CONNECTED DATA

Pushing the Envelope of Data Management Systems

Marco Serafini
UNIVERSITY OF MASSACHUSETTS AMHERST
CONNECTED DATA

• Entities + Relationships
• Each entity can have an arbitrary number of relationships
  • Extreme skew: huge variance in number of relationships per entity
• Relationships are added on the fly
SOCIAL NETWORKS
KNOWLEDGE GRAPHS
RECOMMENDATIONS & PERSONALIZATION
DATA LINAGE / PROVENANCE
LEARNING OVER CONNECTED DATA

• Leverage structural properties of data
MODELING CONNECTED DATA

GRAPH VS. RELATIONAL
GRAPH vs. RELATIONAL DATA MANAGEMENT
CONVENTIONAL WISDOM

“You should not reinvent the wheel”

“When you have a hammer everything looks like a nail”
A PRAGMATIC APPROACH

• It is **not** about graph vs. relational data

• It is about graph vs. relational **workloads**
  • Diverse applications and algorithms
  • Diverse data structures and APIs

• Graph DBMSs should **extend not reinvent**
  • Eventual convergence of implementations is possible and desirable
OPEN ISSUE: REAL-TIME

- **Real-time** analytics and queries on **dynamic** graphs
  - User likes product $\rightarrow$ gets real-time contextual recommendations
  - Failure/attack on system $\rightarrow$ immediate reaction
  - Fraud is attempted $\rightarrow$ blocked before financial loss

- **Challenges**
  - Graph algorithms are complex
  - Hopping edges requires random access
  - Sophisticated indexing, compression, and partitioning works only on read-only data
OPEN ISSUE: SCALE-UP ANALYTICS

• Advanced graph analytics are hard to scale out
  • Impossible to cleanly partition

• SIMD hardware offers massive scale-up parallelism
  • E.g. GPUs, Intel AVX, Intel Phi

• Challenge: hard to leverage SIMD for graph algorithms
  • Same problems as before: random access, poor caching, branching, …
  • But on an even larger scale
VISION

Graph Storage

Execution runtime

Logical graph exploration plan

Transactions, snapshots

Pattern matching

Graph mining

Graph learning

Graph data structure

Physical execution
STORING CONNECTED DATA
AN EVOLUTIONARY APPROACH
VISION

- Pattern matching
- Graph mining
- Graph learning

Logical graph exploration plan

Execution runtime

- Physical execution
- Transactions, snapshots

Graph storage

Graph data structure
RELATIONAL MODEL

• Connected data = dynamic relationships
  • New relationships among entities added all the time
  • Extreme skew: variance in # of relationships per entity

• Needed: flexible physical schema
  • Avoid frequent schema changes!

• Solution: Entity table + Relationship table

<table>
<thead>
<tr>
<th>Entity ID</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source Entity ID</th>
<th>Destination Entity ID</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ENTITY (VERTEX)  RELATIONSHIP (EDGE)
WORKLOADS

• Pattern/path based queries
  • Pattern queries
  • Reachability
  • Random walks

• Subgraph-based queries
  • Frequent subgraphs
  • Densest subgraphs

• Frontier-based queries
  • Shortest path

• Message passing
  • PageRank

Fundamental operation:

EDGE TRAVERSAL, that is,
JOINS ON EDGE TABLE

Hop = Join
Hash-Joining Edge Table

- **Build**: hash table from edge table
- **Probe**: Scan through partial results and join/extend
- Typically, after the join scan (traverse) the joined edges

### Partial Results
(e.g. partial query match)

<table>
<thead>
<tr>
<th>Source ID</th>
<th>Dest 1</th>
<th>Prop 1</th>
<th>...</th>
<th>...</th>
<th>Dest n</th>
<th>Prop n</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Source ID
(e.g. add vertices to partial match)

<table>
<thead>
<tr>
<th>Source ID</th>
<th>Dest 1</th>
<th>Prop 1</th>
<th>...</th>
<th>...</th>
<th>Dest k</th>
<th>Prop k</th>
</tr>
</thead>
<tbody>
<tr>
<td>v2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ADJACENCY LIST REPRESENTATION

- Adjacency lists = edge table optimized for joins
- **Graph storage systems**: optimized for adjacency lists

<table>
<thead>
<tr>
<th>Vertex index</th>
<th>Source ID</th>
<th>Dest 1</th>
<th>Prop 1</th>
<th>…</th>
<th>…</th>
<th>Dest n</th>
<th>Prop n</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>v1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v2</td>
<td>v2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
REAL-TIME WORKLOADS

• Real-time workloads
  • Dynamic data: Entities and relationships are added continuously
  • Queries and analytics on real-time data

• Examples: monitoring, real-time recommendations

• Graph storage requirements
  • Low-latency concurrent (transactional) updates
  • Low-latency reads from graph snapshots
TYPICAL PIPELINE

**UPDATES**

TRANSACTIONAL SYSTEM

DYNAMIC DATA STRUCTURE + TRANSACTIONS
E.g.: B-Tree, LSMT

LOAD

ANALYTICAL SYSTEM

READ-ONLY DATA STRUCTURE NO TRANSACTIONS
E.g.: CSR

RESULTS
LIVEGRAPH

REAL-TIME GRAPH
DATA MANAGEMENT

Updates
Real time queries
Snapshot queries

Results

TRANSACTIONAL
EDGE LOG
LIVEGRAPH

• Features
  • Embedded graph store
  • ACID transactions
  • Real-time reads on the live data (no data loading)
  • Snapshot isolation: wait-free reads
  • Multi-versioned (temporal/incremental queries)

• Key design choices
  • Sequential adjacency list scans
  • Fast insertions in constant time
DATA STRUCTURE COMPARISON

• B+ Trees
  • LMDB
  • Typical RDBMS data structure

• Log-Structured Merge Trees
  • RocksDB
  • Skip-list + compressed runs

• Linked lists
  • Neo4J

• Transactional Edge Log
GRAPH REPRESENTATIONS

Input graph

CSR (read-only)

Linked list

B+ tree

LSMT
MICRO-BENCHMARK

• Seek & scan adjacency list
  • Seek: find adjacency list
  • Scan: get next edge in the adjacency list

• Data: Kronecker graph that fits in memory of one socket
EDGE SCAN

Graph scale, V

ns/edge (scan)

TEL
B+Tree
LSMT
Linked List
BENEFITS OF SEQUENTIAL SCANS

• Better locality
  • Cache utilization

• Sequential execution flow
  • Leverages CPU pipelining and prefetching
  • Reduces the likelihood of branch mispredictions

• Huge gap between pointer-based and sequential data structures

• Total latency improvement
  • 20× over LSMT
  • 18× over linked list
  • 4.5× over B+ tree.
TRANSACTIONAL EDGE LOG

- Fixed-size “dynamic” array
  - Adapts to skew
- Append-only log
  - $O(1)$ insertion
  - Multi-versioning: snapshots and temporal analytics

<table>
<thead>
<tr>
<th>HEADER</th>
<th>PROPERTIES of edge $&lt;V_s, V_D&gt;$ (version $T_1$)</th>
<th>PROPERTIES of edge $&lt;V_s, V_D&gt;$ (version $T_2$)</th>
<th>PROPERTY ENTRIES</th>
<th>EDGE LOG ENTRIES</th>
</tr>
</thead>
<tbody>
<tr>
<td>fixed-size</td>
<td>size = $S_1$</td>
<td>size = $S_2$</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**FORMAT:** EDGE LOG ENTRY
- Destination vertex ID
- Creation TS
- Invalidation TS
- Properties size
TRANSACTIONS + SEQUENTIAL SCANS

- Reads do not need locks
- Writes: double timestamps
  - Atomic timestamp access
  - 64-bit cache-aligned words

**Diagram Description**

**W1**: get lock (in index)

**W2**: append new entry with CreationTS = -TID

**W3**: scan from tail and search for previous version of <V_S, V_D>

**W4**: set InvalidationTS = -TID (atomically)

**R1**: get read timestamp TRE from transaction manager (assume TRE > T1)

**R2**: scan from tail

**R3**: read entry, since TRE > T1 and InvalidationTS either NULL or negative

**R4**: InvalidationTS = -TID (atomically)
**TRANSACTIONAL WORKLOAD**

- LinkBench benchmark
  - Facebook’s back-end graph storage workload
  - RocksDB: Facebook’s back-end storage

- LiveGraph is a good match for latency-sensitive workloads
  - Sub-millisecond tail latency

<table>
<thead>
<tr>
<th>Storage</th>
<th>Optane SSD</th>
<th>NAND SSD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LiveGraph</td>
<td>RocksDB</td>
</tr>
<tr>
<td>mean</td>
<td>0.0450</td>
<td>0.1278</td>
</tr>
<tr>
<td>P99</td>
<td>0.2598</td>
<td>0.6423</td>
</tr>
<tr>
<td>P999</td>
<td>0.9800</td>
<td>3.5190</td>
</tr>
</tbody>
</table>

**Latency (ms)**
# FRONT-END WORKLOAD

- Nano-second latencies!

<table>
<thead>
<tr>
<th>Storage</th>
<th>Optane SSD</th>
<th>NAND SSD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LiveGraph</td>
<td>RocksDB</td>
</tr>
<tr>
<td>mean</td>
<td>0.0039</td>
<td>0.0328</td>
</tr>
<tr>
<td>P99</td>
<td>0.0065</td>
<td>0.0553</td>
</tr>
<tr>
<td>P999</td>
<td>0.6763</td>
<td>4.8716</td>
</tr>
</tbody>
</table>

**Latency (ms)**
SCALABILITY
REAL-TIME ANALYTICS

- LDBC Social Network Benchmark (SNB), in-memory
  - Short reads, transactional updates (possibly involving multiple objects)
  - Complex reads: multi-hop traversals and analytical processing including filters, aggregations, and joins

<table>
<thead>
<tr>
<th>System</th>
<th>LiveGraph</th>
<th>Virtuoso</th>
<th>PostgreSQL</th>
<th>TigerGraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex-Only</td>
<td>9,106</td>
<td>292</td>
<td>3.79</td>
<td>185</td>
</tr>
<tr>
<td>Overall</td>
<td>9,420</td>
<td>259</td>
<td>52.4</td>
<td></td>
</tr>
</tbody>
</table>

Throughput (ops/s)
TRUE REAL-TIME

- Interactive/web analytics must be in the millisecond range!

<table>
<thead>
<tr>
<th>System</th>
<th>LiveGraph</th>
<th>Virtuoso</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex read 1</td>
<td>7.00</td>
<td>23,101</td>
<td>371</td>
</tr>
<tr>
<td>Complex read 13</td>
<td>0.53</td>
<td>2.47</td>
<td>10,419</td>
</tr>
<tr>
<td>Short read 2</td>
<td>0.22</td>
<td>3.11</td>
<td>3.31</td>
</tr>
<tr>
<td>Updates</td>
<td>0.37</td>
<td>0.93</td>
<td>2.19</td>
</tr>
</tbody>
</table>

Average request latency (ms)
VERTEX-CENTRIC COMPUTATION

- Comparison between
  - Running in-database computation with LiveGraph
  - Export to Gemini, dedicated system using compressed read-only storage (CSR)
- Longer running time but no data export delay

<table>
<thead>
<tr>
<th>System</th>
<th>LiveGraph</th>
<th>Gemini</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETL</td>
<td>-</td>
<td>1520</td>
</tr>
<tr>
<td>PageRank</td>
<td>266</td>
<td>156</td>
</tr>
<tr>
<td>ConnComp</td>
<td>254</td>
<td>62.6</td>
</tr>
</tbody>
</table>

Running time (ms)
FUTURE WORK

- Scale out to distributed system
- Multi-hop locality/partitioning
- Improved property indexing
QUERYING CONNECTED DATA

CPU-EFFICIENT PHYSICAL EXECUTION
VISION

Pattern matching

Graph mining

Graph learning

Logical graph exploration plan

Execution runtime

Physical execution

Transactions, snapshots

Graph data structure

Graph Storage
GRAPH PATTERN QUERIES

• Each “hop” is a join in the edge table
• Many graph queries are multi-hop
• This makes query optimization hard
  • Cardinality estimation gets harder at every join
  • Skew: few vertices have very high degree
  • Large intermediate results (e.g. structural or point-to-point path queries)
WORST-CASE OPTIMALITY (WCO)

• WCO: query complexity is the same as the size of the results
  • Example: triangle query should have complexity $O(|E|^{3/2})$

• Multi-way joins
  • Extend partial match by one vertex (not edge) at a time
  • Perform two joins at once

• Set intersection
## Set Intersection Bottleneck

- Set intersection dominates running time
  - Frequent comparisons $\rightarrow$ frequent branch mispredictions
  - Need to fetch lots of data to cache $\rightarrow$ poor caching

<table>
<thead>
<tr>
<th>$v_i$</th>
<th>1</th>
<th>13</th>
<th>32</th>
<th>143</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_j$</td>
<td>4</td>
<td>5</td>
<td>13</td>
<td>43</td>
<td>143</td>
</tr>
</tbody>
</table>

- FIND

[Diagram showing nodes $v_i$ and $v_j$ connected with an edge labeled FIND]
**VECTORIZER**

- **Goal:** optimize CPU efficiency
  - Cache efficiency: Data compression
  - Avoid branch mispredictions
  - SIMD operations

- **Dynamic data:** Cannot afford expensive pre-processing

- **Vectorizer:** On-the-fly vectorization
  - SIMD friendly data structures
  - Materialization and reuse of these data structures
  - **> 3x speedup** compared to state of the art graph tools
  - **> 10x speedup** compared to RDBMS
FREQUENT SUBGRAPH MINING

• Search for initially unknown subgraphs that turn out to be frequent

← Is a Frequent Subgraph
GRAPH EXPLORATION PROCESS

• Enumerate (& prune) embeddings
• Aggregate (e.g. count) by pattern
CHALLENGES

- Exponential number of embeddings

![Diagram showing exponential growth of unique embeddings](image)

- # unique embeddings (log-scale)
- Size of embedding
- Exponential!!!
ARABESQUE

• New execution model & system
  • Think Like an Embedding
  • Purpose-built for distributed graph mining
  • Hadoop-based

• Contributions
  • Simple & Generic API
  • High performance
  • Distributed & Scalable by design
API EXAMPLE: CLIQUE FINDING

```java
boolean filter(Embedding e) {
    return isClique(e);
}

void process(Embedding e) {
    output(e);
}

boolean shouldExpand(Embedding embedding) {
    return embedding.getNumVertices() < maxsize;
}

boolean isClique(Embedding e) {
    return e.getNumEdgesAddedWithExpansion() == e.getNumberofVertices() - 1;
}
```

Previous state of the art (Mace, centralized)

4.621 LOC
FREQUENT SUBGRAPH MINING

• First distributed implementation
• 280 lines of java code
  • … Of which 212 compute frequency metric

• Baseline (Grami): 5,443 lines of Java code
ARABESQUE ARCHITECTURE

Input
Embeddings size n

<table>
<thead>
<tr>
<th>split 1</th>
<th>split 4</th>
<th>split 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>split 2</td>
<td>split 5</td>
<td>split 8</td>
</tr>
<tr>
<td>split 3</td>
<td>split 6</td>
<td>split 9</td>
</tr>
</tbody>
</table>

Worker 1

Output
Embeddings size n + 1

<table>
<thead>
<tr>
<th>split 1</th>
<th>split 4</th>
<th>split 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>split 2</td>
<td>split 5</td>
<td>split 8</td>
</tr>
<tr>
<td>split 3</td>
<td>split 6</td>
<td>split 9</td>
</tr>
</tbody>
</table>

Next step

Previous step
KEY FUNCTIONALITIES

• Avoiding redundant work
• Compression and management of huge intermediate state
• Load balancing
• Efficient pattern aggregation
LIMITATIONS OF API

• Limited control over exploration
  • Not ideal when looking for a specific pattern

• No support for sampling/random traversals

• Related APIs
  • NScale, G-Miner, ASAP, Fractal, …

• Finding the right API is still an active research topic
PARALLEL GRAPH EXPLORATION

• Can we leverage parallel hardware like GPUs?

• Example: graph learning
  • Training uses standard GPU tools for neural networks
  • But mining graph features on GPUs is an open problem

• Challenges
  • Limited CPU-GPU bandwidth
  • Scalability to large graphs
  • Random access and skew make SIMD operations ineffective
VISION

- Pattern matching
- Graph mining
- Graph learning

Execution runtime

Logical graph exploration plan

Transactions, snapshots

Graph data structure

Real-time & long running
PICK THREE?

• Fresh results on dynamic data
• Complex data exploration
  • Random access
  • Query optimization hard
• Low-latency results
THANK YOU

MARCO SERAFININI
University of Massachusetts Amherst
marco@cs.umass.edu