Data Coordination: Supporting Contingent Updates

Michael Lawrence
Dept. of Computer Science
University of British Columbia
Vancouver, Canada
mklawren@cs.ubc.ca

Rachel Pottinger
Dept. of Computer Science
University of British Columbia
Vancouver, Canada
rap@cs.ubc.ca

Sheryl Staub-French
Dept. of Civil Engineering
University of British Columbia
Vancouver, Canada
ssf@civil.ubc.ca

ABSTRACT

In many scenarios, a contingent data source may benefit by coordinating with external heterogeneous sources upon which it depends. The administrator of this contingent source needs to update it when changes are made to the external base sources. For example, when a building design is updated, the contractor's cost estimate must be updated, too. The goal of *data coordination* is to update a contingent source, C, based on changes to an independently maintained base source, B.

This paper introduces a data coordination system which allows C to coordinate its data without imposing significant requirements on B. Our system uses declarative mappings between B and C and performs coordination in two stages *View Differencing* — finding changes to an intermediate view of B based on its mapping to C, and *Update Translation* — translating the view differencing result into updates on C. We present and evaluate novel solutions to both stages and demonstrate their feasibility on real world problems.

1. INTRODUCTION

In many applications, a data source may need to be coordinated with heterogeneous data sources on which it depends. The administrator of this contingent source needs to ensure it is up to date and consistent with the latest data provided by these base sources. A significant challenge when coordinating between autonomous sources is that the base sources only participate in the process to the extent of providing access to their data: the entire coordination process is the responsibility of the contingent source's administrator.

For example, a precise cost estimate is critical for decision making during a building's design phase. Updated building designs are periodically provided by the project's architect, while the cost estimate is created and maintained by the contractor. Efficiently determining how the building design changes and what effects these changes have on the cost estimate are significant challenges for the contractor. We refer to this problem of updating an existing data source based on changes in independently maintained data sources as *data coordination*. Other data coordination scenarios arise; for example, a bus company may need route data to coordinate with the road network data provided by the city, a national department of finance may need to coordinate economic indicators with data provided by local statistics gathering bodies, or a ferry operator may need to coordinate its schedule information based on data from a marine forecast website.

More specifically, data coordination occurs between two autonomous but related data sources, B and C. In particular, one provider (e.g. the building architect) maintains a base relational database, B, while another provider (e.g. the contractor) maintains its own, separate, contingent relational database, C. This is in contrast to the traditional problem of data integration [6], which focuses on querying heterogeneous data; and coordination in peer to peer databases [10, 13, 16], where a number of participants use a common system to share data. We address coordination scenarios where there is a lack of system wide collaboration: C is interested in coordinating with data made available by an autonomous source B, with B periodically changing without notice due to external factors (e.g. the architect updates the design due to client requests). Our proposal for data coordination is a system which can be used by the administrator of C to detect relevant changes in B and compute a set of updates to C in order to maintain consistency. This presents the following new challenges from previous work on data coordination [2, 9, 16] for several reasons: 1) the loose coupling of B and C means that C will not receive notifications when and how B has changed; 2) the base/contingent relationship constrains coordination to only modifications on C, and 3) the contingent source C is interested in resolving a final ground instance and must be careful to avoid changes which result in other side-effects.

In this paper, we propose a data coordination system which represents relationships between B and C using declarative mapping constraints of the form $q_B = q_C$.[1] Such constraints permit the straightforward expression of complex relationships between sources. Our system monitors B for changes which impact C and performs best-effort coordination using a two-phased approach: 1) *View Differencing*, which finds the changes to the view defined by $q_B = q_C$, and 2) *Update Translation*, which translates these into changes to C, allowing its administrator to review and select the

[1]A much altered definition of data coordination appears in [18], which takes a substantially different approach.

Proceedings of the VLDB Endowment, Vol. 4, No. 11

id	type	length	height	area
0	Column	1	1	1
1	Wall	10	3	27
2	Wall	4	3	12

(a) Component

cid	name	thickness
0	column concrete	240
0	rebar	200
1	light concrete	300
1	drywall	15
2	heavy concrete	200

(b) Material

Figure 1: Example building design $\mathbb{B}_t$.

code	qty
2220.00	1
3100.04	30
9250.12	27
9250.14	12
9250.02	27

(a) ProjectItems

code	desc	rate
3100.04	column formwork	6.50
3100.08	column rebar	165.00
9250.12	light concrete	25.00
9250.14	heavy concrete	35.00
9250.02	drywall	3.50
9250.03	190mm studs	12.00

(b) ItemRates

Figure 2: Example cost estimate $\mathbb{C}_t$.

suggested changes.

We make the following specific contributions:

- We define data coordination and decompose the problem into view differencing and update translation.
- We propose and experimentally evaluate two algorithms for view differencing.
- We propose algorithms for view update translation based on data exchange formalisms [7] and incomplete information [8] which find all possible translations, and guide the user to select a unique solution.
- We experimentally evaluate our update translation algorithms on real-world data, demonstrating their feasibility and the benefit of practical heuristics.

The remainder of this section presents an example which outlines our data coordination approach. Section 2 introduces our data coordination system and formally defines the specific problems we address. Section 3 outlines and presents experimental results for two approaches to view differencing; Section 4 describes our solutions for update translation, followed by experimental results in Section 5, related work in Section 6, and conclusions in Section 7.

1.1 Motivating Example

EXAMPLE 1. *A Building Designer's data has schema B containing relations Component and Material. An instance $\mathbb{B}_t$ at time t is shown in Figure 1. A Cost Estimator's data has schema C containing relations ProjectItems and ItemRates; they have created an instance $\mathbb{C}_t$ in Figure 2 for the building design $\mathbb{B}_t$. The Cost Estimator has also created a*

name/desc	ar/qty
light concrete	27
heavy concrete	12
drywall	27

Figure 3: $\mathbf{V}_B$, the evaluation of q_B on building design $\mathbb{B}_t$ (Figure 1) equals $\mathbf{V}_C$, the evaluation of q_c on cost estimate $\mathbb{C}_t$ (Figure 2).

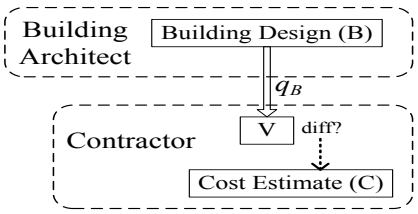

Figure 4: The Cost Estimator stores view V on the building design. When V changes, we translate these changes into a set of changes on C.

declarative mapping constraint $q_B = q_C$ defined by the following queries

$$V_B(nm,\ ar) :- Component(id,\ t,\ l,\ h,\ ar) \wedge Material(id,\ nm,\ th)$$

$$V_C(desc,\ qty) :- ProjectItems(c,\ qty) \wedge ItemRates(c,\ desc,\ r) \wedge c = 9250.*$$

expressing the relationship between the walls in the building design and the cost items pertaining to walls in the cost estimate. The data of Figures 1 and 2 satisfy this mapping because evaluating q_B on $\mathbb{B}_t$ yields the same result as evaluating q_C on $\mathbb{C}_t$ (Figure 3). □

Our data coordination system would allow the Cost Estimator to coordinate C with building design B. The Cost Estimator describes how to connect to B and the desired mapping to be maintained. Our system uses a "pull" model of coordination, which is appropriate for scenarios where B is unwilling to make special provisions to support coordination of contingent sources (for example if B is a dataset available through a public URL). Our system stores a materialized view $\mathbf{V}$ on the building design as defined by the query q_B. It monitors B, and when there are changes to $\mathbf{V}$, it computes a set of possible changes to C as shown in Figure 4.

A major challenge is handling ambiguity. In Example 1 if the building architect removes the drywall material layer from Wall 1, it will cause (drywall, 27) to be deleted from $\mathbf{V}$, which could be reflected in the cost estimate by deleting from ProjectItems, ItemRates, or both. A more challenging example is if the building architect adds paint layers to Walls 1 and 2, causing (beige paint, 88) to be inserted into $\mathbf{V}$. This will require inserting into both ProjectItems and ItemRates, but we cannot tell which value to use for its rate or code (although q_C tells us $c = 9250.*$). If there is an existing "beige paint" tuple in ItemRates, then there is a good chance that we can use its code and insert a single tuple into ProjectItems. However, real world construction data is ambiguous, containing duplicate descriptions and codes.

Because of these ambiguities, both examples remain non-trivial even if $\mathbb{B}_t$ were given.

Our approach to data coordination is to provide a concise description of *all possible* solutions, given the changes and mapping constraint. This approach is motivated by the user's need to resolve a final ground instance of C. It allows them to retain control over how coordination is performed, and provides opportunities to overlay situation-dependent ranking strategies.

2. DATA COORDINATION

Our system uses declarative mapping constraints of the form $q_B = q_C$ to express the coordination relationship of B to C. As emphasized in [15], using declarative mappings allows the mapping designer to focus on *what* it means to be coordinated, without concern to *how* coordination is achieved. In this section, we formally define the specific problems addressed in the implementation of our coordination system.

A database schema A is a set of relations, $\{A_1, \ldots A_k\}$, and an instance $\mathbb{A}$ of schema A consists of a set of relation instances $\{\mathbf{A}_1, \ldots \mathbf{A}_k\}$. A query q over A is a function from instances of A to instances of a view schema, which we denote V, writing the query as $V(\overline{x}) :- q(\overline{x}, \overline{y})$, where $\overline{x}$ is the set of attributes of V, q is a conjunction of relational predicates, and $\overline{y}$ is the set of attributes in these relational predicates which are not in $\overline{x}$. In query q_B of Example 1, $\overline{x}$ consists of *nm, ar*, while $\overline{y}$ consists of *id, t, l, h, th*. When it is clear from the context, we also use $\mathbf{V} = q(\mathbb{A})$ to denote the set of tuples which results from evaluating q on $\mathbb{A}$. Our mappings are defined as follows.

Definition 1. (Mapping Constraint). *A mapping constraint between relational schemas B and C is an expression of the form $q_B = q_C$, where q_B is a query over B, q_C is a query over C, and both q_B and q_C have the same number and types of attributes in $\overline{x}$. A mapping constraint $q_B = q_C$ is said to be* satisfied *with respect to instances $\mathbb{B}$ and $\mathbb{C}$ if $q_B(\mathbb{B}) = q_C(\mathbb{C})$.* □

Our approach in general is for conjunctive queries q_B and q_C, although we note that one of the proposed methods permits arbitrary queries for q_B, and that extending to queries with arithmetic comparisons is a topic of our ongoing work. The problem of creating mappings is outside the scope of this paper. It has recently been argued [5] that precisely engineered mappings are needed in many applications, and we leave this responsibility to a domain expert (e.g. the Cost Estimator), who may draw assistance from automated tools such as Clio [20].

Data coordination involves updating the contingent source, C, in response to changes in the base source, B. We define an update as follows:

Definition 2. (Update) *An update to an instance $\mathbb{A}$ is a set of pairs of tuple sets $(\mathbf{A}_i^+, \mathbf{A}_i^-)$ for each relation A_i. The result of applying an update $(\mathbb{A}^+, \mathbb{A}^-)$ to $\mathbb{A}$ is $\{(\mathbf{A}_1 \cup \mathbf{A}_1^+) - \mathbf{A}_1^-, \ldots (\mathbf{A}_k \cup \mathbf{A}_k^+) - \mathbf{A}_k^-\}$ We assume that for all i $\mathbf{A}_i^- \subseteq \mathbf{A}_i$ and $\mathbf{A}_i^+ \cap \mathbf{A}_i^- = \emptyset$.* □

Since B is autonomous and does not push change notification to C, our system must monitor B for changes. The monitoring policy depends on the freshness requirements of Cs administrator — for example, our system might check B every day or every week. The central problem addressed by our system occurs each time a change is detected in B:

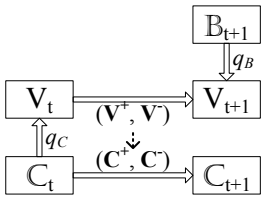

Figure 5: Data coordination solution overview.

Definition 3. (Data Coordination Problem) *Given*

1. *Schema B, database instance $\mathbb{B}_{t+1}$*
2. *Schema C, database instance $\mathbb{C}_t$*
3. *A mapping constraint $q_B = q_C$ s.t. there exists some $\mathbb{B}_t$ s.t. $q_B(\mathbb{B}_t) = q_C(\mathbb{C}_t)$*

Find all possible updates $(\mathbb{C}^+, \mathbb{C}^-)$ such that when applied to $\mathbb{C}_t$, the result is a $\mathbb{C}_{t+1}$ such that mapping $q_B(\mathbb{B}_{t+1}) = q_C(\mathbb{C}_{t+1})$ is satisfied. □

2.1 Our Approach

In our approach to data coordination, the mapping $q_B = q_C$ captures the subset of B's data which, were it to change, should have an effect on C. We can thus focus our data coordination efforts by examining only what has changed with respect to the view defined by q_B. As discussed above, our system implements coordination by storing a materialized view $\mathbf{V}$ on B defined by q_B. C is coordinated with B when $\mathbf{V}$ is identical to the materialized view $\mathbf{V}_C$ resulting from applying q_C to C. Hence, if $\mathbf{V}$ is unavailable, it can be recomputed from C.

With respect to the data coordination problem defined above, at time $t+1$, C will have a materialized view $\mathbf{V}_t$ on B at the time t when it was last coordinated, thus providing a record of the exact changes to B which are pertinent to C. We use $\mathbf{V}_t$ to break data coordination into the following two stages:

1. *(View Differencing)* Given $\mathbf{V}_t$ and $\mathbb{B}_{t+1}$, find updates $(\mathbf{V}^+, \mathbf{V}^-)$ to $\mathbf{V}_t$ s.t. $q_B(\mathbb{B}_{t+1}) = (\mathbf{V}_t \cup \mathbf{V}^+) - \mathbf{V}^-$.
2. *(Update Translation)* Given $(\mathbf{V}^+, \mathbf{V}^-)$, find $(\mathbb{C}^+, \mathbb{C}^-)$ s.t. $q_B(\mathbb{B}_{t+1}) = q_C\big((\mathbb{C}_t \cup \mathbb{C}^+) - \mathbb{C}^-\big)$

This approach is illustrated in Figure 5.

In Section 3 we propose two methods to view differencing: 1) a practical and straightforward method which finds $(\mathbf{V}^+, \mathbf{V}^-)$ by materializing $\mathbf{V}_{t+1}$ and comparing it with $\mathbf{V}_t$; and 2) an approach which is adapted from incremental view maintenance. We experimentally investigate important questions about the behavior and performance of these methods.

Update translation is the more challenging stage and represents the bulk of this paper. The challenge with update translation is ambiguity: there may be many possible translations of $(\mathbb{C}^+, \mathbb{C}^-)$ for a given $(\mathbf{V}^+, \mathbf{V}^-)$. This challenge is compounded by the need for exact equality: how do we find a translation which avoids side-effects? (spurious insertions or deletions.)

The problem is related to data exchange [7]; however, data exchange uses open world semantics and considers the generation of a target source from scratch (as opposed to updating an existing source.) Hence their constraints are more relaxed than in data coordination: the view need only be a *subset* of a given query on a database instance. Our approach is to translate insertions and deletions separately. We combine methods from data exchange, and use incomplete information to define a universal solution for insertion translation which avoids side-effects (Section 4.1). We then formally define the set of minimal, feasible deletion translations and propose a constrained search method which finds them (Section 4.2). In Appendix C, we discuss how these translations are combined into the final result in our system, as well as a number of practical enhancements.

3. VIEW DIFFERENCING

3.1 Materialize and Compare

Our first proposed approach to view differencing is to materialize $\mathbf{V}_{t+1}$ by evaluating q_B on $\mathbb{B}_{t+1}$, and generate $(\mathbf{V}^+, \mathbf{V}^-)$ by direct comparison with $\mathbf{V}_t$. We call this *Materialize and Compare (MAC)*. We briefly describe MAC here; details are in Appendix A. MAC computes $(\mathbf{V}^+, \mathbf{V}^-)$ by scanning $\mathbf{V}_t$ and $\mathbf{V}_{t+1}$ in parallel. The process is similar to a sorted merge: the next tuples from each of $\mathbf{V}_t$ and $\mathbf{V}_{t+1}$ are compared via a total ordering; if one is smaller than the other, it is added to $\mathbf{V}^-$ or $\mathbf{V}^+$ (if it came from $\mathbf{V}_t$ or $\mathbf{V}_{t+1}$ respectively).

MAC for view differencing has several benefits:

1. Since we are simply evaluating q_B, there is no limit to the types of queries which can be used. This allows for mappings expressing very sophisticated relationships, e.g. aggregation, conditionals, negation etc.

2. Minimal support is required from source B. As stated in Section 1, one goal is to support *autonomous* sources. For the MAC algorithm, we only require B to evaluate queries on the its current data instance.

3.2 Incremental View Maintenance

Our second approach to view differencing is based on an adaption of incremental view maintenance (IVM) techniques, hence we call it *IVM for View Differencing (IVM-VD)*. IVM [11] updates a materialized view given changes to the base relations without recomputing that view from scratch. IVM, in addition to the new base relations ($\mathbb{B}_{t+1}$) and existing materialized view ($\mathbf{V}_t$), also requires access to the old base relations (i.e. $\mathbb{B}_t$) and the updates to the base relations. Its result is $\mathbf{V}_{t+1}$.

We use the counting algorithm of [12], which requires a count of the distinct derivations of tuples in the view. Briefly, for a query with k predicates, the counting algorithm evaluates the additive union of $2k$ separate query evaluations (delta rules), each replacing one of the relational predicates with its updates. Implementing this algorithm within the context of an autonomous data coordination system requires two major changes: 1) A local query rewriting to obtain tuple counts from B without requiring any special facilities; and 2) a modified implementation of the additive union operator which allows the view updates to be isolated during the computation of $\mathbf{V}_{t+1}$. The full IVM-VD algorithm and our solutions to its implementation challenges for data coordination are given in Appendix A.

The following section describes the results of experimentally comparing MAC to IVM-VD. While the heuristic of inertia [11] states that IVM-VD should be preferred for small updates, there are a number of practical reasons to prefer MAC. We consider the exploration of such trade-offs an important contribution; to our knowledge, there has been no prior comparison of the performance benefits of incremental view maintenance as opposed to materializing from scratch.

3.3 Comparing MAC to IVM-VD

IVM-VD should be more efficient than MAC when the updates are small and recomputing the view from scratch is expensive. However, there are several possibly limiting factors: IVM-VD makes a trade-off between the number of queries evaluated, and the cost of each of these evaluations. The reasoning is that it should be more efficient to evaluate a large number of delta rules, each involving a term ($\mathbf{B}_i^+$ or $\mathbf{B}_i^-$) significantly smaller than the base relations. Of course, the performance advantage gained by this trade-off depends on the number of relations in the query q_B and the number of inserted/deleted tuples for each of these relations. We explore the scope of this trade-off in depth in Appendix A.1; our findings are as follows:

1. Both MAC and IVM-VD are feasible, offering acceptable performance under realistic conditions.

2. IVM-VD's execution time varies exponentially with the number of joined relations in q_B, but only if most of these relations are updated. MAC is fairly resilient to changes in the problem parameters; its performance depends mostly on the size of $\mathbf{V}_t$ and $\mathbf{V}_{t+1}$. Both algorithms are linear in the size of the instance.

3. MAC appears to be favorable to IVM-VD when q_B contains more than a few joins, and the size of the updates is greater than 2.5% of the database instance.

4. UPDATE TRANSLATION

After our system has computed a set of updates $(\mathbf{V}^+, \mathbf{V}^-)$ to $\mathbf{V}$, it attempts to translate these into a set of updates on $\mathbb{C}$. A satisfactory translation requires that evaluating the query q_C on the updated $\mathbb{C}$ results in the updated view. The problem is extremely challenging because, as shown in [3], unambiguous translation is only possible under extremely limited conditions. Update translation for data coordination additionally poses some new challenges for the following reasons:

1. Our solution must be *exact*. That is, our mapping constraints demand equality (bi-directional logical implications), as opposed the more common subset constraints (uni-directional implication.) This means that special attention must be paid to side-effects; we need to ensure translated insertions $\mathbb{C}$ do not cause additional, spurious insertions into $\mathbf{V}_C$

2. Because changes to $\mathbb{C}$ are dictated by the given changes to $\mathbf{V}$ (which are dictated by changes to $\mathbb{B}$), updates to $\mathbb{C}$ must not assert additional changes to $\mathbf{V}$.

3. We need to consider *sets* of insertions and deletions, and the interaction between them. I.e. translating each

insertion or deletion separately and taking the union will produce incorrect results.

Our methods for translation of insertions $\mathbf{V}^+$ make use of the tuple generating dependency (tgd) chase [1]. In doing so, we generate an uncertain translation containing free variables. A similar approach in a peer to peer setting is taken in Youtopia [9, 16], where individual insertions and deletions are propagated amongst a peer data sharing network. We then find constraints on these variables by using algebra of incomplete information [8]. This translation is presented to the administrator of C, who should have some domain knowledge which allows them to fill in the missing values.

4.1 Insertion Translation

We begin by reformulating the insertion translation problem using first order logic. The constraint $\mathbf{V} = q_C(\mathbb{C})$ can be stated as a pair of tgds of the form

$$\forall \overline{x}\big(V(\overline{x}) \rightarrow \exists \overline{y} q_C(\overline{x}, \overline{y})\big) \quad (1)$$

$$\bigwedge \quad \forall \overline{x}, \overline{y}\big(q_C(\overline{x}, \overline{y}) \rightarrow V(\overline{x})\big) \quad (2)$$

where q_C is a conjunction of relational atoms (not an algebraic function operating on database instances.) Equation (1) algebraically corresponds to $\mathbf{V} \subseteq q_C(\mathbf{J})$, while (2) corresponds to $\mathbf{V} \supseteq q_C(\mathbf{J})$. Insertions into $\mathbf{V}$ amount to additional assertions of $V(\overline{x})$ (i.e. new values for $\overline{x}$ s.t. $V(\overline{x})$ is true) Hence, they can *only* cause violations of (1) and cannot impact (2). Likewise, deletions amount to assertions of $\neg V(\overline{x})$ for some new values of $\overline{x}$, and hence can only violate (2) and will not impact (1).

Our method for translation of insertions takes advantage of this with two steps: chase and constrain. We use the tgd chase to generate a universal solution that satisfies (1), and then use incomplete information to constrain the variables of this solution to satisfy (2). Consider the following example:

EXAMPLE 2. *Consider a database consisting of a single relation instance:*

$\mathbf{C}(a, b)$
$(0, 1)$
$(0, 8)$
$(8, 2)$
$(1, 2)$
$(1, 3)$

and a view $V(x_1, x_2) :- C(x_1, y) \wedge C(y, x_2)$ *The view instance* $(\mathbf{V})$ *corresponding to* $\mathbf{C}$ *is then*

$\mathbf{V}(x_1, x_2)$
$(0, 2)$
$(0, 3)$

Suppose that $\mathbf{V}^+(9, 5)$ *is inserted. The tgd chase on* $\mathbf{V}^+$ *results in* $\mathbf{C}^+(9, z)$ *and* $\mathbf{C}^+(z, 5)$*, where* z *is some unknown value for* y*. This is acceptable if our goal is to find the certain answers to a class of queries over* C *[7, 9]. However, since the user's goal is to eventually resolve a ground instance of* C*, we need to be careful about additional side-effects. In this case, choosing* $z = 2$ *falsely asserts that there is a tuple* $\mathbf{V}(1, 5)$*.* □

4.1.1 Chase Step

The chase step translates the set of insertions by chasing [4, 7] to find a universal solution to the data exchange problem given $\mathbf{V}^+$ and the constraint from Equation (1) in Section 4.1. The correctness of our approach relies on the following theorem.

THEOREM 1. *Let* $\{\mathbf{C}_1^+, \ldots \mathbf{C}_k^+\}$ *be the result of chasing* $\mathbf{V}^+$ *on* (1)*. It follows that*

$$\mathbf{V} \cup \mathbf{V}^+ \subseteq q_C(\{\mathbf{C}_1 \cup \mathbf{C}_1^+, \ldots \mathbf{C}_k \cup \mathbf{C}_k^+\}) \quad (3)$$

PROOF. By the definition of the chase, we have $\mathbf{V}^+ \subseteq q_C(\{\mathbf{C}_1^+, \ldots \mathbf{C}_k^+\})$. The theorem follows from the monotonicity of the conjunctive query q_C. □

Theorem 1 gives a universal solution for insertion translations satisfying Equation (1). Following Example 2, we chase on $\mathbf{V}^+(9, 5)$ to get $\mathbf{C}^+ = \{(9, z), (z, 5)\}$. However, different values for z may result in spurious tuples, violating Equation (2) (such as $z = 2$, from Example 2.) The following section shows how to use conditional tables to find a set of sufficient and necessary conditions on the variables generated during the chase in order to ensure (2) is satisfied.

4.1.2 Constrain Step

The *constrain* step derives constraints on the result of the chase step in order to enforce strict equality, i.e. to satisfy Equation (2) *without* violating Equation (1). We do this by using relational operators on c-tables [8] to determine which tuples may violate the constraints.

Briefly, a conditional table (c-table) [1, 8] is a relation instance where tuples may contain labeled nulls (variables) in addition to literal values. A c-table has a global constraint Φ over its tuples' variables, as well as constraints on the individual tuples. A c-table $(\mathbf{T}, \Phi)$ defines a set of possible relations $Rep(\mathbf{T}, \Phi)$ based on the valuations of its variables which satisfy the conditions.

Given the universal solution $\mathbf{C}^+$ from the chase step, the constrain step finds a Φ^+ s.t. $Rep(\mathbf{C}^+, \Phi^+)$ equals the set of valid insertion translations. Recall Equation (3) from the chase step. Our method is to modify $\mathbf{C}_1^+, \ldots \mathbf{C}_k^+$ so that Equation 3 holds exactly. We do this by finding a Φ^+ which does not allow spurious tuples.

Using the roles for relational operators on c-tables [8], the spurious tuples can be computed as a c-table by evaluating $\mathbf{S} = q_C(\{\mathbf{C}_1 \cup \mathbf{C}_1^+, \ \ldots \ \mathbf{C}_k \ \cup \ \mathbf{C}_k^+\}) - (\mathbf{V} \ \cup \ \mathbf{V}^+)$. Each resulting tuple, t, has some local conditions (e.g., $z = 2$ for $t = \mathbf{V}(1, 5)$ in Example 2); the desired Φ^+ can be found by taking the conjunction of each condition's complement.

Following Example 2, after inserting $(9, z), (z, 5)$ into $\mathbf{C}$, we would find
$\mathbf{S} = \{(9, 1), (9, 8), (9, 2), (9, 3), (0, 5), (1, 5), (5, 5), (9, 9)\}$ for z values of 0, 1, 2, 3, 5, 8, and 9. Hence, our final universal solution for insertion translation would be $\mathbf{C}^+ = \{(9, z), (z, 5)\}$ for $z \neq 0, 1, 2, 3, 5, 8$ or 9.

The evaluation of q_C on c-tables can result in an exponential number of tuples, since each variable in one of the $\overline{y}$-positions could take any of the values present in this position of another predicate. The run time of this algorithm is proportional to the size of $\mathbf{S}$, which is loosely bound by $O\big((n + N)^k\big)$, where k is the number of predicates in q_C *joining another predicate with a* $\overline{y}$ *variable*, n is the number of inserted tuples, and N is the size of the largest relation in $\mathbf{C}$. This severe worst-case is mitigated by factors such as

the join selectivity of the conjunction. For example, if q_C consists of a chain of uniform joins (i.e. a series of many-to-one relationships), then the size of $\mathbf{S}$ becomes a degree-k polynomial over n and N, where the largest exponent of N is $\lceil k/2 \rceil$ and the largest exponent of n is k.

4.2 Deletion Translation

As with insertions, translating deletions also presents challenges in the form of ambiguities and side-effects. The ambiguity is present in that for any deleted tuple $t \in \mathbf{V}^-$, there may be multiple sets of tuples which can be deleted from $\{\mathbf{C}_1 \dots \mathbf{C}_k\}$, each of which achieves the deletion of t. The side-effects are that any or perhaps all of these sets may result in the deletion of other tuples from $\mathbf{V}$. In this section, we use the contrapositive of the tgd in Equation (2) to formulate the enumeration of all possible minimal deletion translations.

Note that deletions from V can only cause violations of Equation (2) — when there is no $V(\overline{x})$ for a corresponding $q_C(\overline{x}, \overline{y})$. Performing a tgd chase in this instance is not a viable solution, because we cannot fix these violations by inserting into V. Our approach is based on transforming the tgd (2) to its contrapositive

$$\forall \overline{x}\ \neg V(\overline{x}) \rightarrow \neg \exists \overline{y}\ q_C(\overline{x}, \overline{y}) \tag{4}$$

With respect to Equation (4), deleted tuples amount to assertions of $\neg V(\overline{x})$ for all values of $\overline{x}$ which are in $\mathbf{V}^-$. We describe a novel chase technique which enforces satisfaction of this contrapositive tgd, while not violating Equation (1). The new challenges are 1) that the right hand side asserts *nonexistence* of $\overline{x}, \overline{y}$, and so our chase rule needs to find all $\overline{y}$ which are in violation for a given $\overline{x}$; and 2) ensuring that (1) is not violated.

Since q_C is a conjunction of relational predicates, $\neg q_C$ is a disjunction of the negation of each predicate. In Example 2, our contrapositive tgd would be
$\forall x_1, x_2\ \neg V(x_1, x_2) \rightarrow \forall y\ \neg C(x_1, y) \vee \neg C(y, x_2)$. For a given value of $\overline{x}$, let $\alpha(\overline{x}) = \{\overline{y} \mid q_C(\overline{x}, \overline{y})\}$ denote the set of values of $\overline{y}$ violating the right hand side of Equation (4) (which can be found by evaluating a variant of q_C.) We satisfy Equation (4) for a given $\overline{x}$ by deleting at least one tuple from any of q_Cs predicates for each element of $\alpha(\overline{x})$. In Example 2, for $(x_1, x_2) = (0, 2)$, we have $\alpha(\overline{x}) = \{1, 8\}$. In order to delete $(0, 2)$ from $\mathbf{V}$, we must delete one of $(0, 1)$ or $(1, 2)$, and one of $(0, 8)$ or $(8, 2)$. Deleting $(0, 1)$ would also cause $(0, 3)$ to be deleted from $\mathbf{V}$, and so any translation having $(0, 1)$ is infeasible. We generalize this to formalize the set of possible deletion translations.

In Appendix B we give an algorithm which constructs all feasible deletion translations. Our algorithm is more general than past approaches [9, 16], and works by first building the set

$$\Gamma = \{\gamma(\overline{x}, \overline{y}) \mid \overline{x} \in \mathbf{V}^-, \overline{y} \in \alpha(\overline{x})\}$$

where each γ is itself a set, defined as

$$\gamma(\overline{x}, \overline{y}) = \{q_C^i(\overline{x}_i, \overline{y}_i) \mid i = 1..k\}$$

(where q_C^i is the i-th predicate of q_C.) A solution to the deletion translation problem consists of a choice of one deleted tuple from each element of Γ. A solution is feasible if deleting all tuples does not violate (1). Our algorithm performs a recursive search for all feasible solutions by considering all possibilities from each γ. Again, since q_C is monotonic we can prune the search space by not considering any deletion translation which is a superset of an infeasible translation.

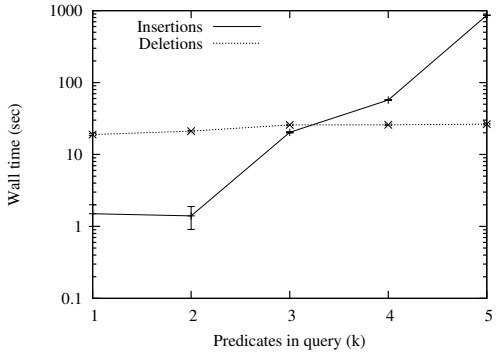

Figure 6: Performance (log scale) of update translation as a function of the number of predicates (k) in q_C.

Our algorithm has a worst case run time of $O(sk^{\alpha n})$, where k is the number of predicates in q_C, $n = |\mathbf{V}^-|$, α is the maximum size of $\alpha(\overline{x})$, and s is the time to determine if a solution yields spurious deletions. In the following section, we demonstrate that it performs quite well on realistic cases, where relational normalization permits a number of practical optimizations. In Appendix C, we discuss combining the insertion and deletion translations into a final result, as well as a number of practical improvements and optimizations.

5. UPDATE TRANSLATION RESULTS

We implemented our update translation algorithms and experimentally evaluated their performance using the TPC-H database benchmark specification [22] and a query from that specification which joins five tables and results in a large view in relation to the size of the base relations. Such a test instance is in line with the scale and complexity of previous experiments on view updating [17]. Our implementation is a C++ program which uses the Berkeley DB storage API [21]; full details of our setup are given in Appendix D.

Our first experiment analyzes performance as the number of predicates k in q_C is varied (by projecting our test query onto $k \leq 5$ relations.) The results (Figure 6) indicate exponential growth of insertion translation, agreeing with the analysis in Section 4.1.2. The difference in performance between $k = 1$ and $k = 2$ is statistically insignificant because the $\overline{y}$-variables do not intersect between the first and second predicates. The effect of increasing k dominates relation size, since the fourth relation contains 25 tuples and the fifth only 5. The performance of deletions scales roughly linearly. There is a jump from $k = 2$ to 3, as the size of $\gamma(\overline{x}, \overline{y})$ increases. The observed scalability is better than would be expected from its worst-case analysis due to the hierarchy of many-to-one relationships in the test query, which causes $\alpha(\overline{x}) = 1$ and also a large pruning benefit.

Our second experiment (Figure 7), varies the size of $\mathbb{C}$. Since our test query consists of a chain of uniform joins and one of its relations is of fixed size, both algorithms scale linearly.

Our final experiment varies n. The performance of insertion translation suffered in initial trials due to the high de-

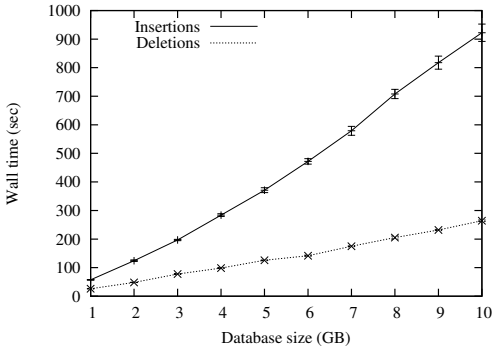

Figure 7: Performance of update translation as a function of the size (N) of $\mathbb{C}$.

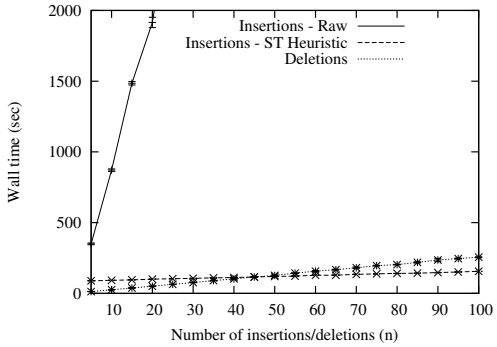

Figure 8: Performance of update translation as a function of update size (n).

gree polynomial in n. Given that the end goal is for the user to select a single unique translation among the many feasible translations, it is wasteful to compute and constrain a large number solutions which will never be contenders for the final result. We propose using a *static tables heuristic* which both reduces the search space to improve the performance of our update translation algorithms, and also reduces the number of solutions which need to be considered by the user. We can use this heuristic to stop our search early, but can always continue computing new solutions should the user request it. The idea behind the heuristic is that many databases contain a hierarchy of many-to-one relationships among tuples (i.e. "is-a" or "has-a".) If q_C contains predicates along such a hierarchy, it is unlikely that an insertion should generate insertions at higher levels. For example, in the TPC-H schema, there is a hierarchy Supplier → Nation → Region. It is unlikely that inserting a tuple into a view on these relations should insert a new Nation or Region into the database, since one would expect the set of Nations/Regions to remain relatively static. The static tables heuristic constrains the set of solutions by fixing a set of tables which should not be modified by insertions or deletions, acting as a hint for the most likely translations.

We modified our algorithms to exclude modifications to three of the tables in our test query. The results of our experiments are shown in Figure 8. We plot the wall time of insertions with and without the static tables heuristic. With the heuristic the performance of insertions scales linearly, a dramatic improvement on its exponential performance without. The performance of deletions is largely unaffected, and appears to scale linearly with a factor roughly equal to 1.

The deletion translation performance exceeds the expectations set in Section 4.2, because the normalized form of the TPC-H schema permits a number of optimizations. In particular, primary key/foreign key relationships between relations allows γ to be efficiently constructed for each tuple by using primary and foreign key indexes, and also for solution feasibility to be easily determined. In Appendix D we discuss a number of other optimizations for insertion translation relating to efficient testing of logical implication between disjunctions of binary inequalities.

6. RELATED WORK

A few projects have studied various forms of data coordination. Hyperion [2] coordinates data amongst autonomous peers through event-condition-action (ECA) rules, which dictate how events at one source trigger actions at another. ECA rules allow powerful update semantics [14] (and once an event is detected, determining the corresponding action is relatively straightforward.) This is an *imperative* approach to coordination; by contrast, our *declarative* mappings make coordination more challenging but allow sophisticated relationships to be encoded with relative ease. Recent Hyperion work [19] coordinates with mapping tables describing pair-wise associations between values, focusing on problems arising in value heterogeneity of relations with direct correspondences, whereas we focus on data sources without direct correspondences (i.e. conjunctions).

ORCHESTRA [9, 10] is a collaborative data sharing system (CDSS) where querying is local and updates are exchanged between neighboring peers. The update exchange problem is transformed from s-t tgds into a Datalog program. The authors use Skolem functions in place of existentially quantified variables. The problems which arise are similar to our update translation problem, but differ in two important ways: 1) In our case, we propagate updates in one direction (from B to C), which additionally constrains the translations to those which operate solely on C; 2) our end goal is not to query C, but to arrive at a final, unambiguous solution. This necessitates attention to side-effects resulting from particular choices of values for the existential variables.

View differencing is similar to incremental view maintenance [11], which finds an updated view given changes to the base relations. Our problem has two major differences: 1) As opposed to an updated view, our objective is to obtain the updates to the view. 2) View maintenance requires the updates to B, and both $\mathbb{B}_t$ and $\mathbb{B}_{t+1}$. In data coordination B is *external* to and *autonomous* from C, and as such may not provide these.

In [17], a side-effect free approach to translating deletions is given by making physical duplicates of joining tuples in the base relations and using a modified join semantics, yielding a fully automatic approach. Our emphasis is on a semi-automatic approach — calling on the user to make decisions in the face of ambiguity, which allows us to leave the operator semantics intact.

In Youtopia [16], insertions and deletions are translated between peers who use s-t tgds to express constraints amongst their data instances. Insertions and deletions are cascaded incrementally by using forward and backward chase rules, similar to our update translation methods from Section 4.

Whereas this incremental cascading is appropriate in a peer network, we feel that in our scenario of the base/contingent relationship, it is appropriate to coordinate data in a batch process. Also, since our data coordination scenario only involves updates to the contingent source C, we need to take extra measures to ensure the translations do not result in spurious insertions/deletions.

7. CONCLUSIONS

This paper introduced a system for the administrator of a contingent data source C to coordinate with a base source B using declarative mapping constraints to express relationships between the two. We have described a two-stage approach to performing coordination of Cs data and identified and proposed solutions to the central problems of this approach using novel methods. We have demonstrated empirically that with the combination of practical heuristics, our methods are able to handle realistic coordination scenarios.

We are currently investigating a number of important features of our coordination system. We have discussed the use of a static tables heuristic to focus the search space of update translation on the best candidate solutions. We think an important feature of our coordination system will be including additional heuristics and ranking methods. For example, the user may naturally prefer updates which are smaller, or respect certain constraints on C. Some of these issues are discussed in more detail in Appendix C.

In order to address large scale coordination tasks, improved algorithms are an important consideration. We are currently working on an algorithm which computes the minimal Φ^+ directly, without evaluating any tuples more than necessary.

Seeking to reduce the burden on the user is a primary concern. It may be desirable to incorporate a training mechanism into our system, so that the user's interaction (say, in choosing a set of updates) can be recorded and used to guide future translations. The first steps in this direction are to formulate the learning problem, including a definition of the objectives and the learning parameters recorded by our system.

Acknowledgments

We thank Raymond Ng, Solmaz Kolahi, and the anonymous reviewers for their suggestions. This project is partially funded by NSERC Canada.

8. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] M. Arenas, V. Kantere, A. Kementsietsidis, I. Kiringa, R. J. Miller, and J. Mylopoulos. The Hyperion project: from data integration to data coordination. *SIGMOD Record*, 32(3):53–58, 2003.

[3] F. Bancilhon and N. Spyratos. Update semantics of relational views. *TODS*, 6(4):557–575, 1981.

[4] P. Barceló. Logical foundations of relational data exchange. *SIGMOD Record*, 38(1):49–58, 2009.

[5] P. A. Bernstein and S. Melnik. Model management 2.0: manipulating richer mappings. In *SIGMOD*, pages 1–12, 2007.

[6] A. Doan and A. Y. Halevy. Semantic integration research in the database community: A brief survey. *AI Magazine*, 25(1):109–112, 2004.

[7] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. In *ICDT*, pages 207–224, 2003.

[8] G. Grahne. *The Problem of Incomplete Information in Relational Databases*, volume 554 of *LNCS*. 1991.

[9] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *VLDB*, pages 675–686, 2007.

[10] T. J. Green, G. Karvounarakis, N. E. Taylor, O. Biton, Z. G. Ives, and V. Tannen. ORCHESTRA: facilitating collaborative data sharing. In *SIGMOD*, pages 1131–1133, 2007.

[11] A. Gupta and I. S. Mumick, editors. *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, 1999.

[12] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166, 1993.

[13] A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The piazza peer data management system. *TKDE*, 16(7):787–798, 2004.

[14] V. Kantere, I. Kiringa, J. Mylopoulos, A. Kementsietsidis, and M. Arenas. Coordinating peer databases using ECA rules. In *DBISP2P*, pages 108–122, 2003.

[15] L. Kot, N. Gupta, S. Roy, J. Gehrke, and C. Koch. Beyond isolation: research opportunities in declarative data-driven coordination. *SIGMOD Record*, 39:27–32, September 2010.

[16] L. Kot and C. Koch. Cooperative update exchange in the youtopia system. *PVLDB*, 2(1):193–204, 2009.

[17] Y. Kotidis, D. Srivastava, and Y. Velegrakis. Updates through views: A new hope. In *ICDE*, page 2, 2006.

[18] M. Lawrence, R. Pottinger, and S. Staub-French. Coordination of data in heterogeneous domains. In *ICDE Workshop NTII*, pages 167–170, 2010.

[19] M. M. Masud, I. Kiringa, and H. Ural. Update processing in instance-mapped P2P data sharing systems. *IJICS*, 18(3/4):339 – 379, 2009.

[20] R. J. Miller, L. M. Haas, and M. A. Hernández. Schema mapping as query discovery. In *VLDB*, pages 77–88, 2000.

[21] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *USENIX*, pages 43–43, 1999.

[22] TPC benchmark H, standard specification, revision 2.11.0, April 2010.

APPENDIX

A. VIEW DIFFERENCING

We now describe our approach to view differencing in more detail. Materialize and Compare (MAC) takes as inputs the updated instance $\mathbb{B}_{t+1}$ and a sorted copy of the materialized $\mathbf{V}_t$ (which is already stored by our system.) Any total ordering on the view tuples will suffice for sorting; In our experiments we order view tuples based on a significance ordering on their attributes. i.e. $\overline{x}_1 < \overline{x}_2$ iff $\overline{x}_1[a] < \overline{x}_2[a]$ for some attribute a, and $\overline{x}_1[b] = \overline{x}_2[b]$ for all attributes b more significant than a. Careful choice of the comparison operator can speed up the sorting. If the mapping designer knows how the data in B is stored and how q_B is evaluated, a comparison operator may be chosen such that sorting does not require moving around a significant number of tuples.

As discussed in Section 3.2, we use the counting algorithm of [12] for IVM-VD. The counting algorithm associates with each tuple $\overline{x}$ a count of the number of derivations of $\overline{x}$ in a relation, under duplicate semantics. For $x \in \mathbf{B}$, we use $count_B(\overline{x})$ to denote this count (which could also be negative.) An additive union operator $\uplus$ is defined over relations of counted tuples $(\overline{x}, count(\overline{x}))$, by adding the counts of tuples (and removing any tuples whose count sums to 0.)

The updated view $\mathbf{V}_{t+1}$ is found by performing the additive union of $\mathbf{V}_t$ with the result of $2k$ *delta rules*, where k is the number of relational predicates in q_B. Each delta rule replaces one of these predicates with the equivalent predicate defined on the on set of inserted/deleted tuples into/from that relation. We obtain tuple counts by rewriting each delta rule as an aggregate query. This rewriting is simple, and incurs little additional cost on the part of B.

Another challenge is computing the updates $(\mathbf{V}^+, \mathbf{V}^-)$ from the set of tuple counts computed by the counting algorithm. We do this during the final additive union operator, by tracking which tuples have a count which changes from ≤ 0 to > 0, and which tuples have a count which changes from > 0 to ≤ 0.

A.1 Evaluation

We experimentally evaluated both MAC and IVM-VD to assess their performance in a variety of circumstances. Our main goals were to determine whether they are both feasible under reasonable conditions, and to what features they are most sensitive. Therefore, our experiments measured the wall execution times of both approaches while varying the amount of data, update size, schema, and view definitions.

A.1.1 Instances

Simple: Contains a single relation $B(id, a, b)$. *id* is a key; a is an integer on $[0, 100)$; and b is a floating point value. The simple instance is to obtain baseline performance for a very simple case, measuring the effect of view size and update size. Our view definition was: $q(b) :- B(id, a, b) \wedge a < c$, where c is a variable between 0 and 100, and which defines the view size as a percentage of the database size. Since b values are randomly distributed, MAC was forced to sort $\mathbf{V}_{t+1}$ after materialization. Naturally, IVM-VD was not required to sort. Data was generated uniformly.

Tree: Recall from Section 3.2 that for a query with k predicates, IVM-VD evaluates $2k$ delta rules and performs $2k$ additive union operations. Our second test instance varied the number of relations in the schema and view definition to determine how well each method scales on progressively complex queries. The tree schema has relations $A_1 \ldots A_k$; each relation $i \neq k$ has a foreign key reference to relation $i+1$, and the data for this schema forms a forest, with each tree rooted by a tuple of A_k relation. The view on the k-relation tree schema contains the id of all tuples in A_1 who are joined to the tuple in A_k with $id_k = 0$.

TPC-H: The *TPC-H* schema is a benchmark decision support schema [22]; TPC-H consists of 8 relations used to track the part ordering system of a retail database. We used the TPC-H schema to evaluate performance under a realistic database scenario. We used TPC-H query q_{16}, as shown in Figure 9. Since the counting algorithm can only be applied for views defined with conjunctive queries, we adapted the TPC-H queries by replacing subqueries with finite value sets, and removing aggregations. Our data was generated with the TPC-H data generator, which has a uniform distribution.

```
SELECT P_BRAND, P_TYPE, P_SIZE, PS_SUPPKEY
FROM PARTSUPP JOIN PART ON P_PARTKEY=PS_PARTKEY
WHERE P_BRAND <> 'Brand#4' AND
P_TYPE NOT LIKE 'SMALL%' AND
P_SIZE IN (1,6,11,16,21,26,31,36,41,46,50)
AND PS_SUPPKEY NOT IN (1, 100, 200, 1000, 4000,
      4800, 5200, 6400, 8800, 9300, 9700, 9999);
```

Figure 9: TPC-H query q_{16}.

A.1.2 Methods

For the simple instance, we conducted experiments which vary c — the view size as a percentage of the data set size — as well as the size of the updates to the relation. For the tree test instance, we varied the number of relations k, and generated a single instance of each size k schema. For the TPC-H instance, we conducted experiments varying the database size (and hence the view size) and varying the size of updates, similarly to the single relation instance. For each data set size, or update size, we performed 10 trials using independently generated updates. We used the TPC-H data generator, which uniformly generates data for all 8 relations proportional to a size factor; this size factor roughly corresponds to its size in GB, or 1/8,660,000th the total number of tuples.

Our implementations of both approaches were written in C++, using the MySQL++ library, and a MySQL database engine, version 5.0. The implementation and engine both ran on the same host. Our experiments were performed on a Intel Xeon 2.93GHz system, with 64GB of RAM, running OpenSUSE Linux.

A.1.3 Experimental Results

First, we varied x (the size of the view as a percentage of the data set size)on the Simple instance (Figure 10). As can be seen, both methods perform reasonably well, with IVM-VD slightly outperforming MAC. In this case, the size of both insertions and deletions are fixed at 5% of the size of the unified data set ($\mathbb{B}_t \cup \mathbb{B}_{t+1}$), which is 2M tuples. We also found that the sorting step occupied 75% of MACs execution time, meaning that it could be made significantly faster if $\mathbf{V}_t$ is already sorted. This would be the case if the view includes any attribute for which there is an index. Our tests on TPC-H highlight the performance advantage of sorting.

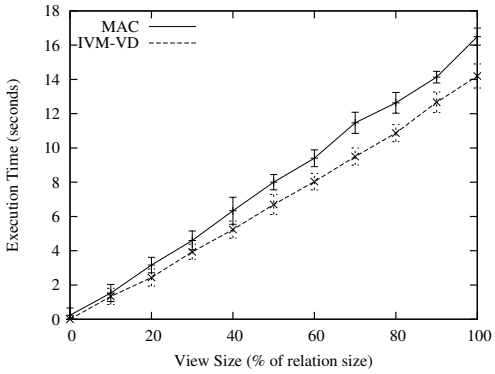

Figure 10: Performance of view differencing on the simple instance.

We ran additional experiments (not shown) varying the size of the updates. As expected, the run time of IVM-VD increases linearly with the size of the updates, roughly tripling over the experimental range, while the performance of MAC was not affected. It should be noted that our experimental range extended to insertions and deletions of 25% of the tuples in the test database, which would be considered an extreme case. For a simple view definition, IVM-VD is not overly sensitive to a normal range of update sizes.

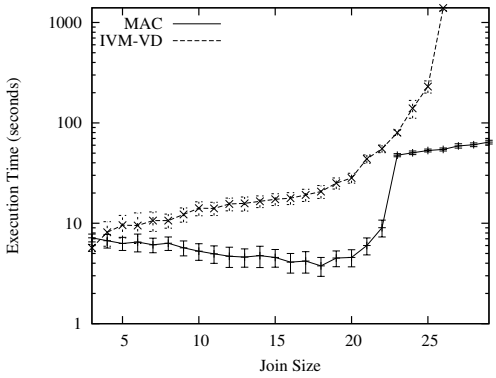

Figure 11: Performance of view differencing on the tree instance.

Next we tested on the Tree instance to determine the algorithms' sensitivity to join size (as a proxy for query complexity). Figure 11 shows the run time (plotted on a logarithmic scale) of each algorithm as the number of joins is varied. The run time of IVM-VD increases exponentially, reaching around 1400 seconds for a 26-way join. While a 26 relation join may seem implausible, it is seen in practice; these results indicate some caution is necessary when considering the application of IVM-VD for large numbers of joins. These results require the number of updated relations to be large, regardless of the number of relations in the view definition. In preliminary trials, we found IVM-VD to be efficient for views with a large number of joins when the number of *updated relations* is small. This is because the majority of delta rules will perform a join with a relation of size 0. Modern query optimizers can detect this case and quickly return an empty result.

MAC scales quite well in this case. Its runtime decreases at first due to a decrease in the size of $\mathbf{V}_{t+1}$ and then increases as the decreased view size is overwhelmed by the time to materialize $\mathbf{V}_{t+1}$. The sharp increase at 23 joins is due to MySQL's execution time of q_B growing from less than 10 seconds to more than 45.

We note that these results may be due in part to the efficiency (or lack thereof) of the query evaluation implementation. While replacing MySQL with an implementation having better performance for large joins may change the results' scale, it should not affect the relative ordering of the two methods, as the execution time of MAC for views involving large joins is more likely to be bound by the time to materialize $\mathbf{V}_{t+1}$.

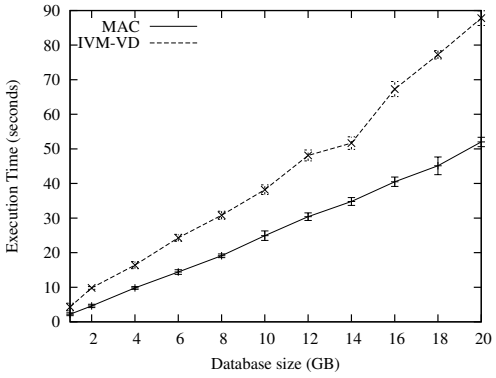

Figure 12: Performance of view differencing as a function of size for TPC-H.

Finally, we present our TPC-H results. In Figure 12 we fixed the update size at 10% of the data set size and varied the size of the database. Both methods scale linearly with the size of the data, with MAC outperforming IVM-VD slightly in each case. This latter trend is largely due to our choice of update size: in Figure 13 we fixed the database size at 10GB and varied the update size. In this case, the run time of IVM-VD increased with update size, while MAC was decreased slightly due to decreasing view size. Our view definition (Figure 9) was a query from TPC-H; it joins four relations and using four selection conditions involving tests of inequality, set membership and substring containment. The size of the resulting view is about 0.64% of the total database size, ranging from 0.11M to 2.37M tuples over the range of the experiments. We experimented with other TPC-H queries, and in all cases, the point at which MAC outperforms IVM-VD is around 2.5%, despite the fact that the three queries used have a varying number of joins and result in varying view sizes.

B. DELETION SEARCH ALGORITHM

Algorithm 1 first builds the set Γ, and then performs a recursive search to find all feasible combinations of 1 tuple from each $\gamma \in \Gamma$. For a set of deletions $\mathbf{V}^- = \{\overline{x}_1, \ldots \overline{x}_m\}$, the number of distinct translations is at most $\prod_{i=1}^{m} k^{\alpha(\overline{x}_i)}$, where k is the number of predicates in q_C.

Consider Example 2 from Section 4.1. If we were given $\mathbf{V}^- = \{(0,2)\}$, TranslateDeletions would first find $\alpha(0,2) = \{(1),(8)\}$. It would build two sets

$$\gamma_1 = \{(0,1),(1,2)\}, \gamma_2 = \{(0,8),(8,2)\}$$

In order to delete $\mathbf{V}^-(0,2)$ we must delete at least 1 tuple from each γ. RecursiveSearch first tries $C(0,1)$, and, finding

Algorithm 1 Finds all combinations of one tuple from each $\gamma \in \Gamma$ using a pruned search tree.

TranslateDeletions($\mathbf{V}^-$)
1: $\Gamma \leftarrow \emptyset$
2: **for all** $\overline{x} \in \mathbf{V}^-$ **do**
3: **for all** $\overline{y} \in \alpha(\overline{x})$ **do**
4: $\gamma \leftarrow \emptyset$
5: **for all** Conjuncts $C_i(\overline{x}_i, \overline{y}_i)$ in $q_C(\overline{x}, \overline{y})$ **do**
6: $\gamma \leftarrow \gamma \cup \{C_i(\overline{x}_i, \overline{y}_i)\}$
7: **end for**
8: $\Gamma \leftarrow \Gamma \cup \{\gamma\}$
9: **end for**
10: **end for**
11: RecursiveSearch($\emptyset$, Γ)

RecursiveSearch($\mathbf{C}^-$, Γ)
1: $\gamma \leftarrow$ any member of Γ
2: **for all** $C_i(\overline{x}, \overline{y}) \in \gamma$ **do**
3: **if** (1) is satisfied w/resp. to $\mathbf{C}^- \cup \{C_i(\overline{x}, \overline{y})\}$ **then**
4: $\mathbf{C}^- \leftarrow \mathbf{C}^- \cup \{C_i(\overline{x}, \overline{y})\}$
5: $\Gamma \leftarrow \Gamma - \gamma$
6: **if** $\Gamma = \emptyset$ **then**
7: Output $\mathbf{C}^-$ as a feasible solution
8: **else**
9: RecursiveSearch($\mathbf{C}^-, \Gamma$)
10: **end if**
11: **end if**
12: **end for**

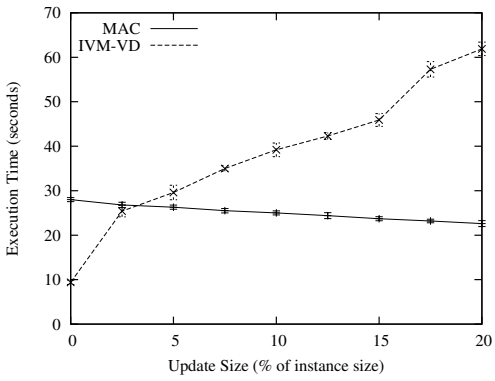

Figure 13: Execution time for view differencing as a function of update size for TPC-H.

that it violates Equation (1) due to $\mathbf{V}(0,3)$ being spuriously deleted, tries $C(1,2)$ instead. On the recursive call, our algorithm would find that neither $C(0,8)$ and $C(8,2)$ violates Equation (1), and hence it would give two possible translations:

$$\mathbf{C}^- = \{(1,2),(0,8)\}, \mathbf{C}^- = \{(1,2),(8,2)\}$$

The set $\alpha(\overline{x})$ can be computed by evaluating a variant of the query $V(\overline{x}) :- q_C(\overline{x}, \overline{y})$ by replacing $\overline{x}$ with its specific values, and adding $\overline{y}$ to V. This query can be efficiently computed if at least one relation C_i has a secondary index on an attribute in $\overline{x}_i$. We can check the constraint on line 4 of RecursiveSearch by maintaining a Γ set for *each* $\overline{x} \in \mathbf{V}_{t+1}$. Each additional deletion on line 5 of RecursiveSearch also removes γs from these additional Γ sets, and the constraint will be violated if $\Gamma = \emptyset$ for any $\overline{x} \in \mathbf{V}_{t+1}$.

C. CHOOSING A TRANSLATION

Sections 4.1 and 4.2 described our solutions to translating view insertions and deletions into sets of possible insertions/deletions on $\mathbb{C}_t$. We now discuss how our system presents translations to the user, guiding them to select what they believe is the correct set of updates for their local data.

The set of insertions is presented as a c-table, along with the constraint Φ^+ on its variables. The user must choose appropriate values for these variables which do not violate Φ^+. Due to the regularity of Φ^+ a relatively simple iterative process can be used.

Φ^+ consists of a conjunction of disjunctions, where the disjuncts are inequality atoms of the form $w \neq z$, or $w \neq c$ for variables w, z and constant c. In our process, each tuple $\overline{x}_i, \overline{y}_i \in \mathbf{C}_i^+$ is presented individually in sequence along with provenance information in the form of the tuple in $\mathbf{V}^+$ which resulted in the generation of $C_i(\overline{x}_i, \overline{y}_i)$. For each variable w, the user is given a list of *suggested values*, where choosing one of the suggested values results in a $(\overline{x}_i, \overline{y}_i)$ which is already in $\mathbf{C}_i$ — i.e. no insertion needs to be performed. The user is also given a list of *forbidden values*, based on the existence of inequalities $w \neq c$ and $w \neq z$ in Φ^+, where z is a variable whose value has already been chosen.

The set of deletion translations can be presented to the user in summary form by using a representative element from each deletion. As with insertion, we can also give as provenance the tuple from $\mathbf{V}^-$ which necessitated each tuple's deletion.

An important question is whether the insertions chosen will dictate the set of deletions which can be chosen — i.e. is it possible for a set of insertions and deletions to conflict? We define a *conflict* as any pair of $\mathbb{C}^+$ and $\mathbb{C}^-$ which are correct translations on their own, but evaluating $q_C((\mathbb{C} \cup \mathbb{C}^+) - \mathbb{C}^-)$ does not generate the updated view. An important question is how to ensure that conflicting translations cannot be chosen, while still giving the freedom of choice.

Our approach relies on the following theorem:

THEOREM 2. *$\mathbb{C}^+$ and $\mathbb{C}^-$ conflict iff $\mathbb{C}_i^+ \cap \mathbb{C}_i^- \neq \emptyset$.*

PROOF. For the *if* direction; let $t \in \mathbb{C}^+ \cap \mathbb{C}^-$. Since $\mathbb{C}^+$ is a universal solution to the data exchange problem given $\mathbf{V}^+$ and q_C, it follows that $\mathbb{C}^+$ is minimal (otherwise it would not homomorphically map to a solution which is minimal.) Also, since q_C is monotonic it follows that $q_C(\mathbb{C} \cup \mathbb{C}^+ - t) \subset \mathbf{V} \cup \mathbf{V}^+$. Again, due to the monotonicity of q_C and the fact that $q_C(\mathbb{C}) = \mathbf{V}$, there exists a tuple $t' \in \mathbf{V}^+$ such that $t \notin q_C(\mathbb{C} \cup \mathbb{C}^+ - t)$. Since by definition $\mathbf{V}^+ \cap \mathbf{V}^- = \emptyset$, it follows that $q_C((\mathbb{C} \cup \mathbb{C}^+) - \mathbb{C}^-) \neq (\mathbf{V} \cup \mathbf{V}^+) - \mathbf{V}^-$.

For the *only if* direction; assume $\mathbb{C}^+ \cap \mathbb{C}^- = \emptyset$. Then it is the case that $q_C((\mathbb{C} \cup \mathbb{C}^+) - \mathbb{C}^-) = q_C((\mathbb{C} - \mathbb{C}^-) \cup \mathbb{C}^+)$ (i.e. the order of operators is irrelevant.) Since $q_C(\mathbb{C} \cup \mathbb{C}^+) = \mathbf{V} \cup \mathbf{V}^+$, and $q_C(\mathbb{C} - \mathbb{C}^-) = \mathbf{V} - \mathbf{V}^-$, and due to the monotonicity of q_C we have $q_C((\mathbb{C} \cup \mathbb{C}^+) - \mathbb{C}^-) \subseteq (\mathbf{V} \cup \mathbf{V}^+) - \mathbf{V}^-$ and $q_C((\mathbb{C} - \mathbb{C}^-) \cup \mathbb{C}^+) \subseteq (\mathbf{V} - \mathbf{V}^-) \cup \mathbf{V}^+$. However, since $\mathbf{V}^+ \cap \mathbf{V}^- = \emptyset$, the right hand side of the previous two expressions are equal, forcing both expressions to hold exactly. □

Given Theorem 2, we can build conflict avoidance into our selection process. After the user has chosen variable values for the inserted tuples, we can eliminate all deletion

```
SELECT S_ACCTBAL, S_NAME, N_NAME, R_NAME,
P_PARTKEY, P_MFGR, S_ADDRESS, S_PHONE, S_COMMENT
FROM SUPPLIER
JOIN PARTSUPP ON PS_SUPPKEY = S_SUPPKEY
JOIN PART ON P_PARTKEY = PS_PARTKEY
JOIN NATION ON N_NATIONKEY = S_NATIONKEY
JOIN REGION ON R_REGIONKEY = N_REGIONKEY;
```

Figure 14: TPC-H query q_2.

translation choices which intersect with the chosen insertions. Conversely, if the user chooses the preferred deletion translation first, we can add constraints to the variables in the translated insertions so that it does not intersect with the deletions chosen.

D. UPDATE TRANSLATION EXPERIMENTAL SETUP

As described in Section 5, we have implemented our update translation solutions in C++ using the Berkeley DB external storage API, and B-Tree index structures. All experiments were run on a 2.93GHz Intel Xeon based system with 64GB of memory running OpenSUSE Linux.

We modeled our experimental instance to be comparable in size and complexity to those used in [17]. We again used the TPC-H benchmark database instance, and a view definition (Figure 14) which joins five tables and is a variant of the cost supplier query (q_2) [22]. For the experiments with < 5 joins (Figure 6) we removed Region, Nation, Supplier, and Part, in that order.

In total, our dataset had ≈1.9M tuples, and the view size is 0.8M tuples. We randomly generated view insertions by equally choosing existing values from the database at random; and newly generated random values. We randomly generated view deletions by choosing tuples from our materialized view at random.

Our implementation of c-tables is completely custom, and we use a nested loop algorithm for joining tables containing incomplete information. We have made a number of optimizations based on an indexing mechanism for disjunctions of inequalities which allows logical implication to be efficiently determined. Let Λ_1 and Λ_2 be two disjunctions of binary inequalities between variables and constants. I.e.

$$\Lambda_1 = \lambda_{1,1} \vee \lambda_{1,2} \ldots \lambda_{1,n}$$
$$\Lambda_2 = \lambda_{2,1} \vee \lambda_{2,2} \ldots \lambda_{2,m}$$

Where each $\lambda_{i,j}$ is a binary inequality of the form $w \neq z$ or $w \neq c$ for variables w, z and constant c. Then $\Lambda_1 \rightarrow \Lambda_2$ iff all $\lambda_{1,i}$ are also in Λ_2. We give each variable a unique identifier, and define a simple total ordering on binary inequalities. We store a disjunction of binary inequalities in sorted order, allowing us to determine if $\Lambda_1 \rightarrow \Lambda_2$ by lexicographical comparison.

During our insertion translation, we store the disjunctions of Φ^+ in lexicographically sorted order. This allows us to minimize Φ^+ in $O(n)$ time by performing an in-order scan, since all of the disjunctions implied by a given disjunction will appear directly after it in this ordering. We also use this property for early termination of nested loop iterations in our join algorithm. We check if completing the current inner loop iterations will require a condition which is already implied by Φ^+, and hence performing these inner loops will be redundant.