Grasper
A High Performance Distributed System for OLAP on Property Graphs

Hongzhi Chen, Changji Li, Juncheng Fang, Chenghuan Huang, James Cheng, Jian Zhang, Yifan Hou, Xiao Yan

The Chinese University of Hong Kong
Outlines

Background

Motivation

System Design

Evaluation
Graph Data is Everywhere

- **Social Networks**
  - Products/Friends recommendation
  - User actions capture

- **Semantic Webs**
  - Real-Time hot-topics tracking
  - Semantic analysis/prediction

- **Biological Networks**
  - DNA sequencing
  - Diseases diagnosis

- **Financial networks**
  - Market forecasts
  - Stock analysis
Graph Data Analytics

Offline: Batch Processing for Graph Data Computation

- PageRank
- SSSP
- Connected Components
- Triangle Counting
- Graph Matching
- ...

Graph processing frameworks / engines
Graph Data Analytics

Online: Graph Querying for Real-time Analytics

Graph Setup:
create (Neo:Crew {name: 'Neo'}), (Morpheus:Crew {name: 'Morpheus'}),
(Trinity:Crew {name: 'Trinity'}), (Cypher:Crew:Matrix {name:
'Cypher'}), (Smith:Matrix {name: 'Agent Smith'}), (Architect:Matrix
{name: 'The Architect'}),
(Neo)-[:KNOWS]->(Morpheus), (Neo)-[:LOVES]->(Trinity), (Morpheus)-[:KNOWS]->(Trinity),
(Morpheus)-[:KNOWS]->(Cypher), (Cypher)-[:KNOWS]->(Smith), (Smith)-[:CODED_BY]->(Architect)

Query:
match (n:Crew)-[r:KNOWS*]-[m] where n.name='Neo' return n as
Neo,r,m

Graph analytics library and toolkit

brainnets        COMBINATORIAL_BLAS
Dracula Graph Library       GraphinnieJS
Graphology           GraphStream
Grph               iGraph
NetworkX          nvGRAPH
ScaleGraph  SNAP

Directed Graph Library

JUNG
Online: Graph Querying for Real-time Analytics

Performance Objectives:

- Low query latency
- High throughput
- Good scalability

Challenging to achieve these objectives on large graphs:

- Graph has flexible structure, no fixed schema
  - hard to store and index for querying
- Graph has diverse query complexity
  - significantly different on workloads
- One query may involve various operators with various access patterns
  - e.g., filter, traversal, aggregator)
- Graph OLAP has high costs on Net and CPU
  - complex processing logics with large portion of data
Graph Model

**Property Graph**

*Nodes*: represent entities (or objects) in the graph
- *Properties*: a set of attributes (key-value pairs)
- *Labels*: roles in a domain

*Edges*: provide directed, semantically connection between two entities.
- Also have *properties* (costs, distances, ratings, time intervals) and *labels*.

![Figure 1. An example of Property Graph.](Image)
**Query Language**

**Gremlin**

A procedural query language supported by *Apache TinkerPop*, which allows users to express queries as a set of query **steps** on a property graph.

```java
// What are the names of Gremlin's friends' friends?
g.V().has("name","gremlin").
    out("knows").out("knows").values("name")
```

```java
// What is the distribution of job titles amongst Gremlin's collaborators?
g.V().has("name","gremlin").as("a").
    out("created").in("created").
    where(neq("a")).
    groupCount().by("title")
```
Outlines

Background

Motivation

System Design

Evaluation
Performance of Some Existing Systems

Figure 2. The query latency breakdown of IC4 in LDBC benchmark.
Figure 3. The CPU and network utilization for a mixed workload formed by \{IS1-IS4\} in LDBC benchmark.
Performance of Some Existing Systems

The limitations of existing graph databases for online query.

High latency for complex analytical query (e.g., IC4 in LDBC)

- Time spent on the query steps varies significantly.
  - e.g., `hasLabel()`, `in()` took up most of the query processing time
- Due to the diverse execution logics and data access patterns of different query steps.
  - `hasLabel()`, a filter operator on nodes by labels
  - `in()`, a traversal operator on adjacent vertices

Low utilization of CPU and network

- Non-native graph storage (e.g., NoSQL or RDBMS) is unfriendly for graph querying
  - e.g., searching neighborhoods starting from vertices, path-based queries, expanding a clique, etc.
- Inefficient query execution model, one-query-one-thread mechanism
Motivation

**Design Goals**

- To propose an efficient query execution model for OLAP on graphs
  - to achieve high utilization on CPU and network

- To implement parallel processing on single complex query, while high concurrency for processing multiple queries
  - to address the diversity of graph query operators

- To avoid using external databases, integrate data store with execution engine tightly to eliminate unnecessary overheads
  - Data storage should be native for graph representation

- By leveraging **RDMA** to reduce the cost of network communication
  - Accordingly, the designs of data store and system components should be RDMA-friendly
Outlines

- Background
- Motivation
- System Design
- Benchmark
- Evaluation
Grasper: An RDMA-enabled distributed OLAP system on property graphs

- Native graph store
- Query-friendly execution model (i.e. Expert Model)
- RDMA-based concurrent query processing
- Performance v.s. state-of-the-art (Titan, JanusGraph, OrientDB, Neo4J, TigerGraph)
  - Better CPU & Net Utilization
  - Orders of magnitude speed-up
  - Higher Throughput
**System Design**

**Data Store**, divide the in-memory space into two parts

- Normal Memory, stores graph topology
- RDMA Memory, stores properties on nodes/edges as KVS

*Figure 4. Data store in Grasper.*
System Design

Data Store

- Index-free adjacency lists to support graph traversal
- RDMA-enabled KVS to achieve low-cost remote access to labels and property values.
- A graph query can be represented as:

  graph traversal + filtering on properties + other control constraints

```sparql
g.V().as('a').out('created').in('created').as('b').
  select('a','b').by('name').where('a', neq('b'))
```
System Design

Memory Layout

<table>
<thead>
<tr>
<th>Normal Mem</th>
<th>RDMA Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Store</td>
<td>Data Store</td>
</tr>
<tr>
<td>Index Buff</td>
<td>Meta Heap</td>
</tr>
<tr>
<td>Meta Heap</td>
<td>V-KVS</td>
</tr>
<tr>
<td>graph topology</td>
<td>E-KVS</td>
</tr>
<tr>
<td>index maps</td>
<td>meta data /tmp buf</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5. Memory layout on a Grasper node.

RDMA Verbs

- KVS.get() → one-sided RDMA read
- Cross-node graph traversal → one-sided RDMA write
- Query logic constraints, e.g., where(), and(), agg(), etc.

Figure 6. RDMA message dispatching in Grasper.
System Design

Query Plan Construction

Flow Type, to describe the execution flow of each query step

- to enable parallel query processing in a distributed setting

1. **Sequential**: query logic is independent, e.g., `in()`, `out()`, `has()`
2. **Barrier**: need sync before moving forward, e.g., `count()`, `max()`
3. **Branch**: can be splitted into subqueries, e.g., `or()`, `and()`, `union()`

(1) Process in parallel
(2) collect all, then go next
(3) split to sub-queries but needs sync at the endpoint
System Design

Query Plan Construction

Query Optimizer, to parse a query string into a logical execution plan in the form of a DAG.

Query: `g.V().hasKey("lang").and( in().count().is(2), out("knows").has("name", "Tom") ).has("age", 20)`

DAG of a Query
System Design

**Execution Engine - Expert Model**

*Design Philosophy*, a top-down query-specific mechanism to address the characteristics of graph OLAP

1. **adaptive parallelism control** at step-level inside each query;
2. **tailored optimizations** for various query steps according to their specific query logic and data access pattern;
3. **locality-aware** thread binding and load balancing

**Expert**: a *physical query operator* in Grasper that expertly handles the processing of one *category of steps*

- to allow fine-grained specialization for querying
- each expert maintains its own
  - opt structures (e.g., indexes, cache) if any
  - execute() function
  - routing rules for out-going msgs
System Design

Execution Engine - Expert Model

The Mechanism of Experts

1) Each node launches only one expert instance for one type – Consequently, all query data belonging to one category of query steps will be processed by its unique expert only, with shared optimizations, i.e., cache, index, etc.

2) Each expert can employ multi-threads to dynamically concurrently process the query steps with above shared optimizations

Case: 2 machines in cluster

Figure 7. (a) adaptive parallelism at step-level; (b) an expert example.
System Design

**Execution Engine - Expert Model**

*Expert pool*: formed by 22 experts currently to represent the query steps in Gremlin language semantics, *driven by* a thread pool.

<table>
<thead>
<tr>
<th>Expert</th>
<th>Query Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>g.V(), g.E()</td>
</tr>
<tr>
<td>End</td>
<td>N/A [to aggregate the final results]</td>
</tr>
<tr>
<td>Traversal</td>
<td>in, out, both, inE, outE, bothE, inV, outV, bothV</td>
</tr>
<tr>
<td>Filter</td>
<td>has, hasNot, hasKey, hasValue</td>
</tr>
<tr>
<td>Range</td>
<td>range, limit, skip</td>
</tr>
<tr>
<td>Order</td>
<td>order</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Group</td>
<td>group, groupCount</td>
</tr>
<tr>
<td>Math</td>
<td>min, max, mean,</td>
</tr>
<tr>
<td>BranchFilter</td>
<td>and, or, not</td>
</tr>
</tbody>
</table>

Table 1. The expert pool in Grasper.
System Design

**Execution Engine - Expert Model**

*Locality-Aware Thread Binding and Load Balancing*

1) To reduce the overhead brought from thread switching
2) To avoid the negative side-effects due to **NUMA** architecture
3) To achieve thread-level load balancing

![Diagram](image)

Figure 8. Core bind and load balancing in Grasper.
System Design

**Execution Engine - Expert Model**

*Work Flow:* when a query engine is launched, its expert pool will be initialized and all expert instances will be constructed and kept alive until the engine shuts down.

Figure 9. The work flow of Expert Model to process concurrent queries in Grasper.
Outlines

Background

Motivation

System Design

Evaluation
**Benchmark**

**LDBC-Social Network Benchmark**
- Interactive Complex IC1 - IC4
- Interactive Short IS1 - IS4

**Self-Proposed**
- 8 query templates for better representation of real-world workloads

<table>
<thead>
<tr>
<th>Q1</th>
<th>g.V().has([filter]).properties([property])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2</td>
<td>g.V().hasKey([filter]).hasLabel([label]).has([filter2])</td>
</tr>
<tr>
<td>Q3</td>
<td>g.V().has([filter]).in([label]).values([key]).max()</td>
</tr>
<tr>
<td>Q4</td>
<td>g.E().has([filter1]).outV().dedup().has([filter2]).count()</td>
</tr>
<tr>
<td>Q5</td>
<td>g.E().has([filter1]).not(outV([label]).has([filter2])).groupCount([key])</td>
</tr>
<tr>
<td>Q6</td>
<td>g.V().has([filter]).and(</td>
</tr>
<tr>
<td></td>
<td>out([label1]).values([key1]).min().is([predicate1]),</td>
</tr>
<tr>
<td></td>
<td>in([label2]).count().is([predicate2])</td>
</tr>
<tr>
<td></td>
<td>).values([key2])</td>
</tr>
<tr>
<td>Q7</td>
<td>g.V().has([filter1]).as(‘a’).union(</td>
</tr>
<tr>
<td></td>
<td>out([label1]),</td>
</tr>
<tr>
<td></td>
<td>out([label2]).out([label3])</td>
</tr>
<tr>
<td></td>
<td>).in([label4]).where(neq(‘a’)).has([filter2])</td>
</tr>
<tr>
<td></td>
<td>.order([property]).limit([number])</td>
</tr>
<tr>
<td>Q8</td>
<td>g.V().has([filter1]).aggregate(‘a’).in([label1]).out([label2]).</td>
</tr>
<tr>
<td></td>
<td>.has([filter2]).where(without(‘a’))</td>
</tr>
</tbody>
</table>

Table 2. The 8 queries in our benchmark.
Evaluation

Setting

- Using 10 machines, each with two 8-core Intel Xeon E5-2620v4 2.1GHz processors and 128GB of memory.
- For fair comparison, we always used 24 computing threads in each machine for all systems we compared with.

Compared Systems

- Titan [1.1.0], JanusGraph [0.3.0], Neo4j [3.5.1], OrientDB [3.0.6] and TigerGraph Developer Edition
- Try our best to tune their configuration (i.e., system parameters) to the setting that gives their best performance.

Datasets

| Dataset | |V| | |E| | |VP| | |EP| |
|---------|--------|--------|--------|--------|--------|--------|
| LDBC    | 59,308,744 | 357,617,104 | 321,281,654 | 101,529,501 |   |
| AMiner  | 68,575,021  | 285,667,220  | 291,161,548  | 120,381,452  |   |
| Twitter | 52,579,682  | 1,963,262,821 | 320,732,961  | 577,955,736  |   |

Table 3. Dataset statistics.
Evaluation

**Latency Breakdown & CPU / Net Utilization**

- **Grasper** needs only about 60ms to process the bottleneck steps (i.e, `hasLabel()`, `in()`).
- The CPU and network utilization have been significantly improved to around 95% and 380+ MB/s respectively.

---

**Figure 10.** (a) The query latency breakdown of IC4 on LDBC by Grasper; (b) CPU and network utilization of Grasper for the mixed workload {IS1-IS4}. 

---
### Query Latency

#### Table 4. Query latency (in msec) of distributed systems on 10 machines.

<table>
<thead>
<tr>
<th></th>
<th>IC1</th>
<th>IC2</th>
<th>IC3</th>
<th>IC4</th>
<th>IS1</th>
<th>IS2</th>
<th>IS3</th>
<th>IS4</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDBC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grasper</td>
<td>271</td>
<td>16.7</td>
<td>388</td>
<td>77.3</td>
<td>0.30</td>
<td>2.19</td>
<td>0.91</td>
<td>0.32</td>
</tr>
<tr>
<td>Titan</td>
<td>66985</td>
<td>13585</td>
<td>5.8E5</td>
<td>11947</td>
<td>0.71</td>
<td>25.9</td>
<td>2.88</td>
<td>1.32</td>
</tr>
<tr>
<td>J.G.</td>
<td>56206</td>
<td>9223</td>
<td>4.5E5</td>
<td>22420</td>
<td>0.83</td>
<td>14.5</td>
<td>2.99</td>
<td>1.17</td>
</tr>
<tr>
<td>AMiner</td>
<td>Q1</td>
<td>Q2</td>
<td>Q3</td>
<td>Q4</td>
<td>Q5</td>
<td>Q6</td>
<td>Q7</td>
<td>Q8</td>
</tr>
<tr>
<td>Grasper</td>
<td>0.17</td>
<td>0.42</td>
<td>17.3</td>
<td>45.2</td>
<td>104</td>
<td>28.8</td>
<td>2.32</td>
<td>4.41</td>
</tr>
<tr>
<td>Titan</td>
<td>1.07</td>
<td>12.4</td>
<td>32341</td>
<td>2.1E5</td>
<td>43809</td>
<td>234</td>
<td>9.11</td>
<td>84.08</td>
</tr>
<tr>
<td>J.G.</td>
<td>1.34</td>
<td>8.70</td>
<td>27466</td>
<td>2.4E5</td>
<td>39155</td>
<td>276</td>
<td>5.61</td>
<td>84.71</td>
</tr>
</tbody>
</table>

#### Table 5. Query latency (in msec) of single-machine systems on one machine.

<table>
<thead>
<tr>
<th></th>
<th>IC1</th>
<th>IC2</th>
<th>IC3</th>
<th>IC4</th>
<th>IS1</th>
<th>IS2</th>
<th>IS3</th>
<th>IS4</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDBC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grasper</td>
<td>1935</td>
<td>75.1</td>
<td>2550</td>
<td>223</td>
<td>0.48</td>
<td>2.51</td>
<td>1.38</td>
<td>0.13</td>
</tr>
<tr>
<td>Neo4J</td>
<td>1448</td>
<td>372</td>
<td>15042</td>
<td>293</td>
<td>20.7</td>
<td>77.6</td>
<td>16.3</td>
<td>21.7</td>
</tr>
<tr>
<td>OrientDB</td>
<td>32869</td>
<td>2140</td>
<td>20721</td>
<td>2582</td>
<td>0.91</td>
<td>25.1</td>
<td>3.46</td>
<td>1.47</td>
</tr>
<tr>
<td>T.G.(install + run)</td>
<td>46517</td>
<td>40739</td>
<td>44048</td>
<td>43685</td>
<td>37745</td>
<td>41629</td>
<td>38799</td>
<td>37708</td>
</tr>
</tbody>
</table>

Table 4. Query latency (in msec) of distributed systems on 10 machines.

Table 5. Query latency (in msec) of single-machine systems on one machine.
### Evaluation

#### Throughput

**Figure 11.** (a) Throughput on LDBC for {IS1-IS4}; (b) CDFs of Grasper’s query latency for {IS1-IS4} (using 10 machines).

**Figure 12.** (a) Throughput on AMiner for {Q1, Q2, Q6}; (b) CDFs of Grasper’s query latency for {Q1, Q2, Q6} (using 10 machines)
Evaluation

Effects of System Designs & Opt{s}

- The performance definitely not only comes from RDMA, but also other system optimizations and Expert Model.

<table>
<thead>
<tr>
<th>LDBC</th>
<th>IC1</th>
<th>IC2</th>
<th>IC3</th>
<th>IC4</th>
<th>IS1</th>
<th>IS2</th>
<th>IS3</th>
<th>IS4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grasper</td>
<td>271</td>
<td>16.7</td>
<td>388</td>
<td>77.3</td>
<td>0.30</td>
<td>2.19</td>
<td>0.91</td>
<td>0.32</td>
</tr>
<tr>
<td>w/o APC</td>
<td>469</td>
<td>24.8</td>
<td>666</td>
<td>131</td>
<td>0.51</td>
<td>3.63</td>
<td>1.43</td>
<td>0.54</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AMiner</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grasper</td>
<td>0.17</td>
<td>0.42</td>
<td>17.3</td>
<td>45.2</td>
<td>104</td>
<td>28.8</td>
<td>2.32</td>
<td>4.41</td>
</tr>
<tr>
<td>w/o APC</td>
<td>0.20</td>
<td>0.62</td>
<td>23.7</td>
<td>59.6</td>
<td>111</td>
<td>35.4</td>
<td>4.50</td>
<td>6.15</td>
</tr>
</tbody>
</table>

Table 6. Query latency (in msec) of Grasper w/ and w/o adaptive parallelism control.

<table>
<thead>
<tr>
<th>LDBC</th>
<th>IC1</th>
<th>IC2</th>
<th>IC3</th>
<th>IC4</th>
<th>IS1</th>
<th>IS2</th>
<th>IS3</th>
<th>IS4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grasper</td>
<td>271</td>
<td>16.7</td>
<td>388</td>
<td>77.3</td>
<td>0.30</td>
<td>2.19</td>
<td>0.91</td>
<td>0.32</td>
</tr>
<tr>
<td>-RDMA</td>
<td>1349</td>
<td>17.97</td>
<td>1253</td>
<td>260</td>
<td>1.04</td>
<td>2.57</td>
<td>2.06</td>
<td>1.26</td>
</tr>
<tr>
<td>-Q.Opts</td>
<td>374</td>
<td>19.39</td>
<td>558</td>
<td>81.26</td>
<td>0.31</td>
<td>2.38</td>
<td>0.93</td>
<td>0.32</td>
</tr>
<tr>
<td>-Steal</td>
<td>488</td>
<td>24.68</td>
<td>671</td>
<td>127</td>
<td>0.57</td>
<td>3.25</td>
<td>1.31</td>
<td>0.54</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AMiner</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grasper</td>
<td>0.17</td>
<td>0.42</td>
<td>17.3</td>
<td>45.2</td>
<td>104</td>
<td>28.8</td>
<td>2.32</td>
<td>4.41</td>
</tr>
<tr>
<td>-RDMA</td>
<td>0.54</td>
<td>1.18</td>
<td>21.54</td>
<td>70.47</td>
<td>222</td>
<td>30.69</td>
<td>9.09</td>
<td>6.23</td>
</tr>
<tr>
<td>-Q.Opts</td>
<td>0.17</td>
<td>4254</td>
<td>22.84</td>
<td>417</td>
<td>131</td>
<td>35.49</td>
<td>2.89</td>
<td>4.34</td>
</tr>
<tr>
<td>-Steal</td>
<td>0.23</td>
<td>0.61</td>
<td>20.91</td>
<td>57.62</td>
<td>111</td>
<td>33.44</td>
<td>4.01</td>
<td>6.07</td>
</tr>
</tbody>
</table>

Table 7. Query latency (in msec) of [Grasper-X] (using 10 machines).
Conclusion

**Grasper**

1. A high performance distributed OLAP system over graphs

2. *RDMA-enable* system design, tightly integrate the data store layer with the execution layer to achieve better performance.

3. We propose a novel *Expert Model*, which enables tailored optimizations on query steps as well as adaptive parallelism control and dynamic load balancing on runtime.
Thank You

**Grasper**

Hongzhi Chen, et al.
Email: hzchen@cse.cuhk.edu.hk

An open-source project, [https://github.com/yaobaiwei/Grasper](https://github.com/yaobaiwei/Grasper)

Husky Data Lab, CSE
The Chinese University of Hong Kong