Polymorphic Queries for P2P Systems

Jie Liu [1] Wenfei Fan [2,3] Jianzhong Li [4] Shuai Ma [2]

[1]*Institute of Computing Technology* lj@kg.ict.ac.cn
[2]*University of Edinburgh* {wenfei,sma1}@inf.ed.ac.uk
[3]*Bell Laboratories*
[4]*Harbin Institute of Technology* lijzh@hit.edu.cn

***Abstract*— When a query is posed on a centralized database, if it refers to attributes that are not defined in the schema, the user is warranted to get either an error or an empty set. In contrast, in a P2P system, one often wants the query to find relevant information from other peers across the system, even if those attributes *cannot* be found in the local database residing in the peer on which the query is posed. Indeed, it is for data sharing that P2P systems are developed. This paper investigates such queries for P2P systems. The novelty of the work consists in the following. (a) A revision of relational queries, referred to as polymorphic queries, which support type variables to accommodate relevant attributes not defined in a local schema. (b) A notion of information closures for the semantics of polymorphic queries, to correlate information about the same object from various peers. (c) A notion of contextual foreign keys, an extension of foreign keys by adding contextual information, to identify objects across peers and to compute information closures. (d) A conceptual evaluation strategy for polymorphic queries with PTIME complexity, and optimization techniques for query processing. (e) Techniques for resolving conflicts of the same attribute collected from different peers, an issue not encountered by previous P2P query models. Our experimental study verifies that polymorphic queries are capable of finding more sensible information than traditional relational queries, while they can still be evaluated efficiently.**

I. Introduction

Consider a relational database D specified by schema S. When a query Q is posed on D, if Q refers to attributes not defined in S, one will get either a syntactic error or an empty set. This is also the semantics adopted by current query models for P2P systems [1], [2], [3], [4], [5], [6], [7], [8]: when a query Q is posed on a database D residing in a local peer P, Q is not allowed to refer to an attribute A if A is not defined in the schema of D. However, while information about attribute A cannot be found in D, it may be available in databases residing in other peers in the system. It is reasonable for the user to expect Q to find the information and include it in the query answer: after all it is to share data that P2P systems are developed. This is illustrated by the example below.

Example 1.1: Alice is interested in John Denver's albums that received a rating of 4. For those albums she wants to query a P2P system and find information about price, label and release. She has only access to peer P_1. To this end Alice poses an SQL query Q_0 on the database residing in P_1.

select album, price, label, release
from review
where artist = "Denver, J" **and** rating = "4"

The database at P_1 is specified by schema: review (album, artist, rating). The review relation collects a number of albums by various artists, and with each album it associates an average rating in the scale $[0, 4]$ given by customers. An instance of the review relation is shown in Fig. 1 (a). Observe that query Q_0 refers to attributes price, label and release, which are not defined in the local schema review. When the query is issued to any current P2P systems, Alice will get either an error or an empty set. This is precisely what one would expect if Q_0 were posed on a centralized database system.

	album	artist	rating
t_1:	Almost Heaven	Denver, J	3
t_2:	Take Me Home	Denver, J	4
t_3:	Greatest Hits	Denver, J	4

(a) An instance of review at peer P_1

	album	artist	price	label	rating
t_4:	Almost Heaven	Denver, J	7.99	DancingBull	good
t_5:	Take Me Home	Denver, J	5.97	Windstar	high
t_6:	Greatest Hits	Denver, J	8.36	BMG	high
t_7:	Diana	Anka, P	5.98	Magic	fair

(b) An instance of sale at P_2

	title	artist	label	release	rank
t_8:	Almost Heaven	Denver, J	DancingBull	12/20/2005	2,175
t_9:	Take Me Home	Denver, J	Windstar	03/07/2006	1,654

(c) An instance of CD at P_3

album	price	label	release
Take Me Home	5.97	Windstar	03/07/2006
Greatest Hits	8.36	BMG	null

(d) Answer to query Q_0 posed on peer P_1

Fig. 1. Example data and answer to query Q_0

However, while the price, label and release information is not provided by peer P_1, it may be available at other peers in the P2P system. For example, P_1 may have a neighboring peer P_2 with a database of schema sale (album, artist, price, label,rating), which in turn has a neighbor P_3 with a database of schema CD (title, artist, label, release, rank). Instances of sale and CD are shown in Fig. 1 (b) and (c), respectively.

Provided these, a P2P system should be able to find the price, label and release information and include it in the answer to Q_0, rather than deny the execution of Q_0. For example, for the album "Greatest Hits" found at P_1, we want to find its price and label from P_2, and its release from P_3. Putting these together, the answer to the query is composed, as shown in Fig. 1 (d). The schema of the answer to Q_0 consists of attributes album, price, label and release. □

This highlights the need for a new P2P query model that allows users to specify, explicitly or implicitly, attributes that do not appear in a local schema. In the example above, Alice specifies in query Q_0 certain attributes that are not

defined in the local schema review. Furthermore, she may be interested in finding other information about John Denver's albums, *e.g.,* rank, although she cannot explicitly name those attributes. The query model should support such queries. In addition, the model should be able to evaluate queries based on *object expansion*, *i.e.,* to augment objects (represented as tuples) found at one peer by incorporating additional attributes about the *same objects* found from other peers in the system.

This query model, however, introduces several challenges.

First, this calls for a revision of relational queries. Recall that in a centralized database system, when a query Q is posed on a database D, the schema of the answer to Q in D, referred to as the output schema of Q, is uniquely determined by Q and D. This is no longer the case for the P2P query model. Indeed, a language for P2P systems should be able to express queries with attributes not defined in a local schema, and to cope with the evolving and dynamic nature of P2P systems. That is, the output schema of a query is "open-ended" and should be *incremented* when the query is evaluated by traversing the linked peers. It is no longer possible to statically determine the output schema based on the query and the local schema alone. For example, one cannot determine the types of price, release and label based on Q_0 and the schema review at compile time.

Second, to support object expansion, it is necessary to correlate attributes and identify tuples in different databases that refer to the same object. In the example above, attributes album at P_1 and title at P_3 both refer to album although they bear different labels; and tuples t_2, t_5 and t_9 at different peers refer to the same album by John Denver. To correlate information from different peers, we need a language to specify correspondences between attributes and tuples.

Third, the query model should not incur significant degradation in performance for query processing in P2P systems. That calls for an efficient query evaluation strategy.

Fourth, to merge and correlate information about the same object from various peers, we often have to resolve conflicts. Indeed, the same attributes may be found from different peers. For example, both review and sale have a rating attribute. Although these attributes bear the same label, they may differ in domains or data values. Effective methods should be employed to resolve conflicts in both types and data values.

Prior work. Several models have been put forward for P2P queries, based on, *e.g.,* schema mapping and certain query answering [1], [2], [3], keyword search [4], mapping (concordance) tables [5], [6], and query expansion [7], [8] (see [9] for a comprehensive survey). To the best of our knowledge, no previous P2P query systems support queries that refer to attributes not defined in a local schema, such as the query Q_0 given in Example 1.1. For relational queries beyond keyword search, query evaluation is often prohibitively expansive, *e.g.,* it is undecidable for computing certain query answers [1] and it incurs a non-elementary complexity in the mapping-table model [5], [6]. Furthermore, issues such as conflict resolution are not encountered in those query models.

On the other hand, a notion of polymorphic records has proved extremely useful in functional programming (see, *e.g.,* [10]). A polymorphic record, *a.k.a.* extensible record, carries *unknown* fields with type variables, in addition to a set of known fields with fixed domains. It allows one to build an object *incrementally* by adding a finite number of fields and instantiating their type variables accordingly.

Contributions. To explore the data sharing nature of P2P systems, we propose a query model for P2P systems based on object expansion. We introduce a revision of conjunctive (SPC) queries, define their semantics based on object expansion, and provide a PTIME evaluation strategy and optimization techniques in unstructured, schema-heterogeneous P2P systems.

(a) Along the same lines as polymorphic records, we propose a class of queries for P2P systems, referred to as *polymorphic queries*. Polymorphic queries extend SPC queries by incorporating *type variables*, and may specify, explicitly or implicitly, attributes that are not defined in a local schema, *i.e.,* the database schema of the local peer on which the query is posed. For example, the query Q_0 given earlier can be expressed as a polymorphic query. In contrast to SPC queries for which the output schema is determined by the queries and the local schema, the output schema of a polymorphic query is computed by instantiating type variables during query evaluation, when traversing various peers in a P2P system.

(b) To support object expansion, we introduce a notion of *information closures* (ICs). The information closure of a tuple t is another tuple t^* that contains t and moreover, all relevant attributes of the object represented by t found across different peers. The computation of ICs is based on a notion of *contextual foreign keys* (CFKs), which specify correspondences between attributes and between values across different peers, and allow us to identify objects at those peers. Contextual foreign keys are an extension of foreign keys that reference primary keys, by incorporating patterns of semantically related data values. They are capable of expressing lexical semantic relations such as WorldNet. In addition, one can efficiently reason about CFKs. Indeed, while the implication problem for keys and foreign keys is undecidable, we show that both the satisfiability problem and the implication problem for primary keys and CFKs taken together are decidable in quadratic time.

(c) Based on ICs and CFKs, we develop algorithms for evaluating polymorphic queries in unstructured, schema-heterogeneous P2P systems. These algorithms yield a conceptual evaluation strategy for polymorphic queries. In contrast to the high complexity of previous P2P query models, the data complexity of polymorphic query evaluation is in PTIME in the P2P setting. In addition, we provide optimization techniques to improve the conceptual evaluation strategy.

(d) We present methods to resolve conflicts that emerge when correlating information collected from different peers. We explore various approaches based on, *e.g.,* OR-sets [11], [12] or aggregate resolution functions [13], [14], [15].

(e) We experimentally verify that polymorphic queries and their evaluation techniques are capable of effectively finding

more sensible information than traditional queries, without incurring significant degradation in the performance of P2P query evaluation. Indeed, we find that polymorphic queries are able to find up to 4.5 times more relevant data than their traditional counterparts. In addition, polymorphic queries can be evaluated efficiently without incurring heavy network traffic. Furthermore, our optimization techniques are effective: they improve the performance by up to 2-3 folds.

Organization. Polymorphic queries are introduced in Section II, followed by contextual foreign keys and information closures in Section III. A conceptual evaluation strategy for polymorphic queries is provided in Section IV, along with optimization techniques. Conflict resolution is investigated in Section V. A preliminary experimental study is presented in Section VI, followed by related work in Section VII and topics for future work in Section VIII. All proofs are in [16].

II. Polymorphic Queries

In this section, we introduce polymorphic queries, an extension of SPC queries (*a.k.a.* conjunctive queries). We present its syntax below, and give its semantics in Section IV. To simplify the discussion we focus on SPC queries, and defer the study of more general polymorphic queries to future work.

SPC queries. Recall that an SPC query [17] is defined on a relational schema $\mathcal{R}$ in terms of the selection (σ), projection (π) and Cartesian product ($\times$) operator. It is of the form:

$$\pi_L(\sigma_F(E_c)), \text{ where } E_c = R_1 \times \ldots \times R_n,$$

where (a) for each $j \in [1, n]$, we assume w.l.o.g. that R_j is a relation atom in $\mathcal{R}$ such that the attributes in R_j and R_l are disjoint if $j \neq l$; (b) F is a conjunction of equality atoms such as $A = B$ and $A =$ 'a' for a constant a in the domain of A, and (c) in the projection $\pi_L(Q_c)$, L is a list of attributes *appearing* in $Q_c = \sigma_F(E_c)$.

Polymorphic queries. We define an extension of SPC, referred to as *polymorphic queries* and denoted by SPC*, by adopting a *polymorphic projection* operator Π with type variables:

$$\Pi_L(Q_c), \text{ where } L = (L_1; L_2; \alpha),$$

where (a) $Q_c = \sigma_F(E_c)$ is the same as above, (b) L_1 is a list of attributes appearing in Q_c, but in contrast, (c) L_2 is a list of attributes *not* appearing in Q_c, and α is an (optional) variable, which, if present, is to be instantiated with a list of attributes appearing in neither L_1 nor L_2. Intuitively,

(1) L_2 denotes attributes that the user *explicitly* wants to find from a P2P system, although they are not defined in the local schema; while the labels of these attributes are known, their *types* are *unknown* and are represented by type variables;

(2) α indicates that other attributes are also demanded, if any. Neither the *labels* nor the *types* of these attributes are known.

We refer to attributes in L_1 and L_2 as *local attributes* and *explicit attributes*, respectively, and to attributes that instantiate α as *implicit* attributes. Note that both L_2 and α carry type variables, along the same lines as polymorphic records [10].

We denote by $\mathsf{sch}(Q)$ the output schema of an SPC* query Q, which consists of attributes in L_1, L_2 as well as implicit attributes that instantiate α.

Example 2.1: The query Q_0 described in Example 1.1 can be expressed as an SPC* query $Q = \Pi_{(L_1;L_2)}(\sigma_F(E_c))$, where (a) F is a conjunction of equality atoms artist = "Denver, J" and rating = "4"; (b) E_c is the review relation schema at the local peer P_1; (c) L_1 = [album] and L_2 = [price, label, release], which are the attributes appearing in $\mathsf{sch}(Q_0)$.

Alice may want to retrieve other relevant attributes, via query Q_1 by adding α to Q_0: $\Pi_{(L_1;L_2;\alpha)}(\sigma_F(E_c))$. When Q_1 is evaluated in the P2P system described in Example 1.1, α will be instantiated with attributes found in the system, including but not limited to rank from P_3. The output schema $\mathsf{sch}(Q_1)$ consists of all these attributes and those in L_1 and L_2, with type variables instantiated with the corresponding domains.

It is straightforward to express an SPC* query in the SQL syntax. For example, the query Q_1 can be written as:

select* album; price, label, release; X
from review
where artist = "Denver, J" **and** rating = "4"

Here X indicates the variable α in the SPC* query. □

Note that SPC queries are a special case of SPC* queries, when L_2 is empty and α is absent.

When posed on the database at a local peer P_l, the query $\Pi_{(L_1,L_2,\alpha)}(Q_c)$ is evaluated as follows: a set of tuples (objects) is first extracted via a modified version of the query $\pi_{L'}(Q_c)$, where each tuple consists of attributes L_1 and possibly other attributes appearing in Q_c. These tuples are then expanded by adding relevant attributes extracted from other peers that are linked to P_l, directly or indirectly. More specifically, at these peers, each tuple found at P_l is expanded with L_2 attributes, and possibly with additional attributes not mentioned in L_1 and L_2, if any. The expansion proceeds until no more attributes can be added. The expanded tuples are collected by P_l and returned as the answer to the query. Here L_2 attributes may take null as their values, and α can be instantiated to an empty list, as shown in Fig. 1 (d). We shall present the semantics in details in the next two sections.

III. Object Expansion

We define the semantics of polymorphic queries in a P2P system based on object expansion. In a nutshell, given a set of tuples found by a query at the local peer, we expand each of these tuples by adding all relevant attributes extracted from other peers in the system. This is formalized in terms of the notion of information closures (ICs). To expand these tuples, we need to specify attribute and value correspondences across databases residing in different peers. These correspondences are expressed as contextual foreign keys (CFKs).

In this section we first introduce CFKs, and then present ICs. Based on these notions we shall give a conceptual evaluation strategy for SPC* queries in Section IV.

A. Contextual Foreign Keys

Recall that a foreign key (FK) fk from relation R_1 to R_2 can be expressed as $R_1[X] \subseteq R_2[Y]$, where X, Y are attribute lists in R_1, R_2, respectively, and Y is a key of R_2. Given instances (D_1, D_2) of (R_1, R_2), fk asserts that Y is a key of D_2 and furthermore, for any tuple t_1 of D_1, there is a tuple t_2 of D_2 such that $t_1[X] = t_2[Y]$, *i.e.,* $t_1[X]$ references the tuple t_2 identified by $t_2[Y]$ (see, *e.g.,* [17]).

Note that an FK specifies a pair of constraints: (a) Y is a key for R_2, and (b) an inclusion dependency from R_1 to R_2.

While FKs help us correlate object across different relations, they may not suffice in practice, as illustrated below.

Example 3.1: Recall relations review, sale and CD from Example 1.1. Suppose that (album, artist) is the primary key of review and sale, and (title, artist) is the primary key of CD. One might want to specify FKs from review to sale, and from sale to CD as follows:

$$\text{review(album, artist)} \subseteq \text{sale(album, artist)}$$
$$\text{sale(album, artist)} \subseteq \text{CD(title, artist)}$$

These FKs do not make sense if (a) the review relation contains information about all albums while sale contains only albums that received a rating above "poor"; and (b) the CD relation contains only albums from either Windstar or DancingBull. In light of this, the correspondences from review to sale and from sale to CD cannot be expressed as traditional FKs or even as more general inclusion dependencies. □

This motivates us to study a revision of FKs.

Contextual foreign key (CFKs). A CFK φ from relation R_1 to R_2 is of the form $(R_1[X] \subseteq R_2[Y],\ t_p[X_p, Y_p])$, where (a) X, X_p (resp. Y, Y_p) are two disjoint lists of attributes in R_1 (resp. R_2); (b) $R_1[X] \subseteq R_2[Y]$ is a traditional FK, where X (resp. Y) is a *primary key* of R_1 (resp. R_2), sorted in a fixed order on attributes; (c) t_p is the *pattern tuple* of φ with attributes in X_p and Y_p, such that for each attribute B in X_p (or Y_p), $t_p[B]$ is a constant in the domain of B. We refer to $t_p[X_p]$ (resp. $t_p[Y]$) as the R_1 (resp. R_2) *pattern* of φ.

An instance (D_1, D_2) of (R_1, R_2) *satisfies* φ, denoted by $(D_1, D_2) \models \varphi$, if X (resp. Y) is the primary key of D_1 (resp. D_2) and moreover, for *each* tuple t_1 in D_1, if $t_1[X_p] = t_p[X_p]$, then *there must exist* t_2 in D_2 such that $t_1[X] = t_2[Y]$ and furthermore, $t_2[Y_p] = t_p[Y_p]$.

Intuitively, the R_1 *pattern* of φ identifies a subset of D_1 that matches $t_p[X_p]$, and the traditional FK $R_1[X] \subseteq R_2[Y]$ is enforced on this subset rather than on the entire D_1. Further, for each tuple t_2 in D_2 that is referenced by $t_2[Y]$ (*i.e.,* $t_1[X]$), the R_2 *pattern* is enforced, *i.e.,* $t_2[Y_p] = t_p[Y_p]$. The latter allows us to express lexical semantic relations (*e.g.,* WorldNet)

Example 3.2: The constraints described in Example 3.1 can be expressed as CFKs as follows:

φ_1: (review (album, artist) $\subseteq$ sale(album, artist),
(review(rating) = "4", sale(rating) = "high"))
φ_2: (review(album, artist) $\subseteq$ sale(album, artist),
(review(rating) = "3", sale(rating) = "good")
φ_3: (sale(album, artist) $\subseteq$ CD(title, artist),
(sale(label) = "Windstar"))
φ_4: (sale(album, artist) $\subseteq$ CD(title, artist),
(sale(label) = "DancingBull"))

Here CFK φ_1 asserts that for each tuple t_1 in the review relation, if t_1[rating] = "4", then there must be a tuple t_2 in the sale relation such that t_1 and t_2 agree on their (album, artist) attributes and moreover, t_2[rating] = "high"; similarly for φ_2. These two CFKs ensure that review tuples can be mapped to sale tuples if and only if their ratings are above 3, and moreover, that a rating of "4" (resp. "3") in review corresponds to "high" (resp. "good") in sale. The CFKs φ_3 and φ_4 assure that sale tuples can find a match in the CD relation only for those albums from either Windstar or DancingBull. Note that no patterns are specified for CD in φ_3 and φ_4. □

Traditional FKs are a special case of CFKs with empty X_p and Y_p lists, when X and Y are the primary keys of R_1 and R_2, respectively. It should also be mentioned that CFKs are a variation of conditional inclusion dependencies (CINDs) studied in [18]. A CFK specifies three constraints: (a) X is the primary key of R_1, (b) Y is the primary key of R_2, and (c) an inclusion dependency from R_1 to R_2 with a pattern. In contrast, CINDs specify (c) alone without requiring (a), (b).

Reasoning about CFKs. When we introduce a class of constraints, it is important to get a balance between its expressive power and its complexity for static analyses. There are two central problems associated with any constraint language $\mathcal{C}$.

The *satisfiability problem* for $\mathcal{C}$ is to determine, given any set Σ of constraints in $\mathcal{C}$ defined on a schema $\mathcal{S}$, whether or not there exists a nonempty instance D of $\mathcal{S}$ such that D satisfies all constraints in Σ, denoted by $D \models \Sigma$. That is, we want to determine whether or not Σ makes sense.

The *implication problem* for $\mathcal{C}$ is to determine, given a set Σ of constraints in $\mathcal{C}$ and another constraint ϕ in $\mathcal{C}$, whether or not Σ *implies* ϕ, denoted by $\Sigma \models \phi$, *i.e.,* whether for any database D that satisfies Σ, D must also satisfy ϕ. It has long been recognized that implication analysis is critical to deriving schema mappings from schema matches (see, *e.g.,* [19]). Furthermore, as will be seen shortly, the computation of object expansion implicitly subsumes the implication analysis.

When $\mathcal{C}$ consists of traditional FKs (with which keys must be present by the definition of FKs), the implication problem is already undecidable [20]. In practice, it is typical that only one key, *i.e.,* the *primary key*, is defined on a relation. This motivated us to define CFKs in terms of primary keys. The result below shows that this restriction makes our lives easier.

Theorem 3.1: *The implication problem for* CFK*s is decidable in quadratic time.* □

One can specify any set of FKs without worrying about their satisfiability. In contrast, there exist sets of CFKs that are not satisfiable. For example, consider a pair of CFKs: $(R_1[A] \subseteq R_2[B], (R_2[C] = \text{"1"}))$ and $(R_1[A] \subseteq R_2[B], (R_2[C] = \text{"2"}))$. These CFKs are not satisfiable. Indeed, for any nonempty instances (D_1, D_2) of (R_1, R_2) and for any tuple t_1 in D_1, the CFKs require the existence of a unique tuple t_2 in D_2 such that $t_2[B] = t_1[A]$, but $t_2[C]$ = "1" *and* $t_2[C]$ = "2". This tells us that the increased expressive power of CFKs comes at a price.

The good news is that the price is not too high:

Theorem 3.2: *The satisfiability problem for* CFKs *is decidable in quadratic time.* □

In the sequel, we consider w.l.o.g. satisfiable CFKs only.

One may want to use CINDs or tuple-generating dependencies (TGDs, see, *e.g.,* [17]) instead of CFKs to specify correspondences across databases. However, for TGDs alone the implication problem is undecidable. When CINDs and keys are put together, both the implication problem and the satisfiability problem are undecidable.

B. Information Closure

Based on CFKs, we next present the notion of *information closures* (ICs), to characterize object expansion.

Consider a collection of databases $\mathcal{D} = (D_1, \ldots, D_n)$, where each D_j is specified by a schema S_j. Moreover, for each S_j, there are some S_i's such that there exists a set $\Sigma_{(j,i)}$ of CFKs from relations of S_j to relations of S_i. Assume that $\mathcal{D}$ satisfies these sets of CFKs. We also assume w.l.o.g. that no relations in S_i and S_j bear the same name if $i \neq j$.

Let t be a tuple consisting of a set of attributes in some relations of $S_1, \ldots, S_n$. The *information closure* (IC) of the tuple t over $\mathcal{D}$ is a tuple t^* computed as follows. The process maintains a set L of attributes to be included in the IC.

(1) Initially, let $t^* = t$ and L consist of all attributes in t.

(2) Repeat the following until t^* cannot be further changed.

(2.a) If there exist a CFK $\varphi = (R_1[X] \subseteq R_2[Y],\ t_p[X_p, Y_p])$ in $\Sigma_{(j,i)}$ and a tuple t' in D_i, such that $t^*[R_1(X)]$ is defined, $t^*[R_1(X_p)] = t_p[X_p]$, and $t^*[R_1(X)] = t'[Y]$, then

1) for each attribute $R_2(B)$ of t' that is not yet in t^*, extend t^* by adding a field $R_2(B) = t'[R_2(B)]$, and extend L by including the attribute $R_2(B)$;
2) for each attribute $C \in Y$, if $R_2(C)$ is not yet in t^*, then extend t^* by adding a field $R_2(C) = t'[R_2(C)]$.

(3) The projection $t^*[L]$ is returned as the IC of t.

Intuitively, we expand t^* by adding R_2 attributes whenever an R_2 tuple is identified by some attributes of t^* via a CFK. The relevant R_2 attributes include but are not limited to those in $R_2[Y_p]$. The set L keeps track of what attributes we shall include in the IC, excluding $R_2[Y]$, since $R_1[X]$ are already included in t^*. Observe that an attribute A may be extracted from different relations in the form of $R_i[A]$ and $R_j[A]$. As will be seen in Section V, we can include a single A attribute in t^* by merging these attributes and resolving conflicts.

Example 3.3: Recall relations review, sale and CD given in Fig. 1. Assume that CFKs φ_1–φ_4 of Example 3.2 are defined. Then the IC of tuple t_2 of Fig. 1(a) can be computed as follows.

(1) The IC t_2^* of t_2 is first expanded via the CFK φ_1, by adding sale attributes price, label, rating as well as album and artist. The list L includes all review attributes and sale attributes price, label and rating.

(2) Then t_2^* is expanded via φ_4, by adding CD attributes label, release, rank as well as title and artist. The list L is incremented by adding CD attributes label, release and rank.

(3) Finally, $t_2^*[L]$ is returned, which includes (a) review attributes album, artist and rating, (b) sale attributes price, label and rating, and (c) CD attributes label, release and rank. Note that rating appears in the IC as review(rating) and sale(rating); these can be merged into a single attribute following the methods of Section V; similarly for label. After this, the IC only includes attributes album, artist and rating from review, price and label from sale, and release and rank from CD, which are all the relevant attributes that can be extracted in the P2P system about the album identified by t_2. □

It is easy to verify that ICs are well defined and can be computed efficiently. Intuitively, the computation of ICs can be expressed as an inflational fixpoint query, which can be evaluated in PTIME (data complexity; see *e.g.,* [17] for fixpoint queries). In addition, any attribute $R_2[B]$ is added to the IC t^* of a tuple t at most once, since tuples across different databases are correlated via primary keys. Thus t^* does not have multiple occurrences of the same attribute (note that each attribute is associated with its relation name).

Theorem 3.3: *Over any collection $\mathcal{D}$ of databases that satisfy* CFKs *$\Sigma_{(j,i)}$, the* IC *t^* of a tuple t can be computed in* PTIME *in the size of $\mathcal{D}$. Furthermore, for any relation D in $\mathcal{D}$, there exists at most one tuple t' in D such that t^* is expanded by adding the attributes of t'.* □

The computation of ICs can be conducted more efficiently if all the sets $\Sigma_{(j,i)}$ of CFKs can be taken together. Indeed, for a tuple t consisting of attributes in S_l, one can determine whether attributes of schema S_k can be added to the IC of t based on the implication analysis of these CFKs, even if $\Sigma_{(l,k)}$ is not defined. However, as will be seen shortly, in a P2P system, $\Sigma_{(j,i)}$'s are distributed across different peers, and it is not very realistic to assume that $\Sigma_{(j,i)}$'s can be collected by a single site and remain unchanged during the static analysis.

IV. Processing SPC* Queries in P2P Systems

We next present a conceptual evaluation strategy for polymorphic queries in unstructured, schema-heterogeneous P2P systems. Consider such a P2P system $\mathcal{P} = (P_1, \ldots, P_n)$, where for each $j \in [1, n]$, the peer P_j is specified by:

- the relational schema S_j of its local database D_j, such that on each relation R in S_j, a primary key is defined;
- for some $i \in [1, n]$, a set $\Sigma_{(j,i)}$ of CFKs from relations of S_j to relations of S_i, each referring the primary key of an S_i relation, such that (D_j, D_i) satisfies $\Sigma_{(j,i)}$.

We assume that $\Sigma_{(j,i)}$ is stored at P_i. Here D_j's and $\Sigma_{(j,i)}$'s are the same as described in Section III-B, except that they are distributed across different peers. We only assume CFKs between neighboring peers, since in P2P networks typically only local knowledge is available (see, *e.g.,* [1], [7]).

In this section, we first outline the conceptual evaluation strategy. We then give the algorithms, which essentially compute ICs in the P2P setting. Finally, we present optimization techniques to improve the conceptual evaluation strategy.

A. A Conceptual Evaluation Strategy

Consider an SPC* query $Q = \Pi_L(Q_c)$, where $Q_c = \sigma_F(R_1 \times \ldots \times R_n)$ and $L = (L_1; L_2; \alpha)$. Given a P2P system $\mathcal{P}$ and the query Q posed on peer P_l (referred to as *the coordinator*), we compute the answer ans to Q in $\mathcal{P}$.

The evaluation of Q in $\mathcal{P}$ consists of three stages.

Stage 1. Upon receiving the query, the coordinator P_l rewrites Q to a normal SPC query Q_l, evaluates Q_l on the local database D_l, and prepares an initial set A_Q of tuples. In the next stage, for each tuple t in A_Q, the IC of t will be computed.

(1.a) We rewrite Q to an SPC query $Q_l = \pi_{L'_1 \cup L_1 \cup L_F}(Q_c)$, by dropping L_2 and α, retaining L_1, and adding relevant attributes $L'_1 \cup L_F$ from the local database D_l. Here L_F is the set of attribute appearing in the selection conditions F of Q_c, for which values are decided by reasoning about equality atoms $A =$ "a" and $A = B$. The set L'_1 includes attributes from relations whose primary keys appear in $L_1 \cup L_F$, *i.e.,* those attributes that can be determined by $L_1 \cup L_F$. These additional attributes help establish links to other peers via CFKs.

More specifically, let $K = \{Z_1, \ldots, Z_k\}$, where for each $i \in [1, k]$, Z_i is the primary key of some relation R_{l_i} in S_l, and Z_i is contained in $L_1 \cup L_F$. Then L'_1 is $\bigcup_{i=1}^{k}(\mathsf{attr}(R_{l_i}) \setminus Z_i)$, where $\mathsf{attr}(R_{l_i})$ denotes the attributes in R_{l_i}.

We use L_k to denote $\bigcup_{i=1}^{k} Z_i$, which will be used in Stage 3.

(1.b) We then evaluate Q_l on the local database D_l at peer P_l, which yields an initial set I_l of tuples.

(1.c) We extend each tuple t in I_l to a template t^* by including a field $(A = x_A)$ for each $A \in L_2$. Here x_A is a distinct (data) variable with its domain specified by a type variable.

In Stage 2 we will compute the IC of each template, by instantiating the variables for attributes in L_2 with concrete data values, and adding an attribute list L_α to t^*, if any.

Let $\mathcal{A}_Q$ be the set $\{t^* \mid t \in I_l\}$ of templates.

(1.d) We then send $\mathcal{A}_Q$ to each neighboring peer P_i of P_l.

Stage 2. Upon receiving $\mathcal{A}_Q$ from a peer P_j, peer P_i expands the ICs of tuples in $\mathcal{A}_Q$ using the local database D_i at P_i. Note that P_j is not necessarily the coordinator P_l.

(2.a) We expand the IC of each tuple t^* in $\mathcal{A}_Q$ by adding relevant attributes from D_i, based on the set $\Sigma_{(j,i)}$ of CFKs and as described in step 2(a) of Section III-B. In particular, when adding a field $R_2(A) = t'[R_2(A)]$ to t^*, if $A \in L_2$, *i.e.,* $A = x_A$ is in t^*, then in this field we instantiate x_A with $t'[R_2(A)]$; otherwise we add $R_2(A) = t'[R_2(A)]$ to L_α. This is done by generating a query Q_e and evaluating it on D_i.

(2.b) If the evaluation of Q_e yields changed tuples in $\mathcal{A}_Q$, then P_i sends $\mathcal{A}_Q$ to each neighboring peer. Otherwise, it sends $\mathcal{A}_Q$ back to the coordinator P_l. In particular, if α is not required and all attributes in L_2 are already instantiated, we also send $\mathcal{A}_Q$ back to P_l without further processing.

The process proceeds until no further changes are incurred to $\mathcal{A}_Q$, or "time to live (TTL)" expires.

Stage 3. Finally, the coordinator collects different versions $\mathcal{S}_{A_Q} = \{\mathcal{A}_{Q_1}, \ldots, \mathcal{A}_{Q_k}\}$ of $\mathcal{A}_Q$ from various peers, merges the tuples in these versions by treating L_k as a key, resolves conflicts, and generates final answer ans to the query Q in $\mathcal{P}$.

More specifically, for each tuple $t^* \in \mathcal{A}_{Q_i}$ ($i \in [1, k]$) and each attribute $A \in L_2$ in t^*, if is x_A is not yet instantiated, we set its value to null. We then group tuples from $\mathcal{A}_{Q_i}$ ($i \in [1, k]$) by their L_k attributes, and merge these tuples into a single tuple, which is an IC. The final answer ans is the outer union of all merged tuples. We shall discuss the details of object merge and conflict resolution in Section V.

Algorithm PolyQuery

Input: An SPC* query $Q = \Pi_{L_1;L_2;\alpha}(\sigma_F(R_1 \times \ldots \times R_n))$
Output: The answer ans to Q in $\mathcal{P}$;

(I). Initialization:
1. $(\mathcal{A}_Q, M_l, L_k)$:= InitQuery(Q);
2. $\mathcal{S}_{A_Q} := \{\mathcal{A}_Q\}$;
3. trigger ObjectExpansion($\mathcal{A}_Q$, M_l) at each neighboring peer P_i of P_l;

(II). Upon receiving $\mathcal{A}_Q$ from participating peers:
4. $\mathcal{S}_{A_Q} := \mathcal{S}_{A_Q} \cup \{\mathcal{A}_\mathcal{Q}\}$.

(III). When no more new $\mathcal{A}_Q$ is received or TTL expires:
5. ans := Merge($\mathcal{S}_{A_Q}, L_k$);
6. **return** ans;

Procedure InitQuery /* executed at coordinator P_l */

Input: An SPC* query $Q = \Pi_{L_1;L_2;\alpha}(\sigma_F(R_1 \times \ldots \times R_n))$.
Output: A template set $\mathcal{A}_Q$, a mapping table M_l, and key attributes L_k.

1. $L_k := \emptyset$; $L'_1 := \emptyset$; $M_l := \emptyset$; $L_F :=$ the set of attributes appearing in F;
2. **for** each relation R appearing in query Q with primary key Z **do**
3. **if** $Z \subseteq (L_1 \cup L_F)$ **then**
4. $M_l := M_l \cup \{(R, Z) \mapsto (R, Z)\}$; $L_k := L_k \cup Z$; $L'_1 := L'_1 \cup \mathsf{attr}(R)$;
5. let Q_l be $\pi_{L'_1 \cup L_1 \cup L_F}(\sigma_F(R_1 \times \ldots \times R_n))$;
6. I_l := the result of executing Q_l on D_l;
7. $\mathcal{A}_Q := \{t^* \mid t \in I_l\}$, where t^* is the template of t extended with attributes of L_2;
8. **return** $(\mathcal{A}_Q, M_l, L_k)$;

Fig. 2. Algorithm PolyQuery executed at coordinator P_l

B. The Evaluation Algorithm

We next present an algorithm, referred to as PolyQuery, for implementing the conceptual evaluation strategy. It consists of four main procedures: InitQuery, ObjectExpansion, ComposeQuery and Merge. While PolyQuery, InitQuery and Merge run on the coordinator P_l, namely, the peer on which the input SPC* query Q is posed, the ObjectExpansion and ComposeQuery run on each participating peers, invoked upon receiving a set $\mathcal{A}_Q$ of partial ICs from another peer.

To simplify the discussion we consider $Q = \Pi_L(Q_c)$, where $\alpha \in L$. The algorithm can be readily modified to deal with the case in the absence of α, as described in Stage (2.b) above.

Algorithm PolyQuery is initiated upon receiving an input query Q at the coordinator P_l. (a) As soon as Q is received, it invokes procedure InitQuery to compute the initial set $\mathcal{A}_Q$ of tuple templates. For each neighboring peer of P_l, it triggers procedure ObjectExpansion by passing $\mathcal{A}_Q$ to it (lines 1–3). This carries out Stage 1 of the conceptual evaluation strategy. (b) It then collects and maintains different versions of $\mathcal{A}_Q$ from various peers generated by procedure ObjectExpansion (line 4), which implements Stage 2 described earlier. (c) Finally, when no more changes can be made to $\mathcal{A}_Q$ at partic-

Procedure ObjectExpansion /* executed at P_i */

Input: $\mathcal{A}_Q$, and M_j received from peer P_j.
Output: An expanded version $\mathcal{A}'_Q$ of $\mathcal{A}_Q$, and updated mapping table M_i.

1. $\mathcal{A}'_Q := \emptyset$; $\mathcal{A}_T := \emptyset$; $M_i := \emptyset$;
2. **for** each template $t^* \in \mathcal{A}_Q$ **do**
3. $Q_t :=$ ComposeQuery$(t^*, M_j, M_i, \Sigma_{(j,i)})$;
4. **if** Q_t is null **then**
5. $\mathcal{A}_T := \mathcal{A}_T \cup \{t^*\}$;
6. **else**
7. execute query Q_t on D_i, and get a single tuple t';
8. **if** t' has attributes not yet in t^* **then**
9. extend t^* based on t'; $\mathcal{A}'_Q := \mathcal{A}'_Q \cup \{t^*\}$;
10. **if** $\mathcal{A}'_Q$ is empty, **then**
11. send $\mathcal{A}_Q$ back to the coordinator;
12. **else**
13. invoke ObjectExpansion$(\mathcal{A}'_Q \cup \mathcal{A}_T, M_i)$ at each neighbor of P_i;

Procedure ComposeQuery /* executed at P_i */

Input: t^*, M_j, M_i, and a set $\Sigma_{(j,i)}$ of CFKs.
Output: Query Q_t.

1. $W := \emptyset$; $F :=$ true; $\mathcal{C} := T$;
2. **for** each CFK $(R_1[X] \subseteq R_2[Y],\ t_p[X_p, Y_p]) \in \Sigma_{(j,i)}$ **do**
3. **if** $((R_1, X) \mapsto (R, Z)) \in M_j$ **and** $t^*[X_p] = t_p[X_p]$, **then**
4. $M_i := M_i \cup \{(R_2, Y) \mapsto (R, Z)\}$; $\mathcal{C} := \mathcal{C} \times R_2$;
5. $F := F$ **and** $T[Z] = R_2[Y]$; $W := W \cup (\mathsf{R}_2 \setminus Y)$;
6. **if** W is empty **then**
7. $Q_e :=$ null;
8. **else**
9. $V :=$ the set of attributes in t^* with concrete data values;
10. Create a temporary table $T(V)$, which contains $t^*[V]$ only.
11. $Q_t := \pi_{W \cup V}(\sigma_F(\mathcal{C}))$;
12. **return** Q_t;

Fig. 3. Algorithm PolyQuery at participating peers P_i: ObjectExpansion

ipating peers or TTL expires, the coordinator P_l assembles the final answer ans to Q in $\mathcal{P}$ by invoking procedure Merge, which merges objects and resolves conflicts (lines 5–6). This carries out Stage 3 of the conceptual evaluation strategy.

Below we present procedures InitQuery, ObjectExpansion and ComposeQuery, as shown in Figures 2 and 3. We defer the details of Merge to Section V.

Procedure InitQuery. This procedure implements Stage (1.a)–(1.c) of the conceptual evaluation strategy. It takes an SPC* query Q as input, and returns as output an initial set $\mathcal{A}_Q$ of tuple templates, a set L_k of attributes that will be used as the merge key in Stage 3 (see Stage (1.a)), and a table M_l that maintains the primary keys of relations involved in the templates, which will possibly be renamed via CFKs later. The use of M_l is to avoid including redundant attributes in the ICs, an alternative implementation of the set L adopted in the computation of ICs (see Section III-B). The procedure also uses variables L_F, L'_1 and Z, as described in Stage (1.a).

For each relation R appearing in query Q, InitQuery checks whether the primary key Z of R is included in $L_1 \cup L_F$. If so, M_l, L_k, and L'_1 are incremented accordingly (lines 2–4). After all those relations are checked, an SPC query Q_l is generated and executed (lines 5–6). This yields a set I_l of tuples, which is further expanded to include attributes in L_2 (line 7). Finally, $\mathcal{A}_Q$, M_l, and L_k are returned as the output (line 8).

Procedure ObjectExpansion. This procedure implements Stage 2 of the conceptual evaluation strategy, and is executed at participating peers P_i. It takes as input a set $\mathcal{A}_Q$ of tuple templates sent over from peer P_j, and a table M_j of mappings between the primary keys of relations in the templates and those of the relations at peer P_j. Given these, it expands $\mathcal{A}_Q$ by extracting data from the local database D_i based on CFKs $\Sigma_{(j,i)}$ from peer P_j to P_i. It uses several variables: (a) a set variable $\mathcal{A}'_Q$ stores the extended templates of $\mathcal{A}_Q$, (b) a set variable $\mathcal{A}_T$ keeps track of unchanged templates in $\mathcal{A}_Q$; and (c) a table M_i contains mappings between primary keys of relations in the templates and those of the relations at peer P_i.

For each template t^* in $\mathcal{A}_Q$, an SPC query Q_t is generated by calling procedure ComposeQuery (lines 2–3). If Q_t is null, t^* cannot be expanded and is added to $\mathcal{A}_T$ (lines 4–5). Otherwise, Q_t is evaluated on the local database D_i. By Theorem 3.3, there is a single tuple t' in the query result of Q_t. If t' has attributes that are not yet in t^*, t^* is expanded as illustrated in step 2(a) of the computation of ICs (see Section III-B). The expanded t^* is included in $\mathcal{A}'_Q$ (lines 6–9). After all templates are processed, if $\mathcal{A}'_Q$ is empty, then no templates are expanded at peer P_i and thus $\mathcal{A}_Q$ is sent back to the coordinator (lines 10–11). Otherwise the same process is invoked at each neighboring peer of P_i, with parameters $\mathcal{A}'_Q \cup \mathcal{A}_T$ and M_i from P_i (lines 12–13).

Procedure ComposeQuery. Given a template t^*, a mapping table M_j for relations at peer P_j, and the set $\Sigma_{(j,i)}$ of CFKs from peer P_j to P_i, this procedure generates a query that conducts all possible extensions at peer P_i (Stage 2.a). It also returns a new mapping table M_i for relations at peer P_i.

To generate query Q_t, we need to create the list W of projection attributes, the selection condition F, and the cross product $\mathcal{C}$ of relations, which are initialized to be empty list, empty list and a temporary table T, respectively (line 1). Here T indicates the template t^* and will be elaborated shortly. For each CFK $(R_1[X] \subseteq R_2[Y],\ t_p[X_p, Y_p]) \in \Sigma_{(j,i)}$, we check whether the primary key of R_1 is mapped to some primary key contained in t^*, by checking whether $(R_1, X) \mapsto (R, Z)$ is in the mapping table M_j. Furthermore, we check whether t^* satisfies the R_1 *pattern*, *i.e.,* $t^*[X_p] = t_p[X_p]$ (line 3). If so, we adjust the mapping to be from the primary key $R_2[Y]$ to $R[Z]$ and add it to the table M_i. In addition, we include relation R_2 in $\mathcal{C}$, attributes $(\mathsf{attr}(R_2) \setminus Y)$ in W, and condition $T[Z] = R_2[Y]$ in F (lines 4–5). After all CFKs in $\Sigma_{(j,i)}$ are checked, if W is empty, then t^* cannot be extended, and the query Q_t is set to be null (lines 6–7). Otherwise we create a temporary table $T(V)$ containing $t^*[V]$ only, where V is the set of attributes in t^* that are already associated with data values. At this stage query Q_t is composed by using F, W, V and $\mathcal{C}$ (lines 8–11). Finally, query Q_t is returned (line 12). This implements step 2(a) of the computation of ICs (Section III-B).

Example 4.1: We show how algorithm PolyQuery evaluates the SPC* query Q_1 of Example 2.1, based on the CFKs given in Example 3.2, in the P2P system described in Example 1.1.

When Q_1 is posed on peer P_1, the algorithm generates the following SPC query by invoking procedure InitQuery:

Q_1': **select** album, artist, rating
from review
where artist = "Denver, J" **and** rating = "4"

Here $L_k = \{\mathsf{album}, \mathsf{artist}\}$, $L_1' = \{\mathsf{artist}, \mathsf{rating}\}$, and M_1 maps album, artist of review to themselves.

The query Q_1' is evaluated at P_1, which yields a set $I_1 = \{t_2, t_3\}$. This set is expanded into the initial $\mathcal{A}_Q$ as shown in Fig. 4 (a). The set $\mathcal{S}_{A_Q}$ consists of $\mathcal{A}_Q$ only. Peer P_1 then triggers procedure ObjectExpansion at peer P_2, since P_2 is the only neighboring peer of P_1, *i.e.,* there is a set of CFKs from the review database of P_1 to the sale database of P_2.

After P_2 receives $\mathcal{A}_Q$ and M_1, it creates a temporary table T_2 (resp. T_3) to store $t_2^*[\mathsf{album}, \mathsf{artist}, \mathsf{rating}]$ (resp. $t_3^*[\mathsf{album}, \mathsf{artist}, \mathsf{rating}]$). Procedure ComposeQuery is called to generate SPC queries Q_{2,t_2} (similarly for Q_{2,t_3}):

Q_{2,t_2}: **select** price, label, rating
from T_2, sale
where T_2.album = sale.album **and** T_2.artist = sale.artist

A new mapping table M_2 is created, which maps sale(album) to review(album) and sale(artist) to review(artist). Queries Q_{2,t_2} and Q_{2,t_3} are evaluated on the sale relation of P_2, and based on the query results, $\mathcal{A}_Q$ is expanded to $\mathcal{A}_{Q2}$ as shown in Fig. 4 (b). Then $\mathcal{A}_{Q2}$ and M_2 are sent to peer P_3.

Procedure ObjectExpansion at P_3 finds that only template t_2^* can be expanded. To do so it generates query Q_{3,t_2}, where T_2 is a temporary table consisting of a single tuple $t_2^*[\mathsf{album}, \mathsf{artist}, \mathsf{rating}, \mathsf{sale(rating)}, \mathsf{price}, \mathsf{label}, \mathsf{rank}]$.

Q_{3,t_2}: **select** release, rank
from T_2, CD
where T_2.album = CD.title **and** T_2.artist = CD.artist

Here the new table M_3 maps CD(title) to review(album) and CD(artist) to review(artist). The query Q_{3,t_2} is evaluated on the CD relation. Based on the result of the query, $\mathcal{A}_Q$ is expanded to $\mathcal{A}_{Q3}$ as shown in Fig. 4 (c). This set is sent to coordinator P_1 since no more changes can be made to $\mathcal{A}_{Q3}$.

The coordinator merges $\mathcal{S}_{A_Q} = \{\mathcal{A}, \mathcal{A}_{Q3}\}$. The final answer to Q_1 is generated, as shown in Fig. 4 (d), where the output schema $\mathsf{sch}(Q_1)$ is (album, artist, rating, sale(rating), price, label, CD(label), release, rank), and L_α consists of implicit attributes artist, rating, sale(rating), CD(label) and rank. □

Complexity. Algorithm PolyQuery implements the computation of ICs. It is essentially an (inflational) fixpoint computation (Theorem 3.3), which is in PTIME (data complexity) [17]. In contrast, in previous models query processing could be undecidable [1] or have a non-elementary complexity [5], [6].

In practice, the coordinator needs to decide when to start merging different versions of $\mathcal{A}_Q$. This can be done by means of a maximum TTL: the assembling process starts right after the TTL expires. Alternatively, we can maintain a log of template expansions at each peer. Note that if a template t^* is extended at a peer P_i, then t^* cannot be expended by visiting P_i again since all the relevant information from P_i is already included in t^*. Thus by using the log we ensure that each P_i is visited at most once for expanding a template t^*.

	album	...	price	label	release
t_2^*:	Take Me Home	...	X_{price}	X_{label}	X_{release}
t_3^*:	Greatest Hits	...	X_{price}	X_{label}	X_{release}

(a) Initial $\mathcal{A}_Q$ computed on P_1

	album	...	price	label	release
t_2^*:	Take Me Home	...	5.97	Windstar	X_{release}
t_3^*:	Greatest Hits	...	7.99	BMG	X_{release}

(b) Extended $\mathcal{A}_{Q2}$ on P_2

	album	...	label	release	rank
t_2^*:	Take Me Home	...	Windstar	12/20/2005	2,175
t_3^*:	Greatest Hits	...	BMG	X_{release}	null

(c) Extended $\mathcal{A}_{Q3}$ on P_3

	album	...	label	release	rank
t_2^*:	Take Me Home	...	Windstar	12/20/2005	2,175
t_3^*:	Greatest Hits	...	BMG	null	null

(d) Final answer ans to query Q_1

Fig. 4. Different versions of $\mathcal{A}_Q$ and the final query result

C. Optimization Techniques

The conceptual evaluation strategy is presented just to give the semantics of polymorphic queries. It can certainly be improved. In particular, ObjectExpansion sends $\mathcal{A}_Q$ to other peers, and moreover, it generates a large number of queries, one for each template in $\mathcal{A}_Q$. Below we present optimization methods to reduce data shipping and the number of queries.

Sending necessary data only. Procedure ObjectExpansion sends all templates in $\mathcal{A}_Q$ from a peer P_j to its neighboring peers P_i (line 13, Fig. 3). In fact, only those templates that are expanded at peer P_j may be further extended at P_i, since the correspondences are only defined for tuples between neighboring peers. Thus it suffices to send only $\mathcal{A}_Q'$ to its neighboring peers, while sending $\mathcal{A}_T$ back to the coordinator.

In addition, in the templates of $\mathcal{A}_Q'$ sent from P_j to P_i, only those attributes that appear in relations at peer P_j or in $L_k \cup L_2$ are useful in later expansion. Indeed, the extension is based on the CFKs in Σ_{ji}, which involve only attributes in relations at P_j and P_i. Thus we partition each template into two parts: one consisting of only P_j attributes and those in $L_k \cup L_2$, while the other consisting of the rest along with L_k (note that L_k serves as the key when merging objects). We send the former to P_i, and the latter to the coordinator.

The standard pipeline techniques [21] can also be used here to improve response time: tuples in $\mathcal{A}_Q$ at P_j are shipped to neighboring peers as soon as they are generated, instead of waiting for the completion of the generation of the entire $\mathcal{A}_Q$.

Reducing the number of queries. Procedure ComposeQuery generates a separate query for each template in $\mathcal{A}_Q$. These queries can be merged via left outer joins, as illustrated below.

Example 4.2: Recall from Example. 4.1 that to expand templates t_2^* and t_3^* at peer P_2, we generate two queries Q_{2,t_2} and Q_{2,t_3}. These queries can be merged into a single query:

Q_2: **select** T.album, price, label, rank
from T **left outer join** sale on
T.album = sale.album **and** T.artist = sale.artist

where T is a temporary table that stores both t_2^* and t_3^* □

Procedure Merge /* executed at coordinartor P_l */

Input: A set $\mathcal{S}_{A_Q}$ of different versions of $\mathcal{A}_Q$, and key L_k.
Output: Answer ans to the input SPC* query.

1. ans := $\emptyset$;
2. **for** each $\mathcal{A}_Q \in \mathcal{S}_{A_Q}$ **do**
3. **for** each template $t^* \in \mathcal{A}_Q$ **do**
4. replace each uninstantiated variable in t^* with a null;
5. **for** each tuple $t \in$ ans such that $t^*[L_k] = t[L_k]$
6. **for** each attribute A in attr$(t^*) \setminus L_k$ **do**
7. **if** $A \in$ attr(t) **then**
8. $t[A] := t[A] \cup \{t^*[A]\}$;
9. **if** $A \notin$ attr(t) **then**
10. expand t by adding A attribute; $t[A] := \{t^*[A]\}$;
11. **if** no tuple $t \in$ ans such that $t^*[L_k] = t[L_k]$ **then**
12. ans := ans $\cup \{t\}$ such that $t[A] = \{t^*[A]\}$ for each attribute $A \in$ attr$(t^*) \setminus L_k$;
13. **return** ans;

Fig. 5. Algorithm PolyQuery at coordinator P_l: Merge

In general, we can compose a single query Q_e at peer P_i to compute the ICs of all the templates in $\mathcal{A}_Q$. The size of Q_e is bounded by $O(m * |M_i|)$, where m is the maximum arity of the relations in Q_e, and M_i is the mapping table for relations at P_i. The query contains $|M_i|$ left outer joins in the **from** clause and $|M_i| - 1$ equality atoms in the **where** clause. However, when M_i is large, Q_e may not be practical as it involves a large number of joins. In this case, we partition M_i into h sets, where h is a constant decided by the capacity of the underlying DBMS. For each set we generate a query, involving h left joins in the **from** clause and $h-1$ equality atoms in the **where** clause. This yields $|M_i|/h$ queries of constant size.

V. Object Merge and Conflict Resolution

A template t^* may contain attributes $R_i[A]$ and $R_j[A]$ extracted from different peers. As mentioned earlier, one often wants an IC to have a single A attribute, *i.e.,* we want to merge $R_i[A]$ and $R_j[A]$ into one. Although $R_i[A]$ and $R_j[A]$ are about the same attribute, they may differ in domains and/or values, *i.e.,* they have *typing conflicts* and/or *value conflicts*.

In this section we present conflict resolution methods and procedure Merge, which is part of algorithm PolyQuery.

Although conflict resolution is not encountered in previous P2P query models, it has been studied for data integration [13], [14], [15], [22] and uncertain data [11], [12]. We adapt these methods to resolve typing and value conflicts that emerge when correlating information collected from different peers.

Resolving typing conflicts. One can maintain a uniform directory over all peers, such that attributes from various relations with the same label are mapped to a uniform type. This is not too expensive to do as we only require the registration of the meta-data in the directory, rather than the data.

Alternatively, we can leverage an automatic type casting mechanism. Consider two templates t^* and t'^*, where $t^*[A]$ has type τ_1 and $t'^*[A]$ has type τ_2. To merge t^* and t'^*, the mechanism casts τ_1 and τ_2 to a uniform type.

Resolving value conflicts. A number of methods have been developed to resolve value conflicts in data integration. A naive method is to set conflicting attributes to be null. A better way is by using conflict resolution functions (*e.g.,* [13], [14], [15]), such as ANY, FIRST, LAST, MIN, MAX, AVG, DISCARD. Given a list Val of values with the same data type, ANY draws a random value from Val, FIRST returns the first one, LAST returns the last one; MIN, MAX and AVG return the minimum, maximum and average values, respectively, when Val consists of numerical values; DISCARD returns null if Val contains more than one value, and it returns the only value in Val otherwise. The user may choose one of these functions.

	album	rating	label
t_2^*:	Take Me Home	{4, high}	{Windstar, Sain}
t_3^*:	Greatest Hits	{4, high}	BMG

Fig. 6. The answer ans to query Q_1

Or-sets. We adopt OR-sets to resolve both typing conflicts and value conflicts. The notion of OR-sets was proposed in [11], [12] for managing incomplete information. An OR-set is a disjunction of a set of data values. As a simple example, a tuple t = ("Take Me Home", {"4", "high"}, {"DancingBull", "Sain"}) indicates that the album "Almost Heaven" has a rating of either "4" or "high" and a label of either "DancingBull" or "Sain". Note that t represents four possible worlds (tuples). Compared to other resolution methods, OR-sets provide a succinct representation for all the relevant information, preventing loss of information. Further, no separate mechanism is required to resolve typing conflicts.

Procedure Merge. We develop procedure Merge, shown in Fig. 5, based on OR-sets. This procedure is executed at the coordinator peer P_l (line 5 of algorithm PolyQuery), and it implements Stage 3 of the conceptual evaluation strategy (Section IV). It takes as input a set L_k of attributes and a set $\mathcal{S}_{A_Q}$ of different versions of $\mathcal{A}_Q$. It groups the tuples in these versions by treating L_k as a key, resolves conflicts, merges the tuples, and generates the answer ans to the input query Q.

Procedure Merge generates ans as follows, where ans is initially an empty set (line 1). In the process, whenever we add a tuple t to ans, we let $t[A]$ be a set of values for attribute A that is not in L_k. The process proceeds as follows. For each template t^* in some $\mathcal{A}_Q$ of $\mathcal{S}_{A_Q}$, we first replace all uninstantiated variables in t^* with null (lines 2–4). Then we check whether there exists a tuple t in the set ans with the same key, *i.e.,* $t^*[L_k] = t[L_k]$ (line 5). If so, we merge tuple t and template t^*. More specifically, for each attribute that is in the set attr(t^*) of attributes in t^* but is not in the key L_k, if A already appears in t, we increment the set $t[A]$ by including $t^*[A]$. Otherwise, we expand t by adding a field $A = \{t^*[A]\}$, with an singleton set as its value (lines 6–10). If there are no tuples in ans with the same key, we simply add t^* to ans, while each field of t^* is made a set (lines 11-12).

Example 5.1: Recall the evaluation of SPC* query Q_1 described in Example 4.1, and the result of Q_1 from Fig. 4 (d). To make the example more interesting, suppose now that tuple t_8 in Fig. 1 is modified such that t_8[label] is "Sain" instead of "Windstar". Then the result of Q_1 contains conflicts in the ICs t_2^* and t_3^*, resulted from conflicts between (a) t_2[rating]

and t_5[rating], (b) t_5[rating] and t_3[rating], and (c) t_5[label] and t_8[label]. When procedure Merge is applied to the result, it yields the answer ans to Q_1, as shown in Fig. 6 (certain attributes are not shown), in which t_2^* represents four possible worlds and t_3^* represents two possible worlds.

Recall query Q_0 from Example 2.1, which demands explicit attributes L_2 = [price, label, release], without implicit attributes L_α. When evaluated in the P2P system above, Q_0 finds the same answer as Q_1 excluding artist, rating and rank. □

Conflict resolution can also be conducted earlier, by procedure ObjectExpansion whenever possible. This effectively reduces data shipping and network traffic.

VI. Experimental Evaluation

In this section we present a preliminary experimental study of algorithm PolyQuery for processing polymorphic (SPC*) queries. We study the effects of the following factors on the performance of PolyQuery: (a) SPC* queries Q and (b) the average number $|\Sigma|$ of CFKs between neighboring peers. We also evaluate the impacts of these factors on the amount of relevant information that SPC* queries find versus their traditional SPC counterparts. In addition, we assess the effectiveness of our optimization techniques.

Experimental setting. We used 10 Linux machines (peers), distributed over a 100Mb/s Ethernet LAN, with memories ranging from 512M to 2G, and CPUs ranging from Intel Pentium 4 (2.80GHz) to Intel Core™ 2 Duo (dual-core, 1.86GHz).

We randomly generated the topology of the P2P network, where each peer had 2 to 4 neighbors. At each peer, we deployed a local database db consisting of at most 20 relations. We designed a generator to produce CFKs and db for each peer. Given $m \leq 20$, the generator creates m relations, each with 10 to 15 attributes. For each relation, we choose its primary key and attributes appearing in CFKs. For each pair of neighboring peers and for a given parameter k, the generator produces k satisfiable CFKs between relations across the peers. For a parameter n, it randomly produces n tuples for each of those relations, such that (1) the relation satisfies its primary key, and (2) there are at least k tuples that reference tuples in the relations at the other peer. Note that different relations at the same peer may share the same primary key.

We implemented three algorithms: (a) PolyQuery as given in Figures 2, 3 and 5, (b) PolyQuery-Op, which is PolyQuery equipped with optimizations presented in Section IV-C, and (c) NaiveQuery, which, given an SPC* query, simply evaluates the local query generated by InitQuery of Fig. 2, *i.e.,* it simulates the behavior of the corresponding SPC query without conducting object expansion.

All algorithms were implemented in Java. For each experiment, we randomly generated 5 similar SPC* queries with fixed parameters, and ran the experiments 5 times on each of the datasets. The average is reported here.

Experimental results. We conducted two sets of experiments, evaluating the impacts of SPC* queries Q and the number $|\Sigma|$ of CFKs, respectively, on the performance of the three algorithms. Here the complexity of Q is measured by the number $|I_l|$ of tuples in its initial query result I_l. For each query Q, we report the running time, the size |ans| of final query result ans, and the size |trans| of network traffic data trans, which mainly consists of different versions of $\mathcal{A}_Q$, of algorithms PolyQuery and PolyQuery-Op. Here |ans| is measured by the number of concrete (non-null) data values in ans and |trans| by the number of characters sent among peers. Note that PolyQuery and PolyQuery-Op produce the same ans. As a result, we only report ans from one of them in the sequel.

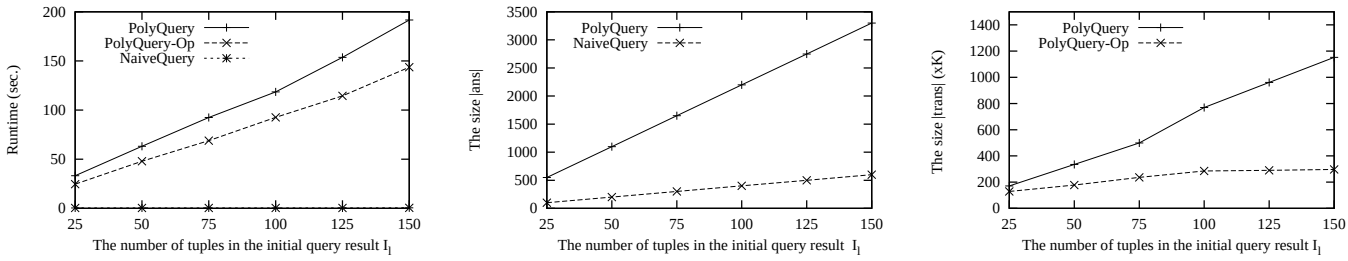
Varying SPC* queries. To evaluate the impact of the SPC* queries, we fixed |db| = 1000K and $|\Sigma|$ = 30, and varied $|I_l|$ from 25 to 150 tuples.

Figure 7(a) shows the running time of the three algorithms PolyQuery, PolyQuery-Op and NaiveQuery. When $|I_l|$ goes up, the running time increases, and all algorithms scale well *w.r.t.* $|I_l|$. Furthermore, algorithm PolyQuery-Op is more efficient than PolyQuery. The former is 34% faster than the later for $|I_l|$ = 150. Compared with PolyQuery, the running time of NaiveQuery is very small because only a small portion of relevant information, *i.e.,* local information, in the P2P network is retrieved. However, the running time of PolyQuery is still acceptable. More importantly, Figure 7(b) shows that algorithm PolyQuery finds much more relevant information than NaiveQuery does. The benefit becomes more prominent as $|I_l|$ goes up. Indeed, PolyQuery finds 4.5 times more information than NaiveQuery when $|I_l|$ = 150. This demonstrates the usefulness of polymorphic queries in a P2P network. Figure 7(c) reports the network traffic of algorithms PolyQuery and PolyQuery-Op. It clearly shows that when the data shipped among peers is concerned, the two algorithms also scale well *w.r.t.* $|I_l|$. It also demonstrates that PolyQuery-Op reduces communication cost, another benefit of PolyQuery-Op in addition to running time.

Varying the number of CFKs. To evaluate the impact of the number of CFKs, we fixed |db| = 1000K and $|I_l|$ = 100, and varied $|\Sigma|$ from 10 to 30 CFKs.

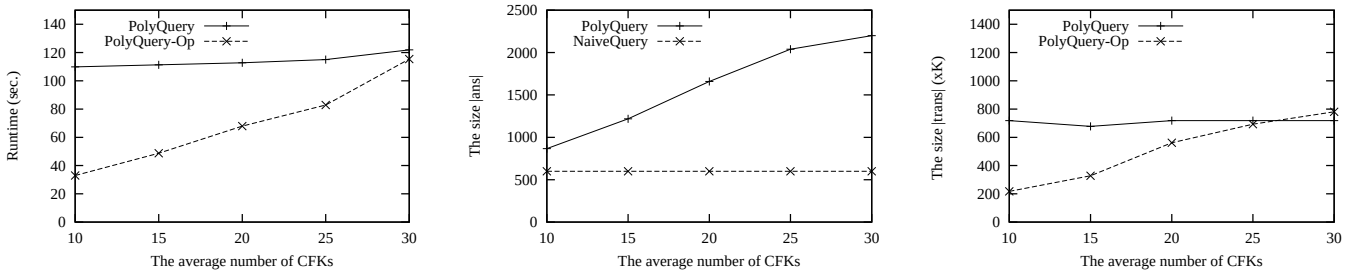
Figure 8(a) shows that both PolyQuery and PolyQuery-Op scale well *w.r.t.* $|\Sigma|$, while again PolyQuery-Op demonstrates better scalability. Figure 8(b) shows that if there are more CFKs between neighboring peers, algorithm PolyQuery finds more sensible information; indeed, these CFKs increase the opportunities of object expansion. Algorithm NaiveQuery is insensitive to $|\Sigma|$ since the information that it finds is decided solely by the initial query result I_l, which is from the local database only, without conducting object expansion via CFKs. Figure 8(c) shows, in addition to running time, these algorithms scale well with $|\Sigma|$ when network traffic is concerned, *i.e.,* both algorithm PolyQuery and PolyQuery-Op do not incur heavy network traffic.

Summary. We have presented several experimental results from our preliminary performance study of algorithms PolyQuery, PolyQuery-Op and NaiveQuery. From these re-



(a) Scalability (b) The size |ans| (c) The size |trans|

Fig. 7. Varying SPC* queries



(a) Scalability (b) The size |ans| (c) The size |trans|

Fig. 8. Varying the number of CFKs

sults we find the following. First, algorithm PolyQuery finds far more relevant information than algorithm NaiveQuery. Second, both PolyQuery and PolyQuery-Op scale well with the size of initial query result $|I_l|$ and with the average number of CFKs. Third, the communication cost incurred by evaluating polymorphic queries increases rather slowly when the queries produce more local results or the number of CFKs goes up. In other words, one can expect that the evaluation of polymorphic queries will not flood a P2P network.

VII. Related Work

As remarked in Section I, none of the previous P2P query models [1], [2], [3], [4], [5], [6], [7], [8] supports SPC* queries.

Several query models have been developed for unstructured, schema-heterogeneous P2P systems, based on semantic mappings (*e.g.,* [1], [2], [3]). Piazza [1], for example, interprets P2P queries based on schema mappings and certain query answering. It is most effective when the schemas of neighboring peers do not have radically different structures. The problem of computing certain answers is, however, undecidable even for SPC queries in a P2P setting, and is already intractable (data complexity) under severe restrictions on the configuration of the systems (acyclic configuration, under the closed world assumption) [1], over which one has no centralized control. In light of this, a semantics of P2P queries was proposed in [2] in terms of epistemic FO, yielding a PTIME data complexity. Our query model is complementary to such query models. While the focus of our model is to expand objects found at a local peer by adding relevant *attributes* retrieved from other peers, models based on semantic mappings are able to add more *objects* retrieved from other peers. On the other hand, such models cannot accommodate attributes in a query that are not defined in the local schema, or cope with attributes from one schema that do not exist in another schema.

PeerDB [4] focuses on keyword searches only and adopts a semantics based on informational retrieval.

Closer to this work is Hyperion, which is based on mapping (concordance) tables [5], [6]. A mapping table maintains value and name correspondences between data in two neighboring peers. Given a query Q, Hyperion traverses the peers in the P2P system, translates Q to a set of queries defined on databases at those peers based on mapping tables, and collects relevant objects found by those translated queries. As opposed to object expansion, Hyperion does not augment objects by including additional attributes; instead, relevant objects are simply put together via outer union, which does not involve conflict resolution. Hyperion does not allow queries to refer to attributes that are not defined in the local schema. Furthermore, when the mapping tables across peers in the system cannot be collected by a coordinator (a common situation as in a P2P system the configuration is rather dynamic), query evaluation incurs a non-elementary complexity. In contrast, our query model is based on CFKs, which suffice to express attribute and value correspondences commonly found in practice, while allowing queries to be processed in PTIME.

A notion of query expansion has been explored for P2P queries [7], [8]. The idea is to enhance queries with vague or surrounding concepts of pre-defined attributes, when users are unable to identify precise keywords. This is quite different from object expansion: query expansion aims to find more relevant results of fixed attributes, and is developed mostly for

keyword queries [23], [24]; in contrast, object expansion is to enrich objects with new relevant attributes, for SPC queries.

To our knowledge, none of the previous models is capable of answering the query Q_0 given in Example 1.1. Furthermore, these models do not consider object merge and conflict resolution for P2P query evaluation.

To simplify the discussion we presented a simple evaluation strategy for polymorphic queries. A number of known techniques can be leveraged to improve the strategy, such as local indices, routing indices and distributed resource location protocol (see, *e.g.,* [25] for various P2P search methods).

There has also been work on polymorphic type inference for relational algebra [26], [27]. The focus is to determine, given a query Q, on what schema Q is well defined, and to infer the "principle" (most generic) type for Q. This is studied for centralized systems, without taking into account the heterogeneous nature of database schemas in a P2P system. In contrast, given a query Q that may not be well defined on the schema of the local peer in the standard semantics, this work studies how to evaluate the query based on object expansion, and to derive the output schema of Q in a P2P system. This also involves object merge and conflict reconciliation, which are not encountered in polymorphic type inference.

SchemaSQL [28] also supports dynamic derivation of output schema based on the contents of input instances. It is developed for federate databases, while polymorphic queries are for P2P systems, in which an SPC* query may be posed without any knowledge of the schemas in other peers, and furthermore, its output schema cannot be determined until all linked peers are traversed. SchemaSQL does not supports object expansion.

VIII. Conclusion

We have proposed a query model for P2P systems, supporting object expansion. The model is based on (a) polymorphic queries, a revision of SPC queries by incorporating type variables to support queries with attributes that are not defined in a local schema, (b) a notion of information closures underlying the semantics of polymorphic queries, (c) a notion of contextual foreign keys for specifying correspondences between schemas across various peers. We have also provided (d) evaluation and optimization methods for processing polymorphic queries, and (e) techniques for conflict resolution. Our preliminary experimental study has verified that our techniques are capable of finding far more relevant information than traditional SPC queries in P2P systems, and furthermore, that polymorphic queries can be evaluated efficiently without incurring heavy network traffic.

A number of issues need further investigation. First, we are currently experimenting with larger datasets and other optimization techniques for P2P query processing. Second, as remarked in Section VII, this work is complementary to previous P2P query models. A natural extension to this work is by using conditional inclusion dependencies (CINDs) [18] instead of CFKs, to specify both schema matching and mapping tables [5], [6]. Capitalizing on CINDs, it is possible to combine query models based on schema mapping [1], [2], the model based on mapping tables [5], [6], and our model based on object expansion. This may lead to an integrated query model that is capable of both enriching objects with additional attributes and retrieving new objects from peers across a P2P system. However, the expressive power does not come for free: the complexity for the computation of information closures based on CINDs will likely be high. Finally, it is interesting to extend SPC* queries to polymorphic relational algebra.

References

[1] A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov, "The Piazza peer data management system," *TKDE*, vol. 16, no. 7, pp. 787–798, 2004.

[2] D. Calvanese, G. D. Giacomo, M. Lenzerini, and R. Rosati, "Logical foundations of peer-to-peer data integration," in *PODS*, 2004.

[3] K. Aberer, P. Cudre-Mauroux, and M. Hauswirth, "The chatty web: emergent semantics through gossiping," in *WWW*, 2006.

[4] W. S. Ng, B. C. Ooi, K.-L. Tan, and A. Zhou, "PeerDB: A P2P-based system for distributed data sharing," in *ICDE*, 2003.

[5] A. Kementsietsidis and M. Arenas, "Data sharing through query translation in autonomous sources," in *VLDB*, 2004.

[6] A. Kementsietsidis, M. Arenas, and R. J. Miller, "Data mapping in P2P systems: Semantics and algorithmic issues." in *SIGMOD*, 2003.

[7] H. F. Witschel and T. Bohme, "Evaluating profiling and query expansion methods for P2P information retrieval," in *P2PIR*, 2005.

[8] B. Yu, L. Liu, B. C. Ooi, and K. L. Tan, "Keyword join: Realizing keyword search in P2P-based database systems," *Technical report, MIT*, 2005.

[9] R. Blanco, N. Ahmed, D. Hadaller, L. G. A. Sung, H. Li, and M. A. Soliman, "A survey of data management in peer-to-peer systems," Technical Report CS-2006-18, University of Waterloo, Tech. Rep., 2006.

[10] J. Garrigue and D. Rémy, "Extending ML with semi-explicit higher-order polymorphism," in *TACS*, 1997.

[11] T. Imielinski, S. A. Naqvi, and K. V. Vadaparty, "Incomplete objects - a data model for design and planning applications," in *SIGMOD*, 1991.

[12] ——, "Querying design and planning databases," in *DOOD*, 1991, pp. 524–545.

[13] L.-L. Yan and M. T. Özsu, "Conflict tolerant queries in aurora," in *CoopIS*, 1999.

[14] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina, "Object fusion in mediator systems," in *VLDB*, 1996.

[15] F. Naumann, A. Bilke, J. Bleiholder, and M. Weis, "Data fusion in three steps: Resolving schema, tuple, and value inconsistencies," *IEEE Data Eng. Bull.*, vol. 29, no. 2, pp. 21–31, 2006.

[16] Full version, *http://homepages.inf.ed.ac.uk/sma1/pqp.pdf*.

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

[18] L. Bravo, W. Fan, and S. Ma, "Extending dependencies with conditions," in *VLDB*, 2007.

[19] L. Haas, M. Hernández, H. Ho, L. Popa, and M. Roth, "Clio grows up: from research prototype to industrial tool," in *SIGMOD*, 2005.

[20] W. Fan and L. Libkin, "On XML integrity constraints in the presence of DTDs," *JACM*, vol. 49, no. 3, pp. 368–406, 2002.

[21] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*. Prentice-Hall, 1999.

[22] K.-U. Sattler, S. Conrad, and G. Saake, "Interactive example-driven integration and reconciliation for accessing database federations," *Inf. Syst.*, vol. 28, no. 5, pp. 393–414, 2003.

[23] E. M. Voorhees, "Query expansion using lexical-semantic relations," in *SIGIR*, 1994.

[24] M. Mitra, A. Singhal, and C. Buckley, "Improving automatic query expansion," in *SIGIR*, 1998.

[25] D. Tsoumakos and N. Roussopoulos, "A comparison of peer-to-peer search methods," in *WebDB*, 2003.

[26] J. V. den Bussche, D. V. Gucht, and S. Vansummeren, "A crash course on database queries," in *PODS*, 2007.

[27] J. V. den Bussche and E. Waller, "Polymorphic type inference for the relational algebra," *JCSS*, vol. 64, no. 3, pp. 694–718, 2002.

[28] L. V. S. Lakshmanan, F. Sadri, and S. N. Subramanian, "SchemaSQL: An extension to SQL for multidatabase interoperability," *TODS*, vol. 26, no. 4, pp. 476–519, 2001.

APPENDIX A: Proofs

Proof of Theorem 3.1

The implication problem for CFKs is decidable in quadratic time.

Proof Sketch: This is verified by providing a quadratic-time algorithm that, given a set Σ of CFKs and a CFK $\varphi = (R_1[X] \subseteq R_2[Y], t_1^p[X_p], t_2^p[Y_p])$, checks whether or not $\Sigma \models \varphi$. The algorithm is based on an extension of the chase technique (see [17] for details about chase).

We first present algorithm Implication shown in Fig. 9, then we show $\Sigma \models \varphi$ if and only if algorithm Implication outputs *true*. Finally we show algorithm Implication terminates in quadratic time.

(1). The main purpose of algorithm Implication is to compute all the inferrable information, based on which we can decide whether or not $\Sigma \models \varphi$. It takes a set Σ of CFKs and a CFK $\varphi = (R_1[X] \subseteq R_2[Y], t_1^p[X_p], t_2^p[Y_p])$ as inputs, and returns as output true or false. If it returns true then $\Sigma \models \varphi$. Otherwise, $\Sigma \not\models \varphi$. Algorithm Implication uses set S to store all the inferrable information, *i.e.,* for any element $(R_a[X'], t_b^p[X'_p]) \in S$, $\Sigma \models (R_1[X] \subseteq R_a[X'], t_1^p[X_p], t_b^p[X'_p])\}$. It also uses variable $change$ to mark whether S is changed or not and control the termination of the loop.

Algorithm Implication initializes the set S as $\{(R_1[X], t_1^p[X_p])\}$. Intuitively, this represents there is a tuple t in R_1 such that $t[X_p] = t_1^p[X_p]$. It also sets the variable $change$ to be *true* initially (line 1). Set Σ' only stores the necessary CFKs to be considered for each run of the loop, which is initialized to be Σ (line 2). The algorithm continues until there are no changes for S (lines 3–14). For each CFK $\varphi' = (R_a[U] \subseteq R_b[V], t_a^p[U_p], t_b^p[V_p])$ in Σ, we test where there exists an element $(R_a[U], t_a'^p[U'_p]) \in S$ such that $U_p \subseteq U'_p$ and $t_a'^p[U_p] = t_a^p[U_p]$ (lines 5–6). If so, we add the its corresponding information to S (lines 8–14). And this CFK is removed from Σ' since it does not need to be considered again. Correspondingly, $change$ is marked to be true (line 7). If there is already an element $(R_b[V], t_b'^p[V'_p])$ in S, we need to merge $(R_b[V], t_b'^p[V'_p])$ and $(R_b[V], t_b^p[V_p])$ (lines 8–10). More specifically, if there exists an attribute B such that $B \in V_b \cap V'_b$ and $t_b^p[B] \neq t_b'^p[B]$, it means some conflicts happen. Intuitively this means a tuple t in R_b needs to have $t[B]$ as both $t_b^p[B]$ and $t_b'^p[B]$, which is impossible. In turn, it shows that there are no tuples in R_1 satisfying $t_1^p[X_p]$ of φ, and we return *true* since $\Sigma \models \varphi$ in this case. Otherwise, $t_b'^p[V'_p]$ is expanded to include $t_b^p[V_p]$. If there are no such elements in S, we simply add $(R_b[V], t_b^p[V_p])$ to S (lines 13–14). When no changes can be made to S, the algorithm jumps out of the loop, then we can decide whether or not $\Sigma \models \varphi$ based on S (lines 15–18). If there is an element $(R_2[Y], t_2'^p[Y'_p])$ in S such that $Y_p \subseteq Y'_p$ and $t_2'^p[Y'_p] = t_2^p[Y_p]$, then $\Sigma \models \varphi$ and variable $result$ is set to true. Otherwise, $\Sigma \not\models \varphi$ and variable $result$ is set to false. Finally, $result$ is returned (line 19).

Procedure Implication

Input: A set Σ of CFKs and a CFK $\varphi = (R_1[X] \subseteq R_2[Y], t_1^p[X_p], t_2^p[Y_p])$.
Output: true or *false*.

1. $S := \{(R_1[X], t_1^p[X_p])\}$; $change := true$;
2. $\Sigma' := \Sigma$;
3. **while** $(change)$ **do**
4. $change := false$; $\Sigma := \Sigma'$;
5. **for** each CFK $\varphi' = (R_a[U] \subseteq R_b[V], t_a^p[U_p], t_b^p[V_p])$ in Σ **do**
6. **if** there exists $(R_a[U], t_a'^p[U'_p]) \in S$ such that $U_p \subseteq U'_p$ **and** $t_a'^p[U_p] = t_a^p[U_p]$, **then**
7. $\Sigma' := \Sigma' \setminus \{\varphi'\}$; $change := true$;
8. **if** there exists $(R_b[V], t_b'^p[V'_p]) \in S$, **then**
9. **if** $\exists B$ such that $B \in V_b \cap V'_b$ **and** $t_b^p[B] \neq t_b'^p[B]$, **then**
10. **return** *true*;
11. **else**
12. Extend $t_b'^p[V'_p]$ to include $t_b^p[V_p]$;
13. **else**
14. $S := S \cup \{(R_b[V], t_b^p[V_p])\}$;
15. **if** $(R_2[Y], t_2'^p[Y'_p]) \in S$ **and** $Y_p \subseteq Y'_p$ **and** $t_2'^p[Y'_p] = t_2^p[Y_p]$, **then**
16. $result := true$;
17. **else**
18. $result := false$;
19. **return** $result$;

Fig. 9. Algorithm Implication

(2). Next we show, if there are no conflicts, $\Sigma \models \varphi$ if and only if there is an element $(R_2[Y], t_2'^p[Y'_p])$ in S such that $Y_p \subseteq Y'_p$ and $t_2'^p[Y_p] = t_2^p[Y_p]$.

Following the definition of CFKs and the description of algorithm Implication, it is easy to verify that if $\Sigma \models \varphi$, then there is an element $(R_2[Y], t_2'^p[Y'_p])$ in S such that $Y_p \subseteq Y'_p$ and $t_2'^p[Y_p] = t_2^p[Y_p]$. Otherwise, the algorithm can not stop.

Conversely, if there is an element $(R_2[Y], t_2'^p[Y'_p])$ in S such that $Y_p \subseteq Y'_p$ and $t_2'^p[Y'_p] = t_2^p[Y_p]$, then $\Sigma \models \varphi$. We first prove the following claim by induction.

Claim: For any element $(R_b[V], t_b^p[V_p])$ in S, $\Sigma \models (R_1[X] \subseteq R_b[V], t_1^p[X_p], t_b^p[V_p])$.

Base: When $S = \{(R_1[X], t_1^p[X_p])\}$, $\Sigma \models (R_1[X] \subseteq R_1[X], t_1^p[X_p])$. It is easy to verify the **Claim** is true.

Induction: (a) An element $(R_b[V], t_b'^p[V'_p])$ in S is extended to $(R_b[V], t_b'^p[V'_p] || t_b^p[V_p \setminus V'_p])$ (line 12).

From assumption, since $(R_a[U], t_a'^p[U'_p])$ in S, we have $\Sigma \models (R_1[X] \subseteq R_a[U], t_1^p[X_p], t_a'^p[U'_p])$. Furthermore, we have $(R_a[U] \subseteq R_b[V], t_a^p[U_p], t_b^p[V_p])$ in Σ. Therefore, we get $\Sigma \models (R_1[U] \subseteq R_b[V], t_1^p[X_p], t_b^p[V_p])$. From assumption, since $(R_b[V], t_b'^p[V'_p])$ in S, we also have $\Sigma \models (R_1[X] \subseteq R_b[V], t_1^p[X_p], t_b'^p[V'_p])$. Since V is the primary key of relation R_b, sorted in a fixed order on attributes, we have $\Sigma \models (R_1[X] \subseteq R_b[V], t_1^p[X_p], t_b'^p[V'_p] || t_b^p[V_p \setminus V'_p])$

(b) A new element $(R_b[V], t_b^p[V_p])$ is added to S (line 14). Along the same lines as (a), we can get $\Sigma \models (R_1[X] \subseteq R_b[V], t_1^p[X_p], t_b^p[V_p])$ based on the same induction.

Therefore, we show for any element $(R_b[V], t_b^p[V_p])$ in S, $\Sigma \models (R_1[X] \subseteq R_b[V], t_1^p[X_p], t_b^p[V_p])$.

Following from the **Claim**, it is easy to verify that if there are no conflicts, $\Sigma \models \varphi$ if and only if there is an element $(R_2[Y], t_2'^p[Y_p'])$ in S such that $Y_p \subseteq Y_p'$ and $t_2'^p[Y_p] = t_2^p[Y_p]$.

In case that there are conflicts (line 9), it means that there are no tuples in R_1 satisfying $t_1^p[X_p]$ of φ. It is easily to know that $\Sigma \models \varphi$.

(3). Finally, we verify algorithm Implication terminates in quadratic time. For each CFK in Σ, once it is used, it won't be considered. In each loop, at least a single CFK is used. Otherwise, $change$ is set to false, and algorithm Implication jumps out of the loop. Following this, it follows that algorithm Implication must terminate in quadratic time. □

Proof of Theorem 3.2

The satisfiability problem for CFKs is decidable in quadratic time.

Proof Sketch: This is verified by providing a quadratic-time algorithm that, given a set Σ of CFKs defined on database $\mathcal{R} = \{R_1, \ldots, R_n\}$, checks whether there exists a non-empty instance $\mathcal{D}$ of $\mathcal{R}$ such that $\mathcal{D} \models \Sigma$. The algorithm is based on an extension of the chase technique.

(1). If there exists a relation R_i, such that, for any CFK $(R_i[X], R_b[Y], t_i^p[X_p], t_b^p[Y_p])$ in Σ, $t_i^p[X_p]$ is not a nil, then Σ is satisfiable.

We can construct a database $\mathcal{D}$, where for $j \in [1, n]$ and $j \neq i$, the instance D_j for relation R_j is empty, and the instance D_i for relation R_i consists a single tuple t such that t does not satisfy the R_1 *pattern* of any CFK in Σ. It is easy to verify $\mathcal{D} \models \Sigma$. Following from this, Σ is satisfiable.

(2). Otherwise, for each relation R_i, we use algorithm Satisfiability to test the satisfiability of Σ .

The algorithm Satisfiability of testing satisfiability is a variation of algorithm Implication. First, S is initialized as $\{(R_i[X], nil)\}$, where X is the primary key of relation R, sorted in a fixed order on attributes. Second, Σ is satisfiable if there exists a relation R_i such that no conflicts are detected in the process. Otherwise, Σ is unsatisfiable, *i.e.,* for all relations, conflicts are detected in the process.

Following from these, the satisfiability problem for CFKs is decidable in quadratic time. □

Proof of Theorem 3.3

(a) Over any collection $\mathcal{D}$ of databases that satisfy CFKs $\Sigma_{(j,i)}$, the IC t^* of a tuple t can be computed in PTIME in the size of $\mathcal{D}$.

(b) Furthermore, if t consists of a set of attributes Z of relation R_1 in $\mathcal{D}$, then, for $t[Z]$ and any relations R_2 in $\mathcal{D}$, there exists at most one tuple t' in R_2 such that t^* is expanded by adding the attributes of t'.

Proof Sketch: (a). We first show that IC t^* of a tuple t can be computed in PTIME in the size of $\mathcal{D}$. Intuitively, the computation of ICs can be expressed as an inflational fixpoint query, which can be evaluated in PTIME (data complexity; see *e.g.,* [17] for fixpoint queries).

In the computation of IC t^* for tuple t, (a) once the information of a tuple t' of some relation R in $\mathcal{D}$ is added to t^*, t' does not need to be considered again; (b) To find t', it needs to search at most all tuples in $\mathcal{D}$; (c) The process repeats until no more tuples can be added to t^*. From these, it follows that IC t^* for tuple t can be computed in PTIME in the size of $\mathcal{D}$.

(b). We then show that if t consists of a set of attributes Z of relation R_1 in $\mathcal{D}$, then, for $t[Z]$ and any relations R_2 in $\mathcal{D}$, there exists at most one tuple t' in R_2 such that t^* is expanded by adding the attributes of t'.

W.l.o.g we assume that R_1 and R_2 are in databases D_j and D_i respectively. Let Σ_t be the set of CFKs, such that for any CFK $\varphi = (R_1[X], R_2[Y], t_1^p[X_p], t_2^p[Y_p])$ in Σ_t, CFK $\varphi \in \Sigma_{j,i}$, $X, X_p \subseteq Z$, and $t[X_p] = t_1^p[X_p]$. For any $\varphi_1, \varphi_2 \in \Sigma_t$, let t_1, t_2 be the tuples in R_2 that $t[Z]$ references to. It suffices to show that t_1 is equal to t_2. Since $t_1[Y] = t[X]$ and $t_2[Y] = t[X]$, $t_1[Y]$ = and $t_2[Y]$. Furthermore, Y is the primary key of R_2. From these, it follows that t_1 and t_2 are actually the same tuple. □