Theo yêu cầu của khách hàng, trong một năm qua, chúng tôi đã dịch qua 16 môn học, 34 cuốn sách, 43 bài báo, 5 sổ tay (chưa tính các tài liệu từ năm 2010 trở về trước) Xem ở đây

DỊCH VỤ DỊCH TIẾNG ANH CHUYÊN NGÀNH NHANH NHẤT VÀ CHÍNH XÁC NHẤT

Chỉ sau một lần liên lạc, việc dịch được tiến hành

Giá cả: có thể giảm đến 10 nghìn/1 trang

Chất lượng: Tạo dựng niềm tin cho khách hàng bằng công nghệ 1.Bạn thấy được toàn bộ bản dịch; 2.Bạn đánh giá chất lượng. 3.Bạn quyết định thanh toán.

Tài liệu này được dịch sang tiếng việt bởi:

www.mientayvn.com

Từ bản gốc:

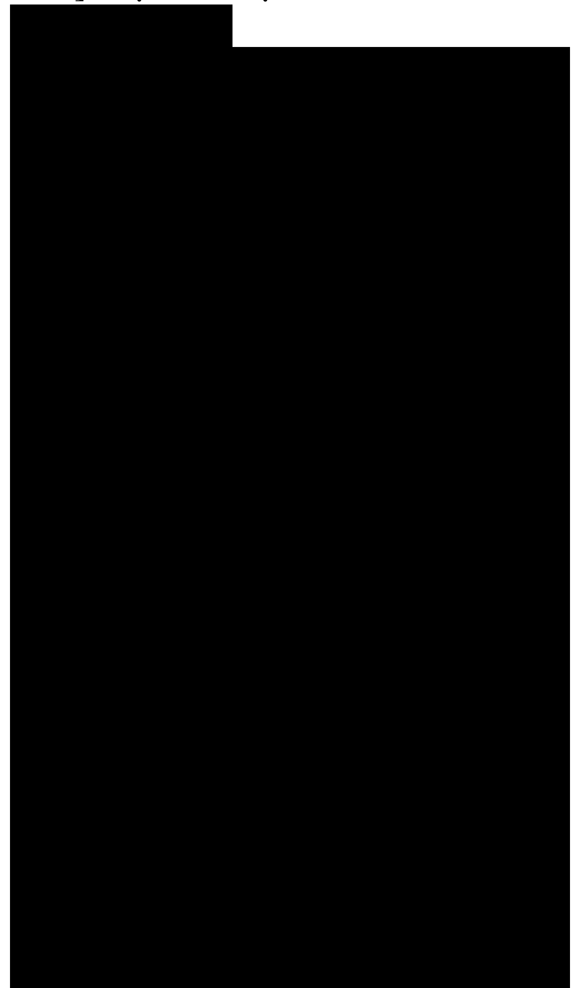
https://drive.google.com/folderview?id=0B4rAPqlxIMRDNkFJeUpfVUtLbk0&usp=sharing

Liên hệ dịch tài liệu :

thanhlam1910_2006@yahoo.com hoặc frbwrthes@gmail.com hoặc số 0168 8557 403 (gặp Lâm)

Tìm hiểu về dịch vụ: http://www.mientayvn.com/dich_tieng_anh_chuyen_nghanh.html

Bidirectional Mappings for Data and Update Exchange 10 h 40 ABSTRACT A key challenge in supporting information interchange is not only supporting queries over integrated data, but also updates. Previous work on update exchange has enabled update propagation over schema mappings in a unidirectional way — conceptually similar to view	Ánh xạ hai chiều (hai hướng) để trao đổi dữ liệu và cập nhật Tóm tắt Thách thức quan trọng trong hỗ trợ trao đổi thông tin không chỉ là hỗ trợ các truy vấn trên các dữ liệu tích hợp, mà còn trên các cập nhật. Nghiên cứu trước đây về trao đổi cập nhật giúp chúng ta có thể truyền cập nhật trên ánh xạ lược đồ theo một hướng-tương tự về mặt khái niệm

maintenance, in that a derived instance gets updated based on changes to a source instance. In this paper, we consider how to support data and update propagation across bidirectional mappings that enable different sites to mirror each other's data. We show how data and update exchange can be extended to support bidirectional updates, implement an algorithm to perform side-effect-free update propagation in this model, and show preliminary results suggesting our approach is feasible.	với bài toán view maintenance, trong đó thể hiện suy diễn (dẫn xuất) được cập nhật dựa trên những thay đổi của thể hiện nguồn. Trong bài báo này, chúng tôi đề cập đến cách truyền dữ liệu và cập nhật nhờ vào các ánh xạ hai chiều giúp các vị trí khác nhau nhân bản dữ liệu của nhau. Chúng tôi trình bày cách mở rộng trao đổi dữ liệu và cập nhật để hỗ trợ các cập nhật hai chiều, thực thi một thuật toán để thực hiện truyền cập nhật không hiệu ứng phụ (hiệu ứng lề) trong mô hình này, và trình bày các kết quả sơ bộ chứng tỏ cách tiếp cận của chúng tôi hoàn toàn khả thi.
Categories and Subject Descriptors	Các loại và các bộ chỉ định Subject
H. 2.5 [Heterogeneous Databases]: Data Translation	H.2.5 [Cơ sở dữ liệu không đồng nhất]: Dịch dữ liệu
I. INTRODUCTION Data integration remains one of the most important problems in today's information-rich world — not only in its traditional enterprise focus, but also among scientists, medical institutions, and researchers. Most research on integration has focused on querying integrated data: either by bringing data into one common schema, such as data warehousing, virtual data integration, or data exchange [9]; or supporting multiple interrelated schemas, such as peer data management systems [1, 4, 14, 19]. Such efforts typically assume data to be stable, clean, and correct; hence the focus is on integrating the data to support querying. However, in many large-scale data sharing efforts, particularly in the sciences, data is neither stable nor clean. It is being continuously annotated, corrected, and hand-cleaned by each user — and a	

major task is not simply to integrate data for querying, but rather to propagate updates across interrelated, independently modified databases. However, surprisingly little work has addressed updates in the data integration literature.

Figure 1: Collaborative data sharing system with bidirectional mappings among 3 peers: PPgus , and PBioSQL •

To address this unmet need, we have been developing an architecture and system to support update propagation across networks of data sources interrelated by schema mappings. The ORCHESTRA collaborative data sharing system [16, 21] (CDSS) builds upon the formalisms of peer data management systems and data exchange in order to provide update exchange [10]: the capability to map updates made over one database instance to other, interrelated instances. Each site sharing data within the CDSS is logically an autonomous peer; it has its own database instance, set of directed schema mappings specifying how to map data from other peers into this instance, and set of schema mappings specifying how to map data out to "downstream" peers' instances.

The existing CDSS model employs mappings from source to target peers,

similar to those used in data integration and exchange. An update made to a peer's instance is applied to the peer's local database instance. Upon request, the CDSS propagates this update to all downstream instances using update exchange. This matches situations where one database is more authoritative than another: updates from a curated database like SWISS-PROT should propagate to individual biologists' databases, but not the converse.

However, in some cases two peers, even with different schemas, want to mirror data: either peer may update data from itself or its neighbor, and the effects should propagate to the other peer. We are aware of no existing solution to this problem in a setting with schema mappings. In this paper, we consider the problems of specifying bidirectional mappings between instances, and propagating updates along these mappings. We briefly illustrate with an example.

Example 1. Figure 1 shows an example bioinformatics CDSS based on a real application. GUS contains gene expression, protein, andtaxon (organism,) information; BioSQL contains very similar concepts; and uBio establishes synonyms and canonical names for taxa. PuBio and PGUS want to propagate updates to each other via mapping mi, and so do PuBio and PBioSQL via m2. Note that update propagation composes: an update to PBioSQL will result in a

update of PuBio, which in turn induces an update over PGUS.

Our problem generalizes two separately studied topics in the traditional relational database realm: a materialized view may be simultaneously maintainable (i.e., updates made to the base instance are propagated to the view instance) and updatable (i.e., updates made to the materialized view are propagated to the base relations). However, a view is a function between source instance and materialized view instance; whereas a schema mapping represents a containment constraint among instances.

Moreover, we consider settings with multiple mappings and peers, some of which can inter-act with one another or share target relations, whereas the view update literature typically focuses on a single view, whose definition is a single (typically conjunctive) query. Another important difference is that in a view definition, one side only contains base tuples, while the other only consists of data derived from those base tuples. In a CDSS with bidirectional mappings between peers, each peer typically contributes its own data as well as imports data from the other peers through mappings. These differences have significant consequences, which we consider in this paper. In particular, we make the following contributions:

- A language and semantics for

bidirectional schema mappings in data and update exchange, useful for propagating both data and updates symmetrically among sets of database instances. We show how update policies can be expressed along with these mappings.

- Techniques by which updates can be made at any instance and propagated to other instances when this does not cause side effects (modifications not made by the user).

- Demonstration that incremental computation performs ac-ceptably both with and without side effect testing.

Roadmap. Section 2 reviews related work. Section 3 presents extensions to data exchange for bidirectional mappings. We expand to update exchange in Section 4, developing techniques for update propagation and update policy specification independently of whether they produce side effects. Section 5 develops strategies to test for and avoid side effects, given actual data instances; this provides greater functionality than instance-independent view update policies. We experimentally demonstrate the feasibility of our approach in Section 6, and conclude and discuss our plans for future work in Section 7.

2. RELATED WORK

Update propagation has typically been considered in the context of relational views. Incremental view maintenance [12] is the task of updating a derived view, given a set of insertions,

deletions, and possibly replacements of one or more base relation tuples. The resulting view instance must be identical to the one that would be computed by directly recomputing the view after updating the base data. Conversely, the view update problem, in which tuples in the base instance are to be changed in order to accomplish updates over the view, is more subtle because each source tuple may produce several tuples in the view. A given view update may thus introduce side effects: in order to modify one tuple in the view, we must modify a tuple in the base, which in turn causes other tuples in the same view to be inadvertently changed (a "side effect"). Dayal and Bernstein identified constraints under which an update does not introduce side effects within the same view [8]; other work has explored a variety of other, generally stricter, restrictions over what data is allowed to be affected [3, 8, 18]. Recent work [5] has considered restricted view definition languages in which view update is side effect-free. Generally the view update literature considers only a single view, which is typically a conjunctive query.

Little work has been done in the context of updates in data integration scenarios — where schema mappings are typically containment constraints between queries over different instances, expressed as tuple-generating dependencies (tgds) [9] or, equivalently, global-local-as-view (GLAV) rules [20]. Recent work on update exchange [10] showed how the

data exchange setting [9] could be generalized to support update propagation among multiple peers with their own data instances and local contributions: updates (including deletions) would be applied to the originating peers' database instance, and their effects would be propagated to all other peers who map data from that instance (but not back to the peers from which this peer imports data, and where the updated data may have originated). The outcome is roughly analogous to that of view maintenance: data derived from the mappings gets updated in response to modifications made at a source peer. For bidirectional mappings, deletions of data at a different peer from the one where they were introduced need to be propagated back to their source peer(s). This is analogous to the view update problem: propagating changes made over a “target” instance back to a source instance, thus removing the original source tuple(s).

3. BIDIRECTIONAL DATA EXCHANGE

The foundations of the CDSS architecture, and its basic capability of update exchange, generalize the semantics of data exchange [9]. Hence we briefly review key results from data exchange and extend these to include support for bidirectional mappings, before considering update exchange in the next section. A data exchange setting involves:

- A source schema S and target schema T
- An instance I of S
- A set of source-to-target tuple

generating dependencies Est, i.e., mappings of the form:
where 0 is a conjunction of atoms over S and ^ is a conjunction of atoms over T.

- A set of target tuple generating dependencies (i.e., where both sides of the dependencies are conjunctions of atoms over T).

The goal of data exchange is to compute an instance for every target relation, such that a conjunctive query (or union of conjunctive queries) over the target will provide all certain answers [20] in accordance with the source tuples and the constraints imposed by the schema mappings. This is a property of all universal solutions: instances J' of T, such that (I, J'} = U and for every other instance J such that (I, J} |= £st U £t, there is a homomorphism h : J' ^ J. We compute and maintain the canonical universal solution of [15], which can be computed as a result of a datalog program, as explained in [10] and briefly sketched below. In the rest of the paper, we use (I, J) to denote that J is the canonical universal solution for I.

The canonical universal solution (according to [15]) of the corresponding data exchange setting is:

3.1 Bidirectional Mappings

We now extend the data exchange setting to support multiple peers, each with its own data instance, and bidirectional schema mappings. Our setting looks like:

- Peer schemas Pi,... , Pn.

• Instances Ii,... , In of Pi,... , Pn, respectively.
• A set of mappings M among the peer relations of Pi,... ,Pn specified as logical expressions of the form:

where the formula in each side of the mappings is a conjunction of atoms over one of the schemas (e.g., 0 is a conjunction of atoms over Pi and ^ is a conjunction of atoms over P2).
Every bidirectional mapping m of the form shown above is logically equivalent to a pair of tgds:
Example 2. The mappings for Figure 1 are:
These mappings are equivalent to the following tgds:
For readability, in the rest of the paper we will omit the universal quantifiers for variables that appear in the left-hand side (LHS) of mappings.

3.2 Bidirectional Data Exchange Semantics
Any set of bidirectional mappings can be converted to a standard data exchange setting (S, T, £st, £t) as follows: Let Pi,... ,P% be the schemas obtained by replacing each relation R of Pi,... , Pn, respectively, by Re (the local contribution relations). In the data exchange setting, let:
• Source schema S = P(U • • • U Pi,
• Target schema T = Pi U • • • U Pn
• Source instance I = Ii U • • • U In
• Source-target mappings = (R£(x R) ^ R(xR) | R e Pi U---U Pn}

• Target mappings = M (i.e., the set of tgds that the bidirectional

mappings are equivalent to)

We define the canonical universal solution for our bidirectional data exchange setting to be the one for this translated data exchange setting.

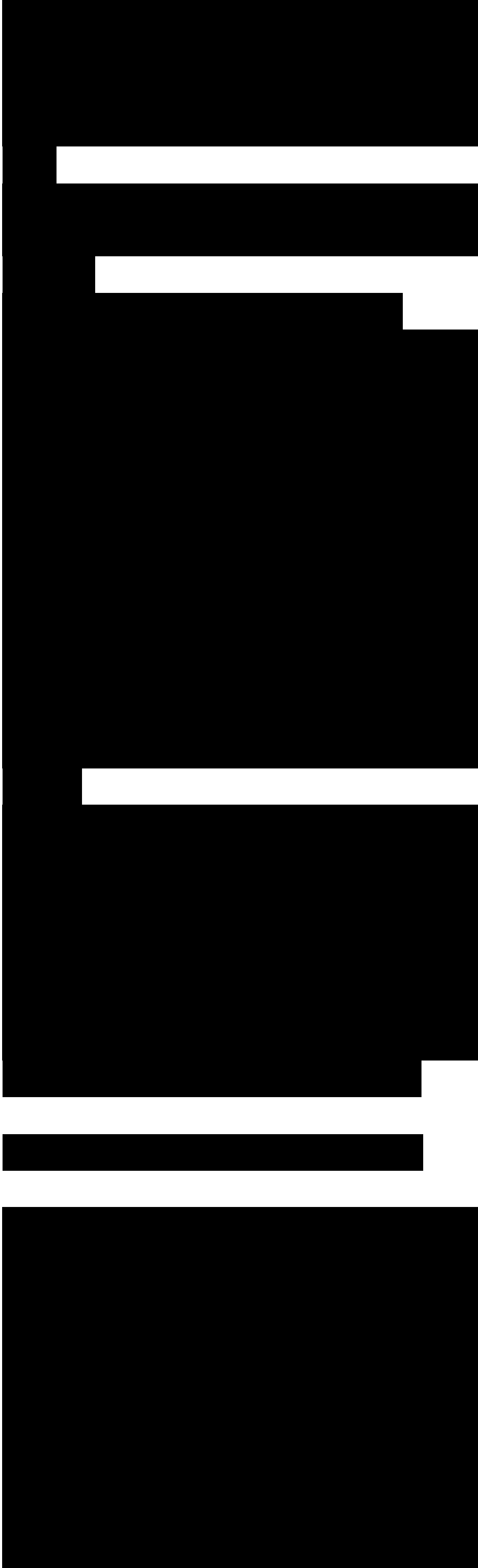
Example 3. For the mappings in Example 2, assume local contribution relations:

(i.e., base data) as appropriate, and then achieve the update over a recomputed version of the canonical universal solution. In essence, this is a version of the view update problem, over the datalog program for generating the canonical universal solution. However, in contrast to a view setting, here tuples may be introduced locally by any peer, and deletion of a tuple must remove data from every peer from which that tuple can be derived.

We first consider insertions and deletions that affect only the local contributions tables at the same peer, before considering how to propagate deletions to local contribution relations at other peers. (Insertions will always be made locally, in accordance with the existing CDSS model.)

4.1 Insertions and Deletions at the Same Peer

For insertions, we start with a previous instance of the CDSS, which is a solution (I, J), and we take a set of insertions A+ that we apply directly over the local contribution relations at the peers that originated the updates. Then we compute a new canonical universal solution (I+A+, J+Y +). We can directly recompute the instance using the datalog program of the



previous section, adding new tuples to the peer relations until the mappings are satisfied. Even better, since bidirectional mappings are equivalent to a pair of unidirectional mappings, we can derive an incremental maintenance program using the delta rules [13] extension that was presented in [10], and perform the recomputation more efficiently.

If a tuple is deleted from relation R at the peer where it originated, we can simply remove the tuple from the local contribution relation R£, and then propagate the effects of the deletion "forward" in incremental fashion, quite similar to the program described for insertions, but with a caveat. As in decremental view maintenance [13], there are subtleties in determining whether to remove a derived tuple, since that tuple could be derived in an alternative way. Two general schemes exist for performing decremental maintenance (when recursion is present, as with our data exchange program of the previous section). The first is the DRed (Delete and Rederive) algorithm of [13], which removes derived tuples, then tries to see if there is an alternate derivation. A more efficient alternative, presented in [10], makes use of data provenance [6, 7, 11], encoded as edge relations in a graph describing which tuples are directly derived from one another, to determine when a tuple is no longer derivable from local contributions.

4.2 Deleting from a Different Peer

9 h 15

When a tuple is deleted from a peer

other than its origin, we must propagate the effects to the local contribution relation(s) of the tuple's originating peers, in a manner analogous to view update. More precisely, we want to derive a set of updates over local contribution relations that perform the update requested on the target peer:

Definition 4.1 (Performs). Let (I,J) be the canonical universal solution and Y- be a set of tuples of J (i.e., peer relations) to be deleted. Let A- be a set of deletions over I (i.e., local contribution relations) and let (I — A-, J') be the canonical universal solution. We say that A- performs Y- iff J' fl Y- = 0.

This generalizes a definition by Dayal and Bernstein [8] to canonical universal solutions in data exchange. As with view update, there may be multiple ways to perform a target deletion. For example, if the LHS of the mapping involves a join, the desired effect may be achieved by deleting tuples from either (or both) of the relations in the join. We now discuss how an administrator may specify policies for performing the updates. We assume that an administrator may wish to manage, or even override, default behaviors. In the next section we consider side effects and how to ensure updates do not produce them. However, we note that in certain settings with many interacting mappings, the administrator may be willing to allow side effects.

4.2.1 Update Policies

We specify update policies as annotations on mappings: if an atom for relation R on one side of a mapping is annotated with *, this means that if a tuple in the opposite side of the mapping is deleted, then any tuples from R, as well as its corresponding local contribution relation Rl, should be deleted. An annotated version of m2 from Example 2 is:

(m2) 3i B(i, n) A *S(i, c)^*U(n, c)

If a tuple is deleted from U, we delete any tuples of S from which it can be derived. Similarly, deleting B and/or S tuples results in a deletion of U tuples, thanks to the update policy in the opposite direction. In some cases, the composition of update policies may cause cascading deletions: e.g., deleting from U as above may trigger further deletions from S. We can show [17] that any update policy of a bidirectional mapping for which there is at least one atom in each side that is annotated with *, is guaranteed to perform any given set of updates.

We generate delta rules for deletion propagation only for the marked relations and their corresponding local contribution relations; the set of such rules for all mappings form the update policy program. The rules for the m2 update policy shown above would be:

Rules 1-3 (and the delta tables U-,B-,S- involved in them) are used to propagate deletions “backwards” along bidirectional mappings, specifying deletions over peer

relations U, B, S, respectively. Rules 4-6 “collect” in the delta tables U£-, B£-, S£- the actual local contribution tuples to delete from Ue,Be,S£, if such tuples exist.

4.2.2 Interactions among Mappings

With bidirectional mappings, a deletion over a peer relation may propagate to deletions over multiple local contribution relations, from both sides of the bidirectional mapping. Moreover, in certain cases tuples can be transitively derived by going back and forth through the two directions of the bidirectional mapping more than once. For instance, in Example 3, B(x3,c) and S(x3,a) were produced by ap-plying mjT to U(c, a), which in turn was derived by applying m^ to B(3, c), S(3, a). The situation gets even more complex when there are multiple bidirectional mappings with relations in common: their update policies can interact. In general, computing the set of local deletions (A-) necessary to perform the deletions in Y- requires us to compute the fixpoint of the update policy program. The computation of the local updates using this update policy program also deletes tuples from peer relations derived “on the path” from the user deletions to the base data in local contribution relations. The update policy program helps us compute two sets of updates, given a set of user updates Y- : A- over local contribution relations and another set Y'- 3 Y- over peer relations. We can compute these sets using the following algorithm:

Algorithm PropagatePeerDeletions

1. Run the update policy program (Sect. 4.2.1) on Y- to compute R£- for each local contribution relation Re

2. For each local contribution relation R£, remove tuples in R£- from R£
3. Run the decremental maintenance program (Sect. 4.1) on the local deletions R£- computed in the previous step. For each peer relation P, this computes a set of deletions P-; the set of all P- is Y'- above
4. For each peer relation P, remove tuples in P- from P
5. ~~AVOIDING SIDE EFFECTS~~

The term side effect was invented in the view update literature to refer to a propagation of an update to a source, which in turn causes other, undesired effects when the contents of a view are recomputed (e.g., because multiple view tuples were derived from the same source tuple). In other words, we propagate an update backwards via a policy, and then its forward effects (via maintenance or recomputation) change tuples that were not part of the original modification. (We do not consider cascading deletions caused by multiple update policies to be side effects.)

Definition 5.1 (Side effects). Let (I, J) be the canonical universal solution, where I is an instance of local contribution relations and J is an instance of peer relations. Let Y- be a

set of updates over J, and A-, Y'- be the output of the update policy program on Y-. Let (I — A-, J') be the canonical universal solution, then the translation that produced A- is side-effect-free iff J' = J — Y'- , while it has side effects iff J' C J — Y'-.

An administrator may wish to propagate updates only if they avoid side effects on a given instance. Previous work typically considers static checking, based on functional dependencies and other constraints, on whether a view can be updated without introducing side effects. We believe such checking is inappropriate for large-scale data sharing: in databases produced by non-expert users, constraints are often under-specified, making static checking overly pessimistic and checking statically may prevent any update to a view, even when some tuples may be updatable without causing side effects. Thus we allow the administrator to request detection and elimination of side effects at update-time, based on the actual contents of the database instances.

The following algorithm identifies which of the local deletions returned by the update policy cause side effects, and only applies to local contribution relations those that do not, before computing the new canonical solution.

Algorithm ~~PropagatePeerDeletionsWithoutSideEffects~~

1. Run the update policy program

on Y- to compute R£- for each local contribution relation Re and P- for each peer relation P (but do not modify R, P)

2. Run the decremental maintenance program on the local deletions R£-, to get sets of peer deletions Pd for every peer relation P (do not apply updates to the peer relations)

3. For each peer relation P, set Pse := Pd — P- and P- := 0. These are the side effects on P

4. For each tuple t £ Pse, compute the set of all tuples in local contribution relations involved in some derivation of t. An algorithm for this was sketched in [10], as part of decremental maintenance. The main idea is to traverse mappings backwards, starting from each side effecting tuple. For each local contribution relation R£, collect all such sources of side effects in a relation Rfnv

5. For each local contribution relation R£, set R£- := R£- — Rfnv. These are the side effect-free source updates

6. For each local contribution relation R£, remove tuples in R£- from R£

7. Run the decremental maintenance program on the local deletions R£- computed in the previous step. For each peer relation P, this computes deletions P-

8. For each peer relation P, remove tuples in P- from P

The algorithm applies deletions of local tuples identified by the update

policy program, when these do not cause side effects (tested in Line 3); it can additionally be relaxed to consider cases where some peers tolerate side effects and others do not. Importantly, each of the steps of the algorithm above (as well as the one in the previous section) can be expressed as a datalog-like program, which can be translated to SQL queries that can be evaluated over an RDBMS.

~~6. EXPERIMENTAL EVALUATION~~

We now investigate the performance of bidirectional update exchange in a CDSS. First, we compare bidirectional and unidirectional update exchange properties, for the same number of peers. Then we compare preliminary implementations of our deletion propagation algorithms, with and without detection of side effects.

We implemented the bidirectional mapping algorithms of the previous sections in the ORCHESTRA system, which is a Java 6 (JDK 1.6.02) layer that runs over a relational DBMS. We used IBM DB2 UDB 9.1 on Windows 2003 as our database engine, running on a dual Xeon 5150 server with 8GB of RAM, and allocated 2GB to DB2 and 768MB for JVM heap space.

We used the synthetic workload generator of [10], which creates different configurations of peer schemas, mappings, and updates. The workload generator takes as input a single universal relation based on the

SWISS-PROT protein database [2], which has 25 attributes. It then creates peers with different partitions of the attributes from SWISS-PROT's schema. Next, mappings are created among the relations via their shared attributes. Finally, we generate fresh insertions by sampling from the SWISS-PROT database and generating a new key by which the partitions may be rejoined. We generate deletions by sampling among our insertions.

For all experiments, each peer is initialized with 2,000 tuples — different for each peer — in its local contributions

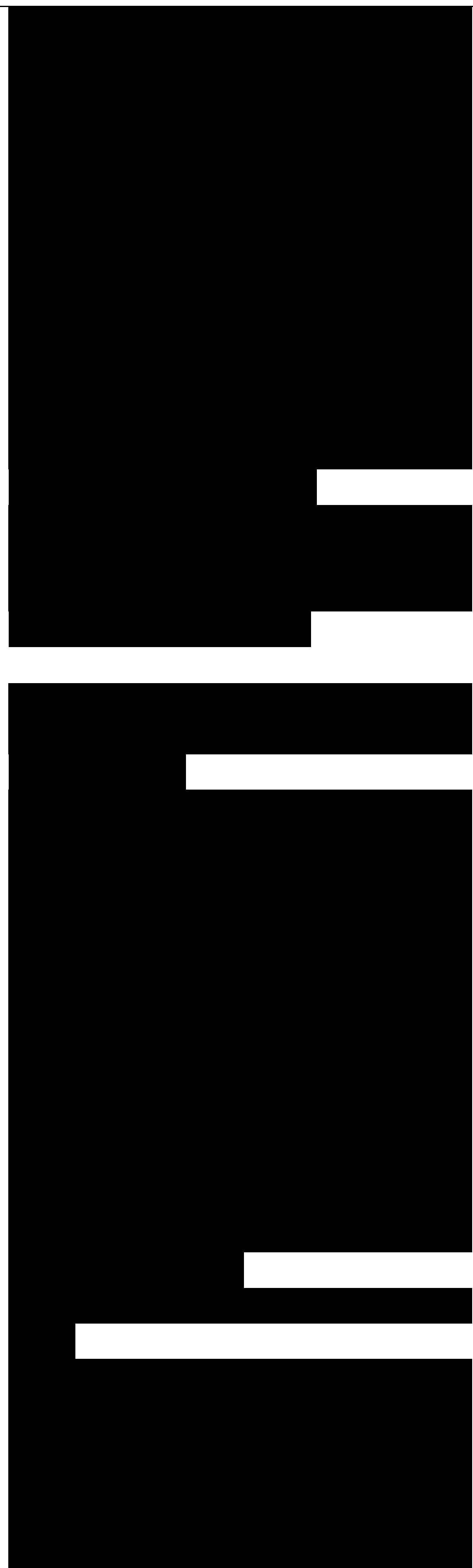


Figure 2: (a) Solution size and computation time, and (b) deletion propagation time

tables. We randomly generate mappings among the peers; for a CDSS of 2 peers there is 1 mapping; for 5 peers, 4 of the peers are connected in a "square" and the 5th peer is mapped to one "corner"; for 10 peers, there is a "grid" of 9 peers, with one additional peer connected to a single neighbor, and one extra mapping that forms a diagonal in the grid. We used "full" mappings, i.e., mappings with no existential variables in either side.

6.1 Unidirectional vs. Bidirectional Mappings

We first consider the effects of unidirectional mappings vs. bidirectional ones: in general, bidirectional mappings should result in larger data instances (since all data will propagate to all peers) and longer

computation times. We see in Figure 2(a) the size of the canonical universal solutions, measured in number of tuples (scale on the left y-axis) and the total running time (scale on the right y-axis). As we scale the CDSS to increasingly larger sizes, we see that for unidirectional mappings, the total instance sizes and running times grow at an approximately linear rate; whereas for bidirectional mappings, the number of tuples and the computation time grow quadratically. This mirrors our expectations, given the topologies and the amount of data exchanged. We note that running times of 200 seconds are tolerable for offline batch operations, which are the emphasis in the CDSS. However, we also observe that these running times suggest opportunities for optimization and indexing.

6.2 Deletion Policies

We separately study deletion, for side effect-free as well as side effecting propagation. We consider total running time, as well as backwards (by update policy) and forwards (by incremental maintenance) computation. For this experiment we start by deleting 10% of the SWISS-PROT entries at every peer (i.e., 200 entries per peer). For 2 and 5 peers, these costs are quite acceptable, especially compared to recomputation, which we can estimate from the previous figure. We observe that backwards propagation is the major factor in side effect-free updates, whereas forwards propagation represents almost the entire cost in the side effecting mode. At 10 peers, the

amount of data and the mapping complexity results in a very expensive operation. Again, we believe this suggests opportunities for future research on optimization.

7. CONCLUSIONS AND FUTURE WORK

We presented a framework for exchange of data and updates between peers connected through bidirectional mappings. Such mappings are important for CDSS settings where peers want to mirror each other's data up to translation between different schemas. To this end, we showed how to extend techniques from view maintenance and view update to compute instances of such peers incrementally, by propagating updates along such mappings. The algorithms presented here are either guaranteed to perform the required updates — ignoring possible side effects — or make an effort to perform them while ensuring that no side effects exist. While our framework in this paper only supports bidirectional mappings, we plan to combine them with related techniques from [10], to develop a framework that supports both unidirectional and bidirectional mappings. We also plan to investigate indexing and optimization techniques to improve performance.