Scalable Multi-Query Optimization for Exploratory Queries over Federated Scientific Databases

Anastasios Kementsietsidis[1] Frank Neven[2] Dieter Van de Craen[2] Stijn Vansummeren[2,*]

[1] IBM T.J. Watson Research Center
New York, USA
akement@us.ibm.com

[2] Hasselt University and Transnational University of Limburg
Belgium
{firstname.lastname}@uhasselt.be

ABSTRACT

The diversity and large volumes of data processed in the Natural Sciences today has led to a proliferation of highly-specialized and autonomous scientific databases with inherent and often intricate relationships. As a user-friendly method for querying this complex, ever-expanding network of sources for correlations, we propose *exploratory queries*. Exploratory queries are loosely-structured, hence requiring only minimal user knowledge of the source network. Evaluating an exploratory query usually involves the evaluation of *many* distributed queries. As the number of such distributed queries can quickly become large, we attack the optimization problem for exploratory queries by proposing several multi-query optimization algorithms that compute a global evaluation plan while minimizing the total communication cost, a key bottleneck in distributed settings. The proposed algorithms are necessarily heuristics, as computing an *optimal* global evaluation plan is shown to be NP-hard. Finally, we present an implementation of our algorithms, along with experiments that illustrate their potential not only for the optimization of exploratory queries, but also for the multi-query optimization of large batches of standard queries.

1. INTRODUCTION

Motivation. The diversity and large volumes of data processed in the Natural Sciences today has led to a proliferation of highly-specialized *scientific databases*. Notable examples from biology include Genbank for genes; SwissProt for proteins; Go for functional descriptions of proteins (among other things); Enzyme for enzymes; OMIM for genetic diseases; and PubMed for publications [1]. Despite being highly-specialized, autonomous, and independently evolving, these sources are inherently *related*: genes produce proteins, proteins affect diseases, and so on. The added scientific value of the sources for conducting research then lies in the ability to query their relationships for interesting correlations (e.g. "What gene produces proteins missing in diabetes patients?"). Currently, such queries are mainly supported through an *ftp-grep* approach [13] in which users are forced to download archives of all sources and to query locally. This approach is defective in two ways:

First, as pointed out by Gray and Szalay [13], an *information avalanche* is causing the sources to grow exponentially, leading to databases that take weeks to months to download. For instance, Genbank roughly doubles in size every two years, and is expected to hit the terabyte boundary within three years [25, section 2.2.8]. It is known that this deficiency can be overcome by adopting a distributed, *federated* architecture where sources cooperate to answer queries and where downloading is hence no longer necessary [13].

Second, users must be knowledgeable of *all* sources that potentially contribute to the desired answer of a correlation query. To illustrate, consider a biologist looking for Genbank genes that are linked to SwissProt 'Fly' proteins with a Go molecular function 'f' and Enzyme description 'd'. Available to the biologist are the relational tables from the sources and *mapping tables* that provide the semantic glue to relate them [10, 18]. In its most basic version as used in this article, a mapping table is nothing more than a n-to-n binary relation over the keys of the participating source tables. Figures 1(d)–1(f) show sample mapping tables for the simplified sample instances in Figure 1(a)–1(c) of some of the biological sources mentioned earlier. There, for example, Genbank gene 4763 is directly connected to SwissProt protein O00662, while Genbank gene 768272 is indirectly connected to SwissProt protein Q9Z167 through the link with PubMed article 18022237. The mapping tables are usually stored along with the regular tables at the corresponding sources and are widely used in biology[1]. For instance, Figure 1(g) shows a graph in which there is an edge between two sources whenever we found actual mapping tables between them. Using the mapping tables, the query above can be expressed by means of the select-project-join expression

$$\pi_{gid}\big(\text{Genbank} \bowtie M_{\text{Gen,Swiss}} \bowtie \sigma_{species\,=\text{fly}}(\text{SwissProt}) \bowtie M_{\text{Swiss,Go}} \bowtie \sigma_{func=f}(\text{Go}) \bowtie M_{\text{Swiss,Enz}} \bowtie \sigma_{desc=d}(\text{Enzyme})\big), \quad (Q_1)$$

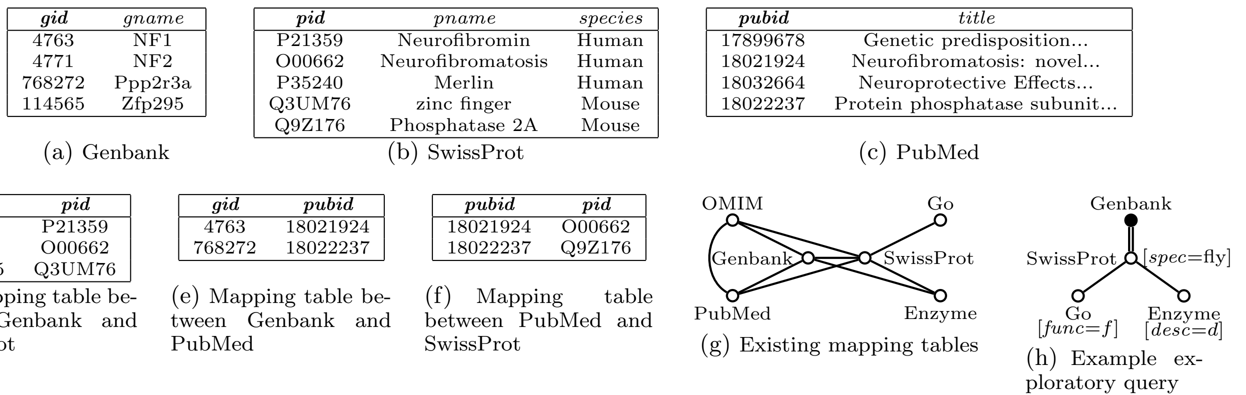
where $M_{\text{Gen,Swiss}}$, $M_{\text{Swiss,Go}}$, and $M_{\text{Swiss,Enz}}$ denote the mapping tables between Genbank and SwissProt; SwissProt and

*Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

[1] Although mapping tables may seem difficult to maintain at first sight as they need to be created manually by domain specialists, recent work has illustrated how mapping tables can be managed semi-automatically [18].

Permission to make digital or hard copies of portions of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright for components of this work owned by others than VLDB Endowment must be honored.
Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists requires prior specific permission and/or a fee. Request permission to republish from: Publications Dept., ACM, Inc. Fax +1 (212) 869-0481 or permissions@acm.org.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-305-1/08/08



gid	*gname*
4763	NF1
4771	NF2
768272	Ppp2r3a
114565	Zfp295

(a) Genbank

pid	*pname*	*species*
P21359	Neurofibromin	Human
O00662	Neurofibromatosis	Human
P35240	Merlin	Human
Q3UM76	zinc finger	Mouse
Q9Z176	Phosphatase 2A	Mouse

(b) SwissProt

pubid	*title*
17899678	Genetic predisposition...
18021924	Neurofibromatosis: novel...
18032664	Neuroprotective Effects...
18022237	Protein phosphatase subunit...

(c) PubMed

gid	*pid*
4763	P21359
4763	O00662
114565	Q3UM76

(d) Mapping table between Genbank and SwissProt

gid	*pubid*
4763	18021924
768272	18022237

(e) Mapping table between Genbank and PubMed

pubid	*pid*
18021924	O00662
18022237	Q9Z176

(f) Mapping table between PubMed and SwissProt

(g) Existing mapping tables

(h) Example exploratory query

Figure 1: Sample instances, mapping tables, and an exploratory query.

Go; and SwissProt and Enzyme, respectively. This only returns genes that are *directly* linked to SwissProt fly proteins with Go function f, however. For genes that are *indirectly* linked, for example through some common PubMed article, separate expressions like

$$\pi_{gid}(\text{Genbank} \bowtie M_{\text{Gen,Pub}} \bowtie \text{PubMed} \bowtie M_{\text{Pub,Swiss}} \bowtie \sigma_{species=\text{fly}}(\text{SwissProt}) \bowtie M_{\text{Swiss,Go}} \bowtie \sigma_{func=f}(\text{Go}) \bowtie M_{\text{Swiss,Enz}} \bowtie \sigma_{desc=d}(\text{Enzyme})), \quad (Q_2)$$

are necessary since $M_{\text{Gen,Swiss}}$ need not contain all information derivable from $M_{\text{Gen,Pub}}$ and $M_{\text{Pub,Swiss}}$. This is clear from Figure 1, for example where gene 768272 is not directly linked to protein Q9Z167 in $M_{\text{Gen,Swiss}}$ although it is indirectly linked through PubMed article 18022237 in $M_{\text{Gen,Pub}}$ and $M_{\text{Pub,Swiss}}$. Enumerating *all* expressions like (Q_2) is impractical, however, as (1) the number of paths between Genbank and SwissProt may be too large to enumerate manually and (2) the biologist must know the whole network of sources and mapping tables to express her query, a rather ominous requirement in the Internet era where new sources are easily added. For instance, at the beginning of 2008 there were 1078 major molecular biology databases, 110 more than in the beginning of 2007 [12].

Exploratory queries. While the information avalanche problem disappears in the federated architecture setting, the need for complete user knowledge persists. We therefore propose *exploratory queries*. An exploratory query takes the form of a tree with nodes representing sources and two kinds of edges: *direct edges* (depicted as single lines) indicating that the sources must be directly linked through some mapping table, and *path edges* (depicted as double lines) indicating that the sources must be linked through a *path* of alternating sources and mapping tables. To illustrate, Figure 1(h) shows the exploratory query corresponding to our biologist's original intent. Due to the declarative nature of exploratory queries, users need only specify local constraints on the sources of interest, and an intent of how these sources should be linked. They need not be aware of other sources, nor of the mapping tables needed to link them.

Exploratory query optimization. Of course, actually evaluating an exploratory query requires the evaluation of many concrete queries. To evaluate the query in Figure 1(h), for example, we need to evaluate (Q_1) and (Q_2), among others. Sequentially evaluating these queries clearly leads to redundant computation and communication (e.g., subquery $\sigma_{species=\text{fly}}(\text{SwissProt})$ is executed multiple times). Since the ratio of communication time to I/O time in a wide area network (WAN) can be of the order of 20:1 [27], it is especially important to minimize the *amount of communicated data* and to share as much as possible the communication of common subresults. *It is this multi-query optimization problem that forms the focus of this article.*

In particular, we show that the problem of finding an exploratory query evaluation plan that maximizes sharing is NP-hard. In response, we optimize an exploratory query E in two steps. In the first step, we generate the concrete queries $Q_1, \ldots, Q_k$ necessary to evaluate E, and determine optimal evaluation plans for each Q_i individually. In the second step we use heuristics to combine these individual plans in a single combined plan. (We discuss each step in more detail in the following paragraphs.) We emphasize that while exploratory query optimization is inherently a multi-query optimization (MQO) problem, existing MQO techniques are unsuitable in our context. Indeed, as further detailed in the Related Work section, they scale only to a small (i.e., 10–15) number of queries whereas in large source networks a single exploratory query may yield hundreds of concrete queries. Moreover, the existing techniques mainly focus on traditional (disk I/O based) optimization instead of communication-based optimization. Hence, while earlier research has shown that combining optimal single-query plans often yields suboptimal multi-query plans with regard to disk I/O cost [31], our experiments show that single-plan combination is not only successful with regard to communication cost, but also scales to thousands of queries.

Single plan generation. An evaluation plan for a single concrete query specifies the direction and order in which sources communicate. Figure 2 for example, shows two evaluation plans P_1 and P_2 for (Q_1), where edge labels indicate the order of communication. So, in P_1, Go and Enzyme compute in parallel the *goid*s with function f and the *eid*s with description d and send them to SwissProt. SwissProt joins the received ids with $M_{\text{Swiss,Go}}$ and $M_{\text{Swiss,Enz}}$ respectively to determine which fly *pid*s it should send to Genbank. From these *pid*s and $M_{\text{Gen,Swiss}}$, Genbank derives the *gid*s to output. In P_2 on the other hand, the order of communication is different. Enzyme first sends to SwissProt the *eid*s with description d. SwissProt joins these with $M_{\text{Swiss,Enz}}$, selects the corresponding fly *pid*s from the result, caches them temporarily, and sends them to Go. Go joins the received *pid*s with $M_{\text{Swiss,Go}}$ and returns to SwissProt the matching *goid*s with function f. SwissProt joins the received *goid*s

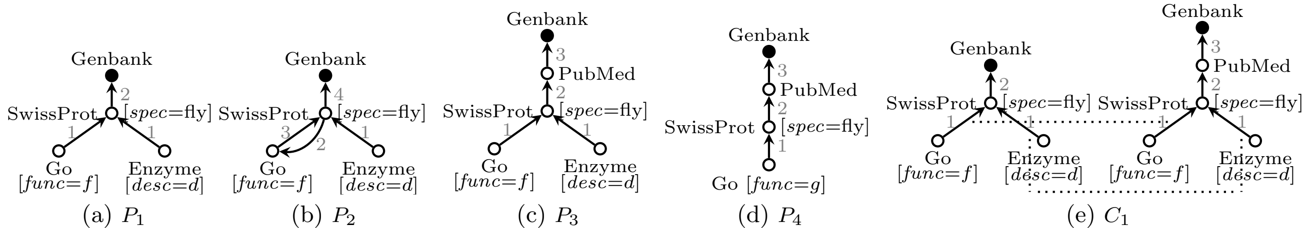


Figure 2: Evaluation plans and combined evaluation plans.

with $M_{\text{Swiss,Go}}$, selects the corresponding fly *pid*s from the result, and intersects them with the *pid*s cached earlier in order to send the correct result to Genbank. Note that the communication cost of P_2 is lower than the cost of P_1 if the number of ids sent from SwissProt to Go and from Go back to SwissProt in P_2 is less than the number of ids sent from Go to SwissProt in P_1. Hence, as in classical query processing, the order in which subresults are joined can have a huge impact on the overall (communication) cost.

As this example illustrates, computing the optimal plan for a single concrete query is far from easy. In fact, for arbitrary cost estimation functions, the problem is known to be NP-hard [36]. We show that optimal plans for a single concrete query can be computed in time quadratic in the size of the query using a cost estimation function inspired by Stocker et al. [34]. This function is very suited for optimization in loosely collaborating environments, as it does not require detailed statistics to be globally available.

Combined plan generation. Intuitively, a *combined evaluation plan* is an evaluation plan that consists of multiple ordinary evaluation plans. These plans execute in parallel, but share communication as indicated by the combined plan. To illustrate, consider plan P_1 for (Q_1) and plan P_3 for (Q_2) as shown in Figure 2. When executed *independently*, the intermediate results $\sigma_{func=f}$(Go) and $\sigma_{desc=d}$(Enzyme) would be transmitted twice. In the combined plan C_1 in Figure 2(e), in contrast, these results are only transmitted once. (The dotted lines indicate for which edges of P_1 and P_3 communication should be shared.) Combined evaluation plans can also take into account that intermediate results do not coincide completely but have a high overlap. To illustrate, consider plans P_1 and P_4 in Figure 2. Although Go is filtered by $\sigma_{func=f}$ in P_1 and by $\sigma_{func=g}$ in P_4, a combined plan containing P_1 and P_4 can still indicate that the communication of these results to SwissProt should be shared. In that case the corresponding ids are transmitted to SwissProt in such a way that shared ids are transmitted only once. Similarly, SwissProt in P_1 can share communication to Genbank with SwissProt in P_4. This is especially effective when the results have significant (expected) overlap.

As these examples indicate, computing a combined optimal plan is even more challenging than the single-query case. Indeed, we *prove* that the problem is NP-hard, even for the cost estimation function mentioned above for which computation of a single optimal plan is in quadratic time. In response, we propose two efficient heuristics that combine individual plans by looking for structural and semantic similarities. Our first heuristic, LEVELWISE MERGING, operates by looking for local similarities. Our second heuristic, ALIGNMENT, is based on a heuristic for solving the multiple partial order alignment problem in Computational Biology [17] and intuitively looks for global similarities.

Multi-exploratory query optimization. Finally, we investigate how effective the proposed heuristics are for the optimization of *multiple exploratory queries.* Such optimization is necessary, for instance, to support continuous exploratory queries that are registered once and that need to be re-evaluated when the sources are updated. In this context, we show that a two-stage approach where (1) a combined plan is computed for each exploratory query and where (2) the resulting plans are later combined in a single super-plan, executes faster than the approach where the super-plan is computed directly, while still yielding super-plans of the same quality.

To summarize our contributions:

1. We introduce the notion of Exploratory Queries (Section 2). We provide an optimization algorithm for single concrete queries that returns optimal plans in quadratic time (Section 3). In strong contrast, we prove the optimization of single exploratory queries to be NP-hard (Section 4).

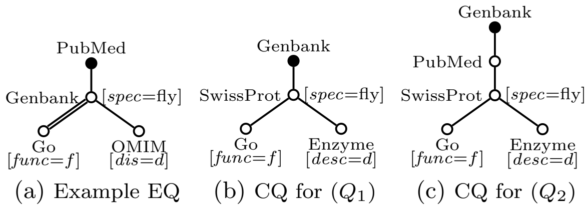
2. In response, we propose two efficient heuristics for exploratory query optimization (Section 4): LEVELWISE MERGING and ALIGNMENT, and derive from these heuristics several two-stage heuristics for the optimization of *multiple* exploratory queries (Section 5).

3. We assess the effectiveness of the proposed heuristics by extending the BioScout-platform [19] (a distributed monitoring system for biological data) with exploratory queries.

Our experiments based on this platform (in Section 6) show that ALIGNMENT is best suited for the optimization of single exploratory queries as it takes the best advantage of the structural similarity of the corresponding concrete queries. The method is less suited for optimization of multiple exploratory queries as in that case the time needed to construct the combined plans becomes a bottleneck, and LEVELWISE MERGING yields better plans. For multiple exploratory queries, the two-stage heuristic based on LEVELWISE MERGING is shown to compute combined plans faster than the one-stage LEVELWISE MERGING, while still yielding plans of the same quality. Finally, we also show LEVELWISE MERGING and ALIGNMENT to be suitable methods for multi-query optimization of sets of *arbitrary* queries, as opposed to sets of queries that implement the same exploratory query and that hence have significantly shared structure. To the best of our knowledge, this makes them the first algorithms for MQO that scale to *hundreds of queries.*

2. PRELIMINARIES

Sources. We consider a fixed, finite set of sources $\mathcal{S}$, together with a set of mapping tables [18] that provide the



(a) Example EQ (b) CQ for (Q_1) (c) CQ for (Q_2)

Figure 3: An exploratory query and two concrete queries.

semantic glue to relate them. To simplify notation, we assume each source to consist of a single relation whose key in turn consists of a single attribute (the tuple id attribute). In this way, a mapping table between sources R and S is nothing more than a binary n-to-n relation between the keys of R and S, as illustrated in Fig. 1. Let the *Data Interconnection Graph* $\text{DIG}(\mathcal{S})$ of $\mathcal{S}$ be the undirected graph over the sources in $\mathcal{S}$ in which edges indicate the presence of a mapping table between the corresponding sources. An example is given in Fig. 1(g). If there is a mapping table between R and S, then we denote this table by $M_{R,S}$. Mapping tables are assumed to be symmetric, and hence $M_{S,R} = M_{R,S}$.

Exploratory queries. Exploratory queries provide a declarative way for stating correlation queries over $\mathcal{S}$ without complete user knowledge of $\text{DIG}(\mathcal{S})$. Formally, an *exploratory query* (EQ) E is a tree with two types of edges, *direct edges* and *path edges*, whose nodes are labeled by *atoms*. An atom A is a pair $R[F]$ consisting of a source name $R \in \mathcal{S}$ and a *filter* F (like "*func=f*") specifying the ids to be selected from R. For each direct edge in E there must be an edge between the corresponding sources in $\text{DIG}(\mathcal{S})$.

Fig. 1(h) and 3(a) show example EQs with direct edges depicted as single lines and path edges depicted as double lines. As mentioned, Fig. 1(h) selects those Genbank genes that are linked *through some path in* $\text{DIG}(\mathcal{S})$ to SwissProt 'Fly' proteins with a Go molecular function 'f' and Enzyme description 'd'. Fig. 3(a), on the other hand, selects those PubMed articles that are directly linked to a Genbank record that is (1) directly linked to OMIM disease d and (2) linked *through some path in* $\text{DIG}(\mathcal{S})$ to Go function f.

Concrete queries. The semantics of EQs is formally defined in terms of the concrete queries that implement them. For Fig. 1(h), these include (Q_1) and (Q_2) from the Introduction. We find it convenient to also represent concrete queries as trees. Formally, a *concrete query* (CQ) Q is a tree whose nodes are labeled by atoms such that for each edge in the tree there is an edge between the corresponding sources in $\text{DIG}(\mathcal{S})$. In other words, a concrete query is an exploratory query in which no path edges occur. To illustrate, Figures 3(b) and 3(c) show the concrete queries corresponding to (Q_1) and (Q_2) from the Introduction.

Semantically, a concrete query Q returns a set of tuple ids. In particular, it returns the set $res(r)$ with r the root node of Q and $res(\cdot)$ defined as follows. Let w be a node in Q labeled by atom $A = R[F]$. Then the set of ids returned by w is inductively given by

$$res(w) := \pi_{\text{id}_A}(A) \cap \bigcap_{v \in \text{children}(w)} \pi_{\text{id}_A}(M_{w,v} \bowtie res(v)),$$

where we abuse notation and simply write A for the set of tuples returned by applying filter F to R; where id_A denotes the key attribute of R; and where $M_{w,v}$ denotes the mapping table between the sources corresponding to nodes w and v. (This mapping table always exists since each edge in M is required to have a corresponding edge in $\text{DIG}(\mathcal{S})$.) Observe in particular that $res(w) = \pi_{\text{id}_A}(A)$ when w is a leaf node.

We should note that, although for ease of exposition we only consider atoms with equality (e.g., *func=f*) and inequality filters (e.g., $val > 10$) in this article, a filter can be anything that is locally supported by the corresponding source. In particular, it can be an SQL statement involving aggregation such as `select id from R where` *func* `=` *f* `group by` *species* `having count(*) > 5`.

Exploratory query semantics. A concrete query Q is said to *implement* an exploratory query E if it can be obtained from E by replacing every path edge in E with a simple path allowed by $\text{DIG}(\mathcal{S})$. (Recall that a simple path in a graph is a path without repeated nodes.) For example, Fig. 4 shows all possible implementations of the EQ in Fig. 3(a). The semantics of an exploratory query is then defined in terms of its implementations, i.e.,

$$res(E) := res(Q_1) \cup \cdots \cup res(Q_k),$$

where $Q_1, \ldots, Q_k$ are all possible implementations of E and where $res(Q_i)$ abbreviates $res(\text{root}(Q_i))$.

Observe that each concrete query is essentially a select-project-join expression (possibly with arbitrary atoms in the selection conditions). The exploratory queries are therefore essentially select-project-join-union expressions. Since both concrete queries and exploratory queries are tree-shaped, however, they cannot express *cyclic joins* [2], and therefore necessarily form a *strict* subclass of the select-project-join(-union) queries. A brief discussion of how our results can be extended to deal with cyclic joins may be found in the full version of this article.

3. OPTIMIZING CONCRETE QUERIES

Since concrete queries form a particular subclass of the exploratory queries (the ones without path edges), it is instructive to discuss their evaluation and optimization before turning our attention to full exploratory queries. Indeed, concrete query optimization will form a key component of exploratory query optimization, as discussed in Section 4.

Evaluation plans. To evaluate a CQ, sources must evaluate atoms; transmit ids to neighboring sources; and translate incoming ids by means of mapping tables. The order and direction in which ids are transmitted is recorded in an *evaluation plan* (or simply plan for short), of which two examples are shown in Figures 2(a) and 2(b). Formally, a *plan on* a CQ Q is a directed, connected graph P whose nodes are labeled by atoms and whose edges are strictly ordered. This strict partial order $\prec_P$ is required to be total on adjacent edges, i.e., for all edges $u \to v$ and $v \to w$ either $u \to v \prec_P v \to w$ or $v \to w \prec_P u \to v$ (but not both). As with exploratory and concrete queries, there must be an edge between the corresponding sources in $\text{DIG}(\mathcal{S})$ for each edge in P. Finally, a single node $\text{root}(P)$ is distinguished as the *root of* P. Intuitively, $\prec_P$ specifies the order of transmission while $\text{root}(P)$ specifies the node whose ids are to be output.

A plan is evaluated as follows. Let $src(v)$ stand for the source corresponding to node v. During evaluation P computes, for every node v, a set of tuple ids $res(v)$. This set

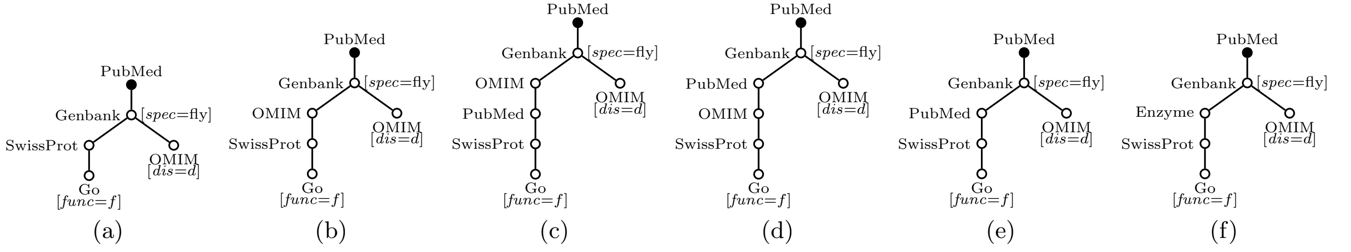


Figure 4: Concrete queries for the exploratory query in Fig. 3(a)

is stored at *src*(v), and initially consists of the ids returned by the atom A at v, i.e., initially $res(v) := \pi_{\text{id}_A}(A)$. When $v \to w$ is an edge without predecessor in $\prec_P$, *src*(v) sends $src(v) \ltimes M_{v,w}$ to *src*(w). In other words, *src*(v) sends to *src*(w) those ids in *res*(v) that are linked to a tuple in *src*(w). On reception, *src*(w) updates *res*(w):

$$res(w) := res(w) \cap \pi_{\text{id}_w}(M_{w,v} \bowtie res(v)),$$

and deletes $v \to w$ from P and $\prec_P$, hence enabling other edges to "execute". Evaluation stops when P has no more edges, at which point *res*(root(P)) is output. In what follows, we will simply write *res*(P) for this output.

For example, in the plan P_2 shown in Fig. 2(b), Enzyme computes the *eid*s with description d and sends them to SwissProt. (For convenience we have labeled edges in Fig. 2 that are incomparable according to $\prec_P$ with equal natural numbers, whereas the other edges are labeled with natural numbers respecting the order in $\prec_P$.) Let v_R be the node in P_2 with source name R. On reception, SwissProt updates $res(v_{\text{SwissProt}})$ to $\pi_{pid}(\sigma_{spec=\text{fly}}(\text{SwissProt})) \cap \pi_{pid}(M_{\text{Swiss,Enz}} \bowtie res(v_{\text{Enzyme}}))$, and transmits to Go. Go in turn updates $res(v_{\text{Go}})$ to

$$\pi_{goid}(\sigma_{func=f}(\text{Go})) \cap \pi_{goid}(M_{\text{Swiss,Go}} \bowtie res(v_{\text{SwissProt}})),$$

and transmits back to SwissProt who updates

$$res(v_{\text{SwissProt}}) := res(v_{\text{SwissProt}}) \cap \pi_{pid}(M_{\text{Swiss,Go}} \bowtie res(v_{\text{Go}})).$$

After reception of the updated $res(v_{\text{SwissProt}})$, Genbank derives the *gid*s to output: $res(\text{Genbank}) := \pi_{gid}(\text{Genbank}) \cap \pi_{gid}(M_{\text{Gen,Swiss}} \bowtie v_{\text{SwissProt}})$.

It is important to note that there can be multiple edges executing simultaneously in a plan. In Fig. 2(a), for example, Go and Enzyme compute and send in parallel the *goid*s with function f and the *eid*s with description d. SwissProt receives both sets of ids and computes

$$\pi_{pid}(\sigma_{spec=\text{fly}}(\text{SwissProt}) \cap \pi_{pid}(M_{\text{Swiss,Go}} \bowtie res(v_{\text{Go}})) \\ \cap \pi_{pid}(M_{\text{Swiss,Enz}} \bowtie res(v_{\text{Enzyme}}))$$

as its final value for $res(v_{\text{SwissProt}})$.

We also note that sources (like Go) always send their own ids to other sources (like SwissProt), and it is the responsibility of the receiving source (SwissProt) to use a mapping table ($M_{\text{Swiss,Go}}$) to map the received ids to its own ids. Alternatively, Go could have used $M_{\text{Swiss,Go}}$ first to send *pid*s to SwissProt, instead of *goid*s. Since in practice the average fan-out of the mapping tables is rather large (see Fig. 7), the latter strategy generates more traffic however, and therefore we opt for the former.

The *transmission cost* of a plan P is the total number of ids transmitted during P's evaluation. That is,

$$\text{cost}(P) := \sum_{v \to w \text{in} P} \#(v \to w),$$

where $\#(v \to w)$ denotes the size of the semijoin $res(v) \ltimes M_{v,w}$ at the time when $v \to w$ executes.

Optimization. Say that a plan P is *a plan for* a CQ Q if $res(P) = res(Q)$ on all possible source relation instances and mapping tables. As usual, the goal of query optimization is to find a plan P for a given CQ Q such that cost(P) is minimal among all possible plans for Q. Since in general there are infinitely many such plans and since the cost of a plan is unknown until after its evaluation, an optimizer in practice will generate only a finite set of *candidate* plans for Q and pick one with minimal *estimated* cost according to some suitably chosen cost estimation method. In this section, we restrict ourselves to those candidate plans built from the nodes in Q that can transmit ids only from child nodes in Q to their parents in Q and vice versa, and we use a particular cost estimation method that allows computation of a candidate with minimal estimated cost in quadratic time. We should stress, however, that our techniques in Section 4 for combining plans of single CQs into plans for EQs are *independent* of the way in which the single plans are computed. In particular, our combination techniques work equally well on single plans that contain more nodes than Q; on single plans in which ids can also be transmitted between, say, ancestors and descendants (although such transmissions only rarely yield a plan for Q, see the full version of this article); and on single plans that are optimal under some other cost estimation method.

Definition 1. *A plan P is a* candidate of *a CQ Q if the nodes of P are the same as the nodes of Q (including labels);* root(P) = root(Q)*; and the edges of P are a subset of $\{v \to w, w \to v \mid v \text{ parent of } w \text{ in } Q\}$.*

For example, both P_1 and P_2 in Fig. 2 are candidate plans for Q_1 in Fig. 3(b) while P_3 in Fig. 2 is not since its nodes are different than those of Q_1.

Observe that cost estimation for a candidate of Q boils down to the estimation of $\#(v \to w)$, the size of $res(v) \ltimes M_{v,w}$ when $v \to w$ executes in P. Since *res*(v) is defined by an ordinary relational algebra expression, one can use the traditional (distributed) size estimation functions based on join selectivities [27] or histograms [24] for this purpose. In that case, however, computing the candidate plan with the least estimated cost is known to be NP-hard [36]. In contrast, the following size estimation function, inspired by Stocker et al. [34], allows such plans to be computed in quadratic time.

Essentially, Stocker et al. propose to conservatively estimate the size of an intersection $R_1 \cap \cdots \cap R_n$ by the minimum

of the cardinalities $|R_1|, \ldots, |R_n|$. Since in our context the set of ids transmitted from $src(v)$ to $src(w)$ when $v \to w$ executes in a plan P consists of the ids satisfying the atom A at v intersected with the translated ids received from all incoming edges $u \to v \prec_P v \to w$ this yields the following *estimation* $est(v \to w)$ of $\#(v \to w)$:

$$est(v \to w) := \min\{est(A), est(u \to v) \times \alpha_{u,v} \mid (u \to v) \in P, (u \to v) \prec_P (v \to w)\}.$$

Here, $est(A)$ with $A = R[F]$ estimates the number of tuples in R satisfying filter F (this estimation can be obtained, for example, by contacting the source R), and $\alpha_{v,w}$ denotes the *average fan-out* of v-ids in $M_{v,w}$,

$$\alpha_{v,w} := \operatorname{avg}\{|\pi_{\mathrm{id}_w}\sigma_{\mathrm{id}_v=i}(M_{v,w})| : i \in \pi_{\mathrm{id}_v}(M_{v,w})\}.$$

Observe that the direction is important here, $\alpha_{v,w} \neq \alpha_{w,v}$.

Although $est(v \to w)$ depends on the estimations $est(u \to v)$ of all predecessor edges $u \to v \prec_P v \to w$, this definition is not cyclic since $\prec_P$ is a strict order. Therefore, $est(e)$ for every edge e in P can be computed by following the order $\prec_P$. The *estimated transmission cost* $\mathrm{ecost}(P)$ of P is simply the sum of all $est(v \to w)$. Say that P is *optimal for* a CQ Q if (1) P is a candidate of Q; (2) P is a plan for Q; and (3) there is no other candidate plan for Q with smaller estimated cost.

EXAMPLE 1. *Assume $est(Enzyme[dis{=}d]) = 5$, $est(Go[func{=}f]) = 75$, $est(\text{SwissProt}[spec{=}fly]) = 1000$, and $est(Genbank) = 50 \times 10^6$. Assume that α is as follows*

v	w	$\alpha_{v,w}$	$\alpha_{w,v}$
Genbank	SwissProt	3	2
SwissProt	*Go*	1.5	3.5
SwissProt	*Enzyme*	3	2.5

Then we estimate as follows for P_2 from Fig. 2:

v	w	$est(v \to w)$
Enzyme	SwissProt	5
SwissProt	*Go*	$\min(1000, 5 \times 2.5) = 12.5$
Go	SwissProt	$\min(75, 12.5 \times 1.5) = 18.75$
SwissProt	*Genbank*	$\min(1000, 5 \times 2.5, 18.75 \times 3.5) = 12.5$

Hence $\mathrm{ecost}(P_2) = 48.75$. A similar reasoning for P_1 yields $\mathrm{ecost}(P_1) = 92.5$. Hence, P_2 is preferable over P_1.

In contrast to estimation techniques based on histograms, the above size estimation function is particularly suited for optimization in loosely collaborating environments, as it only requires a small number of statistics (namely $\alpha_{v,w}$ and $|\pi_{\mathrm{id}_v}(M_{v,w})|$) to be globally available.

Algorithm GENDOWN, shown in Algorithm 1 can be used to compute an optimal plan for a given CQ. Essentially, it computes the set of "downward" edges $v \to w$ from parent nodes v in Q to child nodes w that, when added to the bottom-up evaluation plan of Q consisting solely of Q's nodes and edges, yields an optimal plan for Q. To formalize this claim, we introduce the following definitions. Let a *downward edge* for Q be an edge $v \to w$ over the nodes in Q for which w is a child of v in Q. Let D be a set of downward edges for Q and define $Q \uplus D$ to be the plan P for Q obtained by adding all edges in D to Q such that:

Algorithm 1 GENDOWN

Input: CQ Q and number n
Output: a set of downward edges for Q

```
 1: let v be the root of Q
 2: initialize the result D to the empty set
 3: for every child w of v do
 4:   D_w^{↓↑} := GenDown(Q|_w, n × α_{v,w})
 5:   D_w^{↑} := GenDown(Q|_w, up(w))
 6:   c_w^{↓↑} := n + ecost(Q|_w ⊎ D_w^{↓↑}) + n · α_{v,w}
 7:   c_w^{↑} := ecost(Q|_w ⊎ D_w^{↑}) + up(w)
 8:   if c_w^{↓↑} < c_w^{↑} then
 9:     add all edges in D_w^{↓↑} ∪ {v → w} to D
10:   else
11:     add all edges in D_w^{↑} to D
12: return D
```

1. downward edges ending in a node v are evaluated before all edges leaving v, i.e., for all $u \to v$ in D and all $v \to w$ in $Q \uplus D$ we have $(u \to v) \prec_P (v \to w)$;

2. all edges $u \to v$ in Q for which there is no corresponding downward edge $v \to u$ in D are evaluated before all downward edges $v \to w$ in D leaving v: $(u \to v) \prec_P (v \to w)$; and

3. all edges $u \to v$ in Q are evaluated after all edges $w \to u$ in Q: $(u \to v) \prec_P (w \to u)$.

(Here, $\mathrm{root}(Q \uplus D) = \mathrm{root}(Q)$.) For example, if Q_1 is the CQ shown in Fig. 3(b), then $Q_1 \uplus \{v_{\mathrm{Go}} \to v_{\mathrm{SwissProt}}\}$ yields the plan P_2 from Fig. 2(b).

Let $Q|_w$ stand for the subtree of Q rooted at node w in Q, and let $up(v)$ stand for the estimated size of $res(v)$ when we evaluate Q in a bottom-up manner:

$$up(v) := \min\{est(A), up(w) \cdot \alpha_{w,v} \mid w \in \mathrm{children}(v)\}.$$

In particular, $up(v) = est(A)$ for leaf nodes. Given a CQ Q and a number n, GENDOWN now operates as follows. For every child node w of Q's root v, GENDOWN recursively computes two sets of downward edges in lines 4 and 5: the set $D_w^{\downarrow\uparrow}$ for the case where we would add the downward edge $v \to w$ and the set $P_w^{\uparrow}$ for the case where would not. Based on the estimated costs of $Q|_w \uplus D_w^{\downarrow\uparrow}$ and $Q|_w \uplus D_w^{\uparrow}$, GENDOWN estimates in line 6 the gain of adding $v \to w$ to the final plan and makes its decision accordingly in the following lines. We show in the full version of this paper:

THEOREM 1. *If Q is a CQ without redundant nodes, then $Q \uplus \text{GENDOWN}(Q, up(\mathrm{root}(Q)))$ is an optimal plan for Q. Moreover, GENDOWN runs in time quadratic in the number of edges in Q.*

Here, a node n is said to be *redundant* in Q if Q is equivalent to the query obtained by removing n and all of its descendants. For instance, the second Go node in Fig. 5 is redundant as this CQ is clearly equivalent to the CQ in Fig 3(b) without this node. Although redundant nodes can be identified and removed in PTIME when containment of all mentioned atoms is decidable in PTIME (see the full version of this article), this quickly becomes undecidable for arbitrary atoms [2]. Even in the latter case, however, Theorem 1 shows how to compute optimal plans for Q *modulo the removal of redundant nodes.* In contrast, computing the

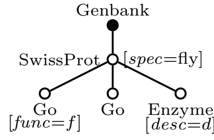


Figure 5: A concrete query with redundant node.

optimal plan using the traditional size estimation techniques is NP-hard, even for CQs without redundant nodes [36].

EXAMPLE 2. *With the same parameters as in Example 1,* GENDOWN$(Q, up(\text{root}(Q)))$ *yields* $\{v_{Go} \rightarrow v_{\text{SwissProt}}\}$*, which results in the optimal plan* P_2 *in Fig. 2(b).*

4. OPTIMIZING EXPLORATORY QUERIES

To evaluate exploratory queries, sources again evaluate atoms; transmit ids to neighboring sources; and translate incoming ids by means of mapping tables. Since EQs in general have *many* implementing concrete queries however (as shown in Figure 4), the transmission of some intermediate results can be merged. The order and direction in which ids are transmitted and how they should be merged is recorded in a *combined evaluation plan* (or simply combined plan for short).

Combined evaluation plans. Formally, a *combined plan* C consists of a set of pairwise node-disjoint plans $P_1, \dots, P_n$, together with an equivalence relation $\equiv_C$ over the edges in $P_1, \dots, P_n$ (called the *merging relation* of C) such that

- only *compatible* edges are merged: if $(v \rightarrow w) \equiv_C (v' \rightarrow w')$ then $src(v) = src(v')$ and $src(w) = src(w')$;
- the ordering of the edges is respected: if $e \equiv_C f$, $e' \equiv f'$ and $e \prec_{P_i} e'$ for some i, then $f' \not\prec_{P_j} f$ for all j.

Figure 2(e) shows an example combined plan in which merged edges are linked by dotted lines.

A combined plan evaluates all the P_i in parallel, and outputs $\bigcup_{i=1}^{n} res(P_i)$ as its final result (we denote this set simply by $res(C)$ in what follows). However, merged edges execute synchronously. In particular, an edge $v \rightarrow w$ can only execute when all $v' \rightarrow w' \equiv_C v \rightarrow w$ are ready to execute. Since all such edges are compatible (i.e., $src(v) = src(v')$ and $src(w) = src(w')$), the transmission of their set of ids from $src(v)$ to $src(w)$ can be combined. Several approaches exist to reduce data transfer in this case. For instance, when $e_1 \equiv_C e_2 \equiv_C e_3 \equiv_C e_4$ are merged edges where e_1 transmits the id $\{a\}$, e_2 transmits $\{b\}$, e_3 transmits $\{a, b\}$, and e_4 transmits $\{a\}$, then one can simply group over the ids, and transmit $\{(a, 1, 3, 4), (b, 2, 3)\}$ instead. Here the '1' indicates that the id is in the result of e_1, the '2' indicates that the id is in the result of e_2, and so on. This simplistic scheme already results in 20% savings in transmission costs. More advanced compression methods may increase these savings even further.

Optimization. Let $plans(C)$ denote the set of plans that C is composed of and let E be an exploratory query. We say that C is *a combined plan for* E if for every implementation Q of E there exists a plan $P \in plans(C)$ that outputs $res(Q)$ and conversely for every plan $P \in plans(C)$ there exists an implementation Q of E such that P outputs $res(Q)$. Clearly, if C is a combined plan for E then $res(C)$ is guaranteed to be equal to $res(E)$.

Similar to CQ optimization, the goal of EQ optimization is to find a combined plan C for a given EQ E such that cost(C) is minimal among all plans for E. In this sense, Theorem 2 below stands in strong contrast to Theorem 1: it shows that computing optimal combined plans for EQs is much harder than computing optimal plans for CQs.

Specifically, adapt the cost estimation method of Section 3 to combined plans as follows. Let $\mathcal{E}$ be a set of compatible edges over C. Let $compr(\mathcal{E})$ be a function that estimates the compression ratio achieved by combining the transmission of all edges in $\mathcal{E}$ under the particular compression method used $(0 < compr(\mathcal{E}) \leq 1)$. The *estimated* number of transmitted ids when all the edges in $\mathcal{E}$ execute simultaneously is then defined as

$$est(\mathcal{E}) := compr(\mathcal{E}) \cdot \sum_{v \rightarrow w \in \mathcal{E}} est(v \rightarrow w),$$

where $est(v \rightarrow w)$ is the estimated number of ids transmitted from v to w in the plan P_i to which $v \rightarrow w$ belongs, as defined in Section 3. The *estimated cost* ecost(C) is then simply the sum of all estimations of synchronously executing edges. That is, if $\{\mathcal{E}_1, \dots, \mathcal{E}_n\}$ is the set of all equivalence classes of $\equiv_C$, then ecost$(C) := est(\mathcal{E}_1) + \cdots + est(\mathcal{E}_n)$.

In the full version of this paperwe show by a reduction from the SHORTEST COMMON SUPERSEQUENCE problem [29]:

THEOREM 2. *Given a number k, an EQ E, and parameters for the cost estimation method, deciding whether there is a combined plan C for E with* ecost$(C) \leq k$ *is* NP*-complete.*

The corresponding optimization problem is hence also NP-hard. In view of this negative complexity result, we will refrain from generating combined plans for EQs directly. Rather, given a EQ E we first compute the concrete queries $Q_1, \dots, Q_k$ implementing E and determine optimal plans for each Q_i individually. We then use the heuristic below to construct a merging relation $\equiv_C$ on the edges of these plans, hence obtaining a combined plan.

4.1 Levelwise merging

Our first heuristic operates by looking for *local* similarities between edges. In particular, it merges edges in a set of plans $\mathcal{P}$ only if they have almost the same *level*, where the level of an edge e in $P_i \in \mathcal{P}$ is defined as

$$level(e) := 1 + \max\{level(f) \mid f \prec_{P_i} e\}.$$

In particular, $level(e) = 1$ if there is no edge f with $f \prec_{P_i} e$. To illustrate, the level of $v_{\text{Enzyme}} \rightarrow v_{\text{SwissProt}}$ in Figure 2(b) is 1, the level of $v_{\text{SwissProt}} \rightarrow v_{\text{Go}}$ is 2, and so on.

Intuitively, the level of an edge measures how early or how late the edge executes in P_i, where edges with a low level execute early and edges with a high level execute late. The motivation for merging an edge only with edges that have a similar level is the following. Consider an edge e in a plan $P_i \in \mathcal{P}$ and suppose that there are two candidate edges f_1 and f_2 in a plan P_j that we can merge e with such that $level(e) \approx level(f_1)$ but $level(e) \ll level(f_2)$. Since merged edges must respect the order in $\prec_{P_i}$ and $\prec_{P_j}$, every successor e' of e in P_i (with $e \prec_{P_i} e'$) can only be merged to the compatible edges f' in P_j with $level(f') > level(f_1)$ if we merge e with f_1. If on the other hand, we merge e with f_2 then every such successor can only be merged to the compatible edges f'' in P_j with $level(f'') > level(f_2)$. Since $level(f_1) \approx level(e) \ll level(f_2)$, we intuitively expect there

to be significantly more such f' than f''. Therefore, it is preferable to merge e with f_1.

When we have several candidate edges to merge an edge $v \to w$ with, we intuitively expect the overlap of transmitted ids to be the largest for those edges $v' \to w'$ whose atom A' at v' is the most similar to the atom A at v. In order to assess this similarity, we assume given a function $\mathrm{sim}(A, A')$ with $0 \leq \mathrm{sim}(A, A') \leq 1$ that encodes available domain knowledge and estimates the overlap between A and A'. For example, if all proteins with function f_1 are guaranteed to also have function f_2, but not f_3, then $\mathrm{sim}(\mathrm{Go}[\mathit{func} = f_1], \mathrm{Go}[\mathit{func} = f_2])$ will be higher than $\mathrm{sim}(\mathrm{Go}[\mathit{func} = f_1], \mathrm{Go}[\mathit{func} = f_3])$. Even when such domain knowledge is not available, sim can syntactically inspect A and A' to estimate similarity. In our experiments, for example, we have taken $\mathrm{sim}(A, A') = 1$ if A and A' are guaranteed to select the same ids due to a syntactic containment check, and $\mathrm{sim}(A, A') = 0$ otherwise.

The heuristic based on these observations, called LEVELWISE MERGING, is shown in Algorithm 2. Starting from the initial relation that merges no edges (i.e., $e \equiv_C f$ iff $e = f$), LEVELWISE MERGING considers the plans in $\mathcal{P}$ one by one and refines $\equiv_C$. To ensure that this refinement respects $\prec_P$ of the plan P under consideration, a queue Q is used. Initially (line 4), Q is populated with the "initial" edges e in P that have no predecessor $d \prec_P e$. For every edge e in Q, LEVELWISE MERGING then considers those compatible edges f (denoted by $e \sim f$) occurring at level i, $i + 1$, ..., $i + k$ that have already been processed. Here, k is an input parameter that determines when edges have a "similar" level, and $i = 1 + \mathit{max}\{\mathit{level}(d') \mid d \prec_P e, d' \equiv_C d\}$ ensures that if e is merged with one of the considered f, then $\prec_P$ will be respected. Let $[f]$ denote the equivalence class of f w.r.t. $\equiv_C$. The benefit gained from fusing e with all edges in $[f]$ is measured as a combination of the expected similarity of combined atoms (higher is better) and the level of $[f]$ (lower is better). We introduce some notation to define this. Let $\mathit{atoms}([f])$ be the multiset consisting of the atoms labeling v, for every $(v \to w) \in [f]$. Let $\mathrm{sim}(A, \mathit{atoms}([f]))$ denote the average similarity of A with regard to the atoms in $\mathit{atoms}[f]$:

$$\mathrm{sim}(A, \mathit{atoms}([f])) := \frac{\sum_{B \in \mathit{atoms}([f])} \mathrm{sim}(A, B)}{n},$$

where n is the cardinality of the multiset $\mathit{atoms}([f])$. Finally, let $\mathit{level}([f]) = \max\{\mathit{level}(f') \mid f' \equiv_C f\}$. The *score* of fusing e to all edges in $[f]$ is then given by

$$\mathrm{score}(e, [f]) := b_1 \times \mathrm{sim}(A_e, \mathit{atoms}([f])) - b_2 \times (\mathit{level}([f]) - i),$$

where $0 \leq b_1, b_2 \leq 1$ are parameters that may be used to bias the score towards similarity rather than level-differences or vice-verse ($b_1 + b_2 = 1$), and A_e is the atom at v in $e = v \to w$. In lines $9 - 11$, LEVELWISE MERGING picks a candidate with maximum score (if it exists), and merges accordingly. Finally, e is marked as processed in line 12, and all unprocessed immediate successors of e (the edges $f \notin D \cup Q$ with $e \prec_P f$ and no f' such that $e \prec_P f' \prec_P f$) are added to Q in line 13.

EXAMPLE 3. *As a simple illustration of* LEVELWISE MERGING*, consider the case where $\mathcal{P}$ consists of the plans P_1 and P_3 in Fig. 2 and suppose that P_1 is processed before P_3. Then in the processing of P_1, no edges are merged since all edges of P_1 are only compatible with themselves. When P_3 is processed, there is a single compatible candidate for $v_{Enzyme} \to v_{\mathrm{SwissProt}}$, namely the corresponding edge in P_1. These edges are therefore merged. Likewise, there is a single compatible candidate for $v_{Go} \to v_{\mathrm{SwissProt}}$, namely the corresponding edge in P_1. The other edges in P_3 have no compatible edges in P_1, and the resulting combined plan is hence the one in Fig 2(e).*

Algorithm 2 LEVELWISE MERGING

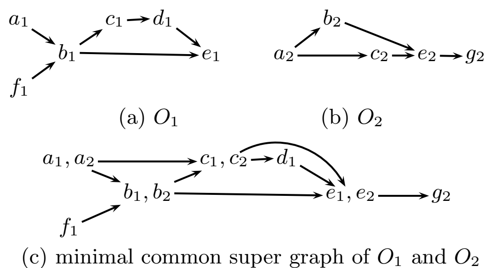
Input: a set of plans $\mathcal{P}$, a parameter k
Output: a fusion relation $\equiv_C$
1: Init $\equiv_C$ to the identity on edges in $\mathcal{P}$
2: Init set of processed edges $D \leftarrow \emptyset$
3: **for** every plan P in $\mathcal{P}$ **do**
4: Init queue $Q \leftarrow \mathit{initial}(P)$
5: **while** Q is not empty **do**
6: $e \leftarrow \mathit{pop}(Q)$
7: $i \leftarrow 1 + \max\{\mathit{level}(d') \mid d \prec_P e, d' \equiv_C d\}$
8: CAN $\leftarrow \{[f] \mid f \in D, f \sim e, i \leq \mathit{level}(f) \leq i + k\}$
9: **if** CAN $\neq \emptyset$ **then**
10: pick $[f] \in$ CAN with max score
11: merge e with every $f' \in [f]$ by setting $e \equiv_C f'$
12: add e to processed edges D
13: add unprocessed immediate successors of e to Q
14: Return $\equiv_C$

4.2 Partial Order Alignment

Our second heuristic operates by looking for *global* similarities and is based on an existing heuristic to solve the *multiple partial order alignment problem* in Computational Biology [17]. In this problem, one is given a set S; a set $\{O_1, \ldots, O_n\}$ of strict partial orders over S; and a set $F \subseteq S \times S$ of *possible fusions* and is asked to construct a *minimal common supergraph* for $O_1, \ldots, O_n$ using only fusions in F. To illustrate the concept of a *common supergraph*, a small example is given in Figure 6 where Figure 6(c) shows a common super graph for O_1 and O_2 given in Figure 6(a,b). In particular, the nodes in the supergraph are *sets* of elements from S where an element s can occur at most once in a set in the supergraph, and where distinct s_1 and s_2 can occur in the same set only if $F(s_1, s_2)$. Furthermore, all edges in $O_1, \ldots, O_n$ must have corresponding edges in the supergraph. Finally, the supergraph must be *acyclic*. For instance, the particular solution in Figure6(c) is obtained by applying the node fusions $\{(a_1, a_2), (b_1, b_2), (c_1, c_2), (e_1, e_2)\}$. Now clearly, if we take (1) S to be all edges in a set of plans $\mathcal{P} = \{P_1, \ldots, P_n\}$; (2) $O_1, \ldots, O_n$ to be $\prec_{P_1}, \ldots, \prec_{P_n}$, respectively; and (3) construct F such that $F(e, f)$ if and only if e and f are compatible edges, then a solution to the multiple partial order alignment problem also gives us a fusion relation $\equiv_C$.

Finding the *minimal* common supergraph (i.e., the common supergraph for which some measuring function yields a minimal result) is NP-complete, however, and for this purpose a heuristic PPOA has been developed [17]. We refer to [17] for the details of this heuristic, but adapt it to our setting by considering a fine-grained scoring mechanism that incorporates the similarity of atoms, similar in spirit to the scoring mechanism used for LEVELWISE MERGING. In particular, the score of aligning f to all edges in a set of edges $\mathcal{E}$ is defined as $\mathrm{score}(e, \mathcal{E}) :=$

$$b_1 \times \mathrm{sim}(A_e, \mathit{atoms}(\mathcal{E})) - b_2 \times \big(\mathit{level}(\mathcal{E}) - \mathit{level}(e)\big),$$



(a) O_1 (b) O_2

(c) minimal common super graph of O_1 and O_2

Figure 6: A minimal common super graph.

Algorithm 3 ALIGNMENT/LEVELWISE MERGING

Input: a set of EQs $\overline{E} = E_1, \ldots, E_n$
Output: a combined plan C for $\overline{E}$
1: **for** each E in $\overline{E}$ **do**
2: $\quad \overline{P}$ = plans of the implementations of E
3: $\quad C = \text{ALIGNMENT}(\overline{P})$
4: $\quad$ add C to $\overline{C}$
5: $C = \text{LEVELWISEMERGING}(\overline{C})$
6: Return C

where $0 \leq b_1, b_2 \leq 1$ are tuning parameters ($b_1 + b_2 = 1$), A_e denotes the atom at v in $e = v \rightarrow w$, $atoms(\mathcal{E})$ denotes the multiset of atoms labeling v' for every $v' \rightarrow w' \in \mathcal{E}$, $level(e)$ is defined as in Section 4.1, and $level(\mathcal{E}) = \max\{level(f) \mid f \in \mathcal{E}\}$. Let ALIGNMENT denote the resulting algorithm.

5. OPTIMIZING MULTIPLE EQs

The proposed heuristics of Sections 4.1 and 4.2 can readily be adapted for the optimization of *multiple exploratory queries*. Such optimization is necessary, for instance, to support continuous evaluation of exploratory queries that are registered once and that need to be re-evaluated when the sources are updated. Given exploratory queries $E_1, \ldots, E_m$ one can first compute plans $P_1, \ldots, P_n$ for all of their implementations, and subsequently compute a merging relation $\equiv_C$ for them using LEVELWISE MERGING or ALIGNMENT. Alternatively, one can apply a two-staged approach: (1) first compute a separate combined plan for each exploratory query; and, (2) subsequently combine these plans into a single combined plan using either LEVELWISE MERGING or ALIGNMENT. Of course, in the second phase LEVELWISE MERGING and ALIGNMENT need to compute merging relations based on existing merging relations, but they are readily extended to this case.

In our experiments, we consider three two-phase strategies. The first two simply consist of applying LEVELWISE MERGING and ALIGNMENT in both phases. Since (as we will demonstrate in Section 6) ALIGNMENT gives better results for a single EQ, we also consider the strategy where in the first phase ALIGNMENT is applied and in the second phase LEVELWISE MERGING. Algorithm 3 gives a schematic overview of the latter approach.

6. EXPERIMENTS

BioScout platform. BioScout is a distributed monitoring system for biological data which we described in [19]. It consists of a relatively small number of sources: the biological sources mentioned in Fig. 1(g). Scientists can register queries which are evaluated periodically. When new results are found for a query, the owner is notified. To assess the algorithms presented in this paper, we extended BioScout's functionality with exploratory queries.

Setup. The experiments were performed on a Pentium IV (3.0 GHz) architecture with 1 GB of internal memory running under Linux 2.6. Figure 7 shows the statistics for (portions of) the databases and mapping tables used in our experiments which were downloaded from the actual biological sources. Specifically, we show the sizes of the databases along with the sizes of some mapping tables in terms of *number of tuples*. Notice that database sizes vary from a few thousands to millions of tuples and that the same is true for mapping tables. Figure 7(c) depicts the number of distinct ids in the left and right-hand side database of each reported mapping table along with the average fan-out. We stress the importance of these statistics for the estimation of transmission costs. Notice that the tables are not *complete*, i.e., not every tuple in a database has a corresponding tuple in another database. For example, from the eleven million Genbank *gid*'s, only seventeen thousand are associated with enzymes. Also the fan-out of tables varies widely from values close to one, to values close to 55. For our experiments, we have generated 500 EQs of varying size (4 to 6 nodes) and shape with at least one path edge, in which all atom filters are conjunctions of the form *attrib* `=` *value* and *attrib* `LIKE` *value*. The resulting EQs correspond to almost 3000 concrete queries. We emphasize that although these EQs are small, the corresponding evaluation plans can be much larger. For example, we encountered plans with up to seventeen edges in our experiments. As briefly touched upon in Section 4, there are various ways to reduce data transfer when transmitting query results over a network. As all of these exploit commonalities of the data in some way or another, we choose to abstract from a concrete compression method and measure in what follows the indicated transmission costs in terms of uniquely transmitted ids. All of our experiments use a value of $k = 2$ as locality parameter for LEVELWISE MERGING, and we have taken $\text{sim}(A, A') = 1$ if A and A' are guaranteed to select the same ids (tested by a mutual containment check on the filters in A and A'), and $\text{sim}(A, A') = 0$ otherwise.

Experiment 1. The first experiment is designed to validate our heuristics for optimizing a *single* EQ E. As a baseline, we also provide the transmission cost for the case where all optimal plans of implementations of E are executed sequentially without any merging (we refer to this method as SEQUENTIAL in what follows). Fig. 8(a) shows the total (aggregated) transmission cost for 50 EQs from our pool of queries. Both heuristics significantly reduce communication: on average by 50% with values ranging from as low as 5% to as high as 85% reduction. Low improvements occur when only few ids are transferred (around 20000) while large improvements occur when many ids (around 1 million) are transferred. Furthermore, for each of the 50 EQs, the transmission cost of the plan computed by ALIGNMENT was always smaller than that of LEVELWISE MERGING. The improvement varied from 1% to 30% with, as can be derived from Figure 8(a), an average improvement of 10%. This discrepancy can be explained by analyzing the structure of the plans generated by the two heuristics. Figure 8(b) depicts box plots of the number of equivalence classes of $\equiv_C$. Clearly, ALIGNMENT generates smaller plans: fewer equiv-

Database	Size
Genbank	11658789
SwissProt	417832
Go	21610
PubMed	329214
Enzyme	4698
OMIM	17850

(a) Database sizes

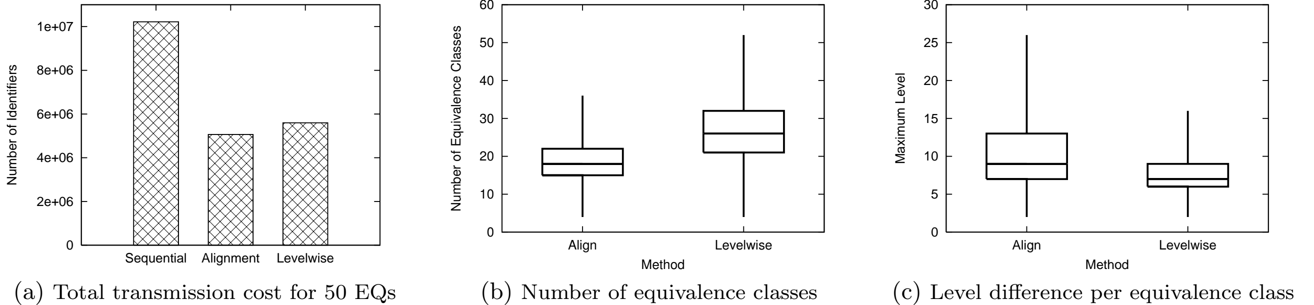
Mapping Table	Size
Genbank → Enzyme	79667
OMIM → Genbank	382177
PubMed → Genbank	4449547
Go → SwissProt	499351
SwissProt → Genbank	666700
OMIM → PubMed	109650

(b) Mapping table sizes

Mapping Table	LHS ids	RHS ids	Avg. fan-out
Genbank → Enzyme	17796	2520	4.47
OMIM → Genbank	11588	182844	32.98
PubMed → Genbank	267903	3841010	16.60
Go → SwissProt	9024	143131	55.33
SwissProt → Genbank	435734	262542	1.53
OMIM → PubMed	13571	87934	8.08

(c) Mapping table statistics

Figure 7: Database statistics



(a) Total transmission cost for 50 EQs (b) Number of equivalence classes (c) Level difference per equivalence class

Figure 8: Results for the optimization of 50 single EQs.

alence classes implies more sharing of communication. As a result, the maximum difference in level between merged edges in the same equivalence class is greater for ALIGNMENT than LEVELWISE MERGING, as shown in Figure 8(c). Constructing the 50 combined plans was reasonably fast: it took LEVELWISE MERGING 8.62 seconds, and ALIGNMENT 10.24 seconds.

Experiment 2. The second experiment is designed to validate our approach for optimizing multiple EQs. We have optimized and evaluated three batches of EQs of size 175, 350 and 500 respectively. These correspond to 1000, 2000, and 3000 concrete queries. As baseline we use again SEQUENTIAL which corresponds to no optimization. In addition we use a second baseline called TEMPLATE inspired by approaches like e.g. NiagaraCQ [8] which group queries based on a common expression signature disregarding selection constants. Intuitively, TEMPLATE checks for all plans P and P' whether the graph of P is isomorphic to a subgraph of P' respecting $\prec_P$. If so, then all edges of P are merged with the corresponding edges in P'. The total transmission cost is shown in Figure 9(a) and in more detail for the third batch of EQs with 3000 EPs in Figure 9(b). It becomes apparent that all proposed approaches significantly reduce the transmission cost with repect to both baselines. Further, the transmission costs only increase moderately for increasing number of evaluation plans. Specifically, when tripling the number of evaluation plans (from 1000 to 3000) the transmission costs of SEQUENTIAL, TEMPLATE and LEVELWISE MERGING increase with 242%, 227% and 55%, respectively. The high transmission cost of TEMPLATE indicates that merging based on subgraph isomorphism is too restrictive for the optimization of multiple EQs. In addition, it stresses that evaluation plans for randomly generated EQs are sufficiently different to warrant more sophisticated approaches like ALIGNMENT and LEVELWISE MERGING.

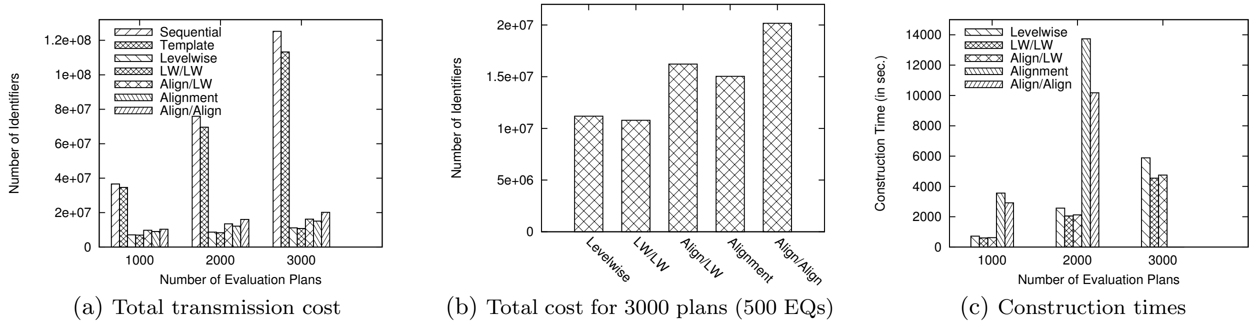
A second observation drawn from Fig. 9(b) is that LEVELWISE MERGING-based methods outperform ALIGNMENT-based methods for the optimization of multiple EQs. The reason for this is that LEVELWISE MERGING merges edges with similar levels from the bottom up. When the number of plans is large and the locality threshold k is low, there will eventually be a phase transition point where there are many (unmerged) edges on the lower levels. After this point, extra edges to be merged almost always find a candidate amongst those present. ALIGNMENT, in contrast, as already explained in Experiment 1, allows merging of edges with large level differences. As a result, there is less saturation at the lower levels, inhibiting later merging.

Finally, one-phase and two-phase LEVELWISE MERGING are comparably succesful in reducing the total transmission cost (91,4% versus 91,1% reduction w.r.t. SEQUENTIAL).

Figure 9(c) shows the construction time for computing the combined plan for the different approaches. For the case of 3000 plans, we have omitted for reasons of presentation the outlier construction times of ALIGNMENT and two-phase ALIGNMENT, which are around 25000 and 30 000 seconds, respectively. Specifically ALIGNMENT scales very poorly. The reason is twofold. Since ALIGNMENT searches for global similarities more and more possible ways of merging edges have to be considered. Furthermore, more time has to be spent computing atom similarities for large number of equivalence classes. LEVELWISE MERGING, in contrast does scale to large batches of EQs. Furthermore, its two-stage variant constructs the combined plans on average 20% faster, while yielding plans of the same quality.

Experiment 3. Our final experiment is designed to validate the behavior of LEVELWISE MERGING and ALIGNMENT as suitable methods for multi-query optimization of sets of *arbitrary* queries, as opposed to sets of queries that implement the same exploratory query and which hence have significantly shared structure. Specifically, for each EQ we randomly choose one concrete query resulting in 500 overall evaluation plans. Figure 10 shows that LEVELWISE MERGING and ALIGNMENT also significantly reduce the transmission cost in this unfavorable setup. Both algorithms save around 70% transmission cost compared to SEQUENTIAL, with a small advantage for LEVELWISE MERGING. As in the previous experiments, the construction time for ALIGNMENT (1296.17 seconds) is again notably larger than for LEVELWISE MERGING (157.58 seconds).

In summary. ALIGNMENT is best suited for the optimization of single exploratory queries as it takes the best advan-



(a) Total transmission cost (b) Total cost for 3000 plans (500 EQs) (c) Construction times

Figure 9: Results for the optimization of multiple EQs.

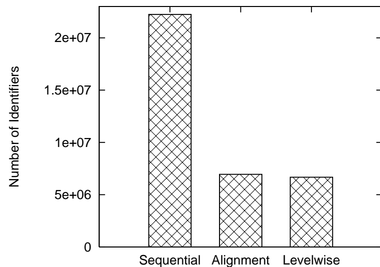


Figure 10: Total cost for 500 random plans

tage of the structural similarity of the corresponding concrete queries. The method is less suited for multiquery optimization both for exploratory and concrete queries as the construction of the actual combined plan does not scale well to larger sets of queries. LEVELWISE MERGING is best suited for multiquery optimization of concrete queries (both in terms of transmission cost and in terms of construction time). Moreover, its two-phase variant LW/LW is the best choice for the optimization of multiple EQs because of faster construction time.

7. RELATED WORK

Distributed databases. Distributed query processing has been investigated since the early days of database systems [21] and the interest is still strong in this area [34]. From the huge literature, most related to ours are the works on the optimization of *semi-joins* [3, 4], where the objective is also to minimize communication cost among participating sources. In this line of work, [9] only considers *chain* queries while we consider the more general class of *full-reducer tree* queries [36]. For this latter class, a number of optimization algorithms have been proposed [28, among others] but they all focus on single-query optimization, while we also consider multi-query optimization. More recently, Stocker et al. [34] have considered *generic* semi-join optimization techniques. The proposed techniques target a particular class of distributed client-server systems, however, in which clients communicate with servers while servers cannot communicate between themselves. Our work, in contrast, is complimentary as it considers exactly the setting where inter-server communication is possible. The difference in the two systems is essential since it influences the evaluation plans considered. Furthermore, we also consider multi-query optimization. From a theoretical standpoint, the problem of optimizing full-reducer tree queries was shown to be NP-hard in [36] w.r.t. a cost-model based on (semi)-join-selectivities. Here, we consider a more conservative cost model (c.f. Section 3) motivated by [34], and consider the complexity both for single- and multi-query optimization.

Multi-query optimization. In [31], queries (and their plans) are represented as AND-OR DAGs [30] and a family of optimization algorithms is presented to (a) group multiple query DAGs into a single one, by exploiting common query sub-expressions, and (b) determine which common sub-expressions to materialize to reduce the cumulative query evaluation cost (in terms of time). The proposed algorithms are effective for a relatively small number of chain queries and some of the underlying optimization principles there can be found in our work. However, we consider a much larger number of more general queries and focus on minimizing the communication cost. Multi-query optimization is also considered in [35], in the context of sensor networks, but only for aggregation queries, while the techniques in [33] for multi-query optimization are expensive and cannot scale to a large number of queries, as [8] also points out. In [32] an NP-hardness result is proved for multi-query optimization for a setting which is entirely different from ours: plans consist of sequences of *abstract* tasks without any meaning associated to them. In strong contrast, in our setting, every local plan has a clear semantics: it must compute the given query. The hardness result in [32] therefore does not imply our Theorem 2.

Scientific Databases. SkyQuery [23] offers subscription of queries over a federation of astronomy databases. The SkyQuery optimizer focuses on minimizing communication cost through a simple strategy that uses *performance queries* and asks each database for its estimate of data for a given query. Then, databases are ordered in decreasing order of selectivity - the one with least data is the first to execute etc. Currently, SkyQuery does not support more complex (non-chain) plans or multi-query optimization. In [26], a system is proposed to monitor biological data. The queries are similar to ours but their setting is a centralized one where every source communicates only with a central source. Furthermore, it is not the amount of data transfer that is minimized but the total number of different database accesses. BioGuide [6,7] is a system which helps scientists in selecting sources and finding alternate paths between them. Optimization of life science queries in the work of [5] focuses on single queries where the largest set of answers, following alternate paths through the graph connecting the sources, has to be computed at the lowest cost. Data integration systems for the life sciences, like Aladin [22], BioFuice [20], Kleisli [11], Orchestra [14] or DiscoveryLink [15] focus on the seamless

integration of data from heterogeneous sources and on the optimization of single-queries and/or updates. As such, they do not consider scalable multi-query optimization like we do. **Peer-to-peer systems.** Distributed query processing has also been studied in the context of unstructured p2p systems. Like us, systems like Piazza [16] and Hyperion [18] focus on heterogeneous sources. However, the focus there is on query translation, through mappings, between the sources. Like Hyperion, our work uses data-level mapping between heterogeneous sources. Unlike Hyperion, a distributed query here involves multiple sources, instead of a single source each time. Furthermore, to our knowledge, no work in this area deals with the optimization of distributed queries.
Publish-subscribe systems. In publish-subscribe systems, multi-query optimization was only considered in the context of *boolean* queries. In NiagaraCQ [8], the plans of multiple queries are grouped together if they have common *expression signatures*, i.e., their plans have common syntactic characteristics. Our work differs since (a) we consider *non-boolean* queries; and (b) apart from syntactic similarities, our algorithms also take into account the communication cost while computing and merging different plans.

8. CONCLUSIONS AND FUTURE WORK

We have proposed exploratory queries as a user-friendly means to query large networks of scientific databases for interesting correlations and have proposed several heuristics for the optimization of exploratory queries and the optimization of multiple exploratory queries. This research is part of an ongoing effort to build a distributed querying and monitoring system for biological data called BioScout [19]. For the querying part, we currently support only a small number of sources (namely six), which makes the enumeration of all implementations of an exploratory query manageable. For much larger networks it could be helpful to define a suitable ranking mechanism to avoid brute-force enumeration of paths [5]. For the monitoring part, we support the continuous evaluation of (exploratory) queries. In this context, huge communication gains can be made by caching subresults of previous evaluations and only transmitting differences with these results upon re-evaluation. In future work, we plan to investigate how the present optimization algorithms can be extended to take such caching into account.

9. REFERENCES

[1] Nucleid acid research database list. `http://www.oxfordjournals.org/nar/database/cap/`.
[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations Of Databases*. Addison-Wesley, 1995.
[3] P. A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, 1981.
[4] P. A. Bernstein, et al. Query processing in a system for distributed databases (SDD-1). *TODS*, 6:602–625, 1981.
[5] J. Bleiholder, et al. Query planning in the presence of overlapping sources. In *EDBT*, p. 811–828, 2006.
[6] S. C. Boulakia, et al. Bioguidesrs: querying multiple sources with a user-centric perspective. *Bioinformatics*, 23(10):1301–1303, 2007.
[7] S. C. Boulakia, et al. Path-based systems to guide scientists in the maze of biological data sources. *J. Bioinformatics and Computational Biology*, 4(5):1069–1096, 2006.
[8] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for internet databases. In *SIGMOD*, p. 379–390. ACM, 2000.
[9] D.-M. W. Chiu, P. A. Bernstein, and Y.-C. Ho. Optimizing chain queries in a distributed database system. *SIAM J. Comput.*, 13(1):116–134, 1984.
[10] S. B. Davidson, G. C. Overton, and P. Buneman. Challenges in integrating biological data sources. *J. of Computational Biology*, 2(4):557–572, 1995.
[11] S. B. Davidson, G. C. Overton, V. Tannen, and L. Wong. Biokleisli: A digital library for biomedical researchers. *Int. J. on Digital Libraries*, 1(1):36–53, 1997.
[12] M. Y. Galperin. The molecular biology database collection: 2008 update. *Nucleic Acids Research*, 36:D2–D4, 2008.
[13] J. Gray and A. S. Szalay. Where the rubber meets the sky: Bridging the gap between databases and science. *IEEE Data Engineering Bulletin*, 27(4):3–11, 2004.
[14] T. Green, G. Karvounarakis, Z. Ives, and V.Tannen. Update exchange with mappings and provenance. In *VLDB*, p. 675–686, 2007.
[15] L. M. Haas, et al. Discoverylink: A system for integrated access to life sciences data sources. *IBM Systems Journal*, 40(2):489–511, 2001.
[16] A. Halevy, Z. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The piazza peer data management system. *IEEE Trans. Knowl. Data Eng.*, 16(7):787–798, 2004.
[17] D. S. Parker Jr. and C. Lee. Pairwise partial order alignment as a supergraph problem. 2003.
[18] A. Kementsietsidis, M. Arenas, and R. Miller. Mapping data in peer-to-peer systems: Semantics and algorithmic issues. In *SIGMOD*, p. 325–336, 2003.
[19] A. Kementsietsidis, F. Neven, and D. Van de Craen. Bioscout: A life-science query monitoring system. Demo to be presented at EDBT, 2008.
[20] T. Kirsten and E. Rahm. Biofuice: Mapping-based data integration in bioinformatics. In *DILS*, p. 124–135, 2006.
[21] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.
[22] U. Leser and F. Naumann. (almost) hands-off information integration for the life sciences. In *CIDR*, p. 131–143, 2005.
[23] T. Malik et al. Skyquery: A web service approach to federate databases. In *CIDR*, 2003.
[24] M. V. Mannino, P. Chu, and T. Sager. Statistical profile estimation in database systems. *ACM Computing Surveys*, 20(3):191–221, 1988.
[25] NCBI. Genetic sequence data bank release notes, December 2007. `ftp://ftp.ncbi.nih.gov/genbank/gbrel.txt`.
[26] F. Neven and D. Van de Craen. Optimizing monitoring queries over distributed data. In *EDBT*, p. 829–846, 2006.
[27] M. Öszu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 2nd edition, 1999.
[28] S. Pramanik and D. Vineyard. Optimizing join queries in distributed databases. *IEEE Trans. Software Eng.*, 14(9):1319–1326, 1988.
[29] K.-J. Räihä and E. Ukkonen. The shortest common supersequence problem over binary alphabet is NP-complete. *TCS*, 16:187–198, 1981.
[30] N. Roussopoulos. View indexing in relational databases. *TODS*, 7(2):258–290, 1982.
[31] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, p. 249–260. ACM, 2000.
[32] T. Sellis and S. Ghosh. On the multiple-query optimization problem. *Trans. Knowl. Data Eng.*, 2(2):262–266,1990.
[33] T. K. Sellis. Multiple-query optimization. *TODS*, 13(1):23–52, 1988.
[34] K. Stocker, et al. Integrating semi-join-reducers into state of the art query processors. In *ICDE*, p. 575–584., 2001.
[35] N. Trigoni, Y. Yao, A. J. Demers, J. Gehrke, and R. Rajaraman. Multi-query optimization for sensor networks. In *DCOSS*, p. 307–321, 2005.
[36] C. Wang and M.-S. Chen. On the complexity of distributed query optimization. *IEEE Trans. Knowl. Data Eng*, 8(4):650–662, 1996.