Recomputing Materialized Instances after Changes to Mappings and Data

Todd J. Green [1], Zachary G. Ives [2]

[1]*Department of Computer Science, University of California, Davis*
Davis, CA 95616 USA
green@cs.ucdavis.edu

[2]*Department of Computer and Information Science, University of Pennsylvania*
Philadelphia, PA 19104 USA
zives@cis.upenn.edu

***Abstract*—A major challenge faced by today's information systems is that of *evolution* as data usage evolves or new data resources become available. Modern organizations sometimes exchange data with one another via declarative mappings among their databases, as in *data exchange* and *collaborative data sharing* systems. Such mappings are frequently revised and refined as new data becomes available, new cross-reference tables are created, and corrections are made.**

A fundamental question is how to handle changes to these mapping definitions, when the organizations each *materialize* the results of applying the mappings to the available data. We consider how to *incrementally* recompute these database instances in this setting, reusing (if possible) previously computed instances to speed up computation. We develop a principled solution that performs cost-based exploration of recomputation versus reuse, and simultaneously handles updates to source *data* and *mapping definitions* through a single, unified mechanism. Our solution also takes advantage of *provenance* information, when present, to speed up computation even further. We present an implementation that takes advantage of an off-the-shelf DBMS's query processing system, and we show experimentally that our approach provides substantial performance benefits.

I. Introduction

In the sciences, in business, and in government, there is increasingly a need to create and maintain *derived* or *transformed data*, often stored externally: consider data warehouses for supporting centralized analysis, master databases for keeping curated results, data replicas that are released after being "sanitized" for privacy reasons, portals for sharing data across the life sciences, and so on. Traditionally, creation and maintenance of these instances has been handled through extract-transform-load (ETL) dataflows, which can perform arbitrary translation and cleaning operations. Increasingly, there is interest in using *declarative schema mappings* relating the source and target instances [1], [2]. Declarative mappings are not as general as ETL, but when applicable (see [3]), they have many advantages: they can be *composed* so data can be mapped from one database to another through transitive connections [4] if direct mappings are unavailable; they can be used to track data provenance [2]; and they can be used to ensure consistency even in the presence of conflicts and different viewpoints [5], [6]. A significant literature now exists on materializing the target data instance when given a set of declarative mappings: when there is a single source and target, as in IBM's Rational Data Architect and the related Clio [7] research project, this is termed *data exchange* [1]; multiple sources and targets are considered in *peer data exchange* systems [8]; and an even more flexible model in which every participant is a source and a target who can make updates is that of *update exchange* as performed in *collaborative data sharing systems* (CDSS) [9].

A major challenge in all such scenarios is that of *evolution*: the target schema and/or the mapping between them might be changed to meet new requirements or a new understanding of the domain, to accommodate a new source, or to leverage a correspondence (synonym) table. How do we efficiently recompute the target instance in a data exchange setting, or the set of all transitively dependent instances in collaborative data sharing, given a new specification of the target instance and its relationship to the source(s)? A natural question we consider in this paper is whether the declarative nature of schema mappings can again provide a benefit, in terms of recomputing the affected instances: intuitively, a minor change to a mapping should result in a relatively inexpensive update to the affected data instances.

We show how this problem of supporting *changes to mappings* (and possibly the target schema) in data exchange and CDSS can be solved through novel techniques for what we term *multi-view adaptation*: the problem of efficiently recomputing a collection of materialized views when the view definitions are changed [10]. In doing so we exploit the close connection between declarative schema mappings and the data exchange "chase" procedure with Datalog programs [2], [11] — reading mappings as view definitions. Our solution enables efficient recomputation of affected view (target) instances. Moreover, because of our more general problem formulation, our techniques are also *applicable in a variety of other view maintenance settings*, even in conventional relational DBMSs.

To get a sense of the problem, we consider a CDSS scenario and how mappings might change over time. (Data exchange represents the simpler case where there is only a single source and a single target.)

Example 1: Suppose we have a series of genomics *databases, exported in the formats preferred by different communities. The initially supplied data (in the* BioSQL *[12] format) includes biological data entries, plus a list of terms. We export a list of genes as a new table related to the original by an integrity constraint, which we supplement with a table specifying which genes correspond to which organisms. Finally, we export a table containing only genes for the mouse (which happens to be the organism given ID 12 in our database). In data exchange and CDSS, this is accomplished by writing schema mappings in the form of constraints relating the source tables with a set of exported* target *tables:*

$$(m_1)\quad bioentry(E,T,N) \wedge term(T,\text{“gene”}) \rightarrow gene(E,N)$$
$$(m_2)\quad gene(G,N) \wedge hasGene(G,12) \rightarrow mousegene(G,N)$$

Later, we modify the mappings populating gene to join terms through a synonym table, since the tables may use alternative terms. Simultaneously, we modify the mappings for mousegene to incorporate a correspondence table *relating organism IDs and scientific names; instead of selecting on organism ID 12, we instead select on organism name "mus musculus."*

$$(m_1)\quad bioentry(E,T,N) \wedge term(T,\text{“gene”}) \rightarrow gene(E,N)$$
$$(m_2')\quad gene(G,N) \wedge hasGene(G,M) \wedge orgname(M,\text{“mus musculus”}) \rightarrow mousegene(G,N)$$
$$(m_3)\quad bioentry(E,T,N) \wedge term(S,\text{“gene”}) \wedge termsyn(T,S) \rightarrow gene(E,N)$$

Now the system must recompute the two exported tables: as it does so, it has the option to reuse data from any existing materialized instances of gene and mousegene, as well as the source BioSQL, GeneOrg, *and* MouseLab *relations. Perhaps it would first update gene, then recompute mousegene using gene.* □

The problem of *multi-view adaptation* poses a number of challenges not directly or adequately addressed by previous work. Existing techniques for *view adaptation* [10] typically consider revisions to single view definitions as simple, atomic changes in isolation, and they apply *ad hoc* case-based (rather than cost-based) reasoning. Mapping adaptation [13] involves modifying mappings in response to schema changes, and thus could be a *source* of changes in our setting.

In general, we may need to *insert* and/or *remove* tuples from the existing materialized views, as a means to efficiently compute an updated instance; to add attributes by joining existing materialized views with additional source data; or to remove or reorder attributes by projection. In very dynamic settings where the data is changing simultaneously with the mappings, we may also have *source data updates* arriving even as the view definitions are changing.

In this paper we tackle these challenges through techniques based on *optimizing queries using materialized views* [14], exploiting an enriched data model that we proposed in our earlier work [15]. We consider the existing data sources, view instances, and even the updated sources as materialized views; we seek to rewrite the modified views to take advantage of these instances via a unified mechanism. Our approach is based on a cost-based search over a rich and well-characterized space of rewritings, and is hence very general. We make the following specific contributions.

- We recast the problem of supporting changes to declarative schema mappings as one of *multi-view adaptation*.
- We extend the theoretical results of our previous work [15] to handle queries and views using *Skolem functions*, as needed in data exchange in CDSS.
- We present a unified strategy for solving multi-view adaptation by enumerating possible query rewritings in an enriched data model, where data and mapping updates can be treated uniformly. The space of rewritings encompasses *differential plans*, which may include both insertion *and removal* of values from existing view instances.
- We develop transformation rules and search strategies, including effective pruning heuristics, for exploring the space of rewritings.
- We build a layer over an existing off-the-shelf DBMS that supports our multi-view adaptation techniques.
- We show that our techniques provide significant performance gains for workloads designed to mimic empirically observed rates of changes in schemas.

The paper is organized as follows. In Section II, we recall background concepts from data exchange and CDSS, and give a formal problem statement. Section III outlines how query reformulation can be used to adapt to changes in schema mappings and/or data. Section IV describes how we search the space of possible reformulations to find an efficient multi-view adaptation plan. Section V experimentally demonstrates that our approach provides significant speedups. We discuss related work in Section VI and wrap up in Section VII.

II. Background and Problem Statement

The problem of *multi-view adaptation* incorporates aspects of view and mapping adaptation, as well as update propagation. Our motivation for studying this problem comes from data exchange systems such as [1], [11] and collaborative data sharing systems (CDSSs) including ORCHESTRA [2] and others [5], [16]. We first recall the data exchange model and its generalization to CDSS, then describe how we represent changes to mappings and to data, before formalizing our problem statement.

A. Data Exchange

A data exchange *setting* consists of a *source* schema **S** and a *target* schema **T**, assumed to be disjoint from **S**, along with sets Σ_{st} and Σ_t of *source-target* and *target-target* dependencies, respectively. In our treatment, these dependencies are assumed to be given as *tuple-generating dependencies* (*tgds*).[1] A tgd is a first-order logical assertion of the form

$$\forall \overline{X}\ \forall \overline{Y}\ (\varphi(\overline{X},\overline{Y}) \rightarrow \exists \overline{Z}\ \psi(\overline{X},\overline{Z})), \qquad (1)$$

[1]Classical data exchange also incorporates *equality-generating dependencies*, but these are unsupported by systems such as Clio and ORCHESTRA. Source-target tgds and egds together are equivalent to another standard formalism, *GLAV mappings* [14].

where the left hand side (LHS) of the implication, φ, is a conjunction of relational atoms over variables $\overline{X}$ and $\overline{Y}$, and the right hand side (RHS) of the implication, ψ, is a conjunction of relational atoms over variables $\overline{X}$ and $\overline{Z}$. For readability, we will generally omit the universal quantifiers and simply write

$$\varphi(\overline{X},\overline{Y}) \rightarrow \exists \overline{Z}\ \psi(\overline{X},\overline{Z}), \qquad (2)$$

as in the examples from Section I. A tgd is called *source-target* (resp. *target-target*) if φ uses only relation symbols from $\mathbf{S}$ (resp. relation symbols from $\mathbf{T}$), while ψ uses only relation symbols from $\mathbf{T}$.

Given a fixed data exchange setting as above, the *data exchange problem* is as follows: given a database instance I over the source schema $\mathbf{S}$, compute a database instance J over the target schema $\mathbf{T}$ such that I and J jointly satisfy the dependencies in Σ_{st}, and J satisfies the dependencies in Σ_t. Moreover, since there may be many such instances J, we require additionally that J be a *universal solution*, which can be used to compute the *certain answers* to (positive) queries over $\mathbf{T}$. The main result of [1] is to show that universal solutions can be computed using the classical *chase* procedure [17]. (See [1] for precise definitions of universal solutions, certain answers, and the chase; they are not crucial here.)

While the classical data exchange literature focuses on the chase procedure for carrying out data exchange, practical systems (including Clio and ORCHESTRA) typically use a different method for computing universal solutions, based on compiling the sets Σ_{st} and Σ_t of dependencies into an executable *Datalog program* using standard techniques [2], [11].[2]

Example 2: The tgds m_1 and m_2 from the introduction can be used to generate the following Datalog program:

```
(m1) gene(E,N) :- bioentry(E,T,N), term(T,"gene").
(m2) mousegene(G,N) :- gene(G,N), hasGene(G,12).
```

Evaluating the program has the effect of "exchanging" data from the source tables `bioentry`, `term`, `hasGene`, `termsyn`, *and* `orgname` *to the target tables* `gene` *and* `mousegene`. *The result will be the set of data instances in Figure 1. Note that this Datalog can be easily converted to SQL for execution by an off-the-shelf DBMS (with additional control logic if the Datalog program is recursive).* □

As can be seen (2) above, tgds may sometimes contain existentially-quantified variables on the RHS. For instance, consider a source schema having table `people(name, address)` and a target schema having tables `names(ssn, name)` and `addresses(ssn, address)` along with the source-target tgd

$$people(N,A) \rightarrow \exists S\ names(S,N) \wedge addresses(S,A).$$

In this case, the compilation procedure will introduce a *Skolem function* `f` into the Datalog rules as a convenient way to "invent" a fresh `ssn` value in the target tables:

[2]Technically, the generated Datalog program implements a variant of the classical chase known as the *oblivious chase* [18]. Clio uses a generalization of the techniques presented here to work with nested collections.

Source tables

bioentry

eid	tid	name
*MGI:*88139	26	*BCL2-like 1*
*MGI:*87904	26	*actin, beta*

term

tid	name
26	*gene*
28	*factor*

termsyn

tid1	tid2
26	28

hasGene

gid	orgid
*MGI:*88139	12
*MGI:*87904	12

orgname

orgid	name
12	*mus musculus*
15	*plasmodium falciparum*

Target tables

mousegene

gid	name
*MGI:*88139	*BCL2-like 1*
*MGI:*87904	*actin, beta*

gene

gid	name
*MGI:*88139	*BCL2-like 1*
*MGI:*87904	*actin, beta*

Fig. 1. Data instances for running example.

```
names(f(N,A), N) :- people(N,A).
addresses(f(N,A), A) :- people(N,A).
```

In this paper, we focus on such generated programs, and in fact we will assume for simplicity of presentation that mappings are simply given as Datalog programs (possibly with Skolem terms). Thus from now on we will use the terms "mappings" and "Datalog rules" interchangeably. Additionally, we assume that the set Σ_t of target-target dependencies is *acyclic*, in which case the generated Datalog program will be non-recursive.

B. Update Exchange

CDSS generalizes the data exchange model to multiple sites or *peers*, each with a database, who agree to share information. Peers are linked with one another using a network of declarative (and compositional) schema mappings of the same kind as are used in data exchange. Each mapping defines how data or updates applied to one peer instance should be transformed [2], [16] and applied to another peer's data instance.[3]

Under the CDSS model, every peer has a materialized local database instance, against which users pose queries and make updates. Periodically, the peer refreshes its materialized instance through an operation called *update exchange*: it first publishes the set of updates made to its local instance, then it uses the set of all others' published updates and the set of schema mappings to recompute a new data instance. For purposes of this paper, update exchange can be thought of as an incremental version of data exchange, which propagates the effects of changes to each peer, in the form of insertions and deletions. (Other aspects of update exchange include trust and local curation updates, which are discussed in [2].) As above, we assume an *acyclic* set of CDSS mappings.

Example 3: Refer to Figure 2 for a CDSS corresponding to our example from the previous section; for simplicity we use the same source and target relations but partitioned them

[3]In addition, *trust conditions* may be used to help arbitrate conflicts [5], [6], but we do not consider these in this paper.

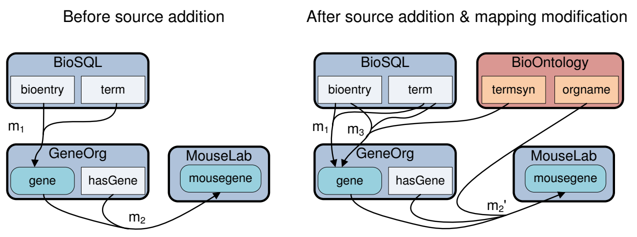

Fig. 2. Example of a CDSS mapping relations to peers.

across peers. Tables `bioentry` *and* `term` *are supplied by* **BioSQL**. *A schema mapping m_1 uses these relations to add data to* `gene` *in participant* **GeneOrg**. *Finally,* **MouseLab** *imports data from* **GeneOrg** *using mapping m_2.* □

Within this update exchange scenario, there are two kinds of changes that can occur, either separately or together. The first is that one may change the *mapping definitions* in the system, requiring that we recompute the instance associated with each affected target schema.

Example 4: Suppose a new participant **BioOntology** *is added to the above CDSS. Now both mappings m_1 and m_2 are modified to incorporate its relations,* `termsyn` *and* `orgname`, *by joining* **BioSQL** *tuples with these.* □

The second is that one may publish changes to the *source data* at one of the peers, requiring again that we recompute the instance associated with each affected target schema.

Example 5: Refer to the data instances of Figure 1. Suppose that **BioSQL** *makes the following updates:*

- *Remove bioentry(MGI:87904, 26, "actin,beta")*
- *Add bioentry(MGI:1923501, 26, "cDNA 0610007P08")*
- *Add term(26, element)*

Update exchange would propagate the effects to the **GeneOrg** *and* **MouseLab** *nodes, in this case, removing the tuples containing $MGI:87904$ from the* `gene`, `hasGene`, *and* `mousegene` *relations. The standard procedure for propagating the effects is a variant of the delta-rules based* count *algorithm [19], discussed in more detail in Section II.* □

C. Representing Mapping Changes

Given that schema mappings can be translated to Datalog programs, the task of adapting to mapping changes is clearly a generalization of view adaptation [10], [20], [21], where the standard approach has been to describe a change in a view definition as a *sequence of primitive updates* (e.g., add a column, project a column, etc.). In general, schema or mapping updates are indeed likely to be formed out of sequences of such steps (we in fact simulate these updates in generating the experimental workload for this paper, in Section V).

However, we argue against using this way to express changes to views, because it encourages a sequential process of adaptation, and enforces *a priori* constraints on allowable changes. Instead, we develop a method that looks *holistically* at the differences between old and new views. We assume we are simply given a new set of mappings or view definitions, and we still have access to the old definitions, as well as their materialized instances.

Example 6: After the CDSS mappings from Example 4 have been updated, we have:

```
(m1)  gene(E,N) :- bioentry(E,T,N), term(T,"gene").
(m3)  gene(E,N) :- bioentry(E,T,N), term(S,"gene"),
                     termsyn(T,S).
(m'2) mousegene(G,N) :- gene(G,N), hasGene(G,M),
                           orgname(M,"mus musculus").
```

As peers perform update exchange, we seek to recompute their data instances in compliance with the new mappings. □

D. Representing Data and Updates: $\mathbb{Z}$-Relations

The original work on CDSS represented updates to each relation as a pair of "update relations:" a table containing a set of tuples to insert plus a table of tuples to delete, where the same tuple could not appear more than once (even in both relations). Ideally we would like a cleaner formalism in which insertions and deletions could be combined in a way that is commutative and associative.

These goals led us to define the device of $\mathbb{Z}$*-relations* (introduced in our previous theoretical work [15], and closely related to the count-annotated relations used in classical view maintenance algorithms [19]). Here "updates" refers to both changes to source data (as in update exchange), and any ensuing updates to derived IDB relations that arise in performing update exchange and in performing multi-view adaptation.

Intuitively, a $\mathbb{Z}$-relation is like a bag (multiset) relation, except that tuple multiplicities may be either positive or negative. In the context of updates, positive multiplicities represent insertions, while negative multiplicities represent deletions. A virtue of $\mathbb{Z}$-relations is that they allow a uniform representation of data, and updates to data. If R is a table (a $\mathbb{Z}$-relation), and ΔR is an *update* to R (another $\mathbb{Z}$-relation), then the *result of applying* ΔR *to* R is just the $\mathbb{Z}$-relation $R \cup \Delta R$. Thus, update application reduces to computing a simple union query.

Example 7: Refer to Example 5. We can capture the updates in two $\mathbb{Z}$-relations, Δbioentry *and* Δterm, *representing changes to apply to* bioentry *and* term, *respectively:*

Δbioentry

MGI:*87904*	*26*	actin, beta	−1
MGI:*1923501*	*26*	cDNA *0610007P08*	1

Δterm

26	element	1

□

As defined in [15], the semantics of the relational algebra on $\mathbb{Z}$-relations is the same as for bag relations, except that the difference operator is allowed to yield negative tuple multiplicities. (In contrast, bag semantics uses "proper subtraction" and truncates negative multiplicities to zero.) We also extend our Datalog syntax to allow both ordinary *positive* rules, as well as *differential* rules, where the head of the rule is marked with a minus sign. For instance, we express the relational algebra query $q = r - s$, where r and s are unary, using a pair of rules `q(X) :- r(X)` and `-q(X) :- s(X)`.

The modified behavior of the difference operator under $\mathbb{Z}$-semantics is very useful for multi-view adaptation, as we argue in Section III-B. It also turns out to be "friendly" with respect to automated reasoning: for example, checking equivalence of relational algebra queries using difference is undecidable under bag semantics [22][4] and set semantics [23], but decidable in PSPACE for queries on $\mathbb{Z}$-relations [15]. We will show in Section III-C that this result extends even further to relational algebra queries using Skolem functions, and discuss how to optimize queries using materialized views with $\mathbb{Z}$-relations (the key to solving the main problems of this paper, discussed shortly).

Example 8: Refer to Figure 3, which consists of updates to be applied to the data instance of Figure 1, given the mapping m_1. The well-known delta rules *[19] reformulation can give the change* Δ`gene` *to apply to* `gene` *(a $\mathbb{Z}$-relation), given as inputs the changes* Δ`bioentry` *and* Δ`term` *(also $\mathbb{Z}$-relations) and the original source relations* `bioentry` *and* `term`*. The change* Δ`bioentry` *can be applied to* `bioentry`*, resulting in the $\mathbb{Z}$-relation* `bioentry`*'. From this intermediate relation, the base relations, and the deltas, we can compute a set of changes* Δ`gene`*, again a $\mathbb{Z}$-relation, to apply to* `gene`*, yielding* `gene`*'.* $\square$

E. Problem Statement

Our goal in this paper is to support efficient recomputation of multiple target instances—*multi-view adaptation*—for situations matching either or both of two subproblems. Here and in the rest of the paper, by “Datalog” we mean non-recursive Datalog extended with Skolem functions (which may be used freely in the heads or bodies of rules) and differential rules as discussed above, evaluated under $\mathbb{Z}$-semantics.

Subproblem 1: Supporting Changes to Mappings. During normal CDSS operation, an administrator may add, remove, or revise a schema mapping. Our goal is to take into account the effects of these modified mappings in recomputing each CDSS peer instance, when each peer next performs update exchange.

More precisely, we are given a Datalog program P, an EDB database instance I, an IDB database instance $J = P(I)$, and an arbitrarily revised version P' of P. The goal is to find an efficient plan to compute $J' = P'(I)$, possibly using J.

Subproblem 2: Supporting Changes to Data. Basic CDSS operation entails *intermittent participation* by the peers. Each peer only periodically performs an update exchange operation, which publishes a list of updates made recently by the peer's users. Between update exchange operations, there is a good chance that the set of mappings changed (as above) *and* several peers in the network have published updates. Thus, as the CDSS is refreshing a peer's data instance during update exchange, it must take into account any changes to mappings and any newly published updates.

In this case, we are given a Datalog program P, an EDB database instance I, an IDB database instance $J = P(I)$, an arbitrarily revised version P' of P, and a collection ΔI of updates to I. The goal is to find an efficient plan to compute $J' = P'(I \cup \Delta I)$, possibly using J.

[4]Actually, [22] proves that bag *containment* of unions of conjunctive queries is undecidable, but the extension to bag-equivalence is immediate, since A is bag-contained in B iff $A - B$ is bag-equivalent to $\emptyset$.

III. Basic Approach to Multi-View Adaptation

In contrast to previous work [10], which focused on taking *sequences of primitive updates to view definitions* and processing each update in step, our basic strategy for multi-view adaptation is to treat both problems uniformly as special cases of a more general problem of *optimizing queries using materialized views* (OQMV).

A. Optimizing Queries Using Materialized Views

The idea of handling changes to mappings (Subproblem 1) using OQMV is very natural: in this case, the materialized IDB instance $J = P(I)$ for the old version P of the Datalog program on I serves as our collection of materialized views, which can be exploited to compute the result $J' = P'(I)$ of the new Datalog program P'.

For handling changes to data (Subproblem 2), we cast the problem as an instance of OQMV as follows. Let $\mathbf{S} = \{S_1, \ldots, S_n\}$ be the EDB relations of P, with updates recorded in EDB relations $\Delta\mathbf{S} = \{\Delta S_1, \ldots, \Delta S_n\}$, and let $\mathbf{T} = \{T_1, \ldots, T_m\}$ be the IDB relations of P. We construct a new Datalog program P' from P by replacing every relation symbol R occurring in P by a relation symbol R', representing the updated version of R. Additionally, for each EDB relation $S_i \in \mathbf{S}$, we add to P' rules of the form

$$S'_i \leftarrow S_i \qquad \text{and} \qquad S'_i \leftarrow \Delta S_i$$

which apply the updates in ΔS_i to S_i, producing the new version S'_i. Now the goal is to find an efficient plan to compute P', given the materialized views of P.

Finally, in the most general case we also allow changes to data and mappings *simultaneously*. Here, the revised version P' of the Datalog program P is assumed to be given over IDB predicates $\mathbf{S}' = \{S'_1, \ldots, S'_n\}$ defined as above, rather than $\mathbf{S}$.

In the remainder of this section, we present foundational aspects of our approach to OQMV in detail, beginning with our novel use of *differential plans* (Section III-B), a term rewrite system for OQMV which supports differential plans (Section III-C), and an extension of our approach to exploit provenance information of the kind present in CDSS (Section III-D). We focus mainly on OQMV from a logical perspective; Section IV will present our actual implementation of these ideas.

B. Differential Plans

Although in data exchange and CDSS, view definitions are typically positive, when view definitions are modified we often want to compute a *difference* between the old and new view. For example, if a view v is defined by two Datalog rules m_1, m_2, and rule m_2 is replaced by m'_2 to produce the new version v' of the view, one plan to consider for adapting v into v' is to compute m_2, subtract it from v, then compute m'_2 and union in the result. The result is precisely v'.[5]

Carrying this plan out as a query requires considering query reformulations using a form of negation or difference.

[5]Under bag or $\mathbb{Z}$-semantics, but *not* (in general) set semantics! (Intuitively, this is because the identity $(A \cup B) - B \equiv A$ fails under set semantics.)

Source table updates as ℤ-relations

Δbioentry

MGI:87904	26	*actin, beta*	−1
MGI:1923501	26	*cDNA* 0610007P08	1
MGI:1923503	26	*cDNA* 0610006L08	2

Δterm

26	*gene*	1
26	*element*	1

Delta rules

```
Δgene(G,N)  :- Δbioentry(E,T,N), term(T,"gene").
Δgene(G,N)  :- bioentry'(E,T,N), Δterm(T,"gene").
gene'(G,N)  :- gene(G,N).
gene'(G,N)  :- Δgene(G,N).
```

Target table updates as ℤ-relations

bioentry′

MGI:88139	26	*BCL2-like* 1	1
MGI:1923501	26	*cDNA* 0610007P08	1
MGI:1923503	26	*cDNA* 0610006L08	2

Δgene

MGI:87904	*actin, beta*	−1
MGI:88139	*BCL2-like 1*	1
MGI:1923501	*cDNA* 0610007P08	2
MGI:1923503	*cDNA* 0610006L08	4

gene′

MGI:88139	*BCL2-like 1*	2
MGI:1923501	*cDNA* 0610007P08	2
MGI:1923503	*cDNA* 0610006L08	4

Fig. 3. Updates and delta rules with ℤ-relations.

We explain how we incorporate such reformulations in our approach to OQMV in the next section.

C. Rule-Based Reformulation

The foundation of our strategy for OQMV in our setting is a small set of equivalence-preserving *rewrite rules*, forming a *term rewrite system*. The approach here extends the theoretical results of our earlier work [15] to deal with a richer class of queries and views, where queries and views may use Skolem functions. In Section IV we will use these rewrite rules as the core building blocks of a practical reformulation algorithm.

Briefly, the term rewrite system contains four rules:

1) **view unfolding.** Replace an occurrence of an IDB predicate v in a rule for IDB predicate q by its definition. If q has n rules and v has m rules, the result will have $n+m-1$ rules for q.
2) **cancellation.** If r_1, r_2 are rules in the definition for q that have different signs but are otherwise isomorphic, then remove them.
3) **view folding.** View unfolding in reverse: replace occurrences of the bodies of an IDB view definition v in the rules for q with the associated view predicate. If q has $n+m$ rules and v has m rules, the result will have $n+1$ rules for q.
4) **augmentation.** Cancellation in reverse: if r_1, r_2 are isomorphic rules with q in their heads, then negate one of them and then add both to the definition of q.

We defer formal definitions of these rewrite rules to the long version of this paper, and here just illustrate with some examples of the rules in action.

Example 9: Consider the view definition for gene *that was updated in Example 6. The old and new versions of* gene*'s definition (we refer to these as* gene *and* gene′*, respectively), may be written in Datalog syntax as follows:*

```
gene(G,N)   :- bioentry(G,T,N), term(T,"gene").
gene'(G,N)  :- bioentry(G,T,N), term(T,"gene").
gene'(G,N)  :- bioentry(G,T,N), term(S,"gene"),
               termsyn(T,S).
```

Since the first rule for gene' *is identical to the definition of* gene*, we can use view folding to produce*

```
gene'(G,N)  :- gene(G,N).
gene'(G,N)  :- bioentry(G,T,N), term(S,"gene"),
               termsyn(T,S).
```
□

Example 10: Continuing with Example 8, the delta rules plan for recomputing view gene *shown in Figure 3 can be seen as a reformulation of the Datalog query* Δ*gene*,

```
 Δgene(G,N)  :- bioentry'(E,T,N), term'(T,"gene").
-Δgene(G,N)  :- gene(G,N).
```

using the materialized views bioentry'*,* term'*, and* gene *as shown below:*

```
bioentry'(E,T,N) :- bioentry(E,T,N).
bioentry'(E,T,N) :- Δ bioentry(E,T,N).
term'(T,"gene") :- term(T,"gene").
term'(T,"gene") :- Δterm(T,"gene").
gene(G,N) :- bioentry(E,T,N), term(T,"gene").
```

The reformulation involves unfolding all occurrences of bioentry' *and* term'*, then applying cancellation, and then using folding with* bioentry' *and* term'*.* □

A conceptually important fact is that by repeatedly applying these four rules, we can in principle find *any* equivalent reformulation of a Datalog query (under ℤ-semantics):

Theorem 1: The above term rewrite system is sound and complete *wrt ℤ-semantics. That is, for any $P, P' \in$ Datalog, we have P rewrites to P' iff P and P' are equivalent under ℤ-semantics.*

The proof of completeness (omitted here) makes use of the fact that the closely related problem of checking query *equivalence* is decidable under ℤ-semantics:

Theorem 2: Equivalence of queries expressed in relational algebra extended with Skolems under ℤ-semantics is decidable in PSPACE. The problem remains decidable for deciding equivalence of queries with respect to a set of materialized views, where views and queries are expressed in relational algebra extended with Skolems.

This is a straightforward extension of a result from our previous work [15] to incorporate Skolem functions, and is nevertheless surprising because the same problems are, as already noted earlier, undecidable under set semantics or bag semantics, even without Skolem functions.

Although the term rewrite system is complete, the space of all possible rewritings is very large—indeed, infinite! (Augmentation, for example, can always be performed *ad infinitum.*) A practical implementation can afford to explore only a small portion of the search space, in a cost-based manner, with the goal of finding a good (not necessarily optimal) plan quickly. Section IV describes how we accomplish this.

D. Exploiting Provenance

In performing multi-view adaptation, it is often useful to be able to "separate" the different disjuncts of a union, or to "recover" values projected away in a view. We would like, therefore, some sort of index structure capturing this information that can be exploited for efficient adaptation. In fact, such a structure already exists in CDSSs in the form of *provenance information* [2], [24], [25]. Intuitively, CDSS provenance records, for each IDB tuple, the *ways that tuple could be derived* from other facts in the database. To accomplish this, the CDSS maintains a *provenance table* for each mapping rule, which captures the relationship between the source tuples used to derive a target tuple, and the target tuple itself.

Example 11: For the two mappings from Example 2, we create a relation that, given the definition of the mapping, records sufficient information to allow us to recover the source and target tuples. This requires us to store the values of the bound variables in the provenance table p_m for each mapping m, yielding tables $p_{m_1}(E,N,T)$ for rule m_1, and $p_{m_2}(G,N)$ for rule m_2. For the data instance of Figure 1, the tables are:

p_{m_1}

E	*N*	*T*
MGI:*88139*	*26*	BCL2-like 1
MGI:*87904*	*26*	actin, beta

p_{m_2}

G	*N*
MGI:*88139*	BCL2-like 1
MGI:*87904*	actin, beta

□

The population and maintenance of provenance tables in CDSS is accomplished via a simple translation of Datalog rules, in effect reducing the problem of update exchange with provenance to the same problem without provenance (but providing additional optimization opportunities).

Formally, for a Datalog rule m in the collection P of mappings

$$A \leftarrow B_1, \ldots, B_n$$

where $X_1, \ldots, X_n$ is the list of variables occurring in the body of the rule (in some arbitrary order), we replace m by the pair of rules

$$\begin{array}{rcl} p_m(X_1,\ldots,X_n) & \leftarrow & B_1,\ldots,B_n, \\ A & \leftarrow & p_m(X_1,\ldots,X_n). \end{array}$$

where p_m is a fresh n-ary IDB predicate called the *provenance table* for m. In other words, we split the original rule into a new projection-free rule to populate the provenance table, and a second rule that performs a projection over the provenance table. Because the rule for p_m is projection-free, it preserves potentially useful information that we can (and will) exploit later for optimization. The table also identifies the particular contribution of this rule to the view definition.

Example 12: The complete set of rules for provenance and target tables would now be:

```
(m1) p_m1(E,T,N) :- bioentry(E,T,N), term(T,"gene").
(m2) p_m2(G,N) :- gene(G,N), hasGene(G,12).
(m3) gene(E,N) :- p_m1(E,T,N).
(m4) mousegene(G,N) :- p_m2(G,N).
```
□

Importantly, from the point of view of OQMV, the provenance tables in CDSS are "just more views." Therefore they are *automatically exploited* (when present) by the rule-based reformulation scheme presented in Section III-C. Moreover, traditional data exchange systems can be easily modified to support this form of provenance, and hence benefit from its presence during multi-view adaptation, by pre-processing the input set of schema mappings using the transformation above.

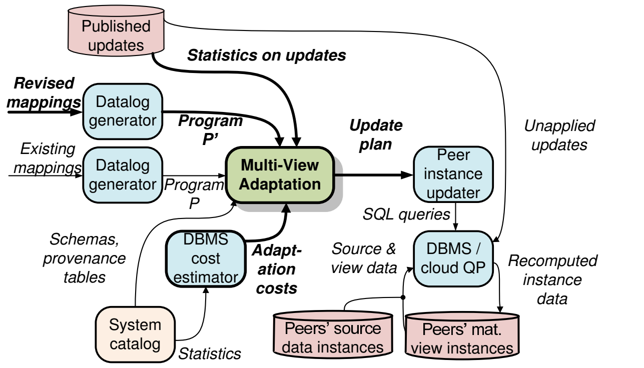

Fig. 4. CDSS system architecture with multi-view adaptation support (new components/dataflows highlighted in bold)

It is worth noting that the tradeoffs to maintaining provenance are much the same as those for other indexing methods: creating or updating a provenance relation adds very little computation overhead, but it of course takes more space and creates higher I/O costs. See [2] for a detailed analysis of the overhead.

Example 13: Suppose the mappings from Example 12 are revised, such that `gene` *now retains an additional attribute from* `bioentry`*, which is ignored in the updated rule for* `mousegene`*:*

```
(m5)  gene'(E,T,N) :- bioentry(E,T,N), term(T,"gene").
(m6)  mousegene'(G,N) :- gene'(G,T,N), hasGene(G,M),
                          orgname(M,"mus musculus").
```

Without provenance tables, we can infer (using the rewrite rules from Section III-C) that `mousegene'` *has not actually changed, so we revise m_6 to read*

```
(m6) mousegene'(G,N) :- mousegene(G,N),
```

However we are forced to recompute `gene'` *from scratch. With provenance tables, though, we can also recompute* `gene'` *more efficiently, directly from the provenance table associated with* `gene`*:*

```
(m5)  gene'(E,T,N) :- p_m1(E,T,N).
```
□

IV. Implementing Adaptation

Next we discuss how we translate the basic approach to multi-view adaptation presented in Section III into a practical implementation, in particular, in the context of ORCHESTRA.

A. Architecture Overview

Our work in this paper involves the addition of a new *multi-view adaptation module* into the CDSS architecture, as indicated by the boldfaced components in Figure 4. The multi-view adaptation module is given Datalog programs based on both old and new mappings, along with information about any pending updates to source data.

The basic strategy of the multi-view adaptation module is to pair the rules for reformulation presented in the last section with cost estimation (invoking a proxy for the DBMS' cost estimator) and a cost-based search strategy. The *output* of the module is an *update plan*. This is not a physical query plan, but rather a sequence of Datalog queries which construct the updated version of the source relations and views. These are then translated to SQL and executed in sequence.

The rest of this section describes the adaptation engine, focusing first on the high-level ideas (Section IV-B) before diving into implementation details (Section IV-C).

B. Cost-Based Search

The multi-view adaptation module begins by performing a topological sort of the updated view definitions in P' (since some views may be defined in terms of other views), then processes the views sequentially, beginning with those views that do not depend on any others. For each view, the engine first finds an optimized plan (using the procedure described below), then invokes the underlying DBMS to execute the plan and materialize the view. The view definition is then added to the list of materialized views, so that it is available to subsequently processed views.

As already noted, the space of possible reformulations is far too large to be explored exhaustively. Instead, our system uses an iterative *hill climbing* algorithm that explores a smaller portion of the space of rewritings. The "exhaustiveness" of the algorithm can be tuned via several parameters that we describe below, and also takes into account a *time budget*. By setting the budget to some fraction of the estimated cost of the input plan, we guarantee that the algorithm imposes at most a bounded overhead in the worst case where no better plan can be found even after extensive search.

The basic idea in the procedure is to start at the bottom of the hill (by unfolding all view predicates in the query q and applying cancellation), then use folding and augmentation to climb up towards plans of lesser cost. One technical issue here involves the use of augmentation: at any point in the process, augmentation can be applied in an unbounded number of ways, and moreover, applying it will invariably result in a plan of higher cost. Meanwhile the real benefit of augmentation is that it enables subsequent folding operations that would not otherwise be possible.

Example 14: Consider a query q and view definition v

```
q(X) :- r(X,Z), r(Z,d), t(X).
q(X) :- r(X,d), r(d,Z), t(X).
v(U,V) :- r(U,W), r(W,V).
v(U,V) :- r(U,V), r(V,W).
v(U,V) :- s(U,c,V).
```

Note that although the bodies of two of the rules in v have matches in q, the third rule in v does not, hence view folding cannot be directly applied. However, by applying augmentation and then view folding, we obtain

```
q(X) :- v(X,d), t(X).   -q(X) :- s(X,c,d), t(X).   □
```

We therefore use in our algorithm a modified version of the term rewrite system from Section III where augmentation and view folding are replaced by a single compound rule

5) **view folding with remainder.** Replace occurrences of the bodies of *some* bodies of a view definition v in the rules for q with the associated view predicate. Account for the unmatched bodies by adding *remainder* terms.

In Example 14, view folding with remainder can be applied to produce the reformulation, with `-q(X) :- s(X,c,d), t(X)` being the remainder.

C. Algorithm Implementation

The algorithm uses several data structures in optimizing a query:

- A *pending queue A* of plans, ordered by their estimated costs, implemented using a priority queue.
- A *completed set B* of plans for the query, disjoint from A, implemented using a hash set.
- Temporary *scratch queues* C_1, C_2 of plans and costs, also sorted by estimated cost using priority queues.

Rewrite rule (5) is encapsulated within a *reformulation iterator*: given a plan p and a materialized view v, $\text{iter}(p,v)$ is the reformulation iterator which, via repeated calls to $\text{iter}(p,v).\text{next}()$, returns all plans that can be produced from p and v using a single application of the rule.[6] While conceptually straightforward, the implementation of the reformulation iterator class is probably the most intricate piece of code in our implementation.

Pseudocode for the algorithm appears as Algorithm 1. Given program P, query q, and time budget t, it starts looping with B empty, A containing the original plan for q, and the unfolded version of q. Each step chooses the cheapest plan from A, moves it to B, uses the reformulation iterator to find adjacent plans in the search space, and then adds them (and their estimated costs) to A. The algorithm is greedy in the sense that it always explores the most promising path first, but a certain number of alternatives are also kept in the pending queue to be considered if time allows.

The scope of the algorithm's search is controlled by several tuning parameters: c_s, the maximum number of rewritings added to A per step; c_v, the maximum number of rewritings using a particular view added to A per step; and c_q, the maximum allowed size of the pending queue. c_v is introduced (in addition to c_s) to ensure diversity despite limited resources.

Note that in line 2 of the algorithm, we insert the original plan for q into A. This ensures that if reformulation does not find a better alternative, the original plan itself will be returned, and q will be "recomputed from scratch."

To get good performance, we also specially optimize two main operations on the critical path.

Isomorphism testing. The search algorithm frequently tests *isomorphism* of rules or view definitions. This must be done when adding plans to the pending queue or completed set, to detect if the plan is already present. It is also used heavily as a subroutine inside the reformulation iterator. We use *hash consing* to make this a fast operation: whenever a new rule

[6]If p and v contain many self-joins, there can be exponentially many such plans, but the number is always finite.

Algorithm 1 optimizeUsingViews(P, q, t)

```
$A$.clear(), $B$.clear()
$A$.insert($q$, cost($q$))
$q' \leftarrow$ unfold($P$, $q$)
$A$.insert($q'$, cost($q'$))
while current time - start time $< t$ do
    $(p, c) \leftarrow A$.poll()
    $B$.insert($p, c$)
    $C_1$.clear()
    for all $v \in P$ do
        $C_2$.clear()
        it $\leftarrow$ iter($p, v$)
        while it.hasNext() do
            $p' \leftarrow$ it.next()
            $C_2$.insert($p'$, cost($p'$))
        end while
        for $i = 1$ to min($C_2$.size(), $c_v$) do
            $C_1$.insert($C_2$.poll())
        end for
    end for
    count $\leftarrow 0$
    while $C_1$.notEmpty() and count $< c_s$ and $B$.size() $< c_q$ do
        $(p', c') \leftarrow C_1$.poll()
        if neither $A$ nor $B$ contains $(p', c')$ then
            $A$.insert($p', c'$)
            count $\leftarrow$ count $+1$
        end if
    end while
end while
return cheapest plan for $q$ in $A$ or $B$
```

or view definition is allocated, we first check to see whether an isomorphic rule or definition already exists; if so, we reuse the existing object. Outside of allocation, we can check isomorphism by simply comparing pointers.

Cost estimation is also frequently performed. The underlying DBMS' estimator (`SQL explain`) proved to be too slow, involving a round-trip to the DBMS and a full planning operation inside the DBMS. We therefore implemented our own proxy cost estimator, which constructs a reasonable physical plan via heuristics, and returns its cost.

D. RDBMS Implementation Issues

Implementing multi-view adaptation over a conventional relational database system requires us to deal with two additional subtleties, since neither $\mathbb{Z}$-relations nor provenance are built into a standard database management system.

Support for $\mathbb{Z}$-relation semantics is accomplished via a straightforward encoding scheme where multiplicities are recorded using an explicit integer-valued `count` attribute, in the spirit of [19]. In the SQL code generated from Datalog rules, joins in a `select-from-where` clause multiply the count attributes of joined tuples; unions and projections sum counts; and difference operations subtract counts.

Provenance information is stored in relational tables—one per mapping—following the implementation of [2]. Such mapping tables are automatically exploited (when present) without any changes to the core adaptation engine, as explained in Section III-D.

V. Experimental Evaluation

We now study how effective our multi-view adaptation techniques are in recomputing ORCHESTRA data instances after mappings and data change. ORCHESTRA can use a variety of centralized or distributed query engines to do data exchange and update exchange; in this case we targeted PostgreSQL 8.4. The adaptation engine comprised approximately 9,500 lines of Java code.

While schema mappings are known to frequently evolve (motivating, e.g., [26]), we are unaware of any standard benchmark for mapping evolution suitable for our needs.[7] Hence we (1) take the CDSS mapping generator of [2] (which creates simulated CDSS instances and mappings based on bioinformatics data), and then (2) develop synthetic updates to the mappings and schemas, informed by the schema modification operation distributions observed in [29].

Peers, instances, and mappings. The CDSS mapping generator takes the SWISS-PROT protein database and "re-factors" it to produce multiple peer's relations, by randomly choosing n attributes from the original database schema. In this way we create 24 peer relation instances. Long strings from SWISS-PROT are encoded in the database using a CLOB-style encoding. As it creates peers, the generator adds mappings (view definitions), consisting of 1-4 mapping rules with 1-5 atoms each. For 16 of the peers' relations, we create corresponding "source" relations populated with approximately 500K-600K tuples by uniform random sampling from the original instance. Data from these source relations is mapped directly to the associated peers' relations, as well as to the other peer relation instances. For each source relation and materialized view, we also create a primary index on the first column.

Synthetic schema changes. We randomly generate change workloads, using compositions of primitive modification operations shown in Table I following the frequencies listed in Table II. For adding, dropping, and renaming columns, we use the distributions empirically observed for Wikipedia in [29, Table 4]. We also add common modifications specific to schema mappings, such as adding or removing a join with a correspondence table.

View definitions are generated as follows:

- For each view, an arity k and a number of rules are chosen uniformly at random from the ranges listed above; a random sequence $t_1, \ldots, t_k$ of distinct attribute types are selected for the columns.
- Each rule was generated by randomly joining a set of source relations "covering" the attribute types in the head.

For the settings above, the average number of tuples in each materialized view v is roughly 250K times its number of rules.

Experimental setup and questions. Experiments were run under OpenJDK 1.6.0 on an Intel Xeon X3450 processor with 8GB RAM and 64-bit Ubuntu Server 10.04. Query processing

[7]For example, STBenchmark [27] uses a fundamentally different data model, nested relations; and the synthetic benchmark of [28] addresses only schema evolution.

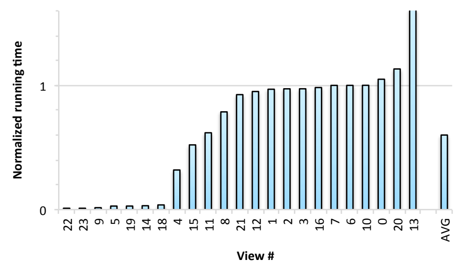

Fig. 5. Mixed workload (mapping changes only); no provenance tables

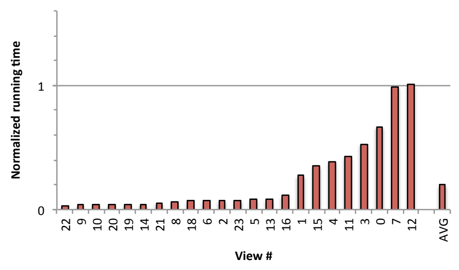

Fig. 6. Mixed workload (mapping changes only); with provenance tables

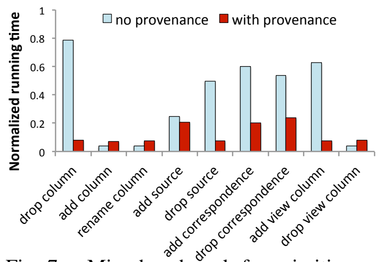

Fig. 7. Microbenchmark for primitive change types

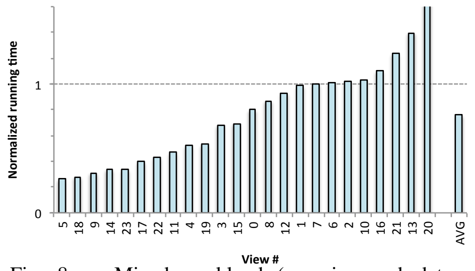

Fig. 8. Mixed workload (mapping and data changes); no provenance tables

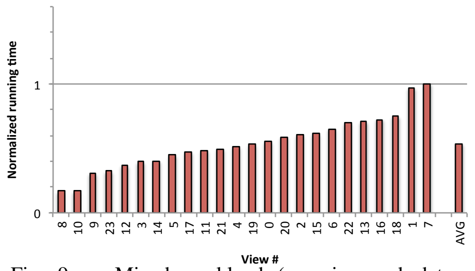

Fig. 9. Mixed workload (mapping and data changes); with provenance tables

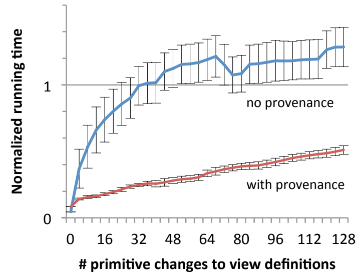

Fig. 10. Running time vs. number of primitive changes

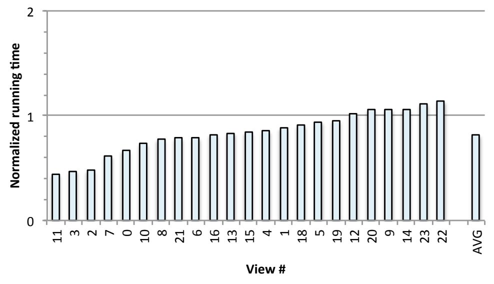

Fig. 11. Sequential vs. combined change handling (mapping and data changes); no provenance

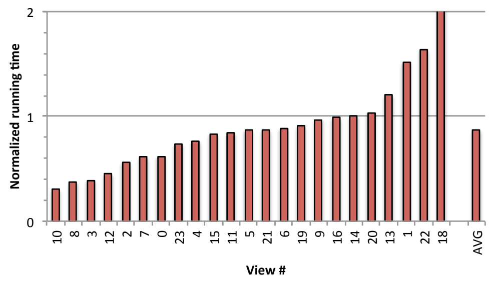

Fig. 12. Sequential vs. combined change handling (mapping and data changes); with provenance tables

in this configuration was primarily CPU rather than I/O bound. We set the constants described in Section IV-C as follows: the search considers up to $c_s = 8$ rewritings per step, at most $c_v = 4$ rewritings per view, and a max pending queue length of $c_q = 128$.

We consider the following questions:

1) What are the performance gains of multi-view adaptation?
2) Do different mapping changes have different savings?
3) How does the multi-view adaptation algorithm handle mixed workloads of mapping changes plus data updates?
4) When does it make more sense to do full recomputation, as opposed to incremental recomputation?

A. Multi-view adaptation performance.

We first study the benefits of our query reformulation methods (compared to complete re-computation). Using the distributions in Table II, we generated compositional sequences of 24 primitive changes to apply to the view definitions, and then ran multi-view adaptation. We timed the combined costs of reformulation, optimization, and execution.

Figure 5 shows the time to adapting each view, normalized against the running time of recomputing the view from scratch. We present the views along the x-axis in decreasing order of speedup (each labeled by its view ID). We see notable speedups for just under half of the queries. Only View 13 shows a significant performance penalty: this is due to the overhead of query reformulation, combined with minor errors in the query cost estimation. The average speedup (the last bar labeled "AVG," is around 40%).

Recall that data provenance provides, in effect, an *index* of derivations for the tuples in a view, separated by each conjunctive subquery. Figure 6 repeats the same experiment when provenance relations are available; sorting the views by speedup results in a different ordering from the previous case, so we again include view IDs for direct comparison. Here we see significantly improved performance versus the prior case: two views have the same performance as before, and all others are sped up. The average speedup is around 80%.

B. Adaptation microbenchmark.

Clearly there was wide variation across views in the previous experiment. To better understand why, we isolate performance with different *types* of mapping changes: see Figure 7. Again, we separately measure speedups based on whether provenance is available; and provenance makes a significant difference here. For dropping a source relation

TABLE I
PRIMITIVE MODIFICATION OPERATIONS.

Add column. *Add a new column to a source relation, with a default value. For a source relation r, of arity n we pick a fresh default value c and model the updated relation as a new materialized view*

```
r'(X_1, ..., X_n, c) :- r(X_1, ..., X_n).
```

We then update views that use r to use r' instead, by replacing every occurrence of `r(X_1, ..., X_n)` *with* `r'(X_1, ..., X_n, c)`.

Drop column. *Drop a column from a source relation. For a source relation r, we model the updated relation as a new materialized view*

```
r'(X_1, ..., X_{n-1}, c) :- r(X_1, ..., X_n).
```

that replaces the dropped column with a fresh default value c. We then replace all occurrences of r in view rules with r'.

Move column. *Swaps two columns within a source relation. For a source relation r, we model the updated relation as a new materialized view*

```
r'(X_n, X_2, ..., X_{n-1}, X_1) :- r(X_1, ..., X_n).
```

We then update the views that use r to use r' instead, by replacing every occurrence of `r(X_1, ..., X_n)` *with* `r'(X_n,X_2, ..., X_{n-1}, X_1)`.

Add source. *Add a new rule to a view v, randomly generated according to the procedure described above.*

Drop source. *Drop a rule from a view v containing at least 2 rules.*

Add correspondence. *Choose a variable X in the head of a rule for view v, choose a source relation r with an attribute of the same type at position i, and choose another position $j \neq i$ in the source relation. Modify the rule for v by replacing X in the head with a fresh variable Y, and by adding an r atom to the rule containing X at position i, Y at position j, and fresh variables at all other positions.*

Drop correspondence. *Choose a variable X in the head of a rule for view v, choose a predicate atom A in the body of the rule containing X and a join variable Y, and choose a predicate atom B in the body of the rule containing Y and another variable Z not occurring in the head of the rule. Replace X with Z in the head of the rule, and remove atom A.*

Add view column. *Choose a position i in the head of a rule v. For each rule for the view, choose a body variable X not occurring in the head of the rule, and insert X at position i in the list of variables in the head of the rule.*

Drop view column. *Choose a position i in the head of a rule v. For each rule for view v, delete variable at that position from the head.*

TABLE II
FREQUENCY OF PRIMITIVE CHANGES.

change	**% freq.**	**change**	**% freq.**
add column	16	drop column	9
move column	5	add source	16
drop source	9	add correspondence	16
drop correspondence	9	add view column	12
drop view column	8		

column, there is high speedup if provenance is available (since provenance tables directly support this operation), whereas without provenance we may need to do a full recomputation. Adding or renaming a source relation column generally does not require recomputation, and is inexpensive. Adding a source merely involves unioning in its results. Dropping a source, adding or removing a join with a correspondence table, or adding a column to the view, achieve their best speedup when provenance is available. Finally, dropping a column in a view is inexpensive as it can be done directly over the materialized relation.

C. Simultaneous mapping and data changes.

Part of the promise of our approach is that it can simultaneously handle updates to data and to the mappings. We repeat the workload sequences of Section V-A, but also apply 1,000 data update operations (insertions and/or deletions) to each source table. For this "mixed workload," we are forcing incremental update propagation as well as view adaptation. Figures 8 and 9 correspond to Figures 5 and 6, respectively, for this case. In these cases, the cost of additionally applying the updates makes the operations more expensive than in the mapping-change-only case. We continue to see a clear benefit from our multi-view adaptation techniques, with an average of an *additional* 15% speedup without provenance, and 46% with provenance.

The benefits of multi-view adaptation over a mixed workload exceed those of applying the mapping changes and the updates separately, as we show next. Figures 11 and 12 isolate the performance impact of combining the optimization of the data and mapping changes. Here, we compare, for the same workloads, the costs of propagating the data updates (deltas) through the original views first, then performing the recomputation; versus applying the deltas and mapping changes at the same time. We see relative speedups of 19% without provenance and 13% with provenance relations.

D. Limits of view reuse.

Finally, we investigate when the optimal multi-view adaptation strategy degrades to simply recomputing the views from the base relations, where the changes become too extensive to make adaptation viable. We use the experimental setup of Section V-A, but vary the number of mapping changes in the input workload from 0-128 updates. Figure 10 shows that without provenance information, the threshold is between 24-32 changes. (The line is not completely monotonic because later updates may reverse the effects of previous ones.) With provenance tables, even sequences of 128 changes show significant benefit; extrapolation of the data suggests that the threshold would be after around 256 changes.

Summary of results. Overall, we conclude from our experiments that (1) our strategy, of using query reformulation with differences, significantly speeds the refreshing of materialized view instances; (2) provenance tables provide significant benefits, as seen in the microbenchmarks as well as the workload sequences; (3) processing mapping changes and deltas together provides significant benefit; (4) especially with provenance tables, the reformulation approach is superior even for significant numbers of changes to the mappings.

VI. RELATED WORK

Our work targets data exchange [1] and CDSS [2], [16], where autonomous databases are interrelated through schema mappings, and where data provenance [25] is maintained. We focus on the Datalog rules (view definitions) generated from the schema mappings. Mapping evolution [30] or adaptation [13] is a complementary problem, in which one of the schemas changes and the view or mapping definitions need to be updated.

Support for evolution of schemas, and of view definitions (*view adaptation*), has previously been studied in data warehouse settings [20], [21], where the emphasis has been on handling schema modification operations, or sequences thereof, as opposed to directly using cost-based query reformulation of the new view definition. Evolution has also been studied when multiple models, possibly at different levels of abstraction, must be co-evolved [26].

Our problem of using materialized views to more efficiently compute results, and its basic techniques, are related to methods for materialized view selection [31], [32], [33]. Such work develops strategies for choosing which common subexpressions to consider materializing and sharing, based on workloads; in fact, [33] uses a transformation rule-based approach to enumerate the search space. Our setting differs in that the materialized views *already* exist; we are searching over the ways of combining (and differencing) views. Our problem is a case of the well-studied problem *optimizing queries using views* [34], [14], for which less attention has been given to queries and views using negation (see [35]). Our work uses a form of negation, the difference operator under $\mathbb{Z}$-semantics, that is incomparable to the classical set difference.

Finally, we propagate changes made to the base relations to the derived instances; this is a case of (deferred) view maintenance [19], [21], [36], [37].

VII. Conclusions and Future Work

This paper presents a novel solution to *multi-view adaptation*, the problem of efficiently supporting changes to schema mappings or view definitions, by applying new techniques for optimizing queries using materialized views and differential query plans. We presented a practical, cost-based search strategy for handling changes to mappings *and data* in a seamless way. Using an implementation built over an off-the-shelf DBMS, we showed that our approach provides significant speedups, particularly when provenance tables are present.

In the future, we hope to extend our results to recursive Datalog, perhaps via a hybrid approach that uses $\mathbb{Z}$-relations when possible and falls back to classical techniques otherwise.

References

[1] R. Fagin, P. Kolaitis, R. J. Miller, and L. Popa, “Data exchange: Semantics and query answering,” *TCS*, vol. 336, pp. 89–124, 2005.

[2] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen, “Update exchange with mappings and provenance,” in *VLDB*, 2007, amended version available as Univ. of Pennsylvania report MS-CIS-07-26.

[3] S. Dessloch, M. A. Hernández, R. Wisnesky, A. Radwan, and J. Zhou, “Orchid: Integrating schema mapping and etl,” in *ICDE*, 2008.

[4] A. Y. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov, “Schema mediation in peer data management systems,” in *ICDE*, March 2003.

[5] W. Gatterbauer and D. Suciu, “Data conflict resolution using trust relationships,” in *SIGMOD*, 2010.

[6] N. E. Taylor and Z. G. Ives, “Reconciling while tolerating disagreement in collaborative data sharing,” in *SIGMOD*, 2006.

[7] M. A. Hernandez, R. J. Miller, and L. M. Haas, “Clio: A semi-automatic tool for schema mapping,” in *SIGMOD*, 2001.

[8] A. Fuxman, P. G. Kolaitis, R. J. Miller, and W.-C. Tan, “Peer data exchange,” in *PODS*, 2005.

[9] Z. G. Ives, T. J. Green, G. Karvounarakis, N. E. Taylor, V. Tannen, P. P. Talukdar, M. Jacob, and F. Pereira, “The ORCHESTRA collaborative data sharing system,” *SIGMOD Rec.*, 2008.

[10] A. Gupta, I. S. Mumick, J. Rao, and K. A. Ross, “Adapting materialized views after redefinitions: techniques and a performance study,” *Inf. Syst.*, vol. 26, no. 5, 2001.

[11] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin, “Translating web data.” in *VLDB*, 2002.

[12] BioSQL project home page, http://www.biosql.org.

[13] C. Yu and L. Popa, “Semantic adaptation of schema mappings when schemas evolve,” in *VLDB*, 2005.

[14] A. Y. Halevy, “Answering queries using views: A survey,” *VLDB J.*, vol. 10, no. 4, 2001.

[15] T. J. Green, Z. G. Ives, and V. Tannen, “Reconcilable differences,” *Theory of Computing Systems*, vol. 49, no. 2, 2011.

[16] L. Kot and C. Koch, “Cooperative update exchange in the Youtopia system,” in *Proc. VLDB*, 2009.

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

[18] A. Calì, G. Gottlob, and M. Kifer, “Taming the infinite chase: Query answering under expressive relational constraints,” in *KR*, 2008.

[19] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, “Maintaining views incrementally,” in *SIGMOD*, 1993.

[20] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, “Update rewriting and integrity constraint maintenance in a schema evolution support system: Prism++,” *PVLDB*, vol. 4, no. 2, 2010.

[21] A. Koeller and E. A. Rundensteiner, “Incremental maintenance of schema-restructuring views,” in *EDBT*, 2002.

[22] Y. E. Ioannidis and R. Ramakrishnan, “Containment of conjunctive queries: Beyond relations as sets,” *TODS*, vol. 20, no. 3, pp. 288–324, 1995.

[23] Y. Sagiv and M. Yannakakis, “Equivalences among relational expressions with the union and difference operators,” *J. ACM*, vol. 27, no. 4, pp. 633–655, 1980.

[24] T. J. Green, G. Karvounarakis, and V. Tannen, “Provenance semirings,” in *PODS*, 2007.

[25] J. Cheney, L. Chiticariu, and W. C. Tan, “Provenance in databases: Why, how, and where,” *Foundations and Trends in Databases*, vol. 1, no. 4, 2009.

[26] J. F. Terwilliger, P. A. Bernstein, and A. Unnithan, “Worry-free database upgrades: automated model-driven evolution of schemas and complex mappings,” in *SIGMOD*, 2010.

[27] B. Alexe, W. C. Tan, and Y. Velegrakis, “STBenchmark: towards a benchmark for mapping systems,” *PVLDB*, vol. 1, no. 1, 2008.

[28] P. A. Bernstein, T. J. Green, S. Melnik, and A. Nash, “Implementing mapping composition,” in *VLDB*, 2006.

[29] C. Curino, H. J. Moon, L. Tanca, and C. Zaniolo, “Schema evolution in Wikipedia – toward a web information system benchmark,” in *ICEIS (1)*, 2008.

[30] Y. Velegrakis, R. J. Miller, and L. Popa, “Preserving mapping consistency under schema changes,” *The VLDB Journal*, vol. 13, no. 3, 2004.

[31] H. Gupta and I. S. Mumick, “Selection of views to materialize under a maintenance cost constraint.” in *ICDT*, 1999.

[32] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham, “Materialized view selection and maintenance using multi-query optimization,” in *SIGMOD*, 2001.

[33] D. Theodoratos, S. Ligoudistianos, and T. K. Sellis, “View selection for designing the global data warehouse,” *TKDE*, vol. 39, no. 3, 2001.

[34] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim, “Optimizing queries with materialized views,” in *ICDE*, 1995.

[35] F. Afrati and V. Pavlaki, “Rewriting queries using views with negation,” *AI Commun.*, vol. 19, pp. 229–237, August 2006.

[36] W. Labio, J. Yang, Y. Cui, H. Garcia-Molina, and J. Widom, “Performance issues in incremental warehouse maintenance,” in *VLDB*, 2000.

[37] J. J. Lu, G. Moerkotte, J. Schue, and V. Subrahmanian, “Efficient maintenance of materialized mediated views,” in *SIGMOD*, 1995.