Extracting and Analyzing Hidden Graphs from Relational Databases

Konstantin Xirogiannopoulos, Amol Deshpande

University of Maryland

Alaa Jarbony
Introduction

**Graph Analytics:**

- Leveraging of connection between entities in a network towards gaining insight about said entities via the use of graph Algorithms

- Analyzing interconnection structures among underlying entities in a dataset through the use of graph analytics can provide tremendous value in many application domains
Graphs are not the primary representation choice for storing most data today.

A large fraction of the data of interest initially resides in relational database systems (or similar structured storage systems like key-value stores).

Relational databases typically include many useful relationships between entities and can contain many hidden, interesting graphs.

- e.g.: **DBLP** (Digital Bibliography and Library Project) dataset
  - A user may want to construct a graph with the authors as the nodes and may create an edge between two authors if they:
    - Co-authored a paper
    - Co-authored a paper recently
    - Co-authored multiple papers together
    - Co-authored a paper with very few additional authors
    - Attended the same conference
Currently a user who wants to explore such structures in an existing database is forced to ETL:

- Manually Extract data from data stores by formulating the right SQL queries
- Write scripts to Transform the results into the format required by some graph database system
- Load the data into it and then write and execute the graph algorithms on the loaded graphs

- costly, labor-intensive and cumbersome process

- In many cases, the extracted graphs can be significantly larger than the initial input stored in database, making it infeasible to construct or analyze such graphs in memory
GraphGen

- The system aims to make it easy for users to extract a variety of different types of graphs from a relational database and execute graph analysis tasks or algorithms over them in memory.
- Supports an expressive Domain Specific Language (DSL) based on Datalog.
- Uses a translation layer to generate the appropriate SQL queries to be issued to the database.
- Creates efficient in-memory representation of the graph.
- Supports general-purpose Java Graph API.
- Supports vertex-centric API for specifying analysis tasks like PageRank.
The main scalability challenge in extracting graphs from relational tables:

*The graph that the user is interested in analyzing may be too large to extract and represent in memory, even if the underlying relational data is small*

- The space explosion because of the types of large-output joins that are often needed when constructing these graphs

- **GraphGen** offers a condensed representation which is more efficient both in terms of **memory requirement** and **extraction time**
The relational model already provides such condensed representation for such queries that we call **C-DUP**

**The key challenge is not generating the representation, but rather dealing with duplicate paths between two nodes**
System Architecture

- Abstraction layer accepts a graph extraction task and constructs the queried graph(s)
- Domain Specific Language (DSL): user specifies how to construct the nodes and the edges of the graph
- Supports direct manipulation of the graph using the Java Graph API
- Supports a vertex-centric API
- Serialized graph output created for other libraries (python wrapper library)
Sure, here is the text:

Ensures that the total size of the graphs (in the *memory-efficient representation*) is less than the total amount of memory available.
Declarative Graph Extraction Language

**Datalog-based DSL:**

- A declarative language based on Datalog to intuitively express the graphs users are interested in extraction
- User specifies the relations and attributes to define the **Nodes** and **Edges** for the graph to be extracted
- User can specify either one or a collection of graphs to be extracted

**Query to extract a single co-authored graph:**

\[
\text{Nodes}(ID, Name): \neg \text{Author}(ID, Name).
\]

\[
\text{Edges}(ID1, ID2): \neg \text{AuthorPub}(ID1, PubID), \text{AuthorPub}(ID2, PubID).
\]
Parse the Datalog query and create a set of SQL queries to execute against the database

The specifics depend on the nature of the Edges query

- **Case 1:** Each of the Edges statements corresponds to an *acyclic, aggregation-free query*.
  - We may load a condensed representation of the graph into memory.

- **Case 2:** At least one Edges statement violates the above condition
  - We create a single SQL statement to construct the edges and execute it to load the expanded graph in memory.

  Aggregate query to connect authors if they have published at least 5 papers together:

\[
\begin{align*}
\text{Nodes}(ID, Name) & : = \text{Author}(ID, Name). \\
\text{Edges}(ID1, ID2) & : = \text{AuthorPub}(ID1, PubID), \text{AuthorPub}(ID2, PubID), \text{COUNT}(PubID) \geq 5.
\end{align*}
\]
Analyzing the extracted Graph

**Basic Data Structure:** two mutable *ArrayLists* for each node for its *incoming* and *out-going* edges.

- The most efficient means to utilize GraphGen is to directly operate on the graph using:

  - *Java Graph API:*
    - `getVertices()`: returns an iterator over all vertices in the graph
    - `getNeighbors(v)`: returns an iterator over the neighbors of v
    - `existEdge(v, u)`: returns true if there is an edge between the two vertexes
    - `addEdge(v, u), deleteEdge(v, u), addVertex(v), DeleteVertex(v)`: allow for manipulating the graph
Vertex-centric API:

- Allows users to implement a `COMPUTE()` function and execute it against the extracted graph regardless of its in-memory representation

Users simply need to:

- Implement the `Executor` interface which contains a single method definition for `Compute()`
- Instantiate `Executor`
- Call the `Run()` method of the `VertexCentric` coordinator object with the `Executor` object as an input
Analyzing the extracted Graph

- **graphgenpy Library:**
  - Python wrapper over *GraphGen* allowing users to run queries in DSL through simple python scripts
  - Serialize the resulting graphs in standard graph format
  - Open up analysis to any graph computation framework
The key efficiency challenge with extracting graphs from relational databases is that:

- In most cases, queries for extracting explicit relationships between entities from a relational dataset requires expensive large-output joins

- The extracted graph may be much larger than the input size itself

- **Solution:** maintaining and operating upon the extracted graph in a condensed fashion.

- The solution guarantees that the condensed representation requires at most as much memory as loading all the underlying tables in worst case.
We will:

• Describe the basic condensed representation (C- DUP)
• Present a general algorithm for constructing such representation
• Propose a series of in-memory variations of the basic condensed representation that handle duplications (DEDUP-1, DEDUP-2, BITMAP-1, BITMAP-2)
• Present algorithms for constructing de-duplicated representations
Let $G = (V, E)$ the output expanded graph

We say $G_C(V', E')$ equivalent C-DUP representation if and only if:

1. $\forall u \in V$, $\exists u_s, u_t \in V'$ (real nodes), remaining nodes in $V'$ are called virtual nodes
2. $G_C$ is DAG
3. In $G_C$ there are no incoming edges to $u_s$ $\forall u \in V$ and no output edges from $u_t$ $\forall u \in V$
4. $\forall (u \rightarrow v) \in E$, there is at least one directed path from $u_s$ to $v_t$
Extracting a condensed graph (C-DUP)

- The key idea is to postpone certain joins

- **Step 1:** Translate the \textit{Nodes} statement into SQL queries and execute them against the database to load the nodes into memory.
  - We assume for every node we have two copies $u_s$ and $u_t$ but physically we store one copy

```sql
Nodes(ID, Name):-Author(ID, Name).
select A.id as ID, A.name as Name from Author;
```
Extracting a condensed graph (C-DUP)

**Step 2:** each *Edge* statement can be represented as:

- **Edges**\((ID1, ID2)\): \(R_1(ID1, a_1), R_2(a_1, a_2), ..., R_n(a_{n-1}, ID2)\)
  \(R_{i(1 \leq i \leq n)}\): database table, \(a_{i1 \leq i \leq n-1}\): join attribute

- **Large-output join:** \(R_i(a_{i-1}, a_i) \Join_{a_i} R_{i+1}(a_i, a_{i+1})\)

\[
If \frac{|R_i||R_{i+1}|}{d} > 2(|R_i| + |R_{i+1}|)

\[d = \text{number of distinct values for } a_i\]

- This formula assumes that the join attribute is uniformly distributed and may miss a large-output join and could be easily substituted with a more sophisticated selectivity estimator
Extracting a condensed graph (C-DUP)

• **Step 3:** Construct an SQL query for each subsequence of the relations *without a large-output join* and execute it against the database.
  - Let \( \{a_l, a_m, ..., a_u\} \) denote the *join attributes* which are marked as *large-output*
  - Then, the queries we execute are:
    - \( \text{res}_1(ID1, a_l) : - R_1(ID1, a_1), ..., R_l(a_{l-1}, a_l) \)
    - \( \text{res}_2(a_l, a_m) : - R_{l+1}(a_l, a_{l+1}), ..., R_m(a_{m-1}, a_m) \)
    - ...
    - \( \text{res}_{k+1}(a_u, ID2) : - R_{u+1}(a_w, a_{u+1}), ..., R_n(a_{n-1}, ID2), \quad k = |\{a_l, a_m, ..., a_u\}| \)
Extracting a condensed graph (C-DUP)

- **Step 4:**
  - \( \forall \text{attr} \in \{a_l, a_m, \ldots, a_u\} \), we create a set of virtual nodes corresponding to all possible values \( \text{attr} \) takes.

- **Step 5:**
  - \( \forall (x, y) \in res_1, \text{add a directed edge from a real node to a virtual node} (x_s \rightarrow y) \)
  - \( \forall (x, y) \in res_i, \text{add an edge between two virtual nodes} (x \rightarrow y), \quad 2 \leq i \leq k \)
  - \( \forall (x, y) \in res_{k+1}, \text{add an edge from a virtual node to a real node} (x \rightarrow y_i) \)

- **Step 6:**
  - for a virtual node if \(|\text{in}| - |\text{out}| \leq (\text{in} + \text{out} + 1)\), remove it and add directed edges from its in-neighbors to its out-neighbors.
Nodes(ID, Name): -Author(ID, Name). →
select A.id as ID, A.name as Name from Author;

Edges(ID1, ID2): -AuthorPub(ID1, PubID),
AuthorPub(ID2, PubID).

E

Query 1: Res1(ID1, PubID): -AuthorPub(ID1, PubID)
Query 2: Res2(PubID, ID2): -AuthorPub(ID2, PubID)

\[ |R_1||R_2| > 2(|R_1| + |R_2|) \]

Query 1 (edges _s → virtual nodes)
select distinct A.aid as X1, A.pid as pubID from AuthorPub A
Query 2 (edges virtual nodes → _t)
select distinct B.aid as X2, B.pid as pubID from AuthorPub B;
C-DUP Condensed Duplication Representation

**Duplication Problem**

- C-DUP representation allows for *multiple paths* between $u_s$ and $v_t$
- A real node may encounter the same real neighbor more than once
- An algorithm whose correctness depends solely on the connectivity structure of the graph can be executed directly
- C-DUP causes correctness issues on all non duplicate-insensitive graph algorithms
Single-layer vs Multi-layer Condensed graph:

- A condensed graph may have one or more layers of virtual nodes
- A condensed graph is called **multi-layer** if it contains a directed path of length > 2

Multi-layer graph:
C-DUP:

- Initial extracted representation from the relational database
- Suffers from the edge duplication problem
- Can be utilized by employing a naïve solution to deduplication using hashset
  - `getNeighbors(u)`:  
    - returns all the real nodes (target nodes) reachable from $u_s$
    - keeps track of which neighbors have already been seen (in a hashset) and skips over them if the neighbor is seen again
- Does not require any preprocessing overhead
- The execution penalty is very high due to hash computations
**EXP- Fully Expanded Graph:**

- Constructed by creating all the direct edges between all the real nodes in the graph and removing all the virtual nodes.
- Has a much larger memory footprint than other representations due to the large number of edges.
- The most efficient representation for operating on, since iteration only requires a sequential scan over direct neighbors.
**DEDUP-1 Condensed Deduplication Representation:**

- Identical to C-DUP in the use of real and virtual nodes
- Does not suffer from duplicated paths
- Does not require the *on-the-fly* deduplication used in C-DUP (*hashset computation*)
- It usually results in a larger number of edges than C-DUP
- Sits in the middle of the spectrum between EXP and C-DUP in terms of both memory efficiency and iteration performance
- **Trade-off:** one-time cost of removing duplication
**BITMAP - Deduplication using Bitmaps:**

- Using **bitmaps** for filtering out duplicated paths between nodes
- A virtual node $v$ may be associated with a set of bitmaps indexed by the **ID’s of the real nodes**
- The size of each bitmap is equal to the number of the **outgoing edges from $v$**
- Considered a depth first traversal starting at $u_s$, we check to see if there is a bitmap corresponding to $u_s$:
  - **if not:** traverse each of the outgoing edges in turn
  - **else:** consult the bitmap to decide which of the outgoing edges to skip:
    - If the corresponding bit set to 1, traverse the edge
    - else, skip that edge

- **Drawbacks:**
  - **memory overhead** and **complexity** of storing bitmaps
  - bitmaps make it **less portable** to systems outside GraphGen
**DEDUP-2: Optimization for Single-layer Symmetric Graphs**

- This representation can significantly reduce the memory requirements for dense graphs for the special case:
  - **single-layer symmetric** condensed graph \( (u_s \rightarrow v_t) \rightarrow (v_s \rightarrow u_t) \)
- For virtual node \( v \) if \( u_s \rightarrow v \) then \( v \rightarrow u_t \)
  - We can omit the target nodes and associated edges
Preprocessing & Duplication

*Processing and deduplication algorithms:*

- **Input:** C-DUP representation in-memory
- **Output:** Condensed Deduplication representation
Prepressing for BITMAP-1:

- **goal:** associate and initialize bitmaps with the virtual nodes to avoid visiting the same real node twice while iterating over the out-neighbors

- **Algorithm:**
  1. Iterate over all the real nodes in turn
  2. For each such node $u$, initiate a depth-first traversal from $u_s$
  3. Keep track of all the real nodes visited during the process using a hashset $H_u$
  4. For each virtual node $v$ visited check if it is the penultimate layer, If so:
     - add a bitmap of size equal to the number of outgoing edges from $v$
     - check for each outgoing edge $v \rightarrow w_t$ if $w_t \in H_u$
       - if so, set the corresponding bit to 0 (if we traverse this edge we will visit a real node that already have been visited)
       - else, set it to 1 (if we traverse this edge we will visit a new real node reachable from $u_s$) and add $w_t$ to $H_u$

- **Complexity:** $O(n_r \cdot d^{k+1})$
  - $n_r = \text{number of real nodes}$
  - $k = \text{number of layers of virtual node}$
  - $d = \text{maximum degree of any node}$
Prepressing for BITMAP-1:

Algorithm BITMAP-1

1: procedure BMP1(graph)
2:    seen ← hashSet()
3:    for each real node \( m \) in graph.vertices() do
4:       for each virtual node \( vn \) in rn.getOutNeighbors() do
5:          for each index, real node \( rn_2 \) in vn.getOutNeighbors() do
6:             if \( rn_2 \notin seen \) then
7:                set bit at index of bitmap vn.bitMaps.get(rn)
8:                seen.add(rn2)
9:          seen.clear()
Prepressing for BITMAP-2:

- This algorithm is based on the standard **greedy algorithm** for **set cover**
- Associates bitmaps with the virtual nodes in **each layer**

```plaintext
Algorithm BITMAP-2
1: procedure BMP2(graph, ordering)
2:   srted ← graph.vertices.sortByDuplication(ordering)
3:   seen ← hashSet()
4:   for each real node rn in srted do
5:     virtSet ← greedySetCover(rn)
6:     for each virtual node v ∈ virtSet do
7:       for each index, real node rn2 in v.getOutNeighbors() do
8:         if rn2 ∉ seen then
9:           v.getBitmap(rn).setBitAt(index)
10:          seen.add(rn2)
11:       else
12:         chosen ← false
13:         for each bitmap bmp in v.getBitmaps() do
14:           if bmp.getBitFor(rn) == 1 then
15:             chosen ← true
16:             break
17:         if !chosen then
18:           v.removeBitmapFor(rn)
19:           removeEdge(rn,v)
20:       v.rebuildBitmapIndex()
21:   seen.clear()
```

- **Complexity:** $O(n_r * d^2k)$
  - $n_r =$ number of real nodes
  - $k =$ number of layers of virtual node
  - $d =$ maximum degree of any node
Deduplication for DEDUP-1:

- **goal**: modify the initial C-DUP graph to reach a state where there is at most one unique path between any two real nodes in the graph
- *Single-layer* condensed graph
- *Multi-layer* condensed graph (for future work)
  - use **BITMAP-2** algorithm instead

Virtual Nodes First Algorithm:

1. Start with a graph containing only the real nodes
2. Add virtual nodes one at a time, always ensuring that the partial graph remains free of any duplication
• When adding a virtual node $v$: $O(n_v)$
  
  ➢ Collect all the virtual nodes $R = \{ r_i \mid I(v) \cap I(r_i) \neq \emptyset \}$ $O(d^2)$
  
  $I(v) = \{ \text{real nodes that points to } v \}$
  
  $O(v) = \{ \text{real nodes that } v \text{ points to} \}$

  ➢ Maintain a processed set which keeps track of the virtual nodes that have been added to the current partial graph

  ➢ $\forall r_i \in R \cap \text{processed}$, if $|O(v) \cap O(r_i)| > 1$ $O(d^3)$

    ▪ Select a real node $x \in O(v) \cap O(r_i)$ at random $O(1)$
    
    ▪ Choose either to remove the edge $(v \rightarrow x)$ or $(r_i \rightarrow x)$ depending in the in-degrees (choose lower-degree virtual node) $O(1)$
    
    ▪ Add direct edges to $x$ to compensate the removal of the edge $O(d)$

      ▪ Suppose we remove $(v \rightarrow x)$ then, add direct edges to $x$ from all the real nodes in $I(v)$, while checking to make sure that $x$ is not already connected to those nodes through other virtual nodes
      
      ▪ add $v$ to processed set. $O(1)$
      
      ▪ Consider the next virtual node

• **Complexity:** $O\left(n_v(d^2 + d^4)\right) = O(n_v \cdot d^4)$

  ▪ $n_v = \text{number of virtual nodes}$
  
  ▪ $d = \text{maximum degree of any node}$
All the experiments were run on a single machine with **24 cores** running at **2.20GHz** and with **64G of RAM**

- **Compression Performance:** *(small Datasets)*
  - VMiner (Virtual Node Miner): graph compression algorithm
**Graph Algorithm Performance: (Small Datasets)**

- **DBLP** and **Synthetic_1** datasets portray a large gap in performance compared to **EXP**
  - This is because these datasets consist of a large number of virtual nodes need to be iterated
  - This also the reason why **DEDUP-1** and **BITMAP-2** typically perform better, they feature a smaller number of virtual neighbors per real node than **C-DUP, BITMAP-1** and **DEDUP-2**
### Large Datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>CDUP</th>
<th>BMP-DEDUP</th>
<th>EXP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mem (GB)</td>
<td>Mem (GB)</td>
<td>Mem (GB)</td>
</tr>
<tr>
<td>Layered_1</td>
<td>1.421</td>
<td>2.737</td>
<td>&gt;64</td>
</tr>
<tr>
<td>Layered_2</td>
<td>1.613</td>
<td>2.258</td>
<td>19.798</td>
</tr>
<tr>
<td>Single_1</td>
<td>1.276</td>
<td>1.493</td>
<td>1.2</td>
</tr>
<tr>
<td>Single_2</td>
<td>9.901</td>
<td>13.042</td>
<td>&gt;65</td>
</tr>
<tr>
<td>TPCH</td>
<td>0.023</td>
<td>0.049</td>
<td>7.398</td>
</tr>
</tbody>
</table>

Comparing the performance (running times in seconds, and memory consumption in GB) of C-DUP, BITMAP, and EXP on large datasets; the table also shows the time required for bitmap de-duplication (DNF → *did not finish* in reasonable time).
• GraphGen is a system that enables users to analyze the interconnection structures between entities in a relational databases

• GraphGen can interoperate with a variety of graph analysis libraries and supports a standard graph API

• We present a series of in-memory condensed representations and deduplication algorithms

• The choice of which representation to use depends on the specific application scenario, and can be made at a per dataset or per analysis level
THANKS FOR LISTENING, ANY QUESTIONS?