Topology Search over Biological Databases

Lin Guo
Cornell University
Ithaca, NY 14853
guolin@cs.cornell.edu

Jayavel Shanmugasundaram
Cornell University
Ithaca, NY 14853
jai@cs.cornell.edu

Golan Yona
Israel Institute of Technology
Technion, Haifa, Israel
golan@cs.technion.ac.il

ABSTRACT

We introduce the notion of a data topology and the problem of topology search over databases. A data topology summarizes the set of all possible relationships that connect a given set of entities. Topology search enables users to search for data topologies that relate entities in a large database, and to effectively summarize and rank these relationships. Using topology search over a biological database, users can ask, for example, *how* transcription factor proteins are related to DNAs in humans. However, detecting topologies in large databases is a difficult problem because entities can be connected in multiple ways. In this paper, we formalize the notion of data topologies, develop efficient algorithms for computing data topologies based on user queries, and evaluate our algorithms using a real biological database.

1. INTRODUCTION

The recent decade has seen a growing number of new high-throughput technologies that have generated massive biological data sets. These heterogeneous data sets often describe different aspects of the same or related biological systems from individual molecules, through molecular complexes and cellular pathways, to cells and organisms. Such inter-related data sets can exhibit a complex structure of relationships, and an emerging direction in biological data analysis is to detect and understand relationships in these large-scale data sets [12, 18, 26]. However, effectively querying and summarizing the rich relationships in large biological database remains a challenge.

To address the above issue, we introduce the notion of a **data topology** (or just **topology**). Given a specific set of entities, a topology defines the *complete* set of relationships that relate the entities. Different relationships between entities might imply different biological meanings and understanding the exact topology is essential for an accurate and in-depth analysis of the biological system that is comprised of these entities. For example, consider two types of entities – proteins sequences and DNA sequences – stored in the Biozon [3, 8, 9] database (Figure 1). According to this schema, the two entities can be related in several different ways (some of which are shown in Figure 2), each one representing a different biological phenomenon. Each graph is a topology that shows *how* the relevant entities are related. The structure of a given topology relays its meaning in the biological context and different results for the same query might have topologies with radically different meanings. When answering the query "how transcription factor (TF) proteins are related to DNAs in humans", knowing for example that a certain TF is linked to a DNA sequence with the third topology of Figure 2 is a substantial finding that indicates that the TF is encoded by the DNA sequence and possibly regulates the same DNA through an interaction. That is, the TF self-regulates itself. Finding instances of such or more complex topologies, where the regulation is mediated through other relations, can reveal very interesting biological systems (for more such topologies, we refer the reader to Figure 16 and http://biozon.org/ftp/data/papers/topologies/graphs).

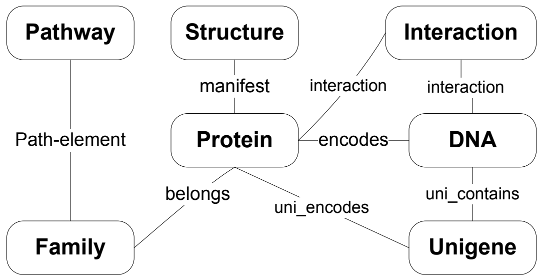


Figure 1: Partial high-level schema of the Biozon Database. The Biozon database (http://www.biozon.org) is a unified biological resource on DNA sequences, proteins, interactions, cellular pathways, and more. The tightly integrated schema allows researchers to explore the rich relationships between biological resources. A snapshot of the actual content is shown in Figure 3.

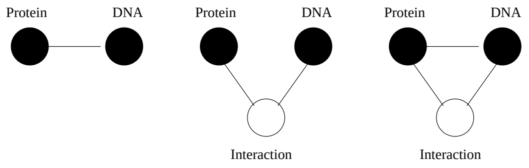


Figure 2: Different topologies over protein and DNA sequences. The following graphs relate the same two data types, but have completely different biological meanings. The first indicates that the DNA encodes for the protein. The second indicates that the protein and the DNA sequence are interacting. The third describes a protein that is encoded by a DNA sequence and interacts with it as well.

Computing topologies for user queries, however, is a challenging task because users may not know (or cannot enumerate) all possible

Protein

ID	Definitions
32	Ubiquitin-conjugating *enzyme* UBCi
78	Ubiquitin-conjugating *enzyme* variant MMS2
34	vitamin D inducible protein [Homo sapiens]
44	ubiquitin-conjugating *enzyme* E2B (homolog)

Uni_encodes

ID	PID	UID
31	78	150
14	34	103
25	78	103
42	44	188
11	44	194

Unigene

ID	Definitions
103	ubiquitin-conjugating enzyme E2
150	hypothetical protein FLJ13855
188	ubiquitin-conjugating enzyme E2S
194	ubiquitin-conjugating enzyme E2S

Uni_contains

ID	DID	UID
93	215	150
62	215	103
121	742	188
37	742	194

DNA

ID	Type	Definitions
214	*mRNA*	Oryctolagus cuniculus ubiquitin-conjugating enzyme UBCi ….
215	*mRNA*	Homo sapiens MMS2 (MMS2) mRNA, complete cds.
742	*mRNA*	Human ubiquitin carrier protein (E2-EPF) mRNA, complete cds

Encodes

ID	PID	DID
44	32	214
57	34	215

Figure 3: The example Biozon database. These are partial tables of the actual tables that reside in Biozon.

topologies for large result sets. Moreover, the query might include arbitrary constraints. For instance, a user might be interested in knowing *how* an "enzyme" protein (i.e., the *definition* attribute of relation Protein contains the keyword "enzyme") and a RNA (the *type* attribute of DNA equals to "mRNA") are related to each other. This problem becomes even more challenging because topologies can include intermediate entities that are not explicitly specified in the query. In our example query above, "interaction" is an intermediate entity type that is not explicitly specified in the query but appears in two of the three topologies in Figure 2.

Existing query approaches such as traditional SQL or some recently proposed database graph search techniques [6, 5, 7, 15, 20, 28] cannot fully and/or efficiently provide the desired topology results. As a simple example, consider using SQL for the above user query where the number of intermediate entities is limited to two. Based on the schema, there are 88453 possible topologies, which correspond to every combination (and possible intermixing) of the ten schema paths of length three or less that connect proteins and DNAs. A naive implementation based on SQL (or any other structured or semi-structured query language) will have to issue 88453 queries to check whether the relationships that correspond to the topologies exist, which is clearly inefficient. Further, each query corresponding to a topology can itself be quite complex for datasets with rich relationships, which further degrades performance.

An alternative approach is to use database search systems such as BANKS [7], DBXPlorer [6] and Discover [19, 20]. However, since these systems are not designed for topology search, they do not produce the desired results. For instance, when the user query described above is issued over the Biozon database in Figure 3, these systems return results similar to Figure 4. These results are unsatisfactory for two reasons. First, these results only produces "isolated" paths and does not produce entire topologies. Specifically, in Figure 3, protein 78 (denoted as p_{78}) is related to DNA 215 (denoted as d_{215}) in three ways: p_{78}–u_{103}–d_{215}, p_{78}–u_{150}–d_{215} and p_{78}–u_{103}–p_{34}–d_{215}. However, they are generated as independent results (possibly intermixed with other results). Consequently, it is not apparent to the user that p_{78} is related to d_{215} by three different paths. More importantly, the fact that the paths share a common intermediate entity is not explicitly captured (paths L_2 and L_6 share the entity u_{103}). Second, existing systems only produce results at the instance-level. These results can be overwhelming: running the query on the actual database returns about 250,000 results. With

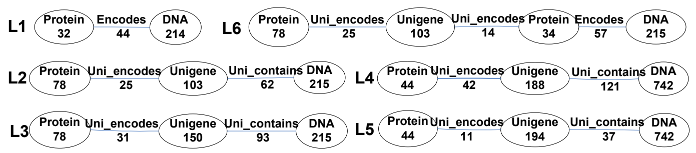


Figure 4: Isolated results of query Q_1.

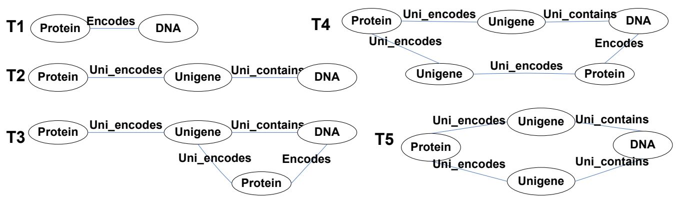


Figure 5: Topology results (except for T_5) of query Q_1.

that many results it becomes practically impossible to identify all possible schema-level topologies and get the "big picture".

To mitigate these problems, we propose methods to organize and group results based on their topology, as in Figure 5. In addition, for each topology we report all instance-level results that adhere to that topology. Given this vision, a large part of the paper is focused on *efficiently* computing topology results for a query, including producing top-k ranked topology results. We also identify some of the shortcomings of existing database systems in processing such queries, and propose extensions to a relational database system to better handle such queries.

We have experimentally evaluated our proposed techniques using the Biozon database. Our results show that our techniques can be used to find meaningful topologies efficiently. Our experiments also demonstrate some subtle issues that arise with large topologies having long paths (with many intermediate nodes). Specifically, there are some long paths with length ≥ 4 that represent weak biological relationships, which "dilute" topologies containing other stronger relationships. These weak relationships also give rise to some inefficiencies because they often have many paths relating the same pair of nodes. In the experimental section, we illustrate this issue and suggest possible solutions.

In summary, the main contributions of this paper are: (a) introducing and formalizing the notion of a topology (Section 2), (b) developing efficient algorithms for quickly computing topologies for user queries (Section 4), (c) developing efficient algorithms for producing top-k topologies (Section 5), and (d) an experimental evaluation of the proposed techniques using a real biological database (Section 6).

2. FORMAL DEFINITION OF TOPOLOGIES

We first present some background on graph databases before formally defining the topology result of queries.

2.1 Background: Graph Databases

We consider a database that has a set $\mathbf{V}$ of entities and a set $\mathbf{E}$ of binary relationships such as in the Entity-Relationship (ER) model. Logically, these are represented as a large (undirected) *data graph* $G = (\mathbf{V}, \mathbf{E})$ where each entity is represented by a node v and each relationship by an edge e. In the rest of this paper, we will use the two concepts 'database' and 'graph' interchangeably.

The entities can be of different types (entity sets), e.g. DNA sequences, Protein sequences, Interaction, Pathways, etc. Similarly, the relationships can be of different types (relationship sets), e.g. encodes, belongs, manifest, similar, etc. Note that each rela-

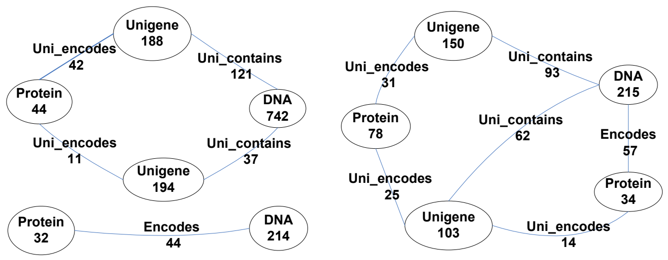


Figure 6: Graph representation of the database in Figure 3

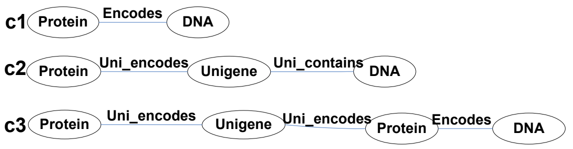


Figure 7: Equivalence classes of graphs in Figure 4

tionship can be reversed and therefore we refer to all of them as undirected (for example, we could say that a DNA sequence encodes a protein sequence or that a protein is encoded by a DNA sequence). We use $Types(V)$ and $Types(E)$ to denote the set of entity types and the set of relationship types, respectively, and assume that the function $type()$ defines the mappings $V \rightarrow Types(V)$ and $E \rightarrow Types(E)$. We refer to a graph where every entity v is labeled with its type $type(v)$ and every edge e is labeled with its type $type(e)$ as a *labeled graph*.

When mapping a relational database to a graph data model, we identify each object/relationship by the value of the primary key of the associated table. For example, Figure 1 and Figure 3 show the schema and a subset of the instance of the Biozon database, respectively, and Figure 6 shows its graph representation.

We now define some useful notation for reasoning about graph databases. A *path* is a sequence of consecutive edges in a graph and the *length* of the path is the number of edges traversed. A *simple path* is a path such that no node is traversed more than once. All paths mentioned in this paper are simple paths. A path P in a graph G can be viewed as a subgraph of G. In a graph G=(V,E), given a limit l, a node pair (a, b) (a,$b$$\in$$V$) determines a *l-path set*, denoted as $PS(a, b, l)$, whose elements are paths of G which connect a and b and are of $length \leq l$.

The *union* of two graphs $G = (\mathbf{V}, \mathbf{E})$ and $G' = (\mathbf{V}', \mathbf{E}')$, denoted by $G \cup G'$, is the graph $(\mathbf{V} \cup \mathbf{V}', \mathbf{E} \cup \mathbf{E}')$.

A labeled graph $G = (\mathbf{V}, \mathbf{E})$ is *subgraph isomorphic* to a graph $G' = (\mathbf{V}', \mathbf{E}')$ if there exists an injection f:$\mathbf{V} \rightarrow \mathbf{V}'$ such that
(1) $type(v) = type(f(v))$ for every node $v \in \mathbf{V}$.
(2) for every edge $e \in \mathbf{E}$ s.t. $e = (u, v)$ then $e' = (f(u), f(v)) \in \mathbf{E}'$ and $type(e) = type(e')$.
A labeled graph G is *isomorphic* to a graph G' if G and G' are subgraph isomorphic to each other, denoted as $G \simeq G'$. It is easy to see that isomorphism is an equivalence relation on a set of labeled graphs. Given a graph G, we use the notion $[G]$ to refer to the equivalence class of G. For instance, among the six graphs in Figure 4, [l_2]={l_2,l_3,l_4,l_5}, [l_1]={l_1} and [l_6]={l_6}. The equivalence class of G is visually represented as a graph preserving the structure of G but only showing the labels of its nodes and edges. For example, in Figure 7, c_1 represents the class of l_1, c_2 represents the class of l_2, l_3, l_4 and l_5 and c_3 represents the class of l_6. (l_1,...,l_6 are in Figure 4.)

2.2 Queries and Topologies

We begin this section by defining queries, and then define the topology results for queries.

A query specifies a list of entity types and the constraints on them. A constraint may contain multiple predicates, including keyword search clauses and structured predicates. In a database G, a *query* is defined as $\{(t_1, con_1), ..., (t_n, con_n)\}$, where $t_i \in Types(V)$ is a data type that corresponds to an entity set t_i and con_i is a constraint defined on attributes of entities v of type t_i. We call such a query an n-query. For simplicity, we focus on 2-queries in this paper.

Example 2.1: The query of the example in Section 1 is: Q = { *(Protein, desc.ct('enzyme')), (DNA, type='mRNA')*}. The first constraint is a keyword search on Protein.*desc*, and the second is a predicate on the *type* attribute of DNA. ⋄

Given a query, existing approaches such as BANKS [7], DBXplorer [6] and DISCOVER [19, 20] return the individual paths connecting the relevant entities separately. Specifically, consider a query Q = {(t_1,con_1), (t_2,con_2)}, the set $\mathbf{A}$ of t_1 entities that satisfy con_1, and the set $\mathbf{B}$ of t_2 entities that satisfy con_2. The result of the query is $\bigcup_{a \in A, b \in B}$ PS(a,b,l), which is the sets of l-results connecting the entities in $\mathbf{A}$ and $\mathbf{B}$ (l is a parameter that limits the size of the paths, as is often needed for practical reasons such as efficiency and producing interpretable relationships). Below is an example.

Example 2.2: Consider the query in Example 2.1 and assume that the set of proteins that satisfy the associated condition is $\{32, 78, 44\}$, and the set of DNAs that satisfy the associated condition is $\{214,$ $215, 742\}$. If we are interested in 3-results, the pair (78, 215) determines a 3-path set $PS(78, 215, 3) = \{l_2, l_3, l_6\}$ and the pair (44, 742) determines a 3-path set $PS(44, 742, 3) = \{l_4, l_5\}$ (see Figure 4). The union of all such 3-path sets is a set of "isolated" results. ⋄

As illustrated in Section 1, these isolated results do not provide the big picture of how satisfied nodes are related, and overwhelm users with relatively redundant (isomorphic) results. To address these problems, we (1) first assemble the paths in each path set into graphs that provide a global view of how result nodes are related; and (2) only output the topologies of the graphs as results. Thus, we suggest a novel approach for exploring complex inter-related data that summarizes the information at the *schema-level*, by listing the topologies that are detected at the *instance-level*, each one corresponds to a different biological phenomena. This is followed by instance level tuples of concrete examples (biological systems) of each topology that is detected. Note that exhaustive enumeration of all possible topologies is not practical nor informative, as the vast majority of possible topologies over a given graph schema is usually not observed in practice.

A straightforward strategy for (1) is to *union* all paths in each path set. However, this can introduce other kinds of redundancy. In Example 2.2, the paths in $PS(44, 742, 3)$ are isomorphic, which thus convey similar biological information. As a result, the topology T_5 hardly provides more valuable information than the simple topology T_2 (see Figure 5). It merely tends to overwhelm users with both the complexity (the topology could be a huge component) and size (different number of isomorphic paths lead to different topologies) of topologies.

We thus adopt an alternative strategy that focuses on the biological diversity and a comprehensive representation of the information associated with a given set of entities. Specifically, For each path set, instead of unioning *all* paths in a path set, we only union paths *across different equivalence classes* within the path set. This eliminates redundancy while capturing the important biological characteristics of the path set. We show the formal definitions step by step. We begin by defining the set of equivalence classes between a given pair of entities.

Definition 1: The *l-path equivalence classes* that relate two entities a and b is the set:
l-PathEC(a,b) = { $[G] \mid G \in PS(a,b,l)$} $\diamond$

As an illustration, consider the entities with ID 78 and 215 in Figure 3. The simple paths of length at most 3 that relate 78 and 215 are: PS(78,215,3) = { l_2, l_3, l_6 } (the paths are depicted in Figure 4). Of these paths, l_2 and l_3 belong to the same equivalence class, while l_6 belongs to a different equivalence class. Hence, 3-PathEC(78,215) contains two equivalence classes, one that corresponds to l_2 (and l_3), and another that corresponds to l_6.

As mentioned earlier, we are interested in the complex graphs that relate these entities using paths from *multiple* equivalence classes. Each such complex graph will give rise to a data topology, as defined next. It is important to emphasize that topology is a schema-level structure but its existence is verified at the instance level.

Definition 2: Consider two entities a and b, and let s be $|l - PathEC(a,b,l)|$ (s is the number of l-path equivalence classes relating a and b). The *l-topologies* that relate a and b is the set:
l-Top(a,b) = { $[G] \mid \exists p_1 \in PS(a,b,l) ... \exists p_s \in PS(a,b,l)(G = \cup_{i=1}^{s}(p_i) \wedge \forall i,j(1 \leq i,j \leq s \wedge [p_i] = [p_j] \Rightarrow i = j))$ } $\diamond$

In other words, an l-topology relating two entities a and b is obtained by creating a graph G that is the union of a path from each path equivalence class, and then obtaining the equivalence class of G. Returning to our example of entities 78 and 215, 3-PathEC(78,215) contains two equivalence classes, one corresponding to l_2 and l_3, and another corresponding to l_6. Hence, 3-Top(78,215) is obtained by unioning l_2 and l_6, and also l_3 and l_6 (paths from different equivalence classes), and then computing the topologies of the resulting graphs. Thus, 3-Top(78,215) = { T_3, T_4 } (the topologies are depicted in Figure 5). Note that T_2 is not in 3-Top(78,215) because it does not depict the full interaction of paths from different equivalence classes.

We now define the l-topology result of a query.

Definition 3: The *l-topology result* of a query Q ={(t_1, con_1), (t_2, con_2)} over a database $G = (\mathbf{V}, \mathbf{E})$ is the set:
l-Topology(Q,G) = { $T \mid \exists a, b(a \in \mathbf{V} \wedge type(a) = t_1 \wedge con_1(a) = true \wedge b \in \mathbf{V} \wedge type(b) = t_2 \wedge con_2(b) = true \wedge T \in$ l-Top(a,b)} $\diamond$

As an illustration, consider the query Q = { (Protein, desc.ct('enzyme')), (DNA, type='mRNA') } over the database in Figure 3. The Proteins that satisfy the predicate are { 78,32,44 }, and the DNAs that satisfy the predicate are { 215, 214, 742 }. As illustrated earlier, 3-Top(78,215) = { T_3, T_4 }. Similarly, 3-Top(32,214) = {T_1} and 3-Top(44,742) = {T_2}. Since there are no path relating the other Protein-DNA pairs, 3-Topology(Q,G) = {T_1, T_2, T_3, T_4}.

3. BASIC METHOD

We now turn our attention to the following problem: given a query Q={(es_1,$cons_1$), (es_2,$cons_2$)} over a database D with schema S, find the l-topology results (for some l) of Q. In this section, we describe some basic methods for solving this problem and discuss their shortcomings. We then describe our optimized algorithms that build upon these basic methods in the next section.

3.1 SQL Method

A simple strategy to compute the l-topology results is as follows. Given the database schema S, we can enumerate all possible l-topologies that connect entity sets es_1 and es_2. Then, for each possible topology T, we can issue an SQL query to check whether

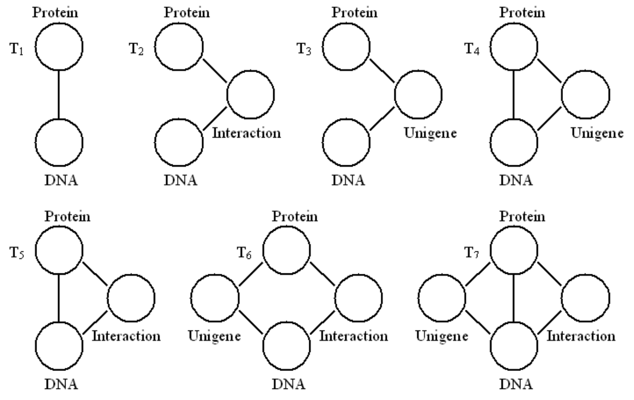


Figure 8: All possible 2-topologies relating *Proteins* and *DNAs*

there exists entities $e_1 \in es_1$ and $e_2 \in es_2$ that (a) satisfy the query constraints, and (b) are connected using topology T. As an illustration, if the query Q is issued over the database scheme shown in Figure 1 and es_1 = "Protein" and es_2 = "DNA", then the set of all possible 2-topologies is shown in Figure 8. Issuing a SQL query corresponding to each of these topologies will be sufficient to determine the set of 2-topology results for the query.

This simple strategy, however, results in poor query performance for the following reasons. First, the number of possible topologies can be very large. For example, if es_1 = 'Protein' and es_2 = 'DNA' in the Biozon database, the number of possible 3-topologies is over 88453 (due to every combination - and possible intermixing - of the ten schema paths of length three or less that connect proteins and DNAs)! Most of these topologies actually do not have any corresponding entities in the database, but even if we restrict our queries to topologies that have at least some corresponding entities (using some priori knowledge), we still have close to 200 topologies. Clearly, issuing such a large number of SQL queries is likely to be very expensive. Second, the SQL queries themselves can be quite complicated for complex topologies; this again leads to bad performance.

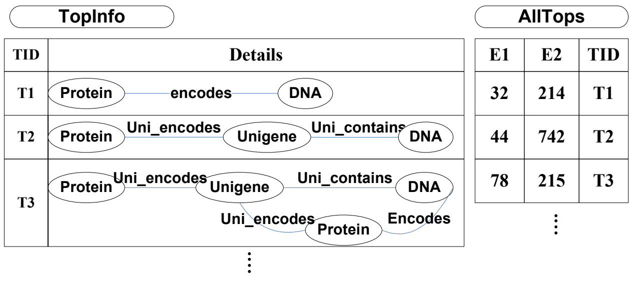
3.2 Full-Top Method

The main drawback of the SQL method is that it performs all computation at query time. Since topology computations are expensive, this is expected to lead to poor performance. Full-Top tries to address this drawback by precomputing all possible information about topologies. Specifically, Full-Top creates a *AllTops* table that stores for every pair of entities in the database, the l-topologies by which they are related. Figure 9 shows a simple *AllTops* table and an associated *TopInfo* table (that stores additional information about topologies). From the table, it can be inferred that Protein 32 and DNA 214 are related through topology T_1 (whose description is in the *TopInfo* table) and so on.

Query processing is very simple in Full-Top. For example, given the query Q={ (Protein, desc.ct('enzyme')), (DNA, type="mRNA") }, we can just issue the following single SQL query to produce the l-topology results:

```
SELECT distinct AT.TID
FROM Protein P, DNA D, AllTops AT
WHERE P.desc.ct('enzyme') and D.type = 'mRNA'
      and P.ID = AT.E1 and D.ID = AT.E2
```

The main disadvantage of Full-Top is it associated space overhead: since a large database will contain many entities, and many entities will be related to other entities, storing all the l-topologies relating any two entities is likely to be expensive in terms of space. As an illustration, for the Biozon database that has about 700MB of



TopInfo
AllTops
TID
Details
T1
Protein
encodes
DNA
T2
Protein
Uni_encodes
Unigene
Uni_contains
DNA
T3
Protein
Uni_encodes
Unigene
Uni_contains
DNA
Uni_encodes
Protein
Encodes
E1
E2
TID
32
214
T1
44
742
T2
78
215
T3

Figure 9: Table *AllTops* and *TopInfo*

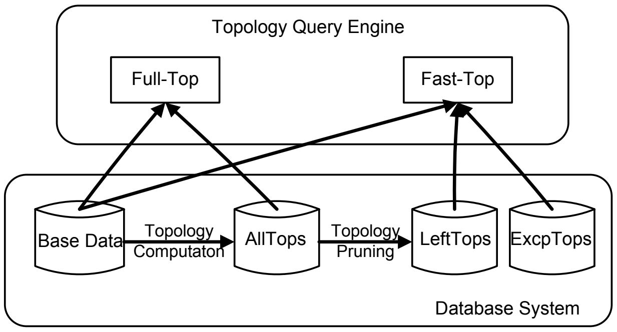


Figure 10: System architecture

queryable data[1], the size of the *AllTops* table is about 15GB. This large storage overhead often translates to poor query performance when querying the *AllTops* table. We note that the update overhead due to precomputation is *not* an issue in biological databases such as Biozon because updates are only done in bulk every few weeks, and the data can recomputed as necessary.

4. FAST-TOP METHOD

We now propose Fast-Top, an optimized method for computing l-topologies. The main idea is to *partially* precompute topology information so that it avoids the space overhead of Full-Top while still leveraging the performance benefits of precomputation. However, this leads to the following interesting challenge: how do we reduce the amount of precomputed data and at the same time *improve* query performance compared to Full-Top? We present a novel solution to this apparent dilemma by exploiting some natural properties of topologies.

Our system architecture is shown in Figure 10. In the offline (non-query processing) phase, the Topology Computation module temporarily computes the *AllTops* table described in Section 3.2 from the Base Data. The Topology Pruning module then dramatically prunes the *AllTops* table to produce the *LeftTops* table. In the online (query processing) phase, the Topology Query Engine efficiently evaluates user queries using the *LeftTops* table and the Base Data. We now describe each component in more detail.

4.1 Topology Computation

The Topology Computation module temporarily computes the *AllTops* table described in Section 3.2. To do this, it considers each pair of entity sets (es_1,es_2) in the database schema and enumerates all possible paths (not topologies) between the two entity sets of length l or less. For each of these paths, it issues a single SQL query to obtain all entities along the path and orders the results based on the IDs of the first and last entities in the path. It then merges

[1]The part of the data that cannot be queried, such as DNA sequences, actually constitutes the bulk of the data.

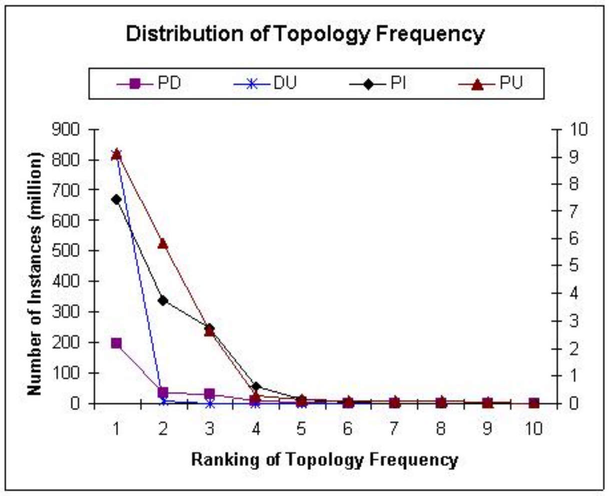


Figure 11: Distribution of topology frequency

the SQL results corresponding to all the paths to compute the l-topology for each pair of entities and stores them in the *AllTops* table.

4.2 Topology Pruning

We now describe how the *AllTops* table can be pruned while at the same time improving the performance for most queries. Our solution is motivated by some topology statistics that we obtained using the Biozon database. Since the Biozon database integrates data from a large number of biological data sources, including GeneBank [14] and SwissProt [4] and many others, these statistics are representative of a large fraction of biological databases. We now present the results of analyzing topologies.

4.2.1 Topology Data Analysis

We analyzed 3-topologies in the *AllTops* table for the Biozon database (we also analyzed other l-topologies, and the results are similar). Specifically, for each pair of entity sets es_1, es_2 in the database schema, we defined the *frequency* of a topology T, $freq$ (es_1, es_2, T), to be: $|\{(e_1, e_2)|e_1 \in es_1 \wedge e_2 \in es_2 \wedge (es_1$ *and* es_2 *are related by topology* T $\}|$. Intuitively, the frequency of a topology T is the number of entity pairs that are related by T.

Figure 11 shows the distribution of topology frequency for various entity set pairs: P stands for Protein, D for DNA, I for Interaction and U for Unigene. The x axis ranks topologies by their frequency and the y axis shows the frequency of topologies (curve PD and DU use the left y axis, while PI and PU use the right axis). As shown, the distribution of topology frequency is approximately Zipfian for all entity set pairs (it is also Zipfian for other entity set pairs not shown here). What this means is that, given a pair of entity sets, most pairs of entities belonging to those entity sets are related using very few distinct topologies. It is only a few rare pairs of entities that are related using uncommon topologies.

To better understand the structure of topologies with large frequency, in Figure 12, we show the details of the top 10 most frequent topologies relating Proteins and DNAs. As shown, all these topologies have a relatively simple structure; most of them are no more complicated than a path. This captures the intuition that most entities are related in simple ways and only a few entities are related in more complex ways.

4.2.2 Pruning Frequent Topologies

The observations in Section 4.2.1 enable an efficient topology

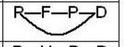
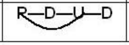
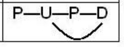
Pair	Rank	Frequency	Topology	Rank	Frequency	Topology
Protein DNA	1	197615546	P—F—P—D	6	1233573	P—D
	2	36131458	P—D—U—D	7	510309	P—U—P—D
	3	30718035	P—U—D	8	194870	P—I—P—D
	4	9016524	P—D—P—D	9	173728	 P—F—P—D
	5	4140324	 P—D—U—D	10	96953	 P—U—P—D

Figure 12: Details of 3-topologies relating Proteins and DNAs. P stands for proteins, F for protein families, D for DNAs, U for unigenes and I for interactions.

LeftTops

E1	E2	TID
32	214	T1
78	215	T3
⋮		

ExcpTops

E1	E2	TID
78	215	T2
⋮		

Figure 13: Table *LeftTops* and *ExcpTops*

pruning strategy. By pruning the few most frequent topologies, we can reduce the size of table *AllTops* dramatically, thereby improving the performance of queries over the non-pruned topologies. Further, since the frequent topologies have a relatively simple structure, it should be efficient to check for the existence of these topologies during query processing using the base data.

There is one subtlety, however, that arises from pruning frequent topologies: although a frequent topology has a simple structure and its existence for a given pair of entities can be checked easily during query processing, we also need to ensure that the pair of entities is *not* related through a more complex topology (for if the entities are related through a more complex topology, then that complex topology – which provides more detailed information about the relationships – should be returned to the user instead of the simple topology). For instance, consider the example after Definition 2 in Section 2.2: entities 78 and 215 are not related by the simple topology T2 (even though they are related by the path represented by T2) because they are related by the more complex topologies T3 and T4.

The above issue gives rise to an interesting question: how do we efficiently check whether a given pair of entities is related by a complex topology so that we can infer that it is not related by a (frequent) simple topology during query processing? In the above example, how do we efficiently detect that entities 78 and 215 are related by complex topologies and hence not related by T2? A simple approach is to issue a complicated SQL query to check for the absence of every different complex topology. However, this is likely to be inefficient, and would degenerate to the SQL method and defeat the main purpose of precomputation.

We thus propose to store an *exception table*, which stores the pairs of entities that are related by the simple path (or graph) condition of a frequent pruned topology, but which should not appear in the query results because they are part of a more complex interaction represented by more complex topologies. For the majority of the pairs of entities, which are only related by the simple topology, we do not need to store them in the exception table. In Figure 13, we illustrate how the *AllTops* table has been pruned to produce the *LeftTops* table by removing topology T2. Also, since the entities 78 and 215 satisfy the path condition of T2, but are in fact related by a more complex topology (T3), they are stored in the exception table *ExcpTops*. Note that the pruned entities 44 and 742 are not stored in the *ExcpTops* table since they *are* related by T2.

In Figure 10, the Topology Pruning module prunes the top few most frequent topologies from the *AllTops* table and produces the *LeftTops* and *ExcpTops* tables. Based on the expected performance gains of pruning (using the query evaluation technique described in the next section), we set an appropriate pruning threshold – all topologies with frequency greater than the pruning threshold are pruned.

4.3 Query Evaluation of Fast-Top

We now show how to evaluate a query based on *LeftTops*, *ExcpTops* and the base data. The query evaluation is similar to that for Full-Top, except that the pruned topologies need to be evaluated on-line. As an illustration, consider the the query Q={(Protein, desc.ct('enzyme')), (DNA, type= 'mRNA')} evaluated over the pruned tables in Figure 13. We issue the following SQL query to evaluate Q (assuming that the IDs of different biological objects are not overlapping).

```
SQL1:
SELECT distinct LT.TID
FROM Protein P, DNA D, LeftTops LT
WHERE P.desc.ct('enzyme') and D.type = 'mRNA'
      and P.ID = LT.E1 and D.ID = LT.E2
UNION
SELECT distinct T2
FROM Protein P, DNA  D,
     Uni_encodes JOIN Uni_contains as PUD
WHERE P.desc.ct('enzyme') and D.type = 'mRNA'
      and P.ID = PUD.PID and D.ID = PUD.DID
      and not exists (SELECT distinct 1
                      FROM  ExcpTops e
                      WHERE e.ID1=P.ID and
                            e.ID2=D.ID and
                            e.TID = T2)
```

The top subquery computes the unpruned topology results of Q as in Full-Top. The lower sub-queries checks to see whether there is some pair of entities in the database that (a) satisfies the query conditions, (b) is related using the path corresponding to T_2, and (c) does not appear in the *ExcpTops* table for T_2. In general, we need to issue as many lower sub-queries as there are pruned topologies. Note that the sub-queries for the pruned topologies are relatively simple, and hence, can be evaluated efficiently. Furthermore, the Zipfian distribution of the topology frequency ensures that only a small number of topologies are pruned – we pruned 19 topologies out of 805 when $l \leq 3$.

5. TOP-K QUERY EVALUATION

So far, we have focused on computing *all* the topology results for a query. However, in large heterogeneous databases such as Biozon, where a query can return many hundreds of topologies, users may only wish to see the top few topology results based on some ranking of topologies (and view the other topology results if needed). In this section, we extend the FastTop method to effectively handle such top-k topology queries by devising new early-termination query evaluation and optimization techniques. The proposed techniques can be easily integrated with a relational database system, and are also applicable to a larger class of SQL queries (including non top-k topology queries).

Our proposed techniques can work with any method for scoring topologies. Hence, for the rest of this section, we simply assume that the score for each topology is stored as the `score` attribute in the *TopInfo* table. In Section 6, we evaluate the performance of the proposed algorithms using three different topology scoring functions, two based on topology frequency and one based on the assessment of a domain expert.

The rest of this section is organized as follows. We first describe a simple extension to the Fast-Top method for producing top-k topology results, and illustrate why existing relational database systems cannot always evaluate such queries efficiently. We then describe new techniques for efficiently evaluating such queries and show how they can be integrated with a relational database system.

5.1 The Fast-Top-k Method

Consider the query Q={ (Protein, desc.ct (‘enzyme’)), (DNA, type=‘mRNA’) }. A simple way to produce the top-k topology results is to extend the Fast-Top query *SQL*1 (Section 4.3) to also produce the topology score, and then order by the score to produce the top-k results. The resulting SQL query for producing the top-10 results is shown below (we use the notation *score(T_2)* to denote the score of topology T_2):

```
SQL3:
SELECT distinct LT.TID, Top.score AS SCORE
FROM Protein P, DNA D, LeftTops LT, TopoInfo Top
WHERE P.desc.ct(’enzyme’) and D.type = ’mRNA’
      and P.ID = LT.E1 and D.ID = LT.E2 and
      Top.TID = LT.TID
UNION
SELECT distinct T2, score(T2) AS SCORE
FROM Protein P, DNA  D,
     Uni_encodes JOIN Uni_contains as PUD
WHERE P.desc.ct(‘enzyme’) and D.type = ‘mRNA’
      and P.ID = PUD.PID and D.ID = PUD.DID
      and not exists (SELECT distinct 1
                      FROM  ExcpTops e
                      WHERE e.ID1=P.ID and
                            e.ID2=D.ID and
                            e.TID = T2)
ORDER BY SCORE DESC
FETCH FIRST 10 ROWS ONLY
```

We can also optimize the evaluation of the above query by first obtaining the top-k results from the top sub-query:

```
SQL4:
SELECT distinct LT.TID, T.score AS SCORE
FROM Protein P, DNA D, LeftTops LT, TopoInfo Top
WHERE P.desc.ct(’enzyme’) and D.type = ’mRNA’
      and P.ID = LT.E1 and D.ID = LT.E2 and
      Top.TID = LT.TID
ORDER BY SCORE DESC
FETCH TOP 10 ONLY
```

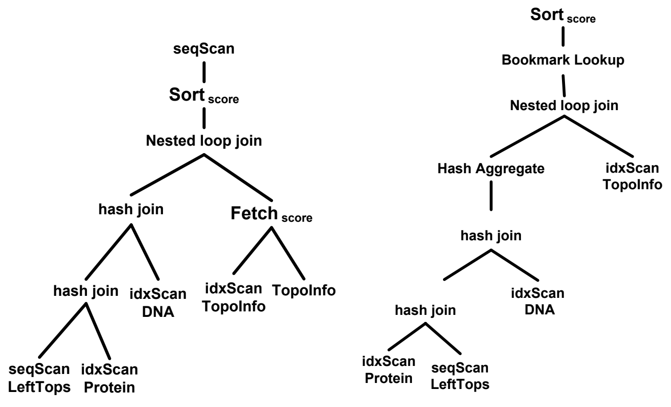
If the query produces at least ten results, and the lowest topology score in the result is higher than the score of the pruned topologies, then there is no need to evaluate the bottom sub-query (or sub-queries, in the case of multiple pruned topologies). Otherwise, the following query is executed for each pruned topology with a potentially higher score to verify if it is in the top-k results.

```
SQL5:
SELECT distinct T2, score(T2) AS SCORE
FROM Protein P, DNA  D
     Uni_encodes JOIN Uni_contains as PUD
WHERE P.desc.ct(‘enzyme’) and D.type = ‘mRNA’
      and P.ID = PUD.PID and D.ID = PUD.DID
      and not exists (SELECT distinct 1
                      FROM  ExcpTops e
                      WHERE e.ID1=P.ID and
                            e.ID2=D.ID and
                            e.TID = T2)
```

5.2 Limitations of the Fast-Top-k Method

The Fast-Top-k method is not always efficient because the underlying relational database cannot process some of the queries efficiently. We illustrate this problem using the query evaluation plans of two commercial relational database systems: IBM DB2 and Microsoft SQL Server. For the rest of this section, we focus on the query $SQL4$; the issues (and proposed solutions) for the other queries are similar.

Figure 14 shows the query evaluation plan for SQL4 chosen by DB2 and SQL Server. As shown, the two plans join the LeftTops table first with the selected tuples in the Protein table, and then join the result with the selected tuples in the DNA table. These results are then joined with the TopInfo table and the result is sorted to produce the top-k topology results.



(a) DB2 **(b) SQL Server**

Figure 14: Query Execution Plans of Commercial DBMSs

There are two main sources of inefficiency in the above plans. First, all the topologies are processed, and the top-k results are produced only in the final step. This results in unnecessary computation for topologies that are not part of the top-k result. Second, within each topology, *all* selected Protein and DNA entities (in their corresponding tables) are joined with the corresponding pairs in the *LeftTops* table – these are produced as a result of the joins. This is wasteful because we only need to verify whether *at least one* Protein-DNA pair corresponding to a topology also appears in the selected Proteins and DNAs, and we can stop processing that topology after identifying one such pair.

Unfortunately, prior solutions proposed in the literature cannot address the combination of the above two issues. For example, some techniques have recently been proposed to incorporate ranking into relational databases (e.g., [13, 21, 22, 27]). However, the focus of the work has mostly been in the context of ranked joins. In our example, if we view the topology score as the rank, the ranked join algorithms will produce all selected Protein-DNA pairs of a topology with a higher score, before producing Protein-DNA pairs of a topology with a lower score. While this join processing will avoid processing topologies that do not appear in the top-k result, it will still process *all* pairs within each topology, even though only a single pair is needed to infer the existence of a topology. In other words, prior work does not focus on *distinct* top-k queries, which is at the heart of inferring topologies.

Another related optimization used in many commercial systems is “early-out” joins [1, 2]. As an illustration, consider R *join* S, where we need to produce only tuples of R that join with at least one tuple of S. Using an early-out join, we do not need to produce all combinations of an R-tuple that joins with an S-tuple, but we can stop processing an R-tuple after we find the first joining S-tuple, and then move on to the next R-tuple. Unfortunately, early-out joins do not help much in our example either. To see why, consider the lower-most hash-join in the query plan in Figure 14-(a). Replacing this join with an early-out join will not help because each tuple in the *LeftTops* table will join with at most one tuple in the *Protein* table (since each *LeftTop* tuple contains at most one ProteinID). Hence, the size of the join result will not be altered by the early-out optimization and will incur a similar overhead. A similar argument holds for the other joins as well.

There are also techniques for pushing "distinct" computation down in a query plan [25]. However, in our example, we cannot push down the "distinct" on topologies to above the lower-most join because that would remove the IDs of the DNAs, and make it impossible to join with the selected DNAs above. Finally, there are techniques for optimizing the fetching of the top-k rows of a relational result [10, 11, 16], but these techniques do not apply to queries that perform a distinct on the result of joins.

5.3 The Fast-Top-k-ET Method

To address the above limitations of the Fast-Top-k method, we now propose the Fast-Top-k-ET method (ET stands for Early Termination). The key idea is to introduce a new family of join operators called *Distinct Group Join* (DGJ) operators. DGJ operators (a) understand the notion of a group of tuples, and preserve the order of groups in the input and propagate it to the output, and (b) allow for efficiently skipping from one group of tuples to the next group of tuples. Property (b) is exposed by means of the *advanceToNextGroup* method, which is in addition to the usual *getNext* method supported by regular operators [17].

The intuition behind a DGJ operator is that property (a) preserves the score ordering of input topologies in the output, and property (b) allows the join to efficiently skip processing a topology as soon as a match for that topology is found.

As an illustration, consider the first query plan in Figure 15, which is an alternative query plan for our running example using DGJ operators (IDGJ is a specific implementation of a DGJ operator, which will be described shortly). In this query plan, the lowermost operator produces topologies in score order. The next IDGJ operator produces all the *LeftTops* tuples for the first topology, before producing all the *LeftTops* tuple for the second topology, and so on. Similarly, the next IDGJ operator produces all the proteins corresponding to the *LeftTops* tuples for the first topology, before producing the tuples for the second topology, and so on. The selection condition only selects the Proteins that satisfy the query condition, and similarly for the next IDGJ operator that joins with DNAs. The interesting aspect of this plan is that once a *LeftTops* tuple that joins with a Protein-DNA pair that satisfies the query predicate is identified for a given topology, the processing of the remainder of the topology can be *skipped* by calling the *advanceToNextGroup* methods on the DGJ operators. Further, once the top-k topology results are determined, processing can be *stopped*. In this way, the DGJ operators address the limitations of the previous approach.

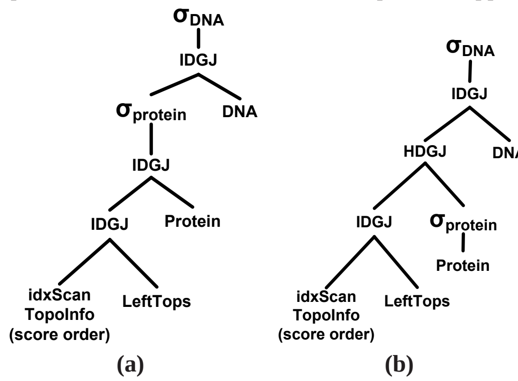


Figure 15: Alternative query evaluation plans

We now describe how we can implement DGJ operators that satisfy the above properties. One simple implementation of a DGJ operator is as an (index) nested loops join (IDGJ). IDGJ preserves property (a) because any nested loops join preserves the order of the outer relation. IDGJ preserves property (b) by simply discontinuing the current loop and invoking *advanceToNextGroup* on its input and starting a new loop. Another implementation of a DGJ operator is as a hash-join (HDGJ). Since the regular hash-join operator does not preserve the order of the outer relation, the HDGJ operator preserves this order by performing the join a group at a time. One implication of this implementation is that the inner relation may be evaluated multiple times, once for each group.

The IDGJ and HDGJ joins can be used interchangeably since they both support the DGJ operator interface. Two query evaluation plans using IDGJ and HDGJ for our running example are shown in Figure 15.

5.4 Query Optimization for Fast-Top-k-ET

While the Fast-Top-k-ET method has the potential to improve performance using the early-termination property of DGJ operators, it may not always be better than the Fast-Top-k method (without DGJ operators). To see why this is the case, consider the implementation of the DGJ operators: the IDGJ operator requires (random) index lookups and the HDGJ operator requires re-scanning the inner relation for each group, while a regular hash-join does not have any of this overhead. So, there can be cases where the benefit of early-termination does not outweigh the extra cost and complexity of DGJ operators. Such cases are expected to arise where the size of each group is small (so early termination within a topology has little benefit) and/or when the value of k is large (so termination across topologies has little benefit). Another practical issue to consider is combining DGJ operators with regular operators because there could be some parts of the query amenable to early-termination, but we may wish to use regular joins for the remaining parts of the query.

To address the above issues in a principled manner, we devise a framework for the cost-based optimization of queries using a mix of DGJ and regular join operators. Thus, depending on the cost, the optimizer may choose a plan consisting solely of DGJ operators, solely of regular join operators, or a mix of the two. We note that existing rank-aware query optimization techniques [22] are not directly applicable in our scenario because they only consider regular ranked joins, and not DGJ joins, which involve skipping tuples within a group (using *advanceNextGroup*) – incorporating skipping tuples within a group requires new cost models and associated techniques, as we shall describe next.

Our optimization algorithm can be tightly integrated with a System R style optimizer [24], and can thus be easily implemented in a relational database system. Consequently, our optimization techniques are applicable not only to topology queries, but to a broader class of SQL queries of the following form (with or without the "order by" and the "fetch top k" clauses):

```
SQL6:
SELECT distinct (O1.ID), O1.score
FROM  Object1 O1,..., Object On
WHERE local_predicate (O1) and ... and
      local_predicate (On) and
      O1 join O2 join ... join On
ORDER BY O1.score DESC
FETCH FIRST k ROWS ONLY
```

There are two main tasks in extending the query optimizer to understand DGJ operators: (1) expanding the search space of the optimizer to include DGJ operator plans, and (2) developing a cost model for DGJ operators. We consider each in turn.

5.4.1 *Expanding the Search Space*

A System R style optimizer [24] performs a bottom-up plan enumeration technique based on dynamic programming to explore the space of join orderings. At each step, various join operators are considered, including hash-join, sort-merge join, and nested-loops join, and the least-cost plan for each "interesting order" is kept (where an interesting order is an ordering of the output tuples on a set of attributes). To expand this search space, we only need to consider DGJ joins in addition to the other join operators. An important aspect of DGJ joins is that they preserve the interesting order of its (outer) input relation. In addition, we need to add a new "interesting property" called the *early-termination property* for DGJ joins because they support the *advanceNextGroup* method that can be exploited by higher operators. Thus, at each stage, the optimizer retains the least-cost plan for each interesting order and interesting property.

5.4.2 Cost Estimation

We now describe how to estimate the cost of a stack of DGJ operators given a value of k, the desired number of results. We note that the cost computation can be done bottom-up, for one DGJ operator at a time as in System-R style dynamic programming, but we consider an entire stack of DGJ operators for ease of exposition. For the purposes of this section, we also make the following assumptions. We consider only IDGJ operators; the extension to HDGJ operators is similar. We assume that we know the values of the following parameters (we show how to compute these parameters using database statistics in the next section):

- m, which is the number of groups (topologies) in the input to the DGJ operators. We refer to the groups as g_1,...,g_m.
- np_i ($1 \leq i \leq m$), which is the probability of the plan not finding any result from group g_i.
- nc_i ($1 \leq i \leq m$), which is the *expected* cost of the plan not finding any result from group g_i.
- ec_i ($1 \leq i \leq m$), which is the *expected* cost of the plan finding the first result from group g_i.

Given the above parameters, we define a random variable $Z^k_{1:m}$ that is equal to the cost of the finding the top k results from groups g_1,...,g_m. The expected cost of the plan is thus $E[Z^k_{1:m}]$.

THEOREM 1. $\forall l.\ 1 \leq l < m$

$$E[Z^k_{l:m}] = ec_l + (1 - np_l) \times E[Z^{k-1}_{l+1:m}] + nc_l + np_1 \times E[Z^k_{l+1:m}]$$

PROOF. Assume the plan is currently processing tuples in group g_l. Let $Card_l$ be the number of tuples in group g_l, and let X be a random variable that represents the first tuple in g_l that satisfies all the joins and predicates in the plan. Let $Cost_i$ be the cost of processing the first i tuples in g_l, when $X = i$ ($i \leq Card_l$). Let Y be a random variable that represents the last tuple in g_l such that it and no previous tuple in g_l satisfy all the joins and predicates in the plan. Let $NCost_l$ be the cost of processing all $Card_l$ tuples of g_l when $Y = Card_l$ (by definiton, we know that $P(Y = Card_l) = np_l$). Thus, for each $1 \leq i \leq Card_l$, we have

(1.1) $P(Z^k_{l:m} = Cost_i + Z^{k-1}_{l+1:m}) = P(X = i)$

(1.2) $P(Z^k_{l:m} = NCost_l + Z^k_{l+1:m}) = P(Y = Card_l)$

Here, (1.1) represents the case that the plan can find a result after processing i tuples in group g_l, and (1.2) represents the case that no result can be found in the group g_l. Therefore,

$$\begin{aligned} E[Z^k_{l:m}] &= \sum_{i=1}^{Card_l} P(X = i)(Cost_i + E[Z^{k-1}_{l+1:m}])) \\ &+ P(Y = Card_l)(NCost_l + Z^k_{l+1:m}) \\ &= ec_l + (1 - np_l)E[Z^{k-1}_{l+1:m}] + nc_1 + np_l \cdot E[Z^k_{l+1:m}]) \end{aligned}$$

□

Given that $E[Z^k_{l:h}]$ = 0 when $l > h$ and when $k = 0$, we can use dynamic programing to compute $E[Z^k_{1:m}]$ efficiently.

5.4.3 Estimating Parameters Using Statistics

We now show how we can estimate the parameters used in the previous section using regular database statistics. We consider a hierarchy of n DGJ join operators, opr_1,...,opr_n, and assume the existence of the following statistics in the database system.

1. m, which is the number of groups (topologies) in the input to the DGJ operators.
2. $Card_i$ ($1 \leq i \leq m$), which is the cardinality of a group g_i.
3. N_i ($1 \leq i \leq n$), which is the cardinality of the ith relation being joined in the stack of DGJ operators.
4. I_i ($1 \leq i \leq n$), which is the cost of an index probe on the joining attribute of the ith relation being joined in the stack of DGJ operators.
5. ρ_i ($1 \leq i \leq n$), which is the selectivity of the ith local predicate.
6. s_i ($1 \leq i \leq n$), which is the selectivity of the ith DGJ join.

The value in 1 is usually stored as histograms associated with a relation, and 2 is the cardinality information stored with a relation. 3 is usually stored as index statistics, and 4 and 5 can be calculated using selectivity and join estimation techniques. We also assume the independence of tuple for joins and selections. Using the statistics, we can determine the value of np_i, nc_i and ec_i ($1 \leq i \leq m$). Please see Appendix A for the details.

6. EXPERIMENTAL EVALUATION

We now experimentally evaluate the various approaches described in the previous sections using the Biozon database. We focus on (a) the relative performance of the different approaches for computing topologies, (b) the effectiveness of the optimizer in choosing the more efficient approach, and (c) the cost of retrieving the instances of a given topology. Our experimental results show that we can achieve interactive response time for topology queries using the various optimization techniques proposed in this paper.

Of course, the ultimate goal of topology search over biological databases is to enable researchers to interactively explore the data and make new scientific discoveries. While a study that quantifies the role of topology search in scientific discoveries is beyond the scope of this paper, we present some very preliminary anecdotal evidence of how a computational biologist in our research team used topology search to identify an interesting topology that appears worthy of further investigation.

6.1 Experimental Setup

We used the Biozon database for our experiments. The database contains 28 million biological objects (stored in seven tables) and 9.6 million binary relationships between the objects (stored in eight tables). We only used the query-able data in Biozon for our experiments, whose size is about 700MB (a large fraction of the Biozon database is DNA sequences, which cannot be queried). We used the IBM DB2 database, and built indices on all the primary keys and queried attributes. All experiments were run on a 1.8GHz Pentium 4 processor running Linux 2.4.21-15.EL, which had 1GB of main memory and 250 GB of disk space.

We evaluated nine methods, five of which have been previously described in the text: SQL, Full-Top, Fast-Top, Fast-Top-k, Fast-Top-k-ET. We also implemented Full-Top-k, Full-Top-k-ET (which are similar to Fast-Top-k and Fast-Top-k-ET, respectively, but without topology pruning), Fast-Top-k-Opt (which uses the proposed optimization technique to choose between Fast-Top-k and Fast-Top-k-ET), and Full-Top-k-Opt (which is similar to Fast-Top-k-Opt, but without topology pruning). Table 1 shows the space requirements for the Full-Top and Fast-Top strategies, where the pruning threshold was set to 2M (based on studying the effect on query

Object Pair		FullTop	FastTop		
object	object	*AllTopo*	*LeftTopo*	*excpTopo*	Ratio
Protein	DNA	3.36GB	30MB	70M	3%
Protein	Interaction	178MB	11MB	1.4M	6.8%
Protein	Unigene	222MB	10MB	3.9M	6.5%
DNA	Interaction	1.0GB	3.1MB	0.12M	0.3%
DNA	Unigene	9.8GB	11.6MB	4.1M	0.1%
Unigene	Interaction	4.5MB	N/A	N/A	N/A

Table 1: Space Requirement

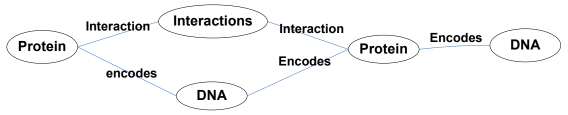


Figure 16: A topology of biological significance

performance, as described in Section 4.2). As shown, the pruning in Fast-Top results in significant space reduction.

All the methods were implemented in C++. Since we could not modify the code for IBM DB2, we implemented the DGJ operators outside the database engine by invoking database sub-queries where necessary. We also implemented our optimizer as an external method that invoked the DB2 optimizer to obtain statistics and sub-plan costs. We used a warm database cache and each query result presented is the average of multiple runs.

We used 3-topologies for most of our experiments (we vary this parameter Section 6.2.3). We used three schemes for ranking topologies. Our first ranking scheme assigned higher scores to topologies with a higher frequency, thereby emphasizing common topologies. Our second ranking scheme assigned higher scores to topologies with a lower frequency, thereby emphasizing rare topologies. Our third ranking scheme relied on a domain expert (one of the co-authors) to rank the interesting topologies based on biological significance. We refer to these three ranking schemes as Freq, Rare, and Domain, respectively.

6.2 Experimental Results

6.2.1 *A Topology of Biological Significance*

The topology search interface is intended to help biologists find interesting topologies. We list all 3-topologies relating Proteins and DNAs at http://biozon.org/ftp/data/papers/topologies/graphs/. One of the more interesting topologies among them was Figure 16. This topology represents a subgraph of two proteins that are encoded by the same DNA sequence, and also interact with each other. This phenomenon is observed for long DNA sequences (such as complete genomes or chromosomes) that encode for multiple proteins. However, the more interesting cases are of shorter DNA sequences, that encode for operons or viral genomes that encode a small number of proteins where the proximity of the proteins on the DNA sequence suggests that they are regulated by the same mechanism.

6.2.2 *Query Performance*

Table 2 compares the performance of the different methods for queries involving *Protein* and *Interaction*. The labels selective, medium, and unselective on the left column indicate the selectivity (15%, 50%, and 85%, respectively) of the predicates on the *Protein* table, and the labels on the top row indicate the selectivity of predicates on the *Interaction* table. For each combination of selectivity, we evaluate the performance of all 9 methods for every ranking scheme (Freq, Domain, and Rare). For these experiments, we produced only the top-10 results for the methods that performed top-k optimization.

We first compare the performance of the non top-k methods (SQL, Full-Top, and Fast-Top). The SQL method, unsurprisingly, performs very poorly because of the overhead of issuing many complex SQL queries. Fast-Top outperforms Full-Top as expected for most medium and unselective queries, because the effect of pruning in Fast-Top. Surprisingly, however, Full-Top outperforms Fast-Top for selective and some medium selective queries. The reason for this phenomenon is that, for selective queries, the overhead of issuing queries to check for pruned topologies outweighs the benefit of using a smaller topology table (since the selective predicates enable Full-Top to scan only a small part of the *AllTops* table). Nevertheless, Fast-Top offers a more stable performance than Full-Top across different selectivities.

We now compare the performance of the top-k methods (Full-Top-k, Fast-Top-k, Full-Top-ET, and Fast-Top-ET). Fast-Top-k in general outperforms Full-Top-k, except for selective predicates, for similar reasons as in the previous section; hence, the benefits of Fast-Top-k are greater for top-k queries because it limits the number of SQL sub-queries that Fast-Top-k has to issue. Full-Top-k-ET and Fast-Top-k-ET perform very well for unselective queries (exactly where Full-Top-k and Fast-Top-k do not perform very well) due to the early termination optimizations. However, they perform very poorly for selective queries because of the overhead of DGJ operators (we actually show the best and worst plans for this case, corresponding to different choices of DGJ operator implementations). Again, Fast-Top-k-ET dominates Full-Top-k-ET in most cases, except for selective predicates.

Finally, we study the effectiveness of the optimization methods (Full-Top-k-Opt and Fast-Top-k-Opt). As shown, Fast-Top-k-Opt (and similarly, Full-Top-k-Opt) almost always makes the right choice between the Fast-Top-k plan (chosen for selective predicates) and the best Fast-Top-k-ET plan (chosen for medium and unselective predicates). This provides users with the “best of both worlds” across various predicate selectivities. In general, Fast-Top-k-Opt is the preferred strategy because it works well for a wide range of selectivities. Although Full-Top-k-Opt sometimes dominates Fast-Top-k-Opt for selective queries, the difference in absolute time is not much (about 3 seconds). Hence, Full-Top-k-Opt appears to be the more robust and efficient method.

6.2.3 *Experiments with larger path length*

We also experimented with 4-topologies (i.e., each path may relate up to 5 nodes). Table 3 shows the space overhead and query performance of Fast-Top-k-Opt (the best) approach. As shown, the query performance and space overhead are comparable to 3-topologies.

However, there are some interesting issues that arise with large l. First, as l increases, the quality of topologies is often diluted by what we call *weak* relationships. For example, one of the most common paths of length 4 is Protein-DNA-Protein-UniGene-DNA. In this path, the first protein is linked to another protein (third node) through a DNA sequence (second node). Further, the middle protein is also linked to a UniGene cluster (which represents a certain gene) that contains an EST (which is a short DNA sequence sampled from the corresponding region along the DNA). However, the first protein and the EST sequence (last node) are most likely unrelated. Such weak relationships are of limited interest to biologists. Further, such relationships also dilute the biological semantics of meaningful topologies, especially when the weak relationship has multiple instances. As an illustration, consider a protein and a DNA sequence that are related by instances of three paths: Protein-DNA-Protein-DNA, Protein-Interaction-Protein-DNA and (the weak relationship) Protein-DNA-Protein-UniGene-DNA. From the biological point of view, the interesting topologies are those that are like the one in Figure 16. However, the presence of many Protein-DNA-

	interaction	*selective*			*medium*			*unselective*		
protein		Freq	Domain	Rare	Freq	Domain	Rare	Freq	Domain	Rare
	SQL	774.3	787.1	784.9	817.3	832.7	822.7	843.1	850.7	836.9
	Full-Top	0.10	0.12	0.11	0.18	0.17	0.17	2.00	3.79	2.62
	Fast-Top	5.56	6.08	5.76	5.84	5.85	5.82	6.13	6.09	6.10
	Full-Top-k	0.075	0.075	0.078	0.156	0.178	0.15	0.237	0.267	0.274
selective	Fast-Top-k	3.95	3.88	3.90	5.848	5.77	5.86	6.105	6.08	6.056
	Full-Top-k-ET	39.42	39.01	38.96	39.19	38.88	38.96	39.23	39.68	38.95
	Fast-Top-k-ET	9.65/2467	10.55/2438	8.96/2540	8.79	10.31	8.26	8.84	10.89	8.82
	Full-Top-k-Opt	0.08	0.08	0.08	0.17	0.16	0.15	0.229	0.243	0.23
	Fast-Top-k-Opt	3.92	3.90	3.91	5.85	5.63	5.77	6.055	6.12	6.076
	SQL	386.7	240.6	783.6	1012	1097	1325	996.5	309.7	794.9
	Full-Top	32.75	39.76	32.88	28.31	47.16	45.92	29.00	28.98	91.23
	Fast-Top	8.06	8.08	7.93	9.84	10.05	9.85	10.88	10.99	10.94
	Full-Top-k	27.20	27.69	27.07	28.95	28.9	29.64	28.92	29.5	29.20
medium	Fast-Top-k	6.21	6.17	6.10	9.83	9.84	9.75	10.88	10.97	10.89
	Full-Top-k-ET	40.47	4.16	28.17	45.33	3.56	28.54	46.0	6.71	2.30
	Fast-Top-k-ET	5.94	4.712	1.615	6.63	5.72	1.96	6.81	5.91	2.19
	Full-Top-k-Opt	27.6	27.3	31.9	30.15	8.20	4.32	47.2	6.85	2.28
	Fast-Top-k-Opt	6.22	6.14	6.20	8.23	5.83	2.02	6.79	6.12	2.31
	SQL	239.9	234.5	187.0	6673	1117	355.0	12549	1182	300.7
	Full-Top	31.78	32.88	36.34	44.12	45.92	45.31	53.66	91.23	56.54
	Fast-Top	18.60	20.23	18.58	34.75	34.89	34.62	44.43	44.66	44.57
	Full-Top-k	30.28	30.71	30.31	44.35	44.85	44.72	52.03	73.32	52.57
unselective	Fast-Top-k	18.77	18.88	18.82	35.19	35.29	35.05	44.69	47.37	44.46
	Full-Top-k-ET	29.74	24.16	26.3	51.85	6.34	3.01	52.01	8.23	3.80
	Fast-Top-k-ET	5.04	4.19	1.64	5.53	5.68	2.13	9.39	5.724	1.68
	Full-Top-k-Opt	29.8	25.6	26.35	51.84	6.41	3.45	51.9	8.31	3.91
	Fast-Top-k-Opt	5.34	4.12	1.78	5.65	5.88	2.31	10.0	5.73	1.57

Table 2: Performance of Different Strategies

interaction	*selective*			*medium*			*unselective*			*space overhead*	
protein	Freq	Domain	Rare	Freq	Domain	Rare	Freq	Domain	Rare	Table	Size
selective	10.5	10.9	10.3	10.9	10.7	10.4	11.8	11.7	11.25	AllTops	186M
medium	13.6	6.38	5.89	17.4	5.84	2.86	8.01	6.4	2.01	LeftTops	18M
unselective	7.68	4.21	0.78	5.97	4.88	2.10	12.6	5.29	1.54	ExcpTops	1.5M

Table 3: Space Overhead and Query Performance (Fast-Top-k-Opt) of 4-topology Data

Protein-UniGene-DNA interacts with the other paths and splits the interesting topology into four topolgies, as shown in Figure 17. Consequently, weak relationships degrade the quality of meaningful topologies.

Second, the intrinsic complexity of computing topologies makes it hard to compute topologies involving weak relationships. For instance, Protein-DNA-Protein-UniGene-DNA has about 600 million instances (because it often connects unrelated end points), and even for a given pair of end points, weak relationships have up to 5000 instances relating the end points. Consequently, it is very expensive to compute topologies involving such paths (which may interact with other paths that have tens of thousands of instances). Note that this is an *intrinsic* complexity of the problem because any solution for topology search has to compute these paths to check whether any one of them interacts with other paths. While any online query evaluation method would have to compute these paths on the fly, we incur this computational overhead during the precomputation phase (when generating topologies), but provide good pruning and performance during query time. Nevertheless, computing such topologies is computational expensive (it took us more than a day to generate topologies for $l = 4$ involving weak relationships).

Given the above analysis, we believe that weak relationships have a detrimental effect on topology search. Consequently, one solution is to use domain knowledge to prune such weak topologies. In Appendix B, we describe weak relationships in Biozon that can potentially be pruned.

6.2.4 Summary of Other Results

We also ran other experiments by varying the parameter k (for top-k queries). Since the results are similar, except for a slight degradation in performance with increasing k, they are omitted here. We also evaluated the performance of retrieving instances of topologies, and found that it ranges from 1-50 seconds depending on the frequency of the topology.

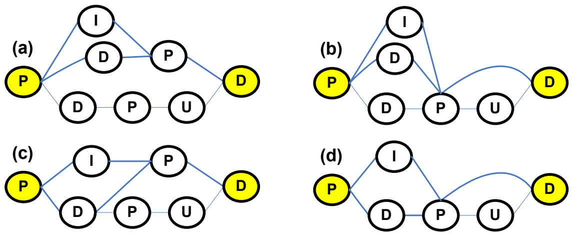


Figure 17: A weak relationship Protein-DNA-Protein-UniGene-DNA dilutes the meaningful topology in Figure 16. P stands for Protein, D for DNA, U for Unigene, and I for Interaction.

7. RELATED WORK

In the biological data domain, topologies is a new concept. Existing methods for querying biological data available on the web are usually limited to the one entity type warehoused in the database being queried. There are a few servers that allow one to query multiple databases at once, such as the NCBI entrez server (*http://www.ncbi.nlm.nih.gov/gquery*) and the EMBL server (*http://www.ebi.ac.uk/services/*). However, these servers do not integrate the results, nor do they analyze the relations between the objects. As such, they are oblivious to data topologies. There also exist systems, like

Biomediator [26], Moby [30], DiscoveryLink [18], Kleisli [12] and others that use a mediated schema and/or wrappers to distribute queries amongst different sources, integrating the information in a mediated middle layer. However, they do not support topology searches.

There have been recent work on keyword search over databases, such as BANKS [7], DataSpot [15], Lore [28], DBXplorer[6] and DISCOVER [5, 20, 19]. However as mentioned in the introduction, these techniques only return "isolated" paths between entities. Further, they do not summarize various similarly related entities at the schema level. ObjectRank[5] supports schema level summaries but it still only considers isolated paths between entities. None of the above work summarizes *all* the rich relationships among entities at the schema level in the form of topologies, which is one of the main contributions of our work. Another point of distinction is that the previous strategies only support "pure" keyword search while our system can support a mix of keyword search and structured queries. There has also been a lot of recent work on top-k queries (e.g., [27, 23, 29]), but these techniques do not consider topology results. For a more detailed discussion of top-k evaluation techniques in relational databases, please see Section 5.2.

8. CONCLUSION AND FUTURE WORK

We have introduced the notion of a data topology, which captures and summarizes relationships between interrelated entities, and shown how (ranked) topology search can be efficiently implemented in a biological database. We have also identified some usability and performance issues that arise when dealing with large topologies containing weak relationships, and have proposed solutions to this problem by exploiting domain knowledge.

We believe that our current work on topologies is only a first step toward a more general study of relationships between entities. Possible directions for future work include extensions to support multiple end-points in a topology, primitives for comparing topologies across multiple queries, and alterative topology formulations for dealing with weak relationships. We also believe that the notion of topology is relevant not only to biological databases, but may also be applicable to the social and physical sciences.

9. ACKNOWLEDGEMENTS

We thank Aaron Birkland for his help and support with the Biozon database and for critically reading the manuscript.

10. REFERENCES

[1] *http://www-306.ibm.com/software/data/db2/udb/v82/*.

[2] *http://www.microsoft.com/sql/default2.mspx*.

[3] http://www.sciencemag.org/cgi/content/short/307/5710/651e. *Science*, 307 (5710):651e, 2005.

[4] A.Bairoch and R.Apweiler. The swiss-prot protein sequence data bank and its supplement trembl in 1999. *Nucl. Acids Res.*, 1999.

[5] A.Balmin, V.Hristidis, and Y.Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *VLDB*, 2004.

[6] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.

[7] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, 2002.

[8] A. Birkland and G. Yona. Biozon: a hub of heterogeneous biological data. *Nucleic Acids Research*, 34:1–8, 2006.

[9] A. Birkland and G. Yona. Biozon: a system for unification, management and analysis of heterogeneous biological data. *BMC Bioinformatics*, 7:70–, 2006.

[10] M. Carey and D.Kossmann. On saying "enough already!" in sql. In *SIGMOD*, 1997.

[11] M. Carey and D.Kossmann. Reducing the braking distance of an sql query engine. In *VLDB*, 1998.

[12] J. Chen, S. Chung, and L. Wong. The kleisli query system as a backbone for bioinformatics data integration and analysis. In *Managing Scientific Data*, pages 147–187. Morgan Kaufmann, 2003.

[13] C.Li and S. K.C. Chang, I.F. Ilyas. Ranksql: query algebra and optimization for relational top-k queries. In *VLDB*, 2003.

[14] D.A.Benson, M. Boguski, D. Lipman, J. Ostell, B.F.Ouellette, B. A. B.A. Rapp, and D. Wheeler. Genbank. *Nucl. Acids Res.*, 1999.

[15] S. Dar, G. Entin, S. Geva, and E. Palmon. Dtl's dataspot: Database exploration using plain language. In *VLDB*, 1998.

[16] D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of top n queries. In *VLDB*, 1999.

[17] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient. In *ICDE*, 1993.

[18] L. Haas, P. Schwarz, P. Kodali, E. Kotlar, J. Rice, and W. Swope. Discoverylink: a system for integrated access to life sciences data sources. *IBM Systems Journal*, 40(2):489–511, 2001.

[19] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, 2003.

[20] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, 2002.

[21] I. Ilyas, W.G.Aref, and A. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, 2003.

[22] I. Ilyas, W.G.Aref, J.S.Vitter, and A. Elmagarmid. Rank-aware query optimization. In *SIGMOD*, 2004.

[23] A. Natsev, Y. Chang, J. Smith, C. Li, and J. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, 2001.

[24] P.G.Selinger, M.M.Astrahan, D.D.Chamberlin, R. Lorie, and T.G.Price. Access path election in a relational database management system. In *SIGMOD*, 1979.

[25] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In *SIGMOD*, 1992.

[26] P.Mork, A. Halevey, and P. Tarczay-Hornoch. A model for data integration systems of biomedical data applied to online genetic databases. In *AMIA Annual Symposium*, 2001.

[27] R.Fagin, A.Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[28] R.Goldman, N.Shivakumar, S.Venkatasubramanian, and H. Garcia-Molina. Proximity search in database. In *VLDB*, 1998.

[29] M. Theobald, G. Weikum, and R. Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB*, 2004.

[30] M. Wilkinson and M. Links. Biomoby: an open-source biological web services proposal. In *Briefings in Bioinformatics*, 2002.

[31] G. Yona, N. Linial, and M. Linial. Protomap: Automatic classification of protein sequences, a hierarchy of protein families, and local maps of the protein space. *Proteins: Structure, Function and Genetics*, 37.

APPENDIX

A. PROOF OF THEOREM 2

Given the statistics in Section 5.4.3, we need to determine np_i, nc_i and ec_i ($1 \leq i \leq m$). We first calculate the formulae for some useful intermediate parameters, before computing the desired formulae.

LEMMA 1. *Let x_i ($1 \leq i \leq n$) be the probability of an input tuple of opr_i (in the outer input) being a result of the plan. We then have:*
(1) $x_{n+1} = 0$;
(2) $x_i = \sum_{j=1}^{s_iN_i}\{\rho_i^j(1-\rho_i)^{s_iN_i-j} \cdot (1-(1-x_{i+1})^j)\}$ where $1 \leq i \leq n$:

PROOF. Since (1) is obvious, we only prove (2). Consider an input tuple u of opr_i (in the outer input). In expectation, u will join with s_iN_i inner input tuples. The probability of exactly j of these inner tuples satisfying the local predicate of opr_i is α_j=$\rho_i^j(1-\rho_i)^{s_iN_i-j}$. Since the probability of each one of the inner tuples producing a result for the plan is x_{i+1}, ζ_j=$1-(1-x_{i+1})^j$ is the probability that at least one of the j tuples will produce a result for the plan. Thus, x_i = $\sum_{j=1}^{s_iN_i}(\alpha_i \times \zeta_i)$ which is formula (2). $\square$

LEMMA 2. *Let δ_i ($1 \leq i \leq n$) be the expected cost of index probes for one input tuple of opr_i (in the outer input) that is* not *a result. We then have:*
(1)$\delta_{n+1} = 0$;
(2)$\delta_i = \sum_{j=0}^{s_iN_i}\{\rho_i^j(1-\rho_i)^{s_jN_j-j} \times [I_i + j \cdot \delta_{i+1}]\}$ where $1 \leq i < n$.

PROOF. Since (1) is obvious, we only prove (2). Consider an input tuple u of opr_i (in the outer input). In expectation, u will join with s_iN_i inner input tuples. The probability of exactly j of these inner tuples satisfying the local predicate of opr_i is α_j=$\rho_i^j(1-\rho_i)^{s_iN_i-j}$. Thus, the cost of index probes for u is ζ_j=$I_i + j\delta_{i+1}$, where I_i is the cost of a local index probe for u, and $j \cdot \delta_{i+1}$ is the cost of index probes for the j tuples in the other operators. Thus, δ_i = $\sum_{j=1}^{s_iN_i}(\alpha_i \times \zeta_i)$, which is formula (2). $\square$

Using Lemmas 1 and 2, we can compute x_i and δ_i ($1 \leq i \leq n$) efficiently using dynamic programming. We now show how we can use these values to compute np_i, nc_i and ec_i (Section 5.4.2).

THEOREM 2. *$\forall i$. $1 \leq i \leq m$: $np_i = (1-x_1)^{Card_i}$*

PROOF. From Lemma 1, x_1 is the probability of an input tuple (to the lower-most operator) being a result. Therefore, the probability of not finding any result from the group g_i is $(1-x_1)^{Card_i}$. $\square$

THEOREM 3. *$\forall i$. $1 \leq i \leq m$: $nc_i = (1-x_1)^{Card_i} \times Card_i \times \delta_1$*

PROOF. Since np_i is the probability of the plan not finding any result from group g_i, and the expected cost of not finding a result in group g_i is $Card_i \times \delta_1$, nc_i = $np_1 \times (Card_i \times E[Y])$ = $(1-x_1)^{Card_i} \times Card_i \times \delta_1$. $\square$

THEOREM 4. *$\forall i$. $1 \leq i \leq m$. ec_i=$EC_{Card_i}^{1:n}$, where:*
(1) $EC_h^{n+1:n} = 0$;
(2) $EC_h^{l:n} = \sum_{j=1}^{h}\{\rho_l(1-x_l)^{j-1} \cdot [(j-1)\delta_l + I_l + EC_{s_lN_l}^{l+1:n}]\}$ where $1 \leq l < n$.

PROOF. Intuitively, $EC_h^{p:q}$ is the expected cost of the plan with opr_p...opr_q ($1 \leq p, q \leq n$) to find the first result from h input tuples. Since (1) is obvious, we only prove (2). Assume that the plan of opr_l,...,opr_n finds a result after processing j input tuples (of opr_l). The probability of this case is α_j=$\rho_l(1-x_l)^{j-1}$, where $(1-x_l)^{j-1}$ is the probability that the first $j-1$ tuples did not produce a result (Lemma 1), and ρ_l is the probability that the j^{th} tuple is a result. The total expected cost of the $j-1$ tuples that did not produce a result is $(j-1)\delta_l$ (Lemma 2). The cost of processing the j^{th} tuple is I_l (the index lookup cost) plus $EC_{s_lN_l}^{l+1:n}$ (because the tuple joins with s_lN_l inner input tuples of opr_l, and these tuples have to be processed by the higher operators). Thus, the total expected cost for this case is ζ_j=$(j-1)\delta_l + I_l + EC_{s_lN_l}^{l+1:n}$. Hence, $EC_h^{l:n} = \sum_{j=1}^{h}(\alpha_j \times \zeta_j)$. $\square$

Relationship	Explanation
DUP	related but weaker than DP
PFP	related/remotedly related (homologous proteins)
PUP	related/remotedly related
PFPD	related/remotedly related
FWF	weak relation (pathway context)
DUPU	remotedly related or completely unrelated
PUPU	remotedly related or completly unrelated
PDP	likely to be unrelated (functionally)
FWFP	likely to be completly unrleated

Table 4: Relationships that can give rise to weak relationships if repeated multiple times

B. WEAK RELATIONSHIPS IN BIOZON

The interest in data topologies emerges from our interest in detecting new biological phenomena. However, as the length of paths l increases (with $l >= 4$) and the size of the topologies grows, it becomes more difficult to discern the truly interesting topologies from *weak topologies*. Weak topologies are subgraphs that contain *weak relationships*, i.e., relationships that most likely connect remotely related entities. Such relationships are typically formed by repeating certain indirect relationships (Figure 4).

For example, the paths PFP or PUP form a relation between two similar (and likely homologous) proteins. Thus, any path that extends on these and relate another entity (say a DNA) to one of the proteins, also relates the same entity to the other protein. However, this relation is not direct and if repeated multiple times, the relation between the two end entities becomes a weak relationship that is less and less reliable, to the point that the resulting topologies have no biological significance. This problem is especially evident for multi-domain proteins and is aggravated by the presence of chance similarities, and transitive relations between remote entities that utilize these paths might end up connecting completely unrelated entities [31].

A different kind of example for a weak relationship arises by extending the path P-D-P. This path represents two distinct proteins (the first and the third nodes) that are encoded by the same DNA sequence (the second node). Some of the DNA sequences are long (including the complete sequences of certain chromosomes or even complete genomes) and encode for multiple proteins, and hence this sub-graph is very common. However, the two proteins might be at far apart locations and completely unrelated functionally. Combining this path with others can create weak relationships that are of limited interest to biologists. For example, when combining PDP with P-U-D the result is one of the most common paths of length 4, P-D-P-U-D. However, the first protein and the EST sequence (last node) are most likely unrelated (note that the second and last nodes do not represent the same instance). Similarly, when combining the relationships PDP and PFP the result is a weak relationship P-D-P-F-P that most likely link two unrelated proteins (the two end nodes).