Distributed Evaluation of Subgraph Queries Using Worst-Case Optimal Low-Memory Dataflows

Khaled Ammar, Frank McSherry, Semih Salihoglu, Manas Joglekar

Presented by: Ramya Nagarajan
Spring 2019
Agenda

- Motivation
- Existing Approaches
- Dataflow Primitive
- Contributions
- Implementation
- Evaluation
- Further Work
Motivation

○ Subgraph queries are a fundamental computation performed by many applications
  □ Clique-finding for related page detection
  □ Diamond-finding for social network recommendation systems
○ Efficiency and scalability as primary goals
○ Linear use of memory, worst-case optimal computation and communication costs
Contributions

- **BigJoin**
  - Distributed algorithm for static graphs
  - Achieves a subset of theoretical guarantees
- **Delta-BiGJoin:**
  - Distributed algorithm for dynamic graphs
  - Achieves same theoretical guarantees in insertion-only workloads
- **BigJoin-S:**
  - Distributed algorithm for static graphs
  - Achieves all theoretical guarantees
  - Notable theoretical guarantee: balances work-load across distributed workers on arbitrary inputs instances
Existing Approaches

○ Distributed Approaches:
  □ Edge-at-a-time
  □ Variants of Shares or Hypercube

○ Serial Approaches:
  □ Vertex-at-a-time
Edge-at-a Time Approaches

- Treat query subgraph as a relational query
- Execute series binary joins to determine result
- Provably worst suboptimal:
  - Worst-case $O(IN^2)$ computations

```plaintext
open-tri(a_1,a_2,a_3):=edge(a_1,a_2),edge(a_2,a_3)
tri(a_1,a_2,a_3):=open-tri(a_1,a_2,a_3),edge(a_3,a_1)
```
Shares Algorithm

- Given a distributed cluster with $w$ workers, $n$ relations, $m$ attributes (i.e. $n$ edges, $m$ vertices)
- Divides the output space equally over $w$ workers
- Replicates edge tuples and distributes each tuple to every worker that can produce an output depending on tuple
- Workers run local join algorithm on received input
- Improved communication and computation costs
- Super-linear cumulative memory growth
Vertex-at-a-Time Approaches

○ Generic Join:
  □ Starts by finding all $a_1$ vertices that will end up in output
  □ Then $(a_1, a_2)$, etc.
Generic Join Algorithm

○ Global Attribute Ordering
○ Extension Indices
  □ $a_1...a_m$ subsets in queries
  □ Maps to $j_1...j_m$ subset
○ Prefix Extension Stages
  □ Iteratively compute result of $Q$ when each relation is restricted to the first $j$ attributes in common global order
Dataflow Primitive

- Starts with a collection of $P_j$ tuples stored across $w$ workers
- Produces the $P_{j+1}$ tuples across the same workers
- 4 steps:
  - Initialization
  - Count Minimization
  - Candidate Proposal
  - Intersection
Dataflow Primitive

- **Initialization:**
  - Tuples of $P_j$ are distributed amongst workers arbitrarily
  - Each prefix transformed into a triple:
    - (prefix, smallest candidate set size, index of relation with that number of candidates)
○ **Count Minimization:**
  □ Workers exchange triples
  □ Place each triple at the worker with access to the corresponding extension set
  □ Each triple per worker updates its extension set
  □ Final result if collection of triples indicating the prefix relations with the fewest extensions
Dataflow Primitive

- Candidate Proposal
  - Produce triple (p, min-c, min-i)
  - Each extension e of P
- Intersection
  - Workers exchange candidate tuples for each relation
Contributions

- BigJoin: distributed algorithm for static graphs
  - Achieves a subset of theoretical guarantees
- Delta-BiGJoin:
  - Distributed algorithm for dynamic graphs
  - Achieves same theoretical guarantees in insertion-only workloads
- BigJoin-S:
  - Distributed algorithm for static graphs
  - Achieves all theoretical guarantees
  - Notable theoretical guarantee: balances work-load across distributed workers on arbitrary inputs instances
BiGJoin: Joins on Static Relations

- Used for evaluating queries on static graphs
- Steps:
  - Arbitrarily order attributes
  - Build indices over each relation for each prefix in global order
  - Assemble dataflows for extending each $P_j$ to $P_{j+1}$ for each attribute $a_i$
BiGJoin Analysis

- $O(mnMaxOut_Q)$ communication and computation costs
  - Equal to GJ
- Cumulative Memory Required:
  - $O(m\text{IN} + mB)$
- Good work-load balance across workers
- No guaranteed workload balance on adversarial inputs
Delta-BiGJoin: Joins on Dynamic Relations

○ Delta-GJ Algorithm
  □ Query Q
  □ For each relation R, have some change to the deletion or addition of records in that relation
  □ New delta query for each relation
    ○ Assume that tuples in record are labeled s.t can tell inserted records apart from existing records
  □ Union of delta queries results in correct output query
Consider the following $n$ delta queries:

\[
\begin{align*}
    dQ_1 &:= \Delta R_1, R_2, R_3, \ldots, R_n \\
    dQ_2 &:= R'_1, \Delta R_2, R_3, \ldots, R_n \\
    dQ_3 &:= R'_1, R'_2, \Delta R_3, \ldots, R_n \\
    \vdots \\
    dQ_n &:= R'_1, R'_2, R'_3, \ldots, \Delta R_n
\end{align*}
\]
Delta-BiGJoin Analysis

- Communication and computation cost: $O(mn^2 + \text{MaxOut}_Q)$
- Cumulative Memory: $O(m\text{NIN}(z) + mB)$
- Rounds of Computation: $O\left(\frac{(mn^2\text{MaxOut}_Q)}{B'} + zmn^2\right)$
BiGJoin-S

○ Sources of Imbalance:
  □ Sizes of extension indices
    ○ A single worker stores the entire extension set for a given prefix
  □ Number of Proposals
    ○ Imbalanced amount of candidate extensions to prefixes
  □ Number of Index Lookups
    ○ If many prefixes originate from the same relation R, there can be an imbalance in the number of prefixes and extensions each worker receives
BiGJoin-S

- Handling Skew
  - Skew-Resilient Indices
  - Modified Dataflow Primitive
    - Extension-Resolve
    - Intersect
    - Count
    - Balance
BiGJoin-S

- Skew-resilient Extension Indices
  - Split extension indices across workers
  - Count Index
  - Extension Resolver Index
  - Original Extension Index
○ Extension-Resolve
  □ In Big-Join:
    ○ Pass (p, k) pair to extension resolver
    ○ Receive candidate extension in return
  □ Skew in number of prefixes an extension has
□ Big-JoinS:
  ○ Locally aggregate extension requests made to a certain relation for a certain (p, k)
  ○ Send only one version of this request
BiGJoin-S

○ Intersect
  □ Big-Join:
    ○ Each (p, e) is routed through each of the Extension sets in order
  □ Big-JoinS:
    ○ Distributed lookup of (p, e) by sending to the worker that holds the Extension set for (p, e)
BiGJoin-S

- **Balance**
  - Skew: Imbalance in the amount of work each worker receives after count minimization
  - Each worker deterministically distributes its amount of work amongst all workers
Theorem 3.4. Suppose $B' \geq \max\{w, \log(IN \times MaxOut_Q)\}$ and let $B = wB'$. Then BiGJoin-S has the following costs:

- Cumulative computation and communication cost of $O(mn \cdot MaxOut_Q)$ and memory cost of $O(mnIN + mB)$.
- $O\left(\frac{mn \cdot MaxOut_Q}{B}\right)$ rounds of computation.
- With at least probability $1 - O\left(\frac{1}{IN}\right)$, each worker performs $O(B')$ communication and computation in each round of the algorithm. In MPC terms, the load of BiGJoin-S is $O\left(\frac{mnIN}{w} + mB'\right)$, so assuming $B' < \frac{IN}{w}$, BiGJoin-S has optimal load.
Evaluation

○ Evaluate triangle finding on standard graphs on different systems
  □ Establish a baseline for running time
○ Implementation scaling
  □ Vary number of workers across single machine and multiple machines
  □ 64 billion-edge graph
○ Evaluate BigJoin and Delta-BiGJoin
○ Batch size of 10,000
Experimental Setup

Table 1: Graph datasets used in our experiments.

<table>
<thead>
<tr>
<th>Name</th>
<th>Vertices</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiveJournal (LJ) [36]</td>
<td>4.8M</td>
<td>68.9M</td>
</tr>
<tr>
<td>Twitter (TW) [35]</td>
<td>42M</td>
<td>1.5B</td>
</tr>
<tr>
<td>UK-2007 (UK) [35]</td>
<td>106M</td>
<td>3.7B</td>
</tr>
<tr>
<td>Common Crawl (CC) [60]</td>
<td>1.7B</td>
<td>64B</td>
</tr>
</tbody>
</table>
COST

- Number of cores that the algorithm needs to outperform an optimized single-threaded version
Evaluation against Frameworks

- EmptyHeaded
  - Highly-optimized shared-memory parallel system
  - Evaluating subgraph queries on static graphs using GJ

<table>
<thead>
<tr>
<th>Query</th>
<th>EH-R</th>
<th>EH-I</th>
<th>BiGJoinT-R</th>
<th>BiGJoinT-I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangle-LJ</td>
<td>1.2s</td>
<td>150.3s</td>
<td>6.5s</td>
<td>1.9s</td>
</tr>
<tr>
<td>Diamond-LJ</td>
<td>31.7s</td>
<td>150.3s</td>
<td>712.3s</td>
<td>1.9s</td>
</tr>
<tr>
<td>Triangle-TW</td>
<td>213.8s</td>
<td>4155s</td>
<td>588s</td>
<td>34.4s</td>
</tr>
</tbody>
</table>
Evaluation against Frameworks

- Arabesque
  - Distributed system specialized in finding subgraphs

<table>
<thead>
<tr>
<th>Query</th>
<th>Arbsq-R</th>
<th>Arbsq-I</th>
<th>BiGJoinT-R</th>
<th>BiGJoinT-I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangle</td>
<td>69.0s</td>
<td>1.46B</td>
<td>3.4s</td>
<td>38M</td>
</tr>
<tr>
<td>4-clique</td>
<td>273.7s</td>
<td>18.7B</td>
<td>21.8s</td>
<td>350M</td>
</tr>
</tbody>
</table>
Future Work

- Improving skew resilience of BigJoin
- Utilizing symmetries of queries
- Practical algorithms that have better than worst-case optimality