terms of relational calculus (Sec. 3), we study in Sec. 4 the applicability of *local search* (LS) [3], in the spirit of *constraint-based local search* (CBLS) [7], to compute second-order DB views. In particular, we study the feasibility of enhancing standard DBMSs with LS capabilities, rather than connecting them to external LS solvers. This choice aims at taking into account the concerns above as much as possible and brings different advantages. (*i*) Data is not duplicated outside the DBMS, hence we do not require that it is kept frozen during solving. The computation of a second-order view is seen as a normal DBMS background process that executes concurrently with the business transactions, with its content *evolving* during the life-cycle of the underlying DB, exploiting standard DBMS *change-interception mechanisms* (i.e., triggers or event-based rules) to react to (often small) data changes. (*ii*) As LS (although incomplete) offers better scalability than global search, our approach can be particularly useful when dealing with large problem instances, as is often the case with business applications. Clearly, DBMSs can offer different view computation engines based on complementary technologies (e.g., backtracking and propagation). (*iii*) Our approach allows

SAC'12 March 25-29, 2012, Riva del Garda, Italy.

the *average* DB user to solve problems where data and constraints cannot be easily modelled in the languages of current CP/MP/ASP solvers, as it offers the *whole spectrum of* SQL *data-types and functions* (e.g., the user can easily model a constraint that performs full-text search, or pattern matching in geo-spatial data or XML trees).

Given that the query cost model of DBMSs is very different from the one of traditional solvers operating exclusively in central memory, we *re-think* LS with an explicit focus on the relational model and on the features of modern DBMSs to manage relational data (potentially large and stored in external memory) efficiently. In particular, we show that the relational information retrieved during the evaluation of constraints in any state of the search (beyond the usual quantitative information about the cost share of each constraint) allows us to reason on the *causes* why constraints are violated and to synthesise automatically a set of moves *explicitly devoted* to remove these causes. We call this new technique *dynamic neighbourhood design* (DND). The set of moves synthesised by DND, in the form of a (typically much smaller) subset of the set of all moves M that can be executed from the current state, is *complete*, in the sense that it will always contain the move that the LS algorithm (whichever in use) would choose, if run on the entire set M. Furthermore, we show that the clever join optimisation techniques of current DBMSs may be exploited to perform a *joint* (i.e., collective) *exploration of the neighbourhood* reachable with these moves, which largely improves the time needed by evaluating them one by one, as is done in traditional LS solvers. We call this new technique *joint incremental neighbourhood evaluation* (JINE).

Sec. 5 briefly describes a proof-of-concept implementation based on DND and JINE, as well as some experiments.

2. MODELLING AS VIEW SYNTHESIS

ConSQL [2] is a non-deterministic extension of SQL for the specification of combinatorial problems. Being a superset of SQL, it provides users with the possibility of safely exploiting the rich set of language features of SQL during problem modelling. In this section, we improve this language from a modelling standpoint, greatly reducing the number of keywords added to standard SQL (from 11 to just 4). In particular, we propose a single general-purpose construct to enable non-determinism. In our opinion, the new language, which we call ConSQL$^+$, makes the modelling task much easier for the average DB user.

We do not give a formal description of the language, but we introduce it by an example. Assume that a set of scientists stored in DB relation Org wants to organise a set of at most w workshops. All scientists in Org will participate, and may be asked to invite others chosen from their coauthors. Each participant attends exactly one workshop. The workshops to be organised (which must have a number of participants between $part_{min}$ and $part_{max}$) do not have predefined topics, as the partitioning of the participants among them is entirely based on a complex criterion on the similarity of their publications. In particular: two scientists s_1 and s_2 are considered *unrelated* iff any of them has more than k publications unrelated to more than k publications of the other. Two publications p_1 and p_2 are considered (un)*related* according to a criterion similar(p_1, p_2) based on the similarity of their titles and/or abstracts (analogously to what happens in current web search engines). Each workshop can have at most u pairs of unrelated participants.

Data is taken from a bibliography DB storing (among possibly many others) the following three tables (primary keys are underlined): Scientist(id) (the set of all scientists in the DB), Pub(id, title) (the set of all publications), Authoring(s,p) (storing authorships, s ∈ Scientist, p ∈ Pub).

Solving this problem with current CP/MP/ASP solvers is extremely difficult, requiring the user to perform an extraordinarily expensive (in programming effort) problem-specific preprocessing and encoding of textual information: in particular, it could be necessary to pre-compute and store the similarity measures for all pairs of publications authored by any two scientists that may be invited to the same workshop. The problem can however be easily and compactly modelled in ConSQL$^+$ as follows (once tables Org(id) and workshop(ws) have been added) exploiting the flexibility of SQL (the few new keywords added to SQL are capitalised):

```
create SPECIFICATION Workshops (
  create view Invitation as
    select sb.id as invitee, CHOOSE(select ws from workshop)
    from Org s, Authoring sa, Pub pa, Authoring sb
    where s.id = sa.author and sa.pub = pa.id and sb.pub = pa.id
```

A problem specification is defined by the new construct create SPECIFICATION, which embeds several elements. Above, we have defined the first of such elements, namely a second-order view on the data (Invitation). The difference of Invitation with ordinary SQL views is that one column (ws) is defined by the new construct CHOOSE. Such columns are called *choose columns*. We are asking the DBMS to populate the choose column *non-deterministically*, picking tuples from the query argument of CHOOSE (an example of a candidate extension of view Invitation is in Fig. 1). The user is supposed to specify *constraints* that the DBMS must satisfy when populating a choose column, as well as an optional optimisation function. Constraints are defined as standard SQL assertions (see App. A for a brief description of the SQL syntax). As an example, the fragment:

```
check "con1" ( not exists (
  select * from Invitation i, Org s where i.invitee = s.id and i.ws is null ))
check "con2" ( not exists (
  select ws from Invitation i where i.ws is not null group by ws
  having count(*)>0 and (count(*)<part_min or count(*)>part_max)  ))
```

defines constraints con1 and con2, which are satisfied iff view Invitation is populated in such a way that all scientists in set Org are invited to a workshop (con1) and each workshop has either 0 (workshop not held) or $part_{min}$ to $part_{max}$ invitees (con2). Function count(*) is one of the several *aggregates* offered by SQL, and counts the number of tuples inside each group. SQL (and hence ConSQL$^+$) follows a paradigm similar to that of logic programming languages and dual to that of, e.g., CP, in that it encodes as $\nexists \vec{x}.\neg c(\vec{x})$ a constraint that a CP practitioner would write as $\forall \vec{x}.c(\vec{x})$. This is because queries are *existentially* quantified formulas. Although ConSQL$^+$ accepts constraints defined by *any* SQL condition, not exists constraints are by far the most commonly used in practice (as are $\forall$ constraints in CP). As constraints are represented *intensionally*, this modelling paradigm does *not* introduce any blow-up in the size of the problem model.

A specification can also define ordinary SQL views (helper views), possibly dependent on second-order views. This eases the modelling task of some constraints. As an example, the following view of the Workshops specification:

```
create view unrelated_scientists_pub as
  select i1.ws as ws, a1.author as s1, a2.author as s2, p1.id as p1, count(*) as n
  from Invitation i1, Invitation i2, Authoring a1, Authoring a2, Pub p1, Pub p2
  where i1.ws is not null and i2.ws is not null and i1.ws = i2.ws and
        i1.s <> i2.s and i1.s = a1.author and i2.s = a2.author and
        a1.pub = p1.id and a2.pub = p2.id and not similar(p1, p2)
  group by i1.ws, a1.author, a2.author, p1.id having n > k
```

evaluates to the set of pairs of scientists (columns s1 and s2 in the select clause) invited to the same workshop ws, such that one publication p1 of s1 is unrelated to *more than* k publications of s2. This view is used to define a second one:

```
create view unrelated_scientists as select ws, s1, s2
  from unrelated_scientists_pub group by ws, a1, a2 having count(*) > k
```

which returns, for every workshop ws, the set of pairs (s1, s2) of unrelated scientists invited to ws.

Constraint con3 is defined on top of the last helper view:

```
check "con3" ( not exists (
  select ws from unrelated_scientists group by ws having count(*) > u ))
```

which enforces view Invitation to be such that no workshop will be attended by more than u pairs of unrelated scientists.

Finally, a specification may define an *objective function* (omitted in this example for brevity) via the new constructs MINIMIZE and MAXIMIZE applied to an arbitrary aggregate SQL query (i.e., a query returning a single numeric value).

In general, a specification may define several second-order views and each of them may contain several CHOOSE constructs.

The language of [2] used different keywords to define the *search space* (i.e., the set of all possible ways to populate choose columns of second-order views): SUBSET, PARTITION, FUNCTION, PERMUTATION. All these keywords have been replaced by the single non-deterministic CHOOSE construct, which models a function (total if not null is specified, partial otherwise) from the pure SQL part of the hosting view (the domain of the function) to the set of tuples of a discrete and bounded interval or SQL query argument of CHOOSE (the codomain of the function). As CHOOSE constructs can take arbitrary queries as argument as well as numeric intervals and modifiers, they are very flexible, offering an easy all-purpose modelling tool to the DB user: e.g., CHOOSE(distinct between 1 to count(*) as n) would ask the DBMS to populate column n with a total ordering of the tuples in the view (this would require the use of the PERMUTATION keyword in the language of [2]). Second-order views cannot refer to other second-order views and their where clause cannot refer to choose columns.

3. FORMALISING ConSQL$^+$

We formalise a ConSQL$^+$ specification as a triple $\langle \mathbf{S}, \mathbf{V}, \mathbf{C} \rangle$, and use a language based on the *safe tuple relational calculus* (TRC) [6] to express queries. $\mathbf{S} = \{\mathsf{S}_1, \ldots, \mathsf{S}_k\}$ is a set of *total functions*, each $\mathsf{S}_i : \mathsf{D}_i \to \mathsf{C}_i$ being defined between two unary DB relations. Each S_i is represented as a relation over $\mathsf{D}_i \times \mathsf{C}_i$ with the same columns $\{dom, codom\}$. These functions, corresponding to ConSQL$^+$ views with a single choose column, are called *guessed functions*. $\mathbf{V} = \{\mathsf{V}_1, \ldots, \mathsf{V}_l\}$ is the set of views, whose extensions are (directly or through other views) functionally dependent on (in the sense of [5]) those of $\mathbf{S}$. View V_i of the form (*conjunctive* view):

$$\text{select * from } \mathsf{V}^1, \ldots, \mathsf{V}^p, \mathsf{S}^1, \ldots, \mathsf{S}^q, \mathbf{T} \text{ where } w \tag{1}$$

defined on top of other views (V^i), guessed functions (S^j), and (fixed) DB relations (list $\mathbf{T}$), defines the set of tuples $\{t \mid t \in \mathsf{V}^1 \times \cdots \times \mathsf{V}^p \times \mathsf{S}^1 \times \cdots \times \mathsf{S}^q \times \mathbf{T} \wedge w(t)\}$. Note that any view or guessed function (e.g., $\mathsf{S}_j \in \mathbf{S}$) may occur multiple times in the definition of V (with different superscripts: e.g., both S^i_j and S^k_j, with $i \neq k$). For our purposes we can always ignore a select clause different from '*' for such queries, also avoiding most issues on the SQL bag-semantics, by (possibly adding and) maintaining a key for all the relations involved. A view of the form:

$$\begin{array}{l}\text{select } \mathbf{g},\mathbf{a},\mathbf{e} \text{ from } \mathsf{V}^1, \ldots, \mathsf{V}^p, \mathsf{S}^1, \ldots, \mathsf{S}^q, \mathbf{T} \\ \text{where } w \text{ group by } \mathbf{g} \text{ having } h\end{array} \tag{2}$$

with grouping attributes $\mathbf{g}$ (a subset of the columns of the input relations), aggregates $\mathbf{a}$, and value expressions $\mathbf{e}$ (arithmetic expressions or functions over $\mathbf{g}$ and $\mathbf{a}$, evaluated for each tuple returned), is formalised as $\Pi_{\mathbf{g},\mathbf{a},\mathbf{e}}\left(\mathsf{V}^{\text{conj}}_i\right)$, where $\mathsf{V}^{\text{conj}}_i$ is the formalisation of (1) (the *conjunctive part* of V) and $\Pi_{\mathbf{g},\mathbf{a},\mathbf{e}}$ is an operator that, when applied to $\mathsf{V}^{\text{conj}}_i$ with parameters $\mathbf{g}$, $\mathbf{a}$, and $\mathbf{e}$, defines the following set:

$$\left\{ \langle t_{\mathbf{g}}, t_{\mathbf{a}}, t_{\mathbf{e}} \rangle \;\middle|\; \begin{array}{l} t_{\mathbf{a}} \text{ and } t_{\mathbf{e}} \text{ are the results of computing } \mathbf{a} \text{ and } \mathbf{e} \\ \text{on } \mathsf{G}, \text{ where } \mathsf{G} \text{ is the set of tuples in } \mathsf{V}^{\text{conj}}_i \\ \text{whose columns } \mathbf{g} \text{ agree with } t_{\mathbf{g}} \text{ and } \mathsf{G} \neq \emptyset \end{array} \right\}$$

Operator Π [6] never returns two tuples that agree on $\mathbf{g}$. The set $\mathbf{e}$ is assumed to contain function $\mathsf{sat}_h(\mathbf{g}, \mathbf{a})$, which computes for each group an integer denoting whether the having condition h is satisfied ($\mathsf{sat}_h > 0$) or not ($\mathsf{sat}_h = 0$). In App. B we give one possible definition for sat_h suitable for our goals. The graph of dependencies among views must be *acyclic* (as SQL forbids cyclic dependencies).

$\mathbf{C} = \{\mathsf{con}_1, \ldots, \mathsf{con}_p\}$ denotes the set of constraints, each defined over a view in $\mathbf{V}$. They may be of two kinds: exists or not exists (SQL supports also $\circ$any and $\circ$all, with $\circ$ being any arithmetic comparison operator, but constraints using them can always be rewritten in terms of exists or not exists).

Evaluating $\langle \mathbf{S}, \mathbf{V}, \mathbf{C} \rangle$ on a finite DB amounts to non-deterministically populating extensions of all functions in $\mathbf{S}$ so that all constraints are satisfied. Such extensions, if exist, represent a *solution* to the problem.

In [2] we introduced NP-Alg, a non-deterministic extension of plain relational algebra (RA), as the formal language to define ConSQL. However, using NP-Alg to present our new techniques on ConSQL$^+$ specifications is impractical, as RA expressions soon become intricate and the structure of the problem (also in terms of dependencies among views) is hidden. Given that TRC and RA are equivalent [6], we can show (details omitted for lack of space) that our TRC-based language is equivalent to NP-Alg, being able to express all and only the specifications of decision problems belonging to the class NP.

4. CBLS TO COMPUTE ConSQL$^+$ VIEWS

Local search (LS) [3] has proved to be an extremely promising approach to solve combinatorial problems. Also, its intrinsic flexibility may be exploited when building systems that need to cope with dynamic and concurrent settings and do not have exclusive access to the data-set. Below, we restate the main notions of LS in terms of a ConSQL$^+$ specification $\langle \mathbf{S}, \mathbf{V}, \mathbf{C} \rangle$. Then, we present novel LS techniques to exploit the relational query cost model.

A *state* is an extension $\overline{\mathbf{S}}$ for guessed functions $\mathbf{S}$. The *search space* is the set of all possible states. The *cost* of a state $\overline{\mathbf{S}}$ is $\Sigma_{\mathsf{con} \in \mathbf{C}}\, cost(\mathsf{con})$, where $cost(\mathsf{con})$ is the cost of constraint con in $\overline{\mathbf{S}}$ (weights may be added to the constraints). For a not exists constraint $\mathsf{con} = \nexists \mathsf{V}_{\mathsf{con}}$, $cost(\mathsf{con})$ is $|\mathsf{V}_{\mathsf{con}}|$ if $\mathsf{V}_{\mathsf{con}}$ is of the form (1) and $\Sigma_{t \in \mathsf{V}_{\mathsf{con}}} t.\mathsf{sat}_h$ if $\mathsf{V}_{\mathsf{con}}$ is of the form (2) (see App. B for a possible definition of sat_h). For an exists constraint $\mathsf{con} = \exists \mathsf{V}_{\mathsf{con}}$, $cost(\mathsf{con})$ is 0 if $|\mathsf{V}_{\mathsf{con}}| > 0$, and 1 otherwise. A *solution* is a state with cost 0.

Most LS algorithms have at their core a *greedy behaviour*, as they iteratively evaluate and perform small changes (*moves*)

from the current state to *neighbour* states, in order to reduce its cost. To escape *local minima* (states from which no moves lead to a cost reduction), they are enhanced with non purely greedy techniques, like simulated annealing or tabu-search.

Although, in general, the universe of possible moves is designed by the programmer depending on the problem, the simple structure of the relational model and guessed functions suggests a natural general-purpose definition for moves: a *move* is a triple $\langle \mathsf{S}, d, c\rangle$ where $\mathsf{S} \in \mathbf{S}$ (with $\mathsf{S} : \mathsf{D} \rightarrow \mathsf{C}$), $d \in \mathsf{D}$, $c \in \mathsf{C}$. In any state, a move $\langle \mathsf{S}, d, c\rangle$ changes the extension of S by replacing with c the co-domain value assigned to domain value d. More complex moves (e.g., swaps) could be defined by composition of these atomic moves. A move m is *improving*, *neutral*, or *worsening* (resp., for a given constraint) in a state $\overline{\mathbf{S}}$ iff the cost of (resp., the cost share of the constraint in) the state reached after performing m is, respectively, less than, equal to, or greater than the cost of (resp., the cost share of the constraint in) $\overline{\mathbf{S}}$.

LS algorithms follow different strategies for selecting an improving move: for example, *steepest descent* chooses the move that *maximally* reduces the cost of the current state. Since steepest descent needs to consider *all* possible moves in order to choose the best one, it could be inefficient on large-scale neighbourhoods. Other algorithms focus on limited portions of the neighbourhood, by e.g. selecting the move that reduces as much as possible the cost share of a *most violated constraint* (one with the highest cost share).

Dynamic Neighbourhood Design (DND).

Given a ConSQL$^+$ specification $\langle \mathbf{S}, \mathbf{V}, \mathbf{C}\rangle$ and a state $\overline{\mathbf{S}}$, the neighbourhood of $\overline{\mathbf{S}}$ can be indirectly represented by the set of all moves that can be performed in $\overline{\mathbf{S}}$. Given our definition of *move*, the neighbourhood may easily become huge, being the set of all states reachable from the current one by changing in *all* possible ways the co-domain value associated to *any* domain value of *any* guessed function.

To overcome these difficulties, the concept of *constraint-directed neighbourhoods* has been proposed [1]. The idea, in the spirit of *constraint-based local search* [7], is to exploit the constraints also to isolate *subsets* of the moves that possess some useful properties, e.g., those that are *improving* w.r.t. one or more constraints. The presence of *symbolic information* (i.e., tuples) in the views defining the constraints allows us to improve these methods. Consider a not exists constraint $\mathsf{con} = \nexists \mathsf{V}$. Each tuple in the extension of V in the current state is a *cause* why con is violated. This knowledge (beyond the classical numeric information about the cost share of con) can be exploited during the greedy part of the search to synthesise dynamically a (typically much smaller) set of moves *explicitly designed to remove these causes* if performed on the current state. When a local minimum is reached, the same approach can be used to synthesise efficiently a worsening move (depending on the chosen LS algorithm). Fig. 1(a) gives an example of the cause why the current state violates con1 in the Workshops problem. The (single, in this example) tuple shows that the current assignment does not invite scientist $\mathsf{s3} \in$ Org to any workshop. To reduce the cost of con1 (during greedy search) *it makes no sense* to consider moves that act on a scientist not occurring in any tuple, as these moves cannot be improving for con1.

Let us formalise this reasoning starting with the simpler case where V is of type (1) and defined as $\{t \mid t \in \mathsf{V}^1 \times \cdots \times \mathsf{V}^\mathsf{p} \times \mathsf{S}^1 \times \cdots \times \mathsf{S}^\mathsf{q} \times \mathbf{T} \wedge w(t)\}$ on top of other views (V^i), guessed functions (S^j), and (fixed) DB relations (list

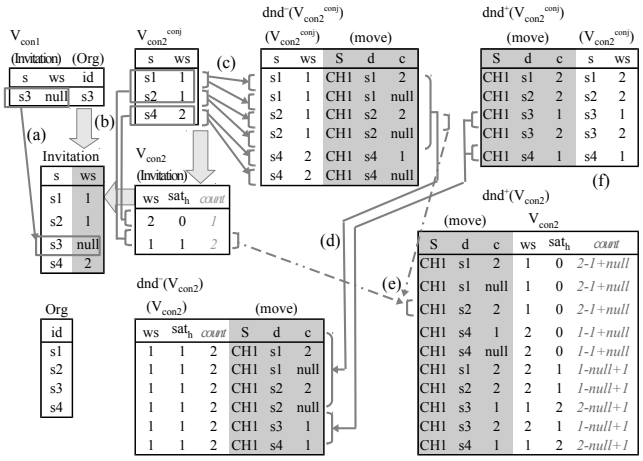


Figure 1: Example behaviour of DND on the Workhops problem (DBMS-generated data in grey areas). CH1 is a reference to the unique choose column in this problem. Thick arrows denote the view-dependency graph.

$\mathbf{T}$), where p or q may be 0. We define for V the following query (in all the queries below, $\mathsf{i} \in [1..\mathsf{p}]$ and $\mathsf{j} \in [1..\mathsf{q}]$):

$$\underset{(\mathsf{V} \text{ of type (1)})}{dnd^-(\mathsf{V})} = \left\{ \langle m, t\rangle \,\middle|\, \begin{array}{r} m \neq \text{ identity move } \wedge\, t \in \mathsf{V} \,\wedge \\ \Big(\exists \mathsf{i} \,.\, \langle m, t|_{\mathsf{V}^\mathsf{i}}\rangle \in dnd^-(\mathsf{V}^\mathsf{i}) \\ \vee\, \exists \mathsf{j} \,.\, m.S = \mathsf{S}^\mathsf{j} \wedge t|_{\mathsf{S}^\mathsf{j}}.dom = m.d\Big) \end{array} \right\}$$

Query $dnd^-(\mathsf{V})$ computes the set of moves m that *aim at improving* the cost of $\mathsf{con} = \nexists \mathsf{V}$, as the moves that would *remove at least one tuple* from V (such moves may actually reveal to be worsening for con if, e.g., they also add new tuples to V, so this reasoning is still incomplete, and will be made complete by a second novel technique called JINE, described next). Given our definition of cost share of a not exists constraint defined over a view of type (1), removing at least one tuple is a necessary (but not sufficient) condition for a move to be improving. The tuples removed by each move m are returned together with m (this comes at no cost and will be precious in the next step). This computation (as those that follow) inductively relies on the prior computation of $dnd^-(\mathsf{V}^\mathsf{i})$ of any view V^i that V depends on. In particular, a move $m = \langle \mathsf{S}, d, c\rangle$ would remove tuple t (and $\langle m, t\rangle$ will be in the result set) either because m refers to a guessed function S^j (for some j) and sub-tuple $t|_{\mathsf{S}^\mathsf{j}}$ is $\langle d, c'\rangle$ (with $c' \neq c$), or because m has been synthesised by $dnd^-(\mathsf{V}^\mathsf{i})$ (for some i). In both cases, performing m would remove a *necessary condition* for the existence of t in V. Fig. 1(c) shows the computation of dnd^- for view $\mathsf{V}^{\mathsf{conj}}_{\mathsf{con2}}$.

Dually, we can compute the set of moves that would *add at least one new tuple* to V (together with the tuples added). Such moves are useful when dealing with violated exists constraints, to cope with non-greedy steps of the search, as well as (as we will see) to compute dnd^- for views of type (2):

$$dnd^+(\mathsf{V}) = \underset{(\mathsf{V}\text{ of type }(1))}{\left\{ \langle m, t' \rangle \;\middle|\; \begin{array}{l} m \neq \text{ identity move } \wedge\, t' \text{ has the schema of } \mathsf{V} \wedge \\ t'|_{\mathbf{T}} \in \mathbf{T} \wedge w(t') \wedge \\ \forall i \,.\, \Big(\langle m, t'|_{\mathsf{V}^i} \rangle \in dnd^+(\mathsf{V}^i) \Big)^{[\dagger]} \;\vee \\ \quad \Big(t'|_{\mathsf{V}^i} \in \mathsf{V}^i \wedge \langle m, t'|_{\mathsf{V}^i} \rangle \notin dnd^-(\mathsf{V}^i) \Big) \\ \wedge \forall j \,.\, \begin{cases} t'|_{\mathsf{S}^j} \in \mathsf{S}^j & (\text{if } m.S \neq \mathsf{S}^j \vee t'|_{\mathsf{S}^j}.dom \neq m.d)) \\ t'|_{\mathsf{S}^j} = m.\langle d, c \rangle & (\text{otherwise})^{[\ddagger]} \end{cases} \\ \wedge \text{ at least one among the } {}^{[\dagger]}\text{s or the } {}^{[\ddagger]}\text{s holds} \end{array} \right\}}$$

Here, $m = \langle \mathsf{S}, d, c \rangle$ is either synthesised because it would change to c the codomain value of tuple $\langle d, c' \rangle$ (with $c' \neq c$) in all the occurrences (if any) of guessed function S, or because it has been synthesised by dnd^+ over some V^i occurring in the definition of V. For every move m, the query predicts how the relations defining V would change if m were executed and computes the set of tuples t' that would be added to V as the result of these changes. App. C contains a step-by-step explanation of this query, while Fig. 1(f) shows the output of $dnd^+(\mathsf{V}^{\mathsf{conj}}_{\mathsf{con2}})$. Note that moves assigning null to column ws of view $\mathsf{Invitation}$ are not generated, as they would not add tuples to $\mathsf{V}^{\mathsf{conj}}_{\mathsf{con2}}$.

In case V is of type (2) and defined by $\Pi_{\mathbf{g},\mathbf{a},\mathbf{e}}\left(\mathsf{V}^{\mathsf{conj}}\right)$ with $\mathsf{V}^{\mathsf{conj}}$ of type (1), tasks $dnd^-(\mathsf{V})$ and $dnd^+(\mathsf{V})$ become more complex. Recall that any tuple $t \in \mathsf{V}$ denotes a group (together with values for aggregates and value expressions), whose *composition* is given by tuples $t^{\mathsf{conj}} \in \mathsf{V}^{\mathsf{conj}}$ such that $t^{\mathsf{conj}}.\mathbf{g} = t.\mathbf{g}$. Fig. 1(b) shows the composition of groups of $\mathsf{V}_{\mathsf{con2}}$ as subsets of the tuples of $\mathsf{V}^{\mathsf{conj}}_{\mathsf{con2}}$. The having condition h is encoded as function sat_h in $\mathbf{e}$, with $\mathsf{sat}_h > 0$ for a group iff that group satisfies h (in Fig. 1, $\mathsf{sat}_h = \mathsf{count}(*) - 1$, modelling $h = \mathsf{count}(*) \geq 2$). So V contains also groups that should be filtered out by h according to the user intention.

As above, $dnd^-(\mathsf{V})$ computes the set of moves m that *aim at improving* $\mathsf{con} = \nexists\mathsf{V}$. However, given the presence of grouping and aggregation and our definition for the cost of con in this case (i.e., the sum of the values of column sat_h), a move might be improving for con also if it *adds* tuples to the conjunctive part $\mathsf{V}^{\mathsf{conj}}$ of V, as also these changes may have a positive effect at the group level (i.e., they may result in different values for the aggregates for some groups, yielding smaller values for sat_h). To this end, $dnd^-(\mathsf{V})$ synthesises moves that, by removing or adding tuples from/to $\mathsf{V}^{\mathsf{conj}}$, would alter the composition of groups $t \in \mathsf{V}$ such that $t.\mathsf{sat}_h > 0$ (i.e., groups that satisfy h in the current state):

$$dnd^-(\mathsf{V}) = \underset{(\mathsf{V}\text{ of type }(2))}{\left\{ \langle m, t \rangle \;\middle|\; \begin{array}{l} t \in \mathsf{V} \wedge \exists t' \,. \\ \quad \langle m, t' \rangle \in (dnd^-(\mathsf{V}^{\mathsf{conj}}) \cup dnd^+(\mathsf{V}^{\mathsf{conj}})) \wedge \\ \quad t'.\mathbf{g} = t.\mathbf{g} \;\wedge\; t.\mathbf{sat}_h > 0 \end{array} \right\}}$$

Fig. 1(d) shows the result of $dnd^-(\mathsf{V}_{\mathsf{con2}})$ computed from the outcomes of $dnd^-(\mathsf{V}^{\mathsf{conj}}_{\mathsf{con2}})$ and $dnd^+(\mathsf{V}^{\mathsf{conj}}_{\mathsf{con2}})$.

Dually, the following query $dnd^+(\mathsf{V})$ synthesises moves that would either add to V a new group t' or modify the composition of an existing group t, ensuring that the new/modified group will have $\mathsf{sat}_h > 0$, hence will satisfy h:

$$dnd^+(\mathsf{V}) = \underset{(\mathsf{V}\text{ of type }(2))}{\left\{ \langle m, t' \rangle \;\middle|\; \begin{array}{l} t' \text{ has the schema of } \mathsf{V} \wedge \\ \exists \langle t, t^-, t^+ \rangle \in \Big[\mathsf{V} \underset{\mathbf{g}}{\bowtie} \Big(\underset{\langle m, \mathbf{g} \rangle, \mathbf{a}}{\Pi} (dnd^-(\mathsf{V}^{\mathsf{conj}})) \underset{m, \mathbf{g}}{\bowtie} \\ \qquad \Pi_{\langle m, \mathbf{g} \rangle, \mathbf{a}}(dnd^+(\mathsf{V}^{\mathsf{conj}})) \Big) \Big] \text{ s.t.:} \\ - \; t'.\langle m, \mathbf{g} \rangle = t^-.\langle m, \mathbf{g} \rangle \vee t'.\langle m, \mathbf{g} \rangle = t^+.\langle m, \mathbf{g} \rangle \\ - \; t'.\langle m, \mathbf{g} \rangle \neq \mathsf{null} \\ - \; t'.\mathbf{a} = revise(t.\mathbf{a}, t^-.\mathbf{a}, t^+.\mathbf{a}) \\ - \; t'.\mathsf{count}(*) > 0 \\ - \; t'.\mathbf{e} = \text{eval. of expr's in } \mathbf{e} \text{ on } t'.\mathbf{g} \text{ and } t'.\mathbf{a} \\ - \; t'.\mathsf{sat}_h > 0 \end{array} \right\}}$$

where we used the right- ($\bowtie$) and full-outer ($\rtimes\ltimes$) join operators (see App. D) borrowed from relational algebra (as their expression in TRC would not be as compact).

As sat_h (which belongs to the set of value expressions $\mathbf{e}$, see also App. B) may (and typically does) depend on the values of the aggregates (which may be modified upon each move), $dnd^+(\mathsf{V})$ *incrementally revises* all of them, before re-evaluating all expressions $\mathbf{e}$ for the groups affected (or the new groups introduced). In this way it is able to predict which tuples will be added to V upon each generated move. Incremental revision of aggregates (performed by function *revise* in the query) is possible for aggregates a (see App. A) being count or sum: $revise(t.a, t^-.a, t^+.a) = t.a - t^-.a + t^+.a$ (treating nulls as 0). Furthermore, avg can be rewritten in terms of these two, and $\mathsf{count(distinct)}$ and $\mathsf{sum(distinct)}$ can be handled by additional grouping. The semantics of min and max does not always permit their full incremental revision. We can define an additional (non-incremental) step to do this job selectively. App. D contains a step-by-step explanation of the query above, using the example in Fig. 1. The following result holds (proofs omitted for lack of space):

PROPOSITION 1. *For each constraint* $\mathsf{con} = \nexists\mathsf{V}$ *(resp.* $\mathsf{con} = \exists\mathsf{V}$*), the moves that are improving for* con *if executed on the current state* all *belong to* $dnd^-(\mathsf{V})$ *(resp.* $dnd^+(\mathsf{V})$*).*

At each step of search, we perform DND depth-first in the dependency graph of the views. Depending on the LS algorithm used, we may start from the views defining all constraints (to perform, e.g., steepest descent, as the best possible move, if improving, must be improving for at least one constraint) or from the view defining a most violated constraint (if we aim at, e.g., finding a move that reduces its cost share as much as possible). Note that only DND queries needed according to the LS algorithm used must be run. E.g., to find moves improving w.r.t. constraint $\mathsf{con} = \nexists\mathsf{V}$ it is enough to run $dnd^-(\mathsf{V})$, which, if V is of type (1), does not need dnd^+ over the views that define V. As we proceed with DND bottom-up, the *set of generated moves shrinks*. The moves returned by DND at the root nodes (the views defining constraints) represent the *only* moves to be actually explored to run the chosen LS algorithm correctly (as none of the other moves would be chosen by that algorithm).

Sec. 5 experimentally shows that DND is able to filter out many moves from further evaluation, and this filtering becomes very selective as the search approaches to a solution.

Joint Incremental Neighbourhood Evaluation (JINE).

Moves synthesised by DND can be stored in a relation $\mathsf{M}(S, d, c)$, where S is a reference to a guessed function from D to C, $d \in \mathsf{D}$, and $c \in \mathsf{C}$. For each move $m \in \mathsf{M}$, during DND we have already partially computed (as a side effect) which tuples m would remove from or add to the various views. Depending on the LS algorithm used, this computation might be incomplete. For example, if the cost of moves is computed by summing their cost share on all the constraints, we may lack information, as we may not have run both dnd^- and dnd^+ on all views that define constraints. We can store the (partial) results brought by dnd^- and dnd^+ for each view V in temporary tables (resp., V^- and V^+) and make them complete with a second depth-first visit of the dependency graph of the views. In this way we achieve a complete (yet incremental) assessment of the contents of each view V. This task is performed *collectively* for all moves in M, running two queries for each view. These queries (omitted here) are similar to dnd^- and dnd^+ but

will have M as an additional input (as they evaluate, but do not generate, moves) and will not have restrictions on the effects of the moves (e.g., we do not ask that $\mathsf{sat}_h > 0$ as we did in dnd^+). We call this technique JINE. The following result holds:

PROPOSITION 2. *$\forall m \in \mathsf{M}$, $\forall \mathsf{V} \in \boldsymbol{V}$, the extension of V in the state reached after performing m is given by:*

$$\left(\overline{\mathsf{V}} \setminus \left\{t^- \mid \langle m, t^-\rangle \in \mathsf{V}^-\right\}\right) \cup \left\{t^+ \mid \langle m, t^+\rangle \in \mathsf{V}^+\right\}$$

where $\overline{\mathsf{V}}$ is the extension of V in the current state. Also, the two arguments of $\cup$ have no tuples in common.

DND and JINE perform a *joint exploration of the neighbourhood*, taking advantage of the economy of scales brought by the use of *join* operations (see forthcoming Sec. 5 for an experimental assessment) and allow us to exploit the *similarities among the neighbours* of a given state, beyond the similarity of each neighbour w.r.t. the current state, exploited by incremental techniques in classical LS algorithms.

Note that, if for a view V we have already run both $dnd^-(\mathsf{V})$ and $dnd^+(\mathsf{V})$, there is no need to run JINE on it, as it would give us no new information. Conversely, always running dnd^- *and* dnd^+ on *all* views would be overkill, as the final set M to be considered is likely to be much smaller.

Given that the two arguments of $\cup$ in the formula of Prop. 2 have no tuples in common, once we have completely populated V^- and V^+ for every view V, we can compute with *one* more query the exact cost of *all* moves in M over *all* constraints, by counting tuples or by summing up the values of sat_h columns (depending on the constraint) of the views they are defined on. If all the V are materialised (i.e., stored in tables), after having chosen the move $m \in \mathsf{M}$ to perform, we can *incrementally update* their content by deleting and adding tuples temporarily stored in V^- and V^+.

5. EXPERIMENTS

We implemented a proof-of-concept ConSQL$^+$ engine based on the ideas above. The implementation choice of using standard SQL commands for choosing, evaluating, and performing moves, *interacting transparently* with *any* DBMS, introduces a bottleneck for performance. However, it was dictated by the very high programming costs that would have arisen if extending a DBMS at its *internal layer* and/or designing storage engines and indexing data structures optimised for the kind of queries run by DND+JINE. Given the observation above, as well as the targeted novel scenario of having data modelled *independently* and stored *outside* the solving engine, and queried by non-experts in combinatorial problem solving using an extension of SQL, our purpose is not and cannot be to compete with state-of-the-art LS solvers like the one of Comet [7]. Rather, we designed our experiments to seek answers to the following questions: what is the impact of DND+JINE on: (*i*) the reduction of the size of the neighbourhood to explore; (*ii*) the overall performance gain of the greedy part of the search; (*iii*) the feasibility of the overall approach to bring combinatorial problem solving to the relational DB world?

Given our objectives, it is sufficient to focus on *single greedy runs*, until a local minimum is reached. Also, we can focus on *relative* (rather than absolute) times and can omit the numbers of moves performed, since DND+JINE do not affect the sequence of moves executed by the LS algorithm.

A first batch of experiments involved two problems: graph colouring (a compact specification with only one guessed column and one constraint, which gives clean information about the impact of our techniques on a per-constraint basis) and university timetabling (a more articulated and complex problem). Given the current objectives, we limit our attention to instances that could be handled in a reasonable time by the currently deployed system: 17 graph colouring instances with up to 561 nodes and 6656 edges (from `mat.gsia.cmu.edu/COLOR/instances.html`) and all 21 `compX` instances of the 2007 International Timetabling Competition (`www.cs.qub.ac.uk/itc2007`) for university timetabling, having up to 131 courses, 20 rooms, and 45 periods. Experiments involved steepest descent and a cause-directed version of min-conflicts, where DND synthesises all moves aiming at removing a random tuple from the view defining a random constraint.

Results (obtained on a computer with an Athlon64 X2 Dual Core 3800+ CPU, 4GB RAM, using MySQL DBMS v. 5.0.75) are in Fig. 2(a). Instances have been solved with and without DND and JINE, starting from the same random seed. Enabling DND and JINE led to *speed-ups of orders of magnitude* on *all* instances, especially when the entire neighbourhood needs to be evaluated (steepest descent), proving that the join optimisation algorithms implemented in modern DBMSs can be exploited to explore the neighbourhood collectively. DND and JINE bring advantages also when complete knowledge on the neighbourhood is not needed (min-conflicts), although speed-ups are unsurprisingly lower.

The ability of DND in *shrinking the neighbourhood* to be explored is shown in Fig. 2(b) (graph colouring, steepest descent): DND often filters out more than 30% of the moves at the beginning of search, and *constantly more than 80%* (with very few exceptions) when close to a local minimum.

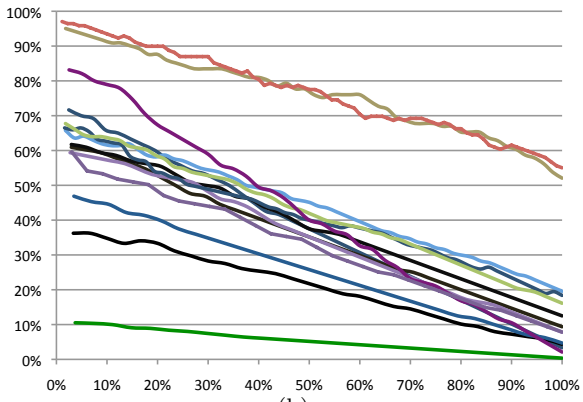
To show the overall feasibility of the approach on large instances and on problems with 'atypical' (w.r.t. CP, MP, ASP, SAT) constraints, we performed some experiments with the Workshops example of Sec. 2. We used a computer with 2 dual-core AMD Opteron 2220SE CPUs and 8 GB of RAM, using PostgreSQL DBMS 8.4, and a DB containing the entire DBLP (`www.dblp.org`) bibliography. We generated 5 instances by filling Org with 20 random scientists and using PostgreSQL-specific full-text ranking features to implement the similar(p1, p2) function ($k = 10, u = 5$). The number of coauthors of the scientists in Org, i.e., the overall number of potential invitees to the $w = 5$ planned workshops, was about 100 in each instance. The number of tuples in view unrelated_scientists_pub (the largest one to be maintained to evaluate the constraints) reached 1.3 million. As these tuples represent pairs of unrelated publications of two scientists invited to the same workshop, an encoding of this problem into an external CP/MP/SAT/ASP/LS solver would hide these high numbers inside a heavy preprocessing step, which would synthesise directly the same aggregate values computed by view unrelated_scientists (for all pairs of scientists if the programmer aims at an instance-independent preprocessing). On the other hand, the DBMS computes view unrelated_scientists only on the pairs of scientists that are invited to the same workshop in the current state, also keeping aggregate values synchronised with the actual publication data.

Notwithstanding these high numbers, the system (running steepest descent) was able to choose the best move at each iteration in about 400 seconds. Running min-conflicts required about a second per iteration. (The performance of evaluating the moves one at the time is very poor; data is omitted.) The impact of DND in the reduction of the size

Problem / Algo	Steepest-d.	Min-conflicts
Graph colouring	8×..665× (avg: 205×)	0.9×..1.6× (avg: 1.4×)
University timetabling	>15×(*) (avg: n/a)	3×..21× (avg: 8×)

(*) Evaluation of all instances without DND+JINE (except one) starved for more than 12 hours at the first iteration.

(a)



(b)

Figure 2: (a) Speed-ups (in nbr. of iterations/hour) of DND+JINE. (b) Ratio of the size of the neighbourhood synthesised by DND w.r.t. the complete neighbourhood, as a function of the state of run (0%=start, 100%=local min. reached) for graph colouring (one curve per instance whose greedy run terminated within the time-limit).

of the neighbourhood is very similar to the graph colouring case: ∼30% (at the beginning of the run) to ∼90% (when close to a local minimum) of the moves were ignored when choosing the best move to perform (steepest descent).

6. CONCLUSIONS

Although there are attempts to integrate DBs and NP-hard problem solving (see, e.g., constraint DBs [4], which however focus on representing *implicitly* and querying a possibly *infinite* set of tuples), to our knowledge ConSQL and ConSQL$^+$ are the only proposals to provide the *average DB user* with effective means to access combinatorial problem modelling and solving techniques without the intervention of specialists. Our framework appropriately behaves in concurrent settings, seamlessly reacting to changes in the data, thanks to the flexibility of LS coupled with change-interception mechanisms, e.g., triggers, that are well supported by DBMSs. This makes our approach fully respect data access policies of information systems with concurrent applications: a solution to the combinatorial problem is represented as a *view* of the data, which is *dynamically kept up-to-date* w.r.t. the underlying (evolving) DB.

This paper is of course only a step in this direction. In particular, the performance of the current implementation can be drastically improved by a tighter integration with the DBMS plus the design of storage engines and indexing data structures to support the queries required by DND and JINE. Also, *parallelism* can be heavily exploited: information systems are often deployed in data centres and implemented as DBs replicated in multiple copies. Divide-and-conquer techniques are exploited during querying, splitting queries into smaller pieces to be run on different servers. DND and JINE can take advantage of replication, as they can be launched simultaneously on different views, as long as their dependency constraints are satisfied. All this would allow a carefully-engineered implementation to scale much better toward large DBs.

7. REFERENCES

[1] M. Ågren, P. Flener, and J. Pearson. Revisiting constraint-directed search. *Information and Computation*, 207(3):438–457, 2009.

[2] M. Cadoli and T. Mancini. Combining Relational Algebra, SQL, Constraint Modelling, and local search. *Theory and Practice of Logic Programming*, 7(1&2):37–65, 2007.

[3] H. H. Hoos and T. Stützle. *Stochastic Local Search, Foundations and Applications*. Elsevier/Morgan Kaufmann, Los Altos, 2004.

[4] G. M. Kuper, L. Libkin, and J. Paredaens, editors. *Constraint Databases*. Springer, 2000.

[5] T. Mancini and M. Cadoli. Exploiting functional dependencies in declarative problem specifications. *Artificial Intelligence*, 171(16–17):985–1010, 2007.

[6] G. Özsoyoğlu, Z. Özsoyoğlu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Transactions on Database Systems*, 12(4):566–592, 1987.

[7] P. Van Hentenryck and L. Michel. *Constraint-Based Local Search*. The MIT Press, 2005.

APPENDIX

A. SYNTAX OF SQL TO DEFINE VIEWS

SQL is a wide language and its full coverage is out of the scope of this paper. *Views* on the data are defined through *queries*, which (for what is of interest to us) may have the form (1) or (2). As an example, the query select ... of the view defining constraint con1 in the Workshops problem returns the set of tuples from the Cartesian product between Invitation and Org (in general, the tuples in the Cartesian product of the relations in the from clause) that satisfy the condition expressed by the where clause, hence yielding one tuple for each scientist in Org that is *not* assigned to a workshop (i.ws is null). Clause i.invitee = s.id (*join condition*) filters out all pairs (i ∈ Invitation, s ∈ Org) of tuples that refer to different scientists. The select clause contains a list of the columns to be returned ('*' meaning all columns). The tuples returned by the query represent scientists that *violate* the not exists constraint.

Constraint con2 in the Workshops problem has the form (2). The query defining it considers all the tuples i in view Invitation that satisfy the where condition (scientists actually invited to a workshop). Then, the group by clause groups together the tuples that refer to the same workshop, and only groups having a number of tuples (count(*)) that violates the constraint are returned.

SQL defines different *aggregate* functions, namely count(), sum(), avg(), min(), max(), which compute respectively the number, the sum, the average, the minimum, and the maximum value of the argument expression evaluated on all the tuples of each group (if the group by clause is omitted, then all the tuples belong to a single group). The distinct modi-

fier would skip identical tuples to be considered more than once.

B. DISTANCE TO FALSIFICATION OF h

We claimed that sat_h can be any function that is > 0 $(= 0)$ if a group satisfies (does not satisfy) the having condition h. This function acts as a *heuristic* and can be regarded as the dual of *violation cost* in CBLS [7]. In particular, we propose (and have experimented with) the following function, called *distance to falsification* of h.

If $h = \mathsf{true}$, then $\mathsf{sat}_h = \mathsf{count}(*)$; if $h = \mathsf{false}$, then $\mathsf{sat}_h = 0$. Otherwise:

- if h is an atom of the form $x < y$, $x \neq y$, or $x = y$ (with x, y being constants or references to columns in $\mathbf{g}$ or $\mathbf{a}$), then sat_h is, respectively, $\max(0, y - x)$, $|x - y|$, and if $x = y$ then 1 else 0;
- if h is $h' \vee h''$, then $\mathsf{sat}_h = \mathsf{sat}_{h'} + \mathsf{sat}_{h''}$;
- if h is $h' \wedge h''$, then $\mathsf{sat}_h = \min(\mathsf{sat}_{h'}, \mathsf{sat}_{h''})$.

Intuitively, the higher sat_h for a group of a view V of type (2), the higher the estimated amount of change that needs to be done on the composition of that group to falsify h. With this definition, queries $dnd^-(\mathsf{V})$ and $dnd^+(\mathsf{V})$ become simpler in the frequent cases where the having condition is simple. As an example, if $h = \mathsf{count}(*) > k$ (with k being a constant), to remove a tuple from V whose $\mathsf{sat}_h = \max(0, \mathsf{count}(*) - k) > 0$, we have that $dnd^-(\mathsf{V})$ does not need to run $dnd^+(\mathsf{V}^{\mathsf{conj}})$, as aiming at adding tuples to $\mathsf{V}^{\mathsf{conj}}$ is not a good strategy to reduce the number of tuples that compose any group of V and reduce sat_h.

C. *DND⁺ FOR VIEWS OF TYPE (1)*

Below we give a step-by-step explanation of query $dnd^+(\mathsf{V})$ in case view V is of type (1). The query returns pairs $\langle m, t' \rangle$, where m is a move and t' is a tuple that m would add to the content of V. Any such t' can be split into $\langle t'_{\mathsf{V}^1}, \ldots, t'_{\mathsf{V}^\mathsf{p}}, t'_{\mathsf{S}^1}, \ldots, t'_{\mathsf{S}^\mathsf{q}}, t'_{\mathbf{T}} \rangle$, with one sub-tuple for each relation in the Cartesian product defining V. For m to insert t' into V, the execution of m must change the V^i and the S^j in such a way that all the sub-tuples above will exist in their respective relations. In particular: (i) for each occurrence S^j (if any) of $m.S$ (the guessed function that m acts on), if $t'_{\mathsf{S}^\mathsf{j}}.dom = d$, then $t'_{\mathsf{S}^\mathsf{j}}.codom = c$, as $\langle d, c \rangle$ is the only tuple that m would introduce in S in replacement of $\langle d, c' \rangle$ for some $c' \neq c$ (case [‡]); (ii) all the other S^j (that would remain unchanged after the execution of m) must already contain $t'_{\mathsf{S}^\mathsf{j}}$; (iii) for each V^i, either $t'_{\mathsf{V}^\mathsf{i}}$ is already in V^i and will not be removed by m ($\langle m, t'_{\mathsf{V}^\mathsf{i}} \rangle \notin dnd^-(\mathsf{V}^\mathsf{i})$) or it will be added to V^i by m itself (case [†]); (iv) $t'_{\mathbf{T}}$ belongs to $\mathbf{T}$ (which contains only DB relations that will not change if m is performed); (v) t' satisfies the where condition w, which is necessary for it to be added to V. The last condition in the formula ensures that t' does not yet belong to V.

D. *DND⁺ FOR VIEWS OF TYPE (2)*

Below we give a step-by-step explanation of query $dnd^+(\mathsf{V})$ in case view V is of type (2). Its most complex part consists in the expression '[...]', which may be regarded as a dual of the *differentiable data-structures* in CBLS [7]. To explain its semantics in a more clear way, we follow the example in Fig. 1(e), which shows the result of $dnd^+(\mathsf{V}_{\mathsf{con2}})$ computed from $\mathsf{V}_{\mathsf{con2}}$ and the outcomes of $dnd^-(\mathsf{V}^{\mathsf{conj}}_{\mathsf{con2}})$ and $dnd^+(\mathsf{V}^{\mathsf{conj}}_{\mathsf{con2}})$. $\mathsf{V}_{\mathsf{con2}}$ has $\mathbf{g} = \{\mathsf{ws}\}$, $\mathbf{a} = \{\mathsf{count}(*)\}$, and $\mathsf{sat}_h = \mathsf{count}(*) - 1$ (modelling $h = \mathsf{count}(*) \geq 2$).

(i) Subexpression $\Pi_{\langle m, \mathbf{g} \rangle, \mathbf{a}}(dnd^-(\mathsf{V}^{\mathsf{conj}}))$ groups the tuples produced by $dnd^-(\mathsf{V}^{\mathsf{conj}})$ according to the move m *and* the grouping attributes of V. Aggregates $\mathbf{a}$ are evaluated for each such group. In the example this computes, for every move m synthesised by $dnd^-(\mathsf{V}^{\mathsf{conj}}_{\mathsf{con2}})$, the number of tuples ($\mathsf{count}(*)$) that m would remove from *each* group of $\mathsf{V}_{\mathsf{con2}}$. Tuples t^- produced by this sub-expression will be of the form: $\langle m, t.\mathbf{g}, t.\mathbf{a} \rangle$.

(ii) Subexpression $\Pi_{\langle m, \mathbf{g} \rangle, \mathbf{a}}(dnd^+(\mathsf{V}^{\mathsf{conj}}))$ makes an analogous computation on $dnd^+(\mathsf{V}^{\mathsf{conj}})$. In the example this computes, for every move m synthesised by $dnd^+(\mathsf{V}^{\mathsf{conj}}_{\mathsf{con2}})$, the number of tuples that m would add to *each* group of $\mathsf{V}_{\mathsf{con2}}$. Tuples t^+ produced by this sub-expression will be of the form: $\langle m, t.\mathbf{g}, t.\mathbf{a} \rangle$.

(iii) The full outer join ($\bowtie_{m,\mathbf{g}}$) returns the set of *pairs* of tuples t^- and t^+ that agree on the move (m) *and* the group $(\mathbf{g})$. If there exist tuples t^- (resp. t^+) referring to a $\langle$move, group$\rangle$ pair for which no matching counterpart t^+ (resp. t^-) exists, the full outer join extends t^- (resp. t^+) with nulls. The result set consists of tuples of the form $\langle t^-, t^+ \rangle$ such that $t^-.m = t^+.m$ and $t^-.\mathbf{g} = t^+.\mathbf{g}$ (unless one between t^- and t^+ is null).

(iv) The right outer join ($\bowtie_{\mathbf{g}}$) matches the tuples t (groups) currently in V with the tuples $\langle t^-, t^+ \rangle$ above, by considering only $\mathbf{g}$ as the matching criterion (as moves are not mentioned in V). The result set consists of tuples of the form $\langle t, \langle t^-, t^+ \rangle \rangle$ whose non-null components refer to the same group. If any tuple $\langle t^-, t^+ \rangle$ has no counterpart t in V, the right outer join enforces a match of $\langle t^-, t^+ \rangle$ with nulls. Intuitively, any tuple $\langle t, \langle t^-, t^+ \rangle \rangle$ returned by expression '[...]' encodes a group (stored in $t.\mathbf{g}$, $t^-.\mathbf{g}$, and $t^+.\mathbf{g}$, ignoring nulls) already existing (if t is not null) or that *will* exist in V (if t is null), together with a move (stored in $t^-.m$ and $t^+.m$, ignoring those being null) and three values for each aggregate in $\mathbf{a}$: the current values (in $t.\mathbf{a}$), and the values computed on the tuples that would be removed (in $t^-.\mathbf{a}$) and added (in $t^+.\mathbf{a}$) from/to $\mathsf{V}^{\mathsf{conj}}$ if the move is executed.

From each $\langle t, \langle t^-, t^+ \rangle \rangle$, $dnd^+(\mathsf{V})$ predicts which tuples t' will be added to V if the move is executed. These tuples $t' = \langle t.\mathbf{g}, t.\mathbf{a}, t.\mathbf{e} \rangle$ will have the same values for $t.\mathbf{g}$ as t, t^-, and t^+, but will have as values for the aggregates the output of $revise(t.\mathbf{a}, t^-.\mathbf{a}, t^+.\mathbf{a})$.

As an example, the third tuple in the result set of $dnd^+(\mathsf{V}_{\mathsf{con2}})$ is computed as follows (see Fig. 1(e)): from $dnd^-(\mathsf{V}^{\mathsf{conj}}_{\mathsf{con2}})$ we know that move $m = \langle \mathsf{CH1}, \mathsf{s2}, 2 \rangle$ would remove tuple $t^- = \langle \mathsf{s2}, 1 \rangle$ from $\mathsf{V}^{\mathsf{conj}}_{\mathsf{con2}}$. From $dnd^+(\mathsf{V}^{\mathsf{conj}}_{\mathsf{con2}})$ we know that m would add no tuple with $\mathsf{ws} = 1$. Hence executing m would (among other things) change the composition of the tuple (group) with $\mathsf{ws} = 1$ in $\mathsf{V}_{\mathsf{con2}}$. In particular, the value for the aggregate $\mathsf{count}(*)$ for this group (currently 2) would be reduced by 1 (the value of $\mathsf{count}(*)$ as computed on the set of tuples removed by m). Note that, as the same move would change also the composition of the group with $\mathsf{ws} = 2$ (in particular it would add tuple $\langle \mathsf{s1}, 2 \rangle$ to $\mathsf{V}^{\mathsf{conj}}_{\mathsf{con2}}$), it occurs again in the result set of $dnd^+(\mathsf{V}_{\mathsf{con2}})$, this time paired with tuple having $\mathsf{ws} = 2$ and $\mathsf{sat}_h = 1$, as the number of tuples in group with $\mathsf{ws} = 2$ upon the execution of move m would become 2.

In general a move could both remove and add tuples from/to a group: in this case the new value for $\mathsf{count}(*)$ would be computed accordingly (see the definition of function *revise* in the paper).