Intensional Associations in Dataspaces

Marcos Antonio Vaz Salles [#1], Jens Dittrich [*2], Lukas Blunschi [+3]
[#]Cornell University [*]Saarland University [+]ETH Zurich
[1]vmarcos@cs.cornell.edu [2]jens.dittrich@cs.uni-sb.de [3]lukas.blunschi@inf.ethz.ch

***Abstract*— Dataspace applications necessitate the creation of associations among data items over time. For example, once information about people is extracted from sources such as webpages and blogs, associations among them may emerge as a consequence of different criteria, such as their city of origin, their elected hobbies, or their age group. In a set of personal data sources, we may wish to associate documents and emails based on their modification dates or their authors. In this paper, we advocate a declarative approach to specifying these associations. We propose that each set of associations be defined by an *association trail*. An association trail is a query-based definition of how items are connected by intensional (i.e., virtual) association edges to other items in the dataspace. The benefit of this mechanism is the creation of an intensional graph of associations among previously disconnected data items coming from different data sources.**

We study in detail the problem of processing neighborhood queries over these intensional association graphs. The naive approach to neighborhood query processing over intensional graphs is to materialize the whole graph and then apply previous work on dataspace graph indexing to answer queries. As the intensional graph may have a number of edges quadratic in its number of nodes, the naive approach has worst-case quadratic indexing cost. We develop in this paper a novel indexing technique, the grouping-compressed index (GCI), that exploits association trail definitions to materialize the same intensional graph with linear cost. In addition, we present a query answering algorithm over GCI that avoids decompressing the graph to its quadratic size. In our experimental evaluation, GCI is shown to provide an order of magnitude gain in indexing cost over the naive approach, while remaining competitive in query processing time.

I. Introduction

Dataspace systems have been envisioned as a new architecture for data management and information integration [1]. The main goal of these systems is to model, query, and manage relationships among disparate data sources. So far, relationships in these systems have been specified at the set or schema level [2], [3]. These relationships are exploited to rewrite queries posed to the system, by reasoning on the containment of queries or on the schema relationships. In several dataspace scenarios, however, it is important to model associations between individual data items across or within data sources. Such scenarios include scientific data management [4], semantic web [5], [6], personal information management [7], social content management [8], and metadata management [9]. In scientific applications, it is necessary to link information about the same entity spread across several databases; in social content management, it is fundamental to create relationships between persons; in personal information management, it is useful to associate messages and documents received in the same context or timespan.

In this paper, we propose a declarative approach, called *association trails*, to specifying associations among items in a dataspace. An association trail is a query-based definition of how items in the dataspace are connected by virtual association edges to other items. A set of association trails defines a logical graph of associations over the dataspace. As this graph is purely logical and in principle does not need to be explicitly materialized, we call it in this paper an *intensional graph*. In addition, we term the edges of this graph *intensional edges* or *intensional associations*. In contrast to solutions that define associations extensionally [4], [5], [6], [10], association trails define associations logically and in bulk. As a consequence, association trails are especially useful in scenarios in which data sources have no explicit associations defined beforehand.

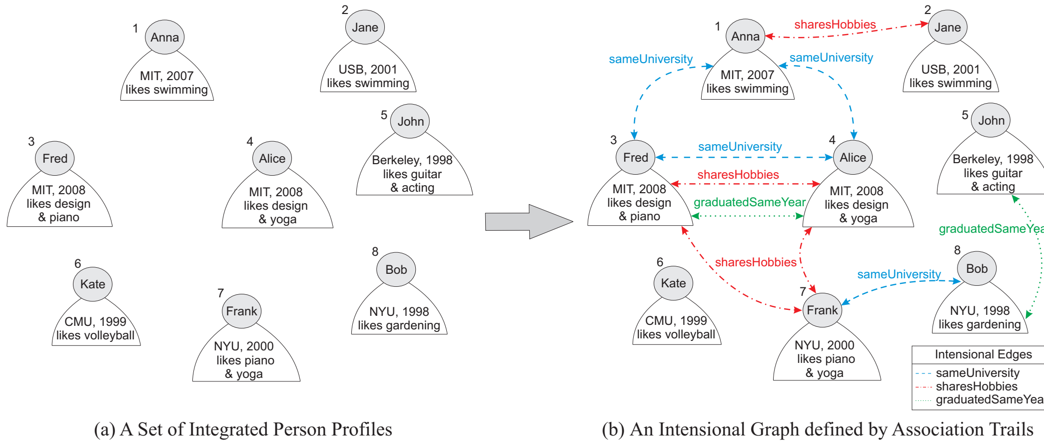
In addition to defining intensional graphs, we propose in this paper novel techniques to process exploratory queries over them. The queries we target are a generalization of the *neighborhood queries* defined by Dong and Halevy [10] to intensional graphs. A neighborhood query extends a search query's results by obtaining their immediate neighborhood in the graph. In the following, we first show example scenarios of association trails. We then discuss the challenges of neighborhood query processing over intensional graphs.

A. Examples

While our techniques are applicable to many dataspace scenarios, we will use as a running example for this paper the modeling of an implicit social network. Figure 1(a) shows a set of person profiles extracted from sources such as webpages and blogs. Each profile states a person's name, along with the university she has attended, her year of graduation, and her hobbies. Such profiles can be obtained by applying information extraction techniques [11].

Example 1 (Implicit Social Networks) *Users would like to navigate their social dataspaces to find other users related to them or to their topics of interest. Unfortunately, data extracted from loosely-connected sources is poor in associations among users.*

State of the art: Users may search their dataspaces with keyword search engines. These systems have no knowlegde about associations between data items. As a consequence, they return only items that match the user's specific request and cannot enrich results with other relevant associated information. While some dataspace approaches extend search results with elements in a dataspace graph [10], they are of little use when connections are not explicitly defined as in Figure 1(a). Users could rely, instead, on a recommender system [12].



(a) A Set of Integrated Person Profiles (b) An Intensional Graph defined by Association Trails

Fig. 1. A set of person profiles extracted from sources such as webpages and blogs is transformed by association trails into an intensional graph of associations.

These systems are, however, limited to hard-coded heuristics such as TF-IDF similarity to recommend related profiles. In addition, current recommender systems can hardly be deployed to a range of loosely-coupled data sources over which the system has no control. Moreover, these systems do not model associations among items, so users cannot browse connected users, select a subset of associations of interest, or even extend the system by defining associations based on new criteria.

Our goal: We provide a declarative technique to model intensional associations among items in the dataspace. Users or administrators provide to the system declarative association definitions, called *association trails*. Consider that the following association trails are given to the system:

1. People that went to the same university are related.
2. People that graduated on the same year are related.
3. People that share a hobby are related.

Now, users will have the view of the dataspace depicted in Figure 1(b). They are able to navigate to other related users by browsing a rich graph of associations. In addition, search query results may be enriched by items in their neighborhoods in the intensional graph. □

As an additional example, we discuss how association trails may enrich a personal dataspace.

EXAMPLE 2 (PERSONAL DATASPACES) *Consider a personal dataspace in which users interact with a set of data sources such as filesystems and email servers. Users have a hard time understanding which items spread across their sources are related to each other in the same context.*

State of the art: While users may search their data sources with search engines, the results returned by these systems are not enriched with contextual information. Users may want to access all other versions of a given file that exist in their dataspace, see files and emails worked on around the same time, or retrieve emails in the same project of a given document. Dataspace approaches, such as iTrails [2] or probabilistic mediated schemas [3], rewrite search queries to obtain results that are semantically integrated across sources. For example, these approaches allow the system to translate queries on a virtual `date` attribute to the `modified` attribute on the filesystem and the `received` attribute in the email server. These approaches, however, do not model an intensional graph of associations that would allow users to navigate to the neighborhoods of query results obtained by their rewritten queries.

Our goal: We propose modeling associations among personal items declaratively via association trails. Examples of association trails are items touched around the same time, documents with similar content, and items that reside in similar folder hierarchies in the email server and in the filesystem. With these association trails defined, users may browse an intensional graph of connections about personal items. In addition, they may find items that are most related to other items in their dataspace by examining by how many edges these items are connected in the intensional graph.

B. The Query Processing Problem

The naive approach to query processing over an intensional graph is to simply materialize the whole graph and then apply previous techniques for dataspace [10] or graph indexing [5], [6], [13]. This approach may have prohibitive cost, however. Given a set of n association trails over a dataspace with N nodes, the number of intensional edges in the intensional graph is $O(n \cdot N^2)$ in the worst case (Section IV-A). Thus, simply materializing the intensional graph has cost quadratic on the number of nodes.

In this paper, we focus on neighborhood queries over intensional graphs. We develop techniques to answer neighborhood queries at cost linear on both n and N. In particular, we propose a new indexing method, the grouping-compressed index (GCI). By eliminating redundant information from the $O(n \cdot N^2)$ naive materialization, GCI represents large classes of intensional graphs in worst-case $O(n \cdot N)$ space (Section IV-E). In addition, we present a query answering algorithm over GCI that avoids decompressing the intensional graph to its quadratic size during query processing.

C. Contributions

In summary, this paper makes the following contributions:

1. We present a new declarative formalism, termed *association trails*, to define an intensional graph of connections among instances in a dataspace. We also discuss how to express neighborhood queries on top this graph of connections

that is not given explicitly, but rather defined by association trails. Association trails are introduced in Section III.
2. We propose query processing techniques for neighborhood queries over the intensional graph defined by association trails. These techniques take advantage of different amounts of materialization of the intensional graph in order to make query processing efficient. We discuss our query processing techniques in Section IV.
3. In a set of experiments with real and synthetic datasets, we evaluate the performance of our query processing techniques over intensional graphs. Our best technique, the grouping-compressed index, exhibits an order of magnitude improvement in indexing cost over the naive approach, while remaining competitive in terms of query processing time. Experimental results are reported in Section V.

II. Preliminaries

A. Data Model

We begin by defining the data model used to represent in the dataspace the data extracted from the data sources.

Definition 1 (Data Model) *The data in the dataspace is represented by a graph $G := (N, E)$, where:*

1. *N is a set of nodes $\{N_1, \dots, N_m\}$. Each node N_i is a set of attribute-value pairs $N_i := \{(a^i_1, v^i_1), \dots, (a^i_k, v^i_k)\}$, where each value is either atomic or a bag of words. We do not enforce a schema over the nodes, i.e., the set of attributes of each node may be different.*
2. *E is a set of directed edges (N_i, N_j), s.t. $N_i, N_j \in N$.* □

Example 3 The data in Figure 1(a) is represented in the data model of Definition 1. Node 4 has a set of attribute-value pairs {(name, Alice), (university, MIT), (gradYear, 2008), (hobbies, design & yoga)} (for better visibility, attribute names have been omitted in the figure). While $E = \varnothing$ in the figure, it may contain explicit source connections in general. For example, in a personal dataspace, E will contain explicit filesystem connections between folders and files. □

B. Query Model

For the purposes of this paper, we will use the following simple keyword and path language.

Definition 2 (Query) *A query Q is an expression that selects a set $Q(G) \subseteq N$. The possible query expressions are:*

1. Keyword Expression: *denoted `K`, returns all nodes such that keyword `K` occurs in some of their attribute-value pairs.*
2. Attribute-value Expression: *denoted `A op V`, returns all nodes such that the condition on attribute `A` with operator `op` and value `V` is true.*
3. Intersect Expression: *denoted `exp1 exp2`, returns all nodes qualifying both `exp1` and `exp2`.*
4. Union Expression: *denoted `exp1 OR exp2`, returns all nodes qualifying either `exp1` or `exp2`.* □

Example 4 In the graph of Figure 1(a), the following queries exemplify the expression types described in Definition 2:

1. `yoga` returns Nodes 4 and 7.
2. `university = NYU` returns Nodes 7 and 8.
3. `yoga NYU` returns Node 7.
4. `yoga OR NYU` returns Nodes 4, 7, and 8. □

C. Basic Index Structures

Given the queries above, we assume two basic index structures, commonly found in state-of-the-art search engines [14]:

1. **Inverted Index:** a mapping from keyword to the list of node identifiers of nodes containing that keyword. In order to support both attribute-value and keyword expressions, the inverted index may be implemented by concatenating keywords with the attribute names in which those keywords occur. Keyword expressions are then translated to prefix queries [15], [10]. Keywords can be of any data type with a total order (e.g., numbers, dates, or strings).
2. **Rowstore (or repository):** a mapping from node identifier to the information associated with that node. This includes the set of attribute-value pairs of that node as well as any explicit edges that connect this node to other nodes.

Search engines use ranking schemes, e.g., PageRank, to support top-K query processing over the data structures desbribed above [14]. In our work, we are agnostic to the ranking scheme used by the search engine. We focus on efficient techniques for neighborhood query processing over intensional graphs.

III. Association Trails

This section formalizes association trails and neighborhood queries over intensional graphs.

A. Basic Form of an Association Trail

An association trail defines a set of edges in the intensional graph. For example, in Figure 1(b), a single association trail would define all sharesHobbies edges. Defining more association trails adds more edges to the graph, potentially between the same nodes. We may thus interpret each association trail as defining an *intensional graph overlay* on top of the original dataspace graph. When we take a set of association trails together, they define an *intensional multigraph*, i.e., a graph in which nodes may be connected by multiple labeled edges (Figure 1(b)). The definition below formalizes this intuition.

Definition 3 (Association Trail) *A unidirectional association trail is denoted as*

$$A := Q_L \overset{\theta(l,r)}{\Longrightarrow} Q_R,$$

where A is a label naming the association trail, Q_L, Q_R are queries, and θ is a predicate. The query results $Q_L(G)$ are associated to $Q_R(G)$ according to the predicate θ, which takes as inputs one query result from Q_L and one from Q_R. Thus, we conceptually introduce in the association graph one intensional edge, directed from left to right and labeled A, for each pair of nodes given by $Q_L \bowtie_\theta Q_R$. We require that the node on the left of the edge be different than the node on the right, i.e., no self-edges are allowed.

A bidirectional association trail is denoted as

$$A := Q_L \overset{\theta(l,r)}{\Longleftrightarrow} Q_R.$$

The latter also means that the query results $Q_R(G)$ are related to the query results $Q_L(G)$ according to θ. □

An association trail relates elements from the data sources by a *join predicate* θ. Therefore, association trails cover relational and non-relational theta-joins as special cases. While knowing the form of θ may allow us to improve performance (see Section IV-E), conceptually θ may be an arbitrarily complex function. This means that Definition 3 also models use cases such as content equivalence and similar documents.

A straightforward extension to our model is to define θ as a matching function generating several edges between a pair of nodes. This may be useful to model individual matches created by multi-valued attributes, e.g., modeling each hobby match in sharesHobbies by a separate intensional edge.

A Word about Ranking. The attentive reader will notice that it is easy to extend the definition of association trails to incorporate edge weights. Each association trail may be given a normalized weight value that represents the strength of the association edges created by that trail. These weights may then be exploited for ranking of results obtained by navigating intensional edges. While a detailed treatment of ranking exceeds the scope of this paper, our query processing techniques compute information necessary as input to ranking, such as the edge in-degree of query results. More information on association trail ranking can be found in [16].

B. Association Trail Use Cases

USE CASE 1 (SOCIAL NETWORKS) The intensional graph of Figure 1(b) is defined by the following association trails:

$$\begin{aligned}
\text{sameUniversity} &:= \texttt{class=person} \overset{\theta_1(l,r)}{\Longleftrightarrow} \texttt{class=person},\\
&\theta_1(l,r) := (l.university = r.university).\\
\text{graduatedSameYear} &:= \texttt{class=person} \overset{\theta_2(l,r)}{\Longleftrightarrow} \texttt{class=person},\\
&\theta_2(l,r) := (l.gradYear = r.gradYear).\\
\text{sharesHobbies} &:= \texttt{class=person} \overset{\theta_3(l,r)}{\Longleftrightarrow} \texttt{class=person},\\
&\theta_3(l,r) := (\exists h \in l.hobbies:\ h \in r.hobbies).
\end{aligned}$$

The association trail sameUniversity (resp. graduatedSameYear) defines that given any two persons, there will be an edge between them if they have the same value for the university (resp. gradYear) field. A more complex existential predicate is introduced in sharesHobbies, which defines that two people are related when they have at least one hobby in common. These association trail examples show how to define a logical graph of associations among elements in the dataspace. This is achieved in terms of queries that select elements to be related and predicates that specify join semantics among those elements. When taken together, the association trails above result in a multigraph (displayed in Figure 1(b)). This intensional multigraph is actually a view, which can be refined over time by adding more association trails in a pay-as-you-go fashion. □

USE CASE 2 (PERSONAL DATASPACES) Consider the following association trails defined over a personal dataspace:

$$\begin{aligned}
\text{similarTime} &:= \texttt{class=file} \overset{\theta_6(l,r)}{\Longleftrightarrow} \texttt{class=file},\\
&\theta_6(l,r) := (r.date - 1 \leq l.date \leq r.date + 1).\\
\text{relatedFolder} &:= \texttt{class=email} \overset{\theta_7(l,r)}{\Longleftrightarrow} \texttt{class=file mimeType=pdf},\\
&\theta_7(l,r) := (\exists f_1, f_2 :\ (f_1,l) \in E \wedge (f_2,r) \in E\\
&\qquad \wedge f_1.name = f_2.name).
\end{aligned}$$

These association trails model the context of personal items as discussed in Example 2. They define, respectively, associations among items changed or received around the same time and emails and files that reside in similar folders in the email server and in the filesystem. The relatedFolder association trail restricts intensional associations between files and emails to occur only for `pdf` documents and not for any arbitrary file. □

C. Neighborhood Queries

The applications described in the Introduction must process exploratory queries over the intensional graph of associations defined by association trails. We focus on a special class of exploratory queries termed *neighborhood queries*. For simplicity, our presentation in the following sections is focussed on unidirectional association trails, as it is simple to extend our techniques to the bidirectional case.

Neighborhood queries were used by Dong and Halevy to explore a dataspace graph [10]. They assume that the dataspace graph is given extensionally, i.e., each edge in the graph is explicitly materialized, and that the original queries to the graph are keyword or union expressions. Unfortunately, their definition does not apply to intensional graphs. As such, we generalize neighborhood queries below to intensional graphs and to any original query described in Definition 2. We first define what a neighborhood is in our context.

DEFINITION 4 (NEIGHBORHOOD) *Given a query Q and a set of association trails A^*, the neighborhood $N^Q_{A^*}$ of Q with respect to A^* is given by*

$$N^Q_{A^*} := \begin{cases} \emptyset, & \text{if } A^* := \emptyset \quad ,\\ (Q \cap Q^i_L) \rtimes_{\theta_i} Q^i_R, & \text{if } A^* := \{A_i\} \quad ,\\ N^Q_{\{A_1\}} \cup N^Q_{\{A_2\}} \cup \ldots \cup N^Q_{\{A_n\}}, & \text{if } A^* := \{A_1, A_2, \ldots, A_n\}. \end{cases}$$

where Q^i_L and Q^i_R are the queries on the left and right sides of trail A_i, respectively, and θ_i is the θ-predicate of A_i. □

The definition above states that the neighborhood includes all instances associated through A^* to instances returned by Q. That is formalized in terms of a semi-join, as we wish to find all instances from Q^i_R which are connected to some element of Q also appearing on Q^i_L. Note that in the definition above, nothing is said about self-edges. Self-edges are disallowed by Definition 3. It is simple to exclude self-edges by removing from Q^i_R all nodes in $Q^i_L \cap Q^i_R$ for which θ_i generates a self-edge and that do not have an edge to at least one other node in Q^i_L. For the remainder, we will not consider self-edges in order to simplify our presentation.

EXAMPLE 5 Consider the keyword query `alice`, posed over the intensional multigraph of Figure 1(b). It returns Node 4. As Node 4 is of class person, then it qualifies the left sides of all association trails in Use Case 1. If we take the semi-join of Node 4 with all other person nodes for predicate θ_1 of association trail sameUniversity, then we obtain Nodes 1 and 3. Repeating the semi-join for all other association trails in Use Case 1 and taking unions yields Nodes 1, 3, and 7, i.e., the neighborhood of Node 4 in Figure 1(b). □

DEFINITION 5 (NEIGHBORHOOD QUERY) *For a query Q and a set of association trails A^*, the neighborhood query $\overset{\leadsto}{Q}_{A^*}$ is given by*

$$\overset{\leadsto}{Q}_{A^*} := Q \cup N^Q_{A^*}$$

We call the results obtained by Q primary query results *and the results obtained by $N^Q_{A^*}$* neighborhood results. □

In contrast to recursive queries over network overlays [17], which traverse multiple hops of a single intensional graph, neighborhood queries inspect a single-hop neighborhood of multiple intensional graph overlays. As we will see below, this allows us to process them much more efficiently than by simply adapting methods from recursive query processing.

IV. QUERY PROCESSING TECHNIQUES

In this section, we present techniques to process neighborhood queries over intensional graphs. We start by discussing the naive approach in Section IV-A and then proceed to present our more specialized techniques in Sections IV-B to IV-E.

A. Naive Approach

The most intuitive query processing strategy is to explicitly materialize all edges in the intensional graph at indexing time. At query time, we can lookup this materialization to obtain the neighborhoods for each element returned by the original query Q. This query strategy is roughly equivalent to the technique of [13] adapted to neighborhood queries. Other dataspace [10] and graph indexing techniques [5], [6] could also be applied on top of the naive materialization to reduce query time. However, they will make indexing time even larger for the naive approach. As we will see in our experiments (Section V), indexing is the most dramatic cost driver for this strategy.

Given a set of association trails $A^* = \{A_1, A_2, \ldots, A_n\}$, the materialization of the intensional graph can be obtained by the join $Q^1_L \bowtie_{\theta_1} Q^1_R \cup \ldots \cup Q^n_L \bowtie_{\theta_n} Q^n_R$. This materialization could be stored in a join index [18]. In our scenario, that would entail using a single clustered B^+-tree with the left OID as a key. This representation has some unnecessary overhead. If a given OID in the left maps to several OIDs in the right, then the join index will repeat the left key several times, one time in each $\langle OID_{left}, OID_{right}, trailList \rangle$ tuple. By employing an inverted list, we factor out each occurrence of a left OID, representing it only once for all right matching OIDs. Figure 2(a) shows the resulting inverted list for the first two trails in Use Case 1.

Clearly, if the queries Q^i_L and Q^i_R each return N nodes, then the association trail A_i may generate up to $O(N^2)$ edges in the multigraph. We see an example of that effect in Figure 1(b): Nodes 1, 3, and 4 form a clique when we consider the sameUniversity association trail. We believe this situation will occur frequently in practice, given that association trails are based on N:M join semantics. As association trails generate edges in the multigraph independently, for n association trails, there will be $O(n \cdot N^2)$ edges in the multigraph in the worst case. Thus, we expect the naive approach to exhibit large indexing times. In addition, query times will grow significantly whenever we must retrieve a sizable portion the naive materialization at query time. This situation may occur when the selectivity of the original query Q is low. Both of these observations are confirmed by our experiments (see Section V).

B. Canonical Plan

Instead of rushing into full materialization, we observe that neighborhood queries can be answered without any materialization at all. In fact, we can use a query plan that follows trivially from Definitions 4 and 5. We call this query plan the *canonical plan*. We show an example in Figure 2(b).

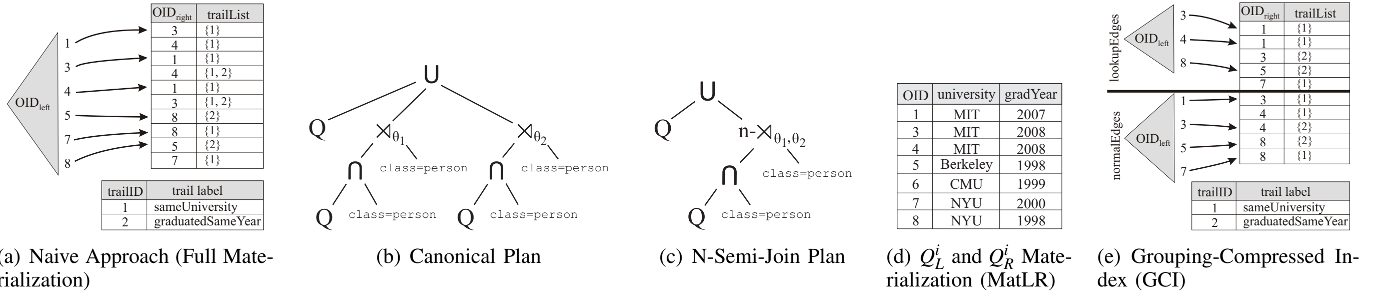
The canonical plan must compute the query Q once for each association trail. In addition, it always recomputes the queries on left and right sides of the association trails. As we have illustrated in Use Case 1, we frequently find common expressions being used for these queries. What this means is that the canonical plan will naturally contain a large number of common subexpressions. Techniques to reuse common subexpressions include spooling and push-based plans [19]. Spooling is simple to be implemented in pull-based query processors, such as search engines and database systems. On the other hand, it incurs in higher runtime overhead than push-based plans. As the investment to convert a pull-based query processor into push-based one is often equivalent to a reimplementation of the query processing stack, we would like to use pull-based plans while reducing the overhead of materializing and re-reading results from common subexpressions. We discuss how to achieve this in the next subsection.

C. N-Semi-Join Plan

Instead of converting our algebra to push-based, we introduce a new operator: the n-semi-join n-$\rtimes_{\theta_1,\ldots,\theta_n}$. Given a list of θ-predicates $\theta_1, \ldots, \theta_n$ and two inputs R and S:

$$R \text{ n-}\rtimes_{\theta_1,\ldots,\theta_n} S \;:=\; (R \rtimes_{\theta_1} S) \cup \ldots \cup (R \rtimes_{\theta_n} S).$$

N-semi-joins may improve the canonical plan by compressing all semi-joins that have the same left and right inputs into one single operator. For example, if $Q^1_L = Q^1_R = \ldots = Q^n_L = Q^n_R$, then the canonical plan is rewritten to a plan with a single n-semi-join. An example is shown in Figure 2(c). If all left and right sides are different, however, the n-semi-join plan is equivalent to the original canonical plan. In this respect, n-semi-join plans are "safe", as they will not produce a plan that is worse than the canonical plan. In practice, we expect n-semi-joins to significantly compress query plans, replacing the n semi-joins in the canonical plan by $k \ll n$ n-semi-joins. For example, the canonical plan for all association trails of Use Case 1 would have 5 semi-joins; the equivalent n-semi-join plan would have 2 n-semi-joins. If another hundred association trails were



(a) Naive Approach (Full Materialization)

(b) Canonical Plan

(c) N-Semi-Join Plan

(d) Q_L^i and Q_R^i Materialization (MatLR)

(e) Grouping-Compressed Index (GCI)

Fig. 2. Query processing alternatives for association trails sameUniversity and graduatedSameYear.

defined relating people as in the first four association trails of Use Case 1, we would still have only 2 n-semi-joins, compared to 105 semi-joins in the canonical plan.

It is straightforward to implement a hash-based version of an n-semi-join operator. A hash-based n-semi-join is able to handle all association trails presented in Use Case 1. We have adapted this operator to also handle existential semantics, as in sharesHobbies or in Experiment V-C. In fact, depending on the θ-predicate, we may have to use a different join implementation, even not only hash-based joins. We refer the reader to the significant work on join processing in the literature (e.g. [20], [21]), as discussing how to handle different types of θ-joins efficiently exceeds the scope of this paper.

D. Materialize Q_L^i and Q_R^i (MatLR)

Recall from Section IV that even the improved n-semi-join plan must execute the original user query Q as well as the left (Q_L^i) and right (Q_R^i) sides of the association trails. One important observation is that although the query Q is not directly available to us at indexing time, the association trails are. Thus, we may pre-compute Q_L^i and Q_R^i and save the results as materialized views. The materialized views for Q_L^i and Q_R^i are physically represented as lists of node identifiers (OIDs) along with a projection of the attributes from the nodes. This projection comprises all attributes referenced in the θ-predicates of the association trails. We choose those attributes to avoid costly lookups to the rowstore if Q_L^i and Q_R^i are executed by accessing the inverted index (see Section II-C). We show the materialized view for association trails sameUniversity and graduatedSameYear in Figure 2(d). The advantage of this technique is to improve at query time over pure n-semi-joins, while only incurring modest indexing cost.

E. Grouping-Compressed Index (GCI)

The quadratic growth in the size of the naive materialization of Section IV-A is often a consequence of grouping effects in the association trail join predicates. In an equi-join, for example, all nodes with the same value for a join key will join with one another, generating a clique in the graph for that association trail. In a clique of size C, we would naively store C^2 edges in the naive materialization. In this section, we show how to eliminate grouping effects in the join at both indexing and query time.

Compressing the Full Materialization with Lookup Edges. As an alternative representation, we could: (1) explicitly represent the edges from a given node n_1 in the clique to all the other nodes $\{n_2,\dots,n_C\}$ and (2) for each remaining node in the clique, represent special *lookup edges* $\langle n_j, n_1, lookup\rangle$ that state n_j connects to the same nodes as n_1. Thus, we represent the information in the clique with C normal edges for n_1 plus $C-1$ edges for the lookup edges of all remaining nodes. In short, a reduction in storage space (and consequently join indexing time) from C^2 edges to $2\cdot C-1$ edges. We call this representation *grouping compression*.

The grouping compression may be applied not only to equi-join predicates in the association trails, but also to other types of joins in which edges outgoing from one node n_l point to exactly the same nodes as the edges outgoing from another node n_j. In such situations, we may compress the edges of n_j by emitting the lookup edge $\langle n_j, n_l, lookup\rangle$. For example, in the existential predicate of association trail sharesHobbies in Use Case 1, whenever two nodes have the exact same set of hobbies, they must point to the same set of nodes. So that predicate is another example where grouping effects on the join exist and, thus, grouping compression may be used.

Figure 2(e) shows our representation for the *grouping-compressed index*. Unlike in the naive materialization, we store two lists for each node n_i: a list of all normal edges for that node and a list of all lookup edges. For example, Node 3 has a lookup list which includes the lookup edge $\langle 3, 1, lookup\rangle$ for association trail sameUniversity (trail 1 in Figure 2(e)). If we wish to know which nodes are connected to Node 3, then we should include Node 1 along with nodes in the normal list of Node 1. This means that Node 3 is connected to Node 1 and to Node 4 by that association trail (note that we ignore self-edges). In addition, the normal list of Node 3 includes Node 4 for association trail graduatedSameYear. Therefore, Node 3 is also connected to Node 4 by this other association trail.

Grouping-Compressed Neighborhood Query Algorithm. We present below a method to process neighborhood queries over the grouping-compressed index that does not require each clique to be decompressed to its original C^2 edges. The core idea of our method is to operate in two phases. In the build phase, we create an aggregate table with information from all lookup edges for the original query results. In the probe phase, we may use this table to avoid looking up normal edges for nodes in the same clique several times.

Algorithm 1 is a detailed description of this idea. The algorithm is described assuming that all trails in A^* should be taken into consideration for query processing. Filtering for

a given subset of the trails, however, is straightforward. First, we begin by processing lookup lists (Lines 1 to 8). For every node in $Q(G)$, we check the lookup list for that node (Line 4). For each element in the lookup list, we record the trails in which that element appears along with the number of times it appears per trail (Lines 5 and 6). This comprises the build phase, in which for each element in the lookup lists, we have built a small mapping containing in-degree information for each association trail (map *oidToTrailCountMap*). The next phase comprises probing the normal lists of all nodes either in $Q(G)$ or *oidToTrailCountMap* in order to expand the result (Lines 9 to 29). We first update the in-degree of each node according to its lookup edges (Line 12). For each element in the normal list of that node (Line 13), we calculate its appropriate in-degree, depending on whether the lookup was triggered by a primary query result (Lines 15 to 18) or by an element from a lookup list (Lines 19 to 22). Note that if the element appears in a lookup list, we must increase its in-degree only for association trails in the intersection of its trail list and the trails in *oidToTrailCountMap* (function intersectTrailCounts, Line 21). Finally, if the element is indeed connected through some edge, then we add it to the result, updating its in-degree (Lines 23 to 27).

EXAMPLE 6 Suppose $Q(G) := \{3,4\}$ and the grouping-compressed index of Figure 2(e) are given as input to Algorithm 1. In the build phase, the algorithm will inspect the lookup lists for Nodes 3 and 4. It will find Node 1 represented twice for sameUniversity and Node 3 represented once for graduatedSameYear. That information will be saved in the map $oidToTrailCountMap := \{(1 \rightarrow \{(1,2)\}), (3 \rightarrow \{(2,1)\}\}$. The probe phase then calculates which nodes are related to one of Nodes 1, 3, or 4 ($Q(G) \cup$ getNodes(*oidToTrailCountMap*.keySet())). We first process Node 1, adding it to the result with in-degree equal to 2 (Line 12). We then lookup the normal list for Node 1. Both entries, again Nodes 3 and 4, qualify for association trail sameUniversity and are thus added to the result with a partial in-degree of 1 (Line 21; note that we deduce self-edges). We proceed by processing Node 3. As Node 3 is in *oidToTrailCountMap* with an edge count of 1, we increase its in-degree to 2 (Line 12). We then process the normal list of Node 3, updating the in-degree of Node 4 to 2 (Line 17) as Node 3 is also in $Q(G)$. Note that the count of Node 4 is not increased again in Line 21, as we deduce self-edges. Finally, we try to lookup the normal list of Node 4, but it is inexistent. The final result contains the nodes and in-degrees $Res := \{(1,2),(3,2),(4,2)\}$, i.e., Nodes 1, 3, and 4 are all returned with in-degree 2. Contrast that result with Figure 1. If Nodes 3 and 4 are in the original query result, then Node 1 is in the neighborhood of both of them via association trail sameUniversity. Nodes 3 and 4 are in the neighborhood of one another and are connected by both association trails sameUniversity and graduatedSameYear. □

F. Handling updates

Following the same design philosophy used for traditional inverted list indexing, we may incorporate updates to the index

Algorithm 1: Grouping-Compressed Neighborhood Query

Input: Primary Query results $Q(G)$
Grouping-compressed index *GCI*
Output: Result Set $Res = \overset{\leadsto}{Q}_{A^*}$

```
1  Create Map oidToTrailCountMap of type
     OID → {(trail_1, count_1), ..., (trail_n, count_n)}
2  // build phase — build oidToTrailCountMap from lookup
   edges:
3  for Node n_i ∈ Q(G) do
4      for Entry entry_Lookup ∈ GCI.lookupList(n_i.OID) do
5          TrailCountList trailCountList :=
             oidToTrailCountMap.getOrCreate(entry_Lookup.OID)
6          trailCountList.increaseTrailCounts(entry_Lookup.trailList)
7      end
8  end
9  Res := (Q(G) ∪ getNodes(oidToTrailCountMap.keySet()))
10 // probe phase — scan normal lists to expand Res:
11 for Node n_i ∈ Res do
12     n_i.inDegree := n_i.inDegree + ∑ all counts in list
         oidToTrailCountMap.get(n_i.OID)
13     for Entry entry_Normal ∈ GCI.normalList(n_i.OID) do
14         int entryInDegree := 0
15         // count all trails when access caused by primary query
           result:
16         if n_i ∈ Q(G) then
17             entryInDegree := entry_Normal.trailList.length
18         end
19         // count extra lookup edges for entry_Normal.OID:
20         if oidToTrailCountMap.containsKey(n_i.OID) then
21             entryInDegree := entryInDegree + ∑ all counts in
               list
               intersectTrailCounts(oidToTrailCountMap.get(n_i.OID),
                 entry_Normal.trailList), deducing self-edges when
                 entry_Normal.OID ∈ Q(G).getOIDs()
22         end
23         // add entry_Normal.OID to result if there is some edge
           to it:
24         if entryInDegree > 0 then
25             Node n_Normal := Res.getOrCreate(entry_Normal.OID)
26             n_Normal.inDegree :=
               n_Normal.inDegree + entryInDegree
27         end
28     end
29 end
```

structures described above via differential indexing [22]. When association trails are added to the system, we build indexes for those association trails and combine results from all indexes at query processing. Likewise, when elements are added or removed from the dataspace, we record their neighborhoods in a differential index structure that is merged at query processing time with the main indexes. Thus, update performance will be proportional to the performance observed for small index creations (see Section V).

V. EXPERIMENTS

In this section, we evaluate the query processing techniques of Section IV. The main goal of our experiments is to understand how our techniques compare to the naive approach. Section V-B evaluates the query performance and indexing costs of each technique as we scale on the number

of association trails. Section V-C explores the sensitivity of all methods to the selectivity of the primary query Q.

A. Setup and Datasets

Datasets. We have implemented a synthetic data generator that creates a set of person profiles as in the running example of this paper. The generator creates N person nodes in the dataspace. We scale up to a maximum of n association trails. All association trails follow the template $A_i :=$ `class = person` $\overset{\theta_i(l,r)}{\Longrightarrow}$ `class = person`, $\theta_i(l,r) := l.a_k = r.a_{k+1}$, where a_k, a_{k+1} are attributes from the profiles of people in the dataset. Therefore, we create person profiles with $2n$ attributes, where each attribute is generated following a Zipf distribution with skew z and cardinality c [23]. Other than the profile attributes, each person also has a randomly generated person name. For our synthetic dataset, we have set $N =$ 1,600,000; $z = 0.5$; and $c =$ 100,000. Each generated profile contained 200 attributes.

In order to generate data similar to online social networks, our generator creates a scale-free graph [24] among person nodes and also creates nodes for comments and communities. We have generated on the order of 500,000 comments and 160 communities. This additional data is immaterial for our evaluation, however, as we focus our evaluation on the intensional graphs among person nodes created by association trails.

We have also evaluated our system with a real dataset of biographies and filmographies from IMDb [25]. Person profiles were obtained from the biographies of actors, actresses, writers, and directors, totalling 1,909,796 people. Each profile included the person's first and last names, birthdate, place of birth, and height. In addition, we imported all the explicit connections between people and the movies they worked in. There were a total of 1,414,654 movies and 14,250,548 person-movie connections. We have created the following (self-explanatory) association trails: sameLastName, sameBirthdate, samePlaceOfBirth, sameHeight, and moviesInCommon. We have applied the extension discussed in Section III-A for association trails on multi-valued attributes to moviesInCommon, i.e., this association trail generates one intensional edge for each movie in common between two persons. The association trails above create an intensional graph of person nodes from a dataset in which these connections are not explicit.

parameter	setting
number of trails n	$0 \ldots 100$
query Q selectivity s	0.01% … **0.1%** … 10%

TABLE 1

PARAMETER SETTINGS USED IN THE EXPERIMENTS

We summarize the main parameters varied in the experiments in Table 1. Default settings are highlighted.

Setup. All experiments have been run on a dual AMD Opteron 280 2.4 Ghz dual-core server, with 6 GB of RAM, and a 400 GB ATA 7200 rpm hard disk with 16 MB cache. Association trails have been implemented in the open-source iMeMex Dataspace Management System [26]. The system already provided the basic index structures described in Section II-C and supported all query types described in Section II-B. In addition, it provided reusable data structures for materialized views, bulk-loaded B^+-trees, and inverted lists. In our implementation, we have not made use of multi-threaded parallelism for querying, so our code takes explicit advantage of only a single core; for indexing, on the other hand, we have used a multi-threaded mergesort implementation, resulting in a partial usage of the cores available in the server. Our techniques could easily be extended to work on a cluster. Consider the grouping-compressed index. At indexing time, we compute association trail joins and then load the results into a sorted structure. Both of these activities are clearly parallelizable. At querying time, all lookups performed during the build phase are independent and may be done in parallel. Likewise, the lookups performed in the probe phase may also be performed in parallel.

Dataset	Orig Data Size [MB]	Total Index Size [MB]	Rowstore Size [MB]	Inv Index Size [MB]	Indexing Time [min]
Synthetic	2,768	3,138	2,056	1,082	43
IMDb	564	924	676	248	48

TABLE 2

SIZE AND CREATION TIME FOR BASIC INDEXES (SECTION II-C)

We report in Table 2 the indexing time and index sizes taken by the index structures of Section II-C for the datasets above. In order not to have our query time measurements influenced by the time necessary to process the original query Q, we have simulated Q by choosing nodes randomly from the persons in each dataset and creating a materialized view as an OID list for those nodes. Thus, processing Q is equivalent to a scan in an OID list and can be typically carried out in subsecond response times. This means that the performance differences observed in the experiments are due to association trail processing strategies alone.

For all query processing strategies, we obtain as results not only the nodes that qualify $\overset{\leadsto}{Q}_{A^*}$, but also an in-degree count that states how many edges connect each node to a primary query result. This information could be used as input for a ranking algorithm on top of our query processing methods. Note that exploring top-K algorithms in the style of [27] exceeds the scope of this paper.

Methods Compared. We compare the following strategies:

1. **Naive**: materialization of the whole intensional graph as an inverted list (Section IV-A). Note that the naive approach will compute the join of all association trail queries at indexing time. In order to optimize this computation, we have applied a technique similar to n-semi-joins to reuse common subexpressions in the joins.
2. **Canonical**: canonical plans without indexing except for the basic structures in Section II-C (Section IV-B).
3. **N-Semi-Join**: n-semi-join plans also without indexing (Section IV-C).
4. **MatLR**: materialization of the left- and right-side queries of association trails as materialized views (Section IV-D). For this strategy, note that, in our setup, the queries on both sides of the association trails are the same (`class=person`) and thus only one materialization is necessary.
5. **GCI**: the grouping-compressed index (Section IV-E).

We would like to point out that techniques such as those proposed by Dong and Halevy [10] or graph indexing techniques [28], [5], [6] expect as input a fully materialized, extensional graph. As a consequence, any of these techniques will have indexing times that equal the Naive approach.

B. Scalability in Number of Association Trails

We present below performance results when scaling on the number of association trails. All results in this section are reported using our synthetic dataset, as it allows us to scale on the number of association trails.

Query Response Time. Figure 3(a) shows that query performance degrades linearly as we scale on the number of association trails, given an original query Q selecting 0.1% of the person nodes in our dataset. The first point in the graph is measured without association trails and the second with a single association trail. Query response time without association trails is in the order of 10 ms. With the addition of the first association trail, query response time for strategies without indexing, namely Canonical and N-Semi-Join, rises sharply, reaching a time in the order of 3 min. This is due to our having to process both the association trail left- and right-side queries and the additional semi-join to calculate the association trail neighborhood. As expected, N-Semi-Join brings no benefit for a single association trail over canonical plans; in fact, it has slight overhead. As we scale on the number of trails, however, the differences between the two techniques become apparent. N-Semi-Join processes the queries in the left and right sides of the trails only once, while Canonical spools the association trail queries and scans this spool multiple times.

Overall, association trail indexing has a significant impact in query response time. The indexing strategies do not have as dramatic a jump in query processing time when the first association trail is added to the system as Canonical and N-Semi-Join. N-Semi-Join has a processing time of 2.9 min for that point, while MatLR takes around 4 sec and both Naive and GCI take under 0.5 sec. Furthermore, all indexing strategies exhibit faster query response times than both Canonical and N-Semi-Join for all numbers of association trails.

Out of the indexing strategies, only MatLR and GCI could be scaled up to 100 association trails, with GCI consistently outperforming MatLR. At 100 association trails, GCI outperformed MatLR by a factor 4.6 (37.4 sec vs. 2.9 min) and was up to a factor 18.8 better than Canonical. Naive exhibited an interesting behavior: its query response time, while also linearly increasing, remains rather low. It reaches 6.9 sec for 90 trails, in contrast to 29 sec for GCI. GCI must perform more random lookups to process neighborhood queries using Algorithm 1 than Naive at the selectivity level of 0.1% for the original query Q. We could not scale Naive above this point, however, due to large index sizes (reported below). In particular, we did not have enough temporary disk space in our server to materialize the whole intensional graph as required by Naive. This point is marked in Figure 3(a) by a vertical line. In addition, in a set of separate experiments, we have also observed that the query performance of Naive is sensitive to skew in the data, performing significantly worse than GCI when the skew of the attributes is increased to 0.95. We omit the detailed results for brevity.

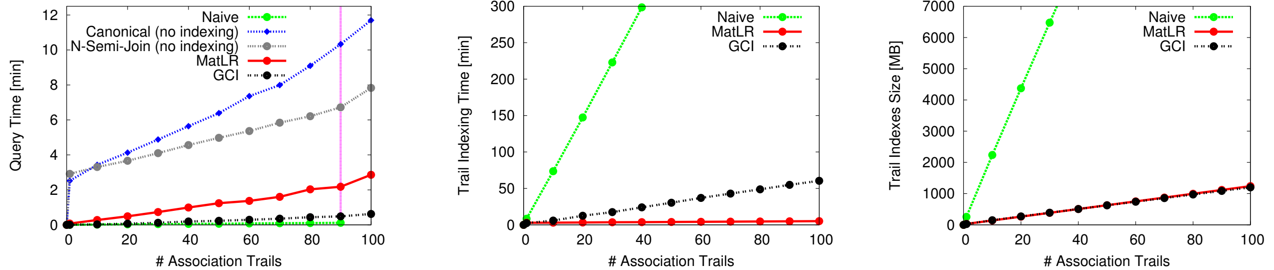
In summary, while Naive can deliver good response times, it is strongly affected by indexing time and skew in the dataset, limiting its scalability. Among the methods that could support up to 100 association trails, GCI had the best performance, with response time gains of over an order of magnitude when compared to Canonical.

Indexing Time and Index Size. Figures 3(b) and 3(c) report indexing times and index sizes for the indexing strategies discussed above when we scale in the number of association trails. The figures show that both indexing time and index size for Naive are dramatic. For 40 association trails, indexing time is already 5 hours in comparison to 24.0 min for GCI (factor 12.4) and 3.7 min for MatLR (factor 80). We have not scaled Naive above 90 trails, a point in which indexing time reached about 13.2 hours and index size was 17.8 GB. The temporary disk space needed to build the index was excessive beyond that point. The second most time-demanding strategy, GCI, required only 54.9 min to build an equivalent index. In addition, the indexing time for Naive is over an order of magnitude higher than the 43 min time needed to build the basic indexes over the dataset (Table 2, factor 18.4).

In spite of the large indexing times and index sizes obtained for Naive, all indexing strategies scaled linearly on the number of association trails. The second most time-demanding indexing strategy, GCI, took about 1 hour to index 100 association trails, while MatLR took 5.1 min (factor 11.8). Those differences in indexing times are not impressive given the comparable differences in favor of GCI in query response times. In terms of index sizes, GCI was in fact slightly more space-efficient than MatLR. Note that GCI only needs to store OIDs, while MatLR needs to store a projection containing the attributes used in the θ-predicates of the association trails. As more association trails are indexed, more attributes must be materialized by MatLR.

In summary, the additional indexing investment necessary for GCI can be easily offset by the savings it provides in terms of query response times. The trade-off is much worse for Naive, which exhibits index sizes and indexing times larger by at least an order of magnitude. While MatLR has the fastest indexing time, its non-interactive query response times greatly limit its applicability.

Number of Join Tuples. Recall from Section IV-E that Naive stores all edges in the association trail multigraph, while GCI stores normal and lookup edges. Each edge, regardless of its type, is counted in Figure 4(a) as one join tuple. The number of join tuples is an implementation-independent metric for both Naive and GCI. The number of join tuples for both strategies grows linearly on the number of association trails. That is in accordance with the number of join tuples being $O(n \cdot N^2)$ for Naive and $O(n \cdot N)$ for GCI. For comparison, we show in the figure a curve with the predicted number of edges that Naive would require if attributes were uniformly distributed. As expected, the number of join tuples created on the actual

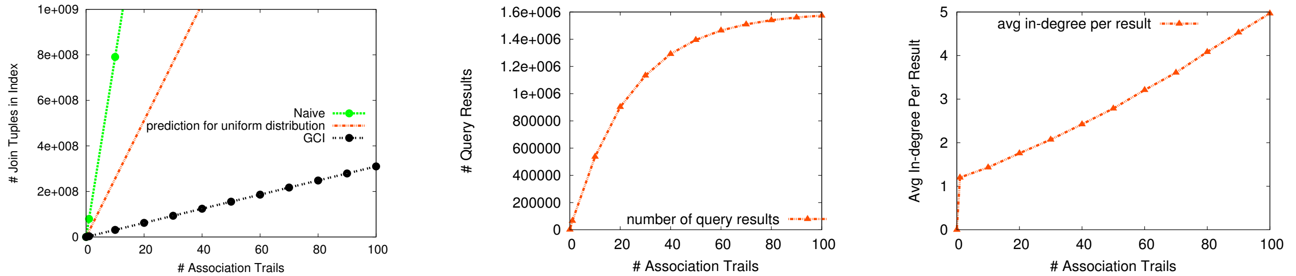


(a) Query execution time vs. number of association trails

(b) Indexing time vs. number of association trails

(c) Index size vs. number of association trails

Fig. 3. Association trails indexing: query processing and indexing performance.



(a) Number of materialized join tuples (edges) vs. number of association trails

(b) Number of query results vs. number of association trails

(c) Query-dependent in-degree vs. number of association trails

Fig. 4. Implementation-independent metrics: number of join tuples, number of query results, and average result in-degree.

Naive materialization was even higher than predicted, because the attributes in our generated dataset follow a Zipf distribution with medium skew. For 90 trails, the maximum we could scale Naive to, we had amazing 7.1 billion join tuples being created by Naive vis-à-vis 279 million for GCI.

Number of Query Results and In-degree. We show in Figure 4(b) the number of query results returned by neighborhood queries as we scale on the number of association trails. The non-linear increase observed in the figure is expected, because we have generated attribute values using a Zipf distribution with medium skew. This means that some attribute values are more likely to appear in query results and thus join with more neighbors. We observe that not only is the increase in the number of neighborhood query results sharp, but also the absolute number of results reaches almost the whole set of person nodes in our dataset for 100 association trails. This result suggests that ranking could be beneficial for neighborhood queries. We plot in Figure 4(c) the query-dependent in-degree derived from connections between primary query results and neighborhood results. This metric scales linearly with the number of association trails. Contrasting the sharp increase in number of results with the linear increase in in-degree, we can conclude that some neighborhood results will be more densely connected to the primary query results than others. As a consequence, we recommend future work on ranking over intensional graphs to consider query-dependent in-degree as a feature for ranking.

C. Sensitivity to Selectivity of Original Query

We report in this section the sensitivity of the methods to the selectivity of the original query Q. To run these experiments, we have used the IMDb dataset. First, we show the indexing performance for all methods in Table 3. The numbers we obtained confirm the results discussed for synthetic data. In short, Naive's indexing time and index sizes are, respectively, 27 and 28 times larger than GCI, a difference of over an order of magnitude. MatLR has modest indexing cost, being lower than GCI by about a factor 2. Canonical and N-Semi-Joins do not perform indexing.

Metric	Strategy				
	Naive	Canonical	N-Semi-Join	MatLR	GCI
Indexing Time (min)	194	-	-	3	7
Index Size (MB)	4769	-	-	206	170

TABLE 3

ASSOCIATION TRAIL INDEXING TIMES AND INDEX SIZES FOR IMDB.

Figure 5 shows the corresponding query performance of the methods. As expected, Canonical and N-Semi-Joins have the worst query performance overall, with query response time steadily increasing for lower selectivities. The performance of these two methods was comparable, as we have only 5 association trails defined over the dataset. At 1% selectivity, MatLR performed 9.3 times better than N-Semi-Joins, but still 3.6 times worse than GCI. It also proved to be robust to lower selectivities. The gap in performance between GCI and MatLR tended to diminish for lower selectivities, as more random lookups in GCI lead to poorer processing times. The query processing times for Naive were highly dependent on query selectivity, increasing sharply as selectivity is lowered. At 10% selectivity, Naive took 2.4 min, a factor 8.8 worse than the 16.2 sec taken by GCI. That behavior is consistent with the fact that lower selectivities imply that a proportionally larger

fraction of the fully materialized intensional graph must be processed in order to answer a neighborhood query.

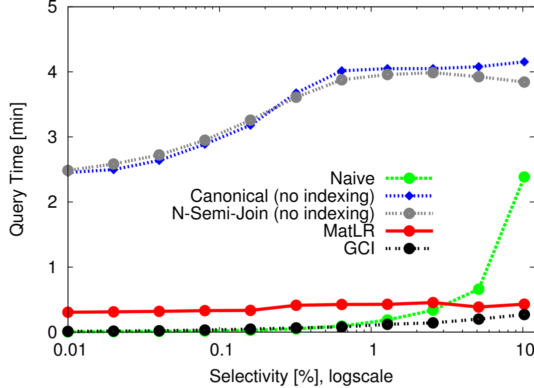


Fig. 5. Query execution time vs. selectivity of original query

VI. Related Work

Metadata and Associations. Systems such as Mondrian extend the relational model to include annotations [4]. These systems, however, represent annotations extensionally, while our approach allows users to relate any item in the dataspace (including data and annotations) declaratively. In order to model associations between relational tuples, Srivastava and Velegrakis extend the relational model by using queries as data values [9]. Their approach is similar to ours in two aspects: (i) associations are defined intensionally and (ii) no distinction is made between data and metadata. In contrast to [9], however, our approach does not require a schema to be created from the start for the data. Moreover, our approach applies not only to the relational model but to a general graph data model. Furthermore, our query interface is a simple extension of keyword search with structural hints, which allows an end-user to explore the dataspace once association trails are defined by an administrator, while [9] adopts full-blown SQL.

Chapman et al.'s approach for provenance storage relies on compressing an existing extensional database of provenance information [29]. In contrast, we propose a grouping compression scheme that can be applied without ever having to materialize the full materialization created by the naive approach. In addition, we also present an algorithm to avoid full decompression at query time. The idea of multiple hierarchies modeled by different colors in Colorful XML is similar to creating graph overlays that model different relationships on the data [30]. Our framework, in contrast, uses a general graph model for each overlay.

Graph Querying Languages. Many graph querying languages represent queries as graph patterns to be matched against the input graph [31], [32]. Given an algebra of operators on graphs, complex queries and graph patterns may be specified in a compositional manner. While querying general graph patterns is powerful, it is possible to make query processing more efficient by restricting attention to a specific query type. Our choice of neighborhood queries enables us to propose indexing techniques that greatly improve over the naive strategy of materializing the whole intensional graph. The grouping-compressed index, in particular, is capable of representing the intensional graph in linear space, even in situations where it has a quadratic number of edges.

Recursive Queries. Much work has been done on the efficient processing of recursive queries in deductive databases [33]. More recently, declarative networking has been proposed to process multi-hop, recursive queries over a single intensional graph, defined via datalog rules [17]. In contrast, we process *single-hop*, neighborhood queries over *multiple* intensional overlay graphs, defined by association trails. Applying semi-naive evaluation techniques [17] to multiple intensional overlays is equivalent to processing neighborhood queries using the canonical plan of Section IV-B. As we have shown in our experiments, it is possible to outperform this canonical strategy by at least an order of magnitude.

Indexing Graphs and XML. Extensive work has been done on processing path queries on tree-structured XML data [34], [35] and on combining paths with keyword queries [13]. Our focus is different, as we target exploratory neighborhood queries on intensional graphs and not general path queries on trees. An extension of that line of research explores indexing techniques to efficiently answer reachability queries in graphs instead of trees [28], [36]. Recent work has explored indexing support for RDF query patterns [5], [6]. This previous graph indexing work assumes that the graph is given extensionally as input to the indexing system. In contrast, our approach enables users to define and query the graph intensionally.

Some approaches study how to combine items in the database to generate meaningful results for keyword queries [37], [38]. XML/IR search engines tackle a similar problem in a different way, through score aggregation on related items that satisfy a keyword query [39], [15]. These systems inspect (implicitly or explicitly) the neighborhood of primary query results to form answer graphs (e.g. LCAs) or compute aggregated scores. In contrast to our work, all of this previous work is concerned with extensional graphs and their techniques would have to be revisited when the graph is defined intensionally.

Similarity Search. One could argue that association trails actually create a declaratively-specified metric space among the elements of a dataspace. That argument would naturally lead to the use of metric-space structures to index association trails [40]. In reality, however, applying metric-space structures to association trails implies that the metric computation itself must be made efficient. As such, the query processing techniques proposed in Section IV are necessary to avoid an inefficient metric implementation based on the naive approach.

Dataspaces. Dataspaces were envisioned by Franklin, Halevy, and Maier [1]. Indexing for exploratory queries in dataspaces has been studied by Dong and Halevy [10]. As in our approach, they also aim to process neighborhood queries; however, their approach is restricted to extensional graphs. Directly applying [10] to intensional graphs implies using the the naive approach of Section IV-A to build the input graph. Therefore, in order to create an inverted list in the style of [10]

over intensional graphs, the techniques studied in this paper are a prerequisite.

iTrails. One could argue that the set-level trails introduced by the authors in previous work [2] could be used to represent item-level association trails. In order to do that, we would have to create one set-level trail with a set of one item to another set of one item for each intensional edge in the association trail multigraph. As a consequence, we would need to define a quadratic number of set-level trails to represent an intensional graph that could be alternatively specified with a single association trail. Using the trails of [2] here would in fact be equivalent to the naive approach (full materialization of the intensional graph). Apart from the incovenience of defining such a large number of trails, our experiments demonstrate that this approach is an order of magnitude less efficient than the grouping-compressed index.

VII. Conclusions

In this paper, we have presented *association trails*, a declarative technique to define a logical, intensional graph of associations among instances in a dataspace. Our technique is general and may be applied to model such intensional graphs on a variety of scenarios, such as social networks and personal dataspaces. We have shown how to process exploratory neighborhood queries on top of the intensional graph defined by association trails. Our query processing techniques combine partial materialization of the intensional graph with specialized query processing algorithms in order to avoid the naive approach of completely materializing the intensional graph.

Our evaluation showed that the grouping-compressed index (GCI), our best query processing technique, scales well when the number of association trails is increased and when the selectivity is varied. In addition, GCI may bring over an order of magnitude gain in query response time when compared to the canonical plan without indexing. At the same time, GCI may provide over an order of magnitude gain in indexing time when compared to the naive approach.

As future work, we plan to adapt our techniques to better support top-K query processing in the style of [27]. In that vein, we would like to compare different ranking schemes when computing top-K answers over the association trail multigraph. In addition, we would like to evaluate how our techniques perform over large real social networks.

References

[1] M. Franklin, A. Halevy, and D. Maier, "From Databases to Dataspaces: A New Abstraction for Information Management," *SIGMOD Record*, vol. 34, no. 4, pp. 27–33, 2005.

[2] M. A. V. Salles *et al.*, "iTrails: Pay-as-you-go Information Integration in Dataspaces," in *VLDB*, 2007.

[3] A. D. Sarma, X. Dong, and A. Halevy, "Bootstrapping Pay-As-You-Go Data Integration Systems," in *ACM SIGMOD*, 2008.

[4] F. Geerts, A. Kementsietsidis, and D. Milano, "MONDRIAN: Annotating and Querying Databases through Colors and Blocks," in *ICDE*, 2006.

[5] T. Neumann and G. Weikum, "RDF-3X: a RISC-style Engine for RDF," *JDMR (formerly Proc. VLDB)*, vol. 1, 2008.

[6] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: Sextuple Indexing for Semantic Web Data Management," *JDMR (formerly Proc. VLDB)*, vol. 1, 2008.

[7] J.-P. Dittrich and M. A. V. Salles, "iDM: A Unified and Versatile Data Model for Personal Dataspace Management," in *VLDB*, 2006.

[8] S. Amer-Yahia, L. V. S. Lakshmanan, and C. Yu, "SocialScope: Enabling Information Discovery on Social Content Sites," in *CIDR*, 2009.

[9] D. Srivastava and Y. Velegrakis, "Intensional Associations Between Data and Metadata," in *ACM SIGMOD*, 2007.

[10] X. Dong and A. Halevy, "Indexing Dataspaces," in *ACM SIGMOD*, 2007.

[11] R. Grishman, "Information Extraction: Techniques and Challenges," in *SCIE*, 1997.

[12] G. Adomavicius and A. Tuzhilin, "Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 6, pp. 734–749, 2005.

[13] R. Kaushik *et al.*, "On the Integration of Structure Indexes and Inverted Lists," in *ACM SIGMOD*, 2004.

[14] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.

[15] D. Carmel *et al.*, "Searching XML Documents via XML Fragments," in *ACM SIGIR*, 2003.

[16] M. V. Salles, "Pay-as-you-go Information Integration in Personal and Social Dataspaces," Ph.D. dissertation, ETH Zurich, 2008.

[17] B. T. Loo *et al.*, "Declarative Networking: Language, Execution and Optimization," in *ACM SIGMOD*, 2006.

[18] P. Valduriez, "Join Indices," *ACM Transactions on Database Systems (TODS)*, vol. 12, no. 2, pp. 218–246, 1987.

[19] T. Neumann, "Efficient Generation and Execution of DAG-Structured Query Graphs," Ph.D. dissertation, University of Mannheim, 2005.

[20] P. Mishra and M. H. Eich, "Join Processing in Relational Databases," *ACM Computing Surveys*, vol. 24, no. 1, pp. 63–113, 1992.

[21] E. H. Jacox and H. Samet, "Spatial Join Techniques," *ACM Transactions on Database Systems (TODS)*, vol. 32, no. 1, 2007.

[22] D. Severance and G. Lohman, "Differential Files: Their Application to the Maintenance of Large Databases," *ACM Transactions on Database Systems (TODS)*, vol. 1, no. 3, 1976.

[23] J. Gray *et al.*, "Quickly Generating Billion-Record Synthetic Databases," in *ACM SIGMOD*, 1994.

[24] A.-L. Barabási and R. Albert, "Emergence of Scaling in Random Networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[25] "IMDb. `http://www.imdb.com/`."

[26] iMeMex project web-site. `http://www.imemex.org`.

[27] R. Fagin, "Combining Fuzzy Information: an Overview," *SIGMOD Record*, vol. 31, no. 2, pp. 109–118, 2002.

[28] S. Trissl and U. Leser, "Fast and Practical Indexing and Querying of Very Large Graphs," in *ACM SIGMOD*, 2007.

[29] A. Chapman, H. V. Jagadish, and P. Ramanan, "Efficient Provenance Storage," in *ACM SIGMOD*, 2008.

[30] H. V. Jagadish *et al.*, "Colorful XML: One Hierarchy Isn't Enough," in *ACM SIGMOD*, 2004.

[31] H. He and A. Singh, "Graphs-at-a-Time: Query Language and Access Methods for Graph Databases," in *ACM SIGMOD*, 2008.

[32] M. Fernández, D. Florescu, A. Y. Levy, and D. Suciu, "Declarative Specification of Web Sites with Strudel," *VLDB Journal*, vol. 9, no. 1, pp. 38–55, 2000.

[33] F. Bancilhon and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies," in *ACM SIGMOD*, 1986.

[34] N. Bruno, N. Koudas, and D. Srivastava, "Holistic Twig Joins: Optimal XML Pattern Matching," in *ACM SIGMOD*, 2002.

[35] T. Milo and D. Suciu, "Index Structures for Path Expressions," in *ICDT*, 1999.

[36] H. Wang *et al.*, "Dual Labeling: Answering Graph Reachability Queries in Constant Time," in *ICDE*, 2006.

[37] C. Yu and H. V. Jagadish, "Querying Complex Structured Databases," in *VLDB*, 2007.

[38] S. Agrawal, S. Chaudhuri, and G. Das, "DBXplorer: A System for Keyword-Based Search over Relational Databases," in *ICDE*, 2002.

[39] S. Amer-Yahia and M. Lalmas, "XML Search: Languages, INEX and Scoring," *SIGMOD Record*, vol. 36, no. 7, pp. 16–23, 2006.

[40] E. Chávez *et al.*, "Searching in Metric Spaces," *ACM Computing Surveys*, vol. 33, no. 3, pp. 273–321, 2001.