Grasper

A High Performance Distributed System for OLAP on Property Graphs

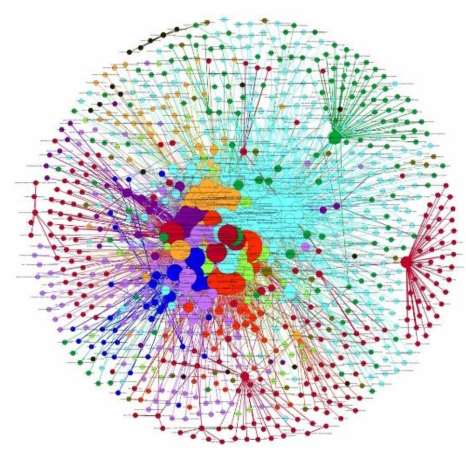
Hongzhi Chen, Changji Li, Juncheng Fang, Chenghuan Huang, James Cheng, Jian Zhang, Yifan Hou, Xiao Yan

香港中文大學
The Chinese University of Hong Kong

Outlines

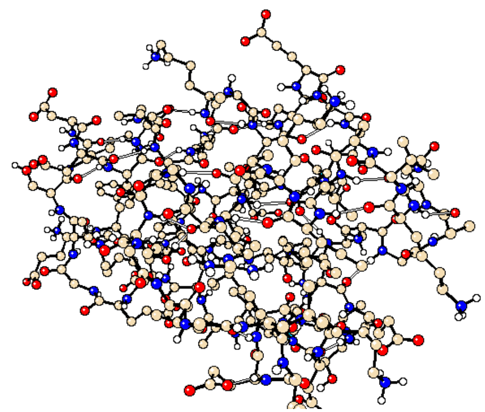
- ***Background***
- ***Motivation***
- ***System Design***
- ***Evaluation***

Graph Data is Everywhere

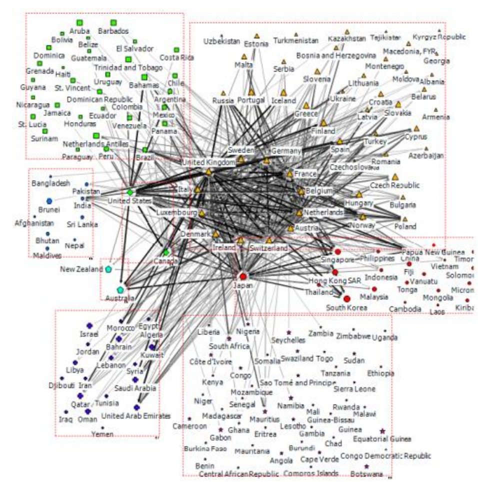


- Social Networks
  - Products/Friends recommendation
  - User actions capture

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



- Biological Networks
  - DNA sequencing
  - Diseases diagnosis

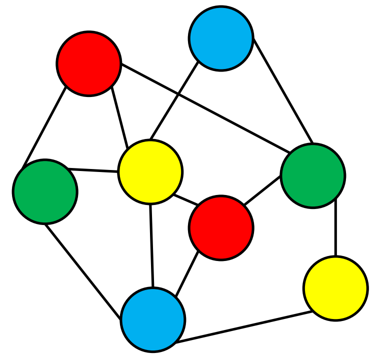


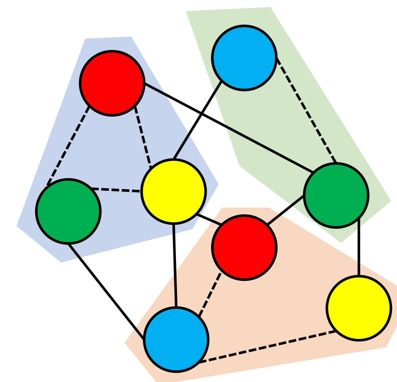
- Financial networks
  - Market forecasts
  - Stock analysis

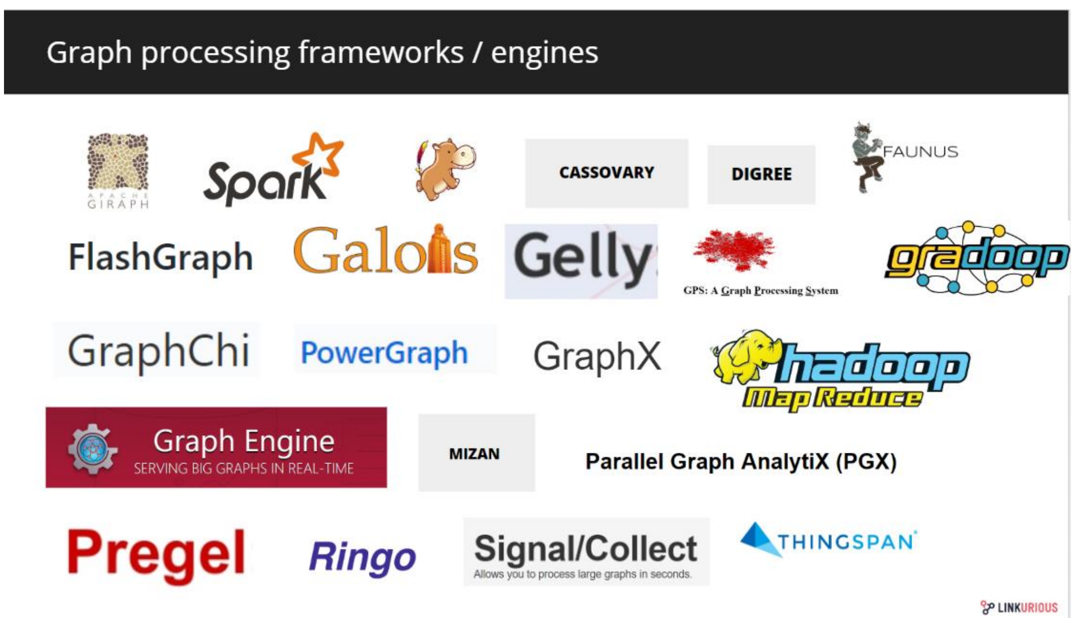
Graph Data Analytics

Offline: Batch Processing for Graph Data Computation

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







Graph 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
```

Neo	r	m
(0:Crew {name:"Neo"})	[(0)-[0:KNOWS]->(1)]	(1:Crew {name:"Morpheus"})
(0:Crew {name:"Neo"})	[(0)-[0:KNOWS]->(1), (1)-[2:KNOWS]->(2)]	(2:Crew {name:"Trinity"})
(0:Crew {name:"Neo"})	[(0)-[0:KNOWS]->(1), (1)-[3:KNOWS]->(3)]	(3:Crew:Matrix {name:"Cypher"})
(0:Crew {name:"Neo"})	[(0)-[0:KNOWS]->(1), (1)-[3:KNOWS]->(3), (3)-[4:KNOWS]->(4)]	(4:Matrix {name:"Agent Smith"})

Graph analytics library and toolkit

brainnets
COMBINATORIAL_BLAS
Directed Graph Library
Dracula Graph Library
GraphiniusJS
Graphology
GraphStream
Grph
igraph
JGraphT
JUNG
NetworKit
NetworkX
nvGRAPH
ScaleGraph
SNAP
LINKURIOUS

Graph Data Analytics

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***.

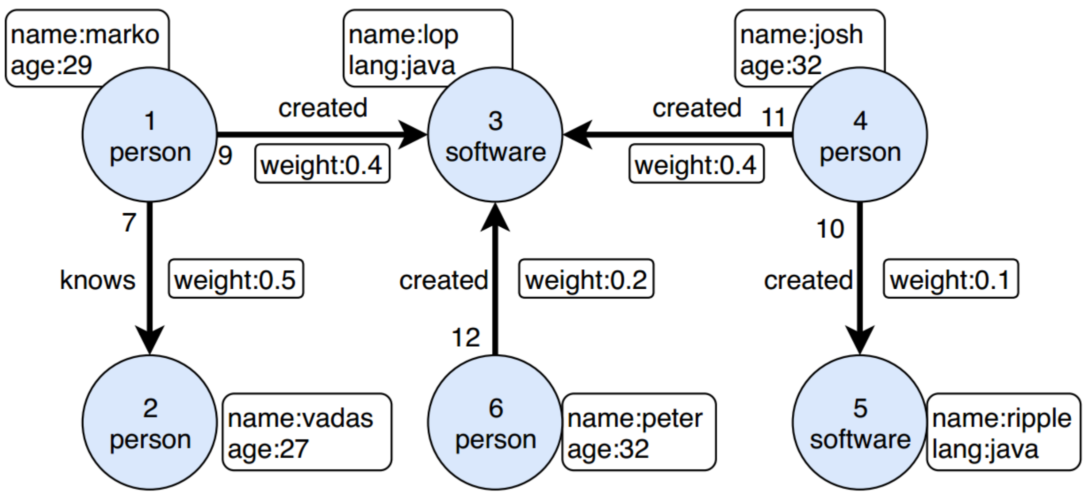


Figure 1. An example of Property Graph.

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.

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

```
// 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

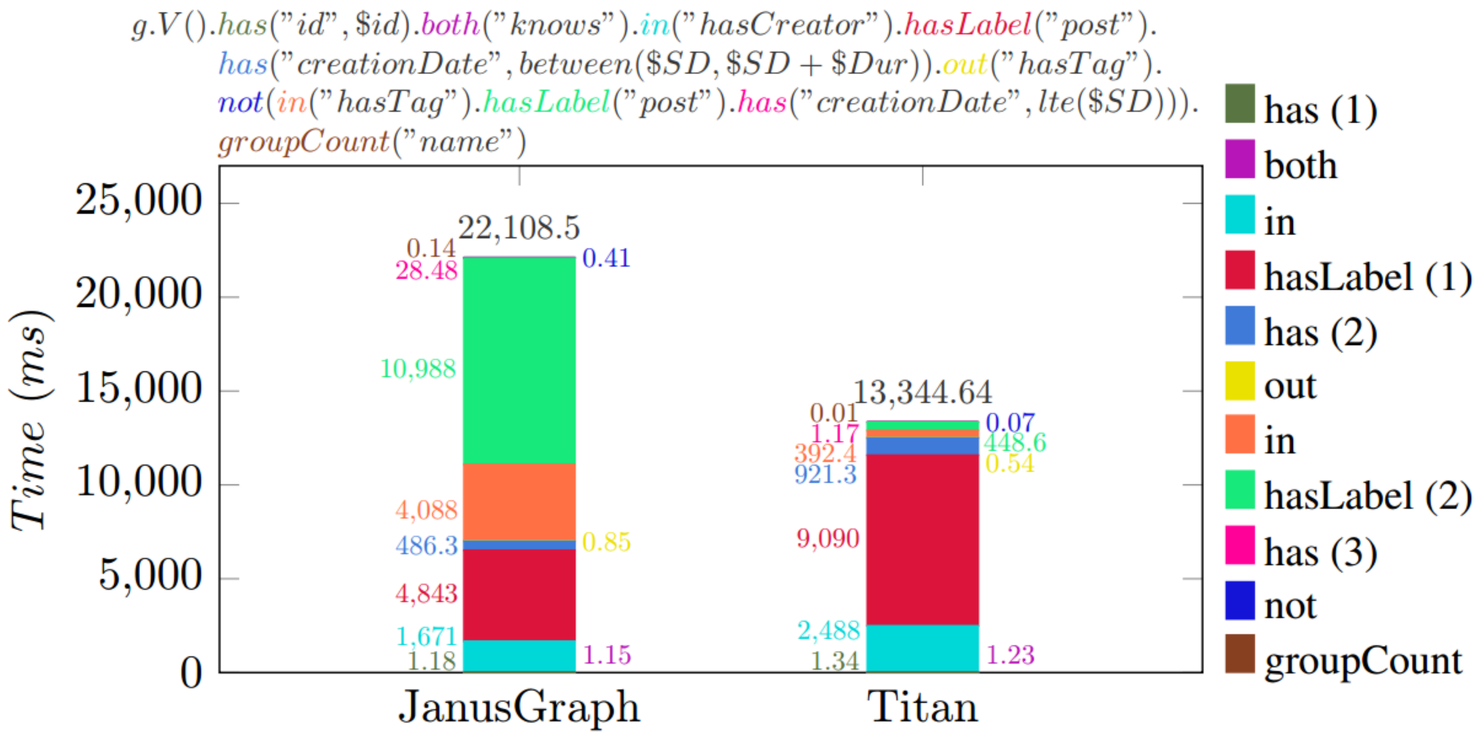


Figure 2. The query latency breakdown of IC4 in LDBC benchmark.

Performance of Some Existing Systems

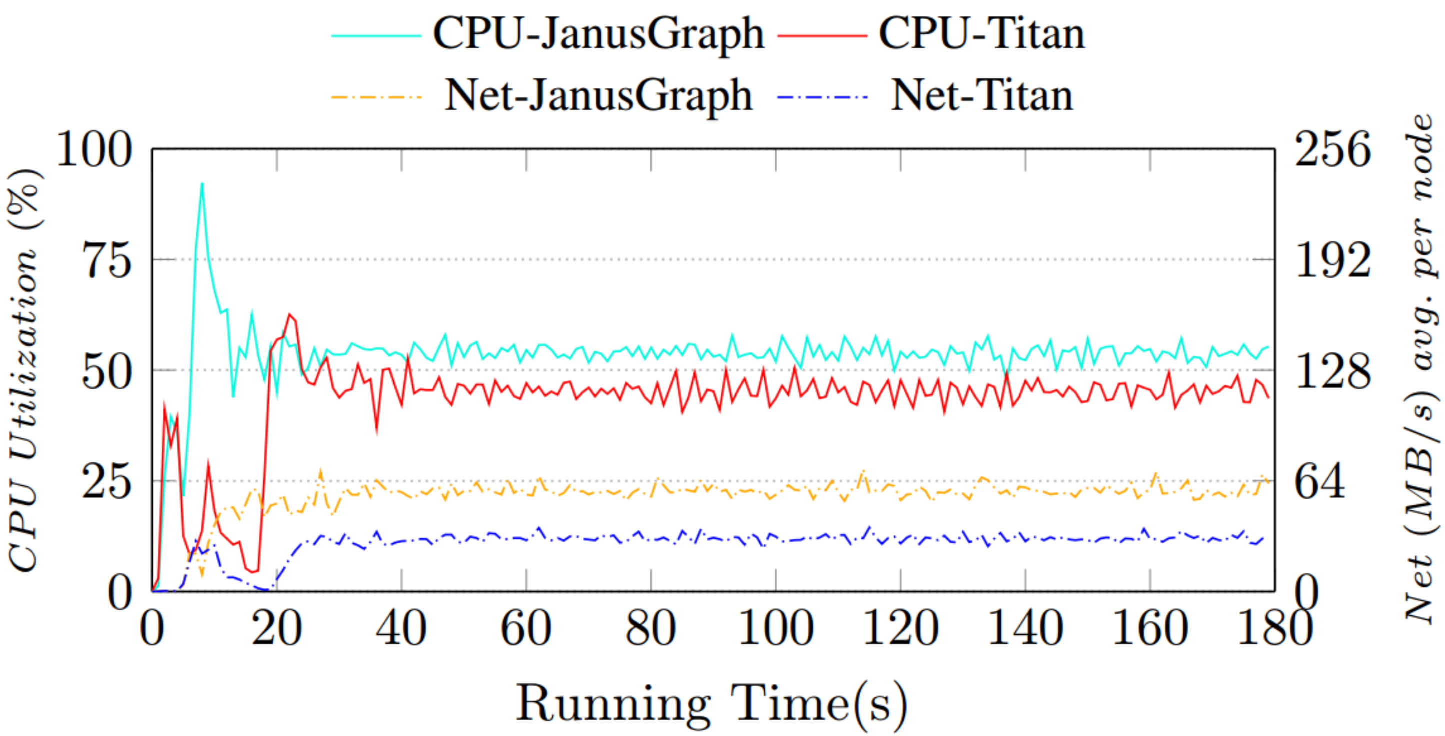


Figure 3. The CPU and network utilization for a mixed workload formed by {IS1-IS4} in LDBC benchmark.

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

System Overview

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

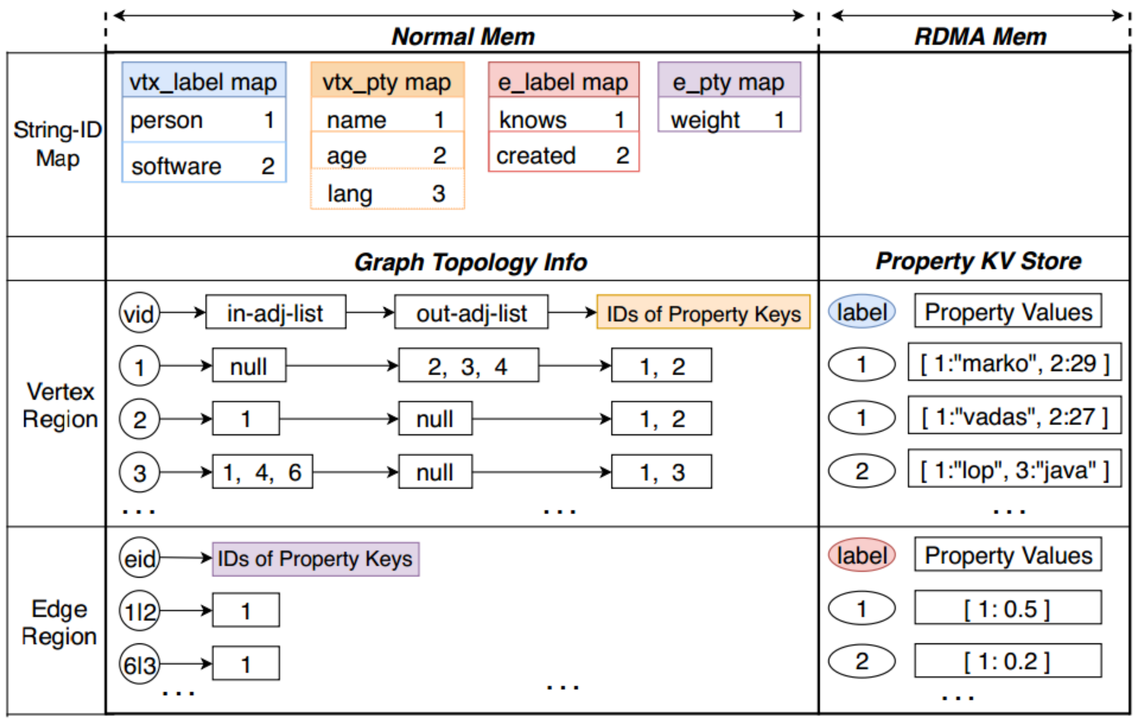


Figure 4. Data store in Grasper.

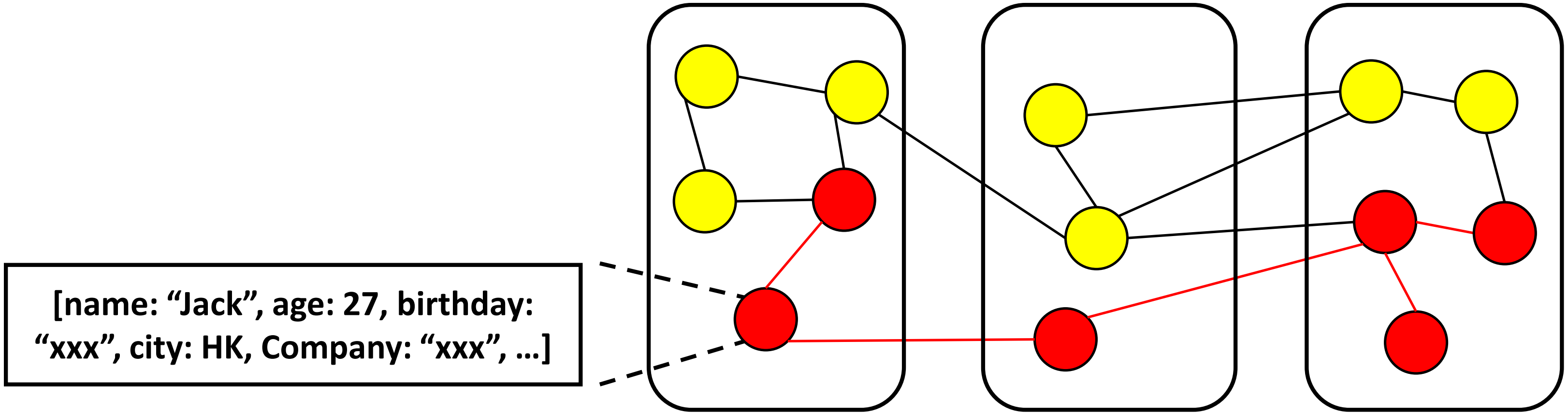
System Design

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

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



System Design

Memory Layout

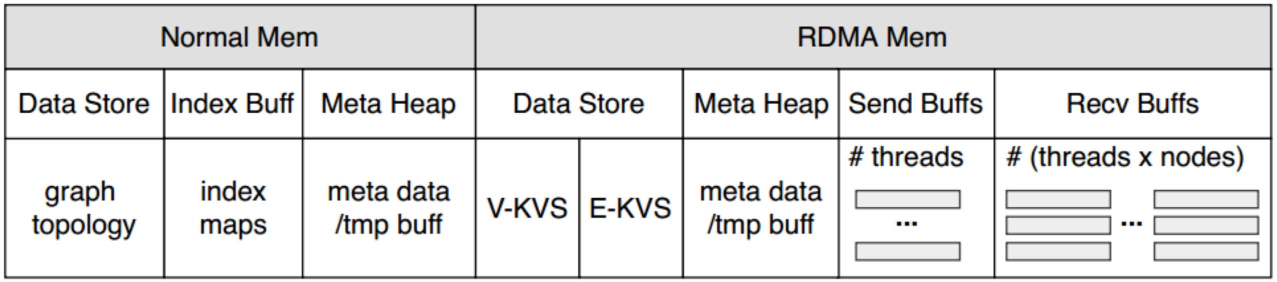


Figure 5. Memory layout on a Grasper node.

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***.

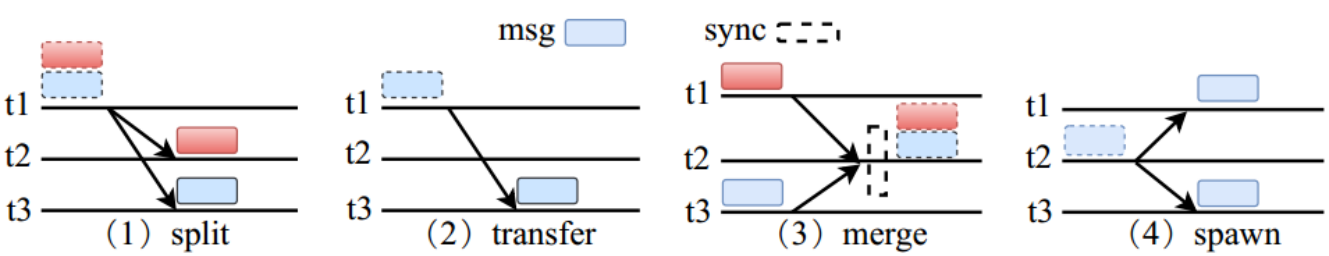


Figure 6. RDMA message dispatching in Grasper.

System Design

Query Plan Construction

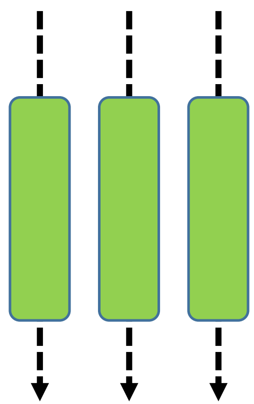
Flow Type, to describe the execution flow of each query ***step***

- to enable ***parallel query processing in a distributed setting***

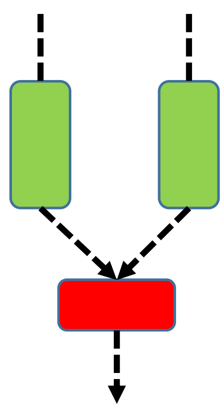
(1) *Sequential*: query logic is independent, e.g., in(), out(), has()

(2) *Barrier*: need sync before moving forward, e.g. count(), max()

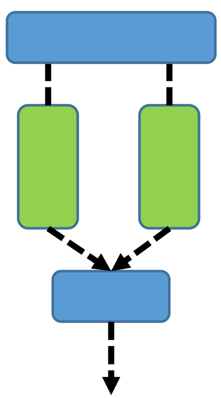
(3) *Branch*: can be splitted into subqueries, e.g., or(), and(), union()



(1) Process in parallel



(2) collect all, then go next

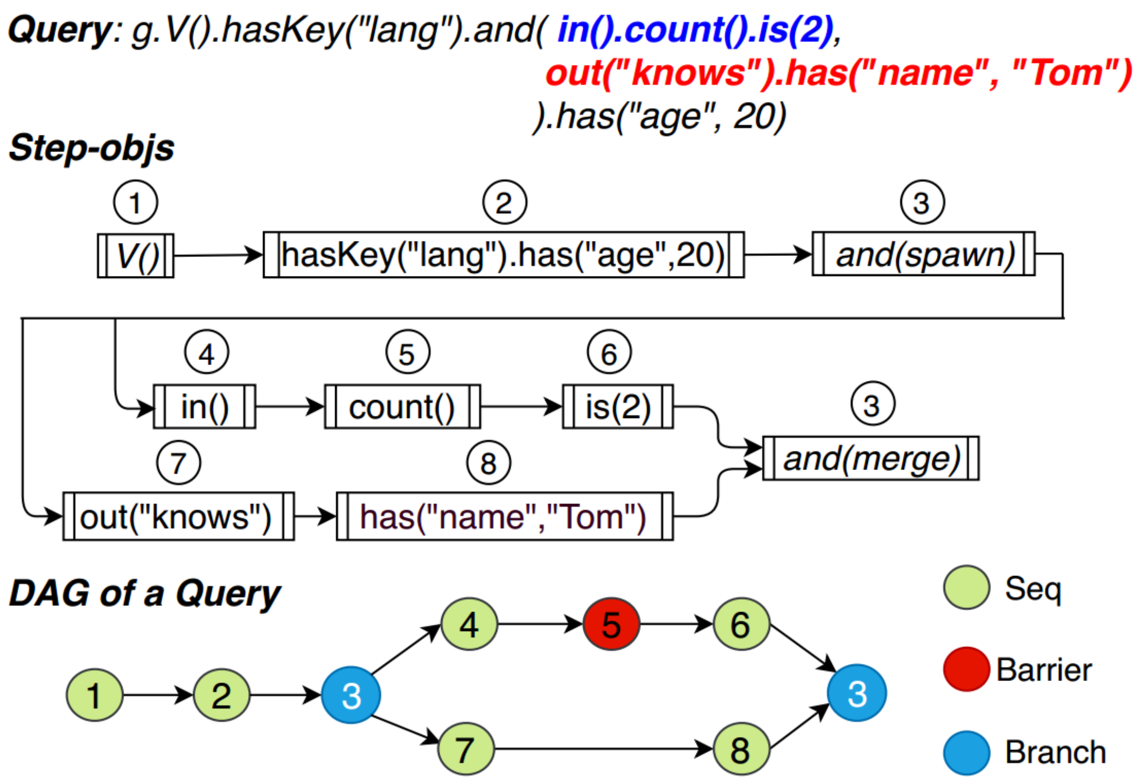


(3) split to sub-queries but needs sync at the endpoint

System Design

Query Plan Construction

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



System Design

Execution Engine – Expert Model

Design Philosophy, a top-down query-specifc 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

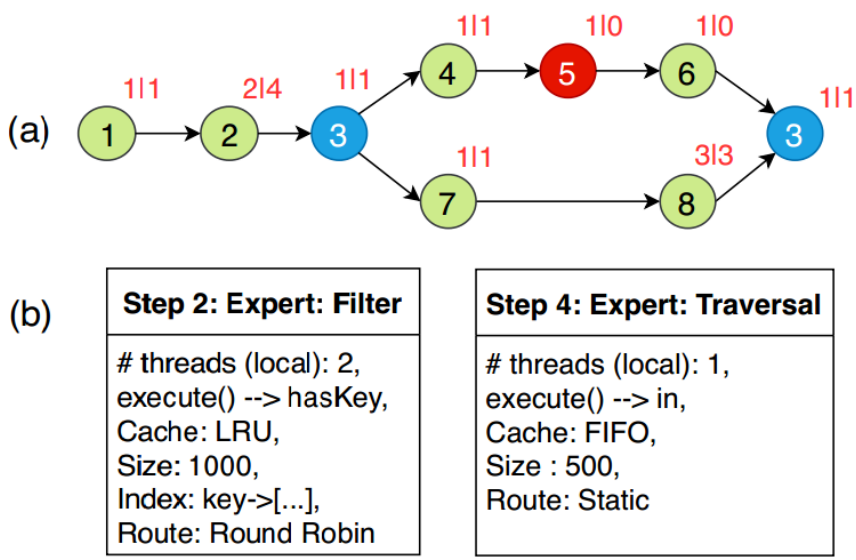


Figure 7. (a) adaptive parallelism at step-level; (b) an expert example.

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.

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

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

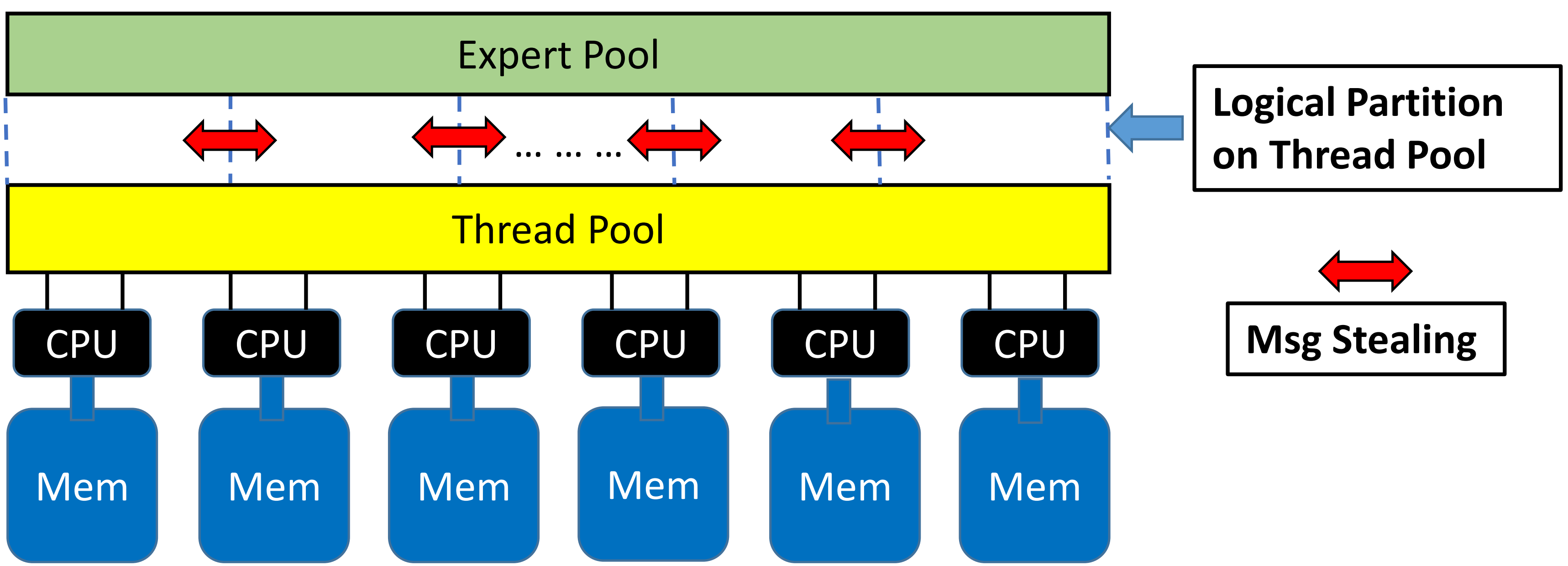


Figure 8. Core bind and load balancing in Grasper.

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.

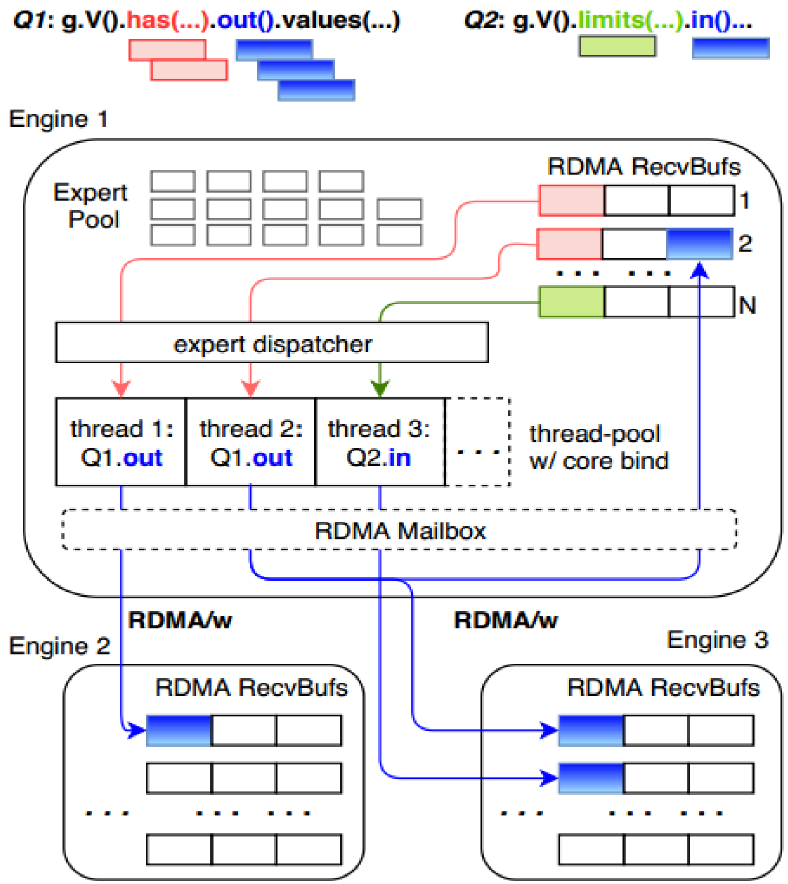


Figure 9. The work flow of Expert Model to process concurrent queries in Grasper.

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

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

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 confguration (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.

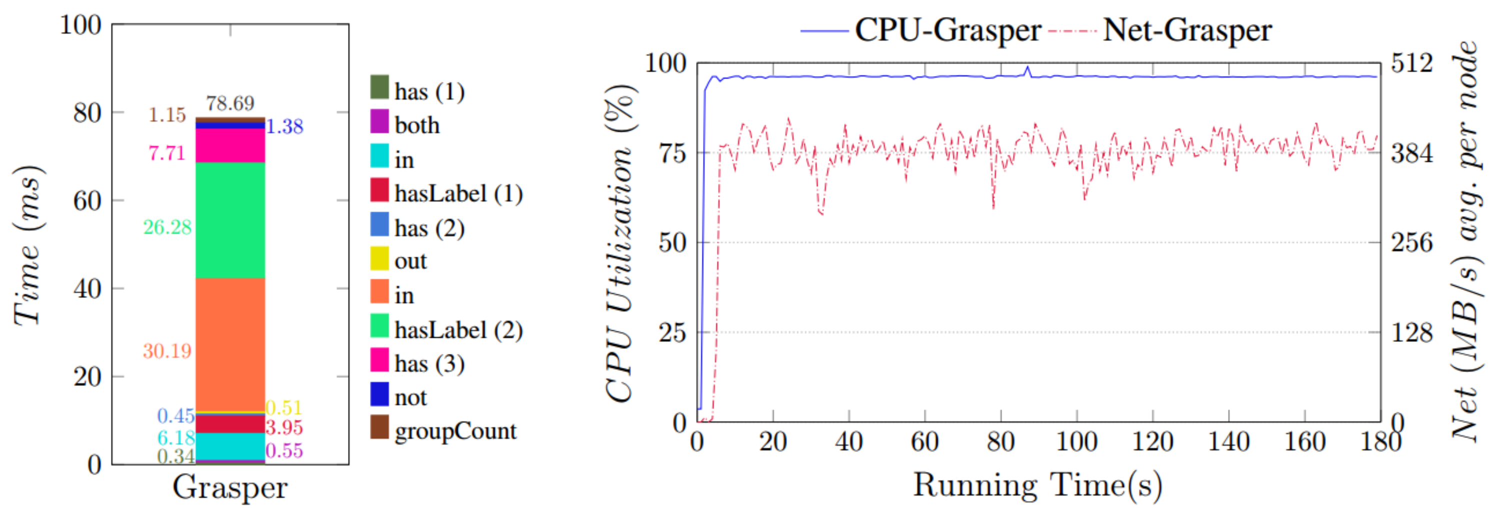


Figure 10. (a) The query latency breakdown of IC4 on LDBC by Grasper;
(b) CPU and network utilization of Grasper for the mixed workload {IS1-IS4}.

Evaluation

Query Latency

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

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

LDBC	IC1	IC2	IC3	IC4	IS1	IS2	IS3	IS4
Grasper	1935	75.1	2550	223	0.48	2.51	1.38	0.13
Neo4J	1448	372	15042	293	20.7	77.6	16.3	21.7
OrientDB	32869	2140	20721	2582	0.91	25.1	3.46	1.47
T.G.(install + run)	46517 +55.3	40739 +18.2	44048 +117	43685 +30.1	37745 +8.03	41629 +11.1	38799 +9.39	37708 +7.66

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

Evaluation

Throughput

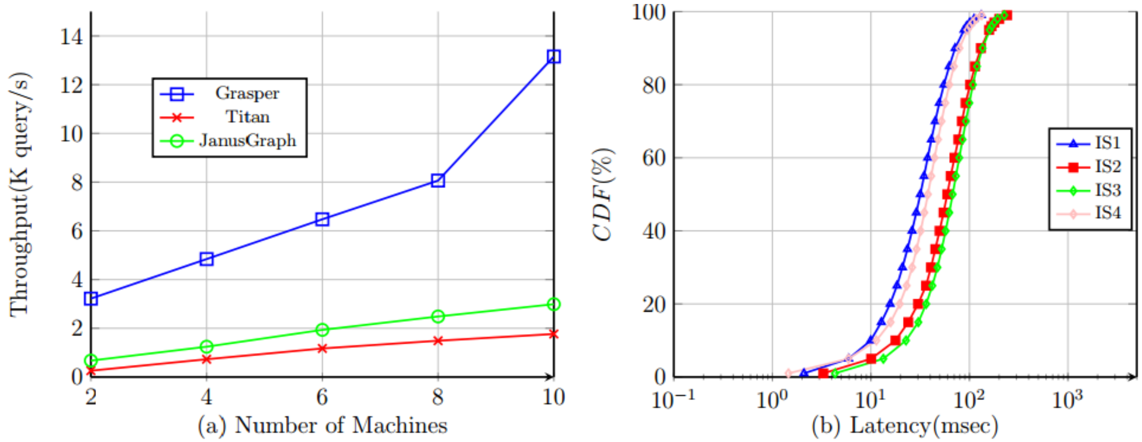


Figure 11. (a) Throughput on LDBC for {IS1-IS4}; (b) CDFs of Grasper's query latency for {IS1-IS4} (using 10 machines).

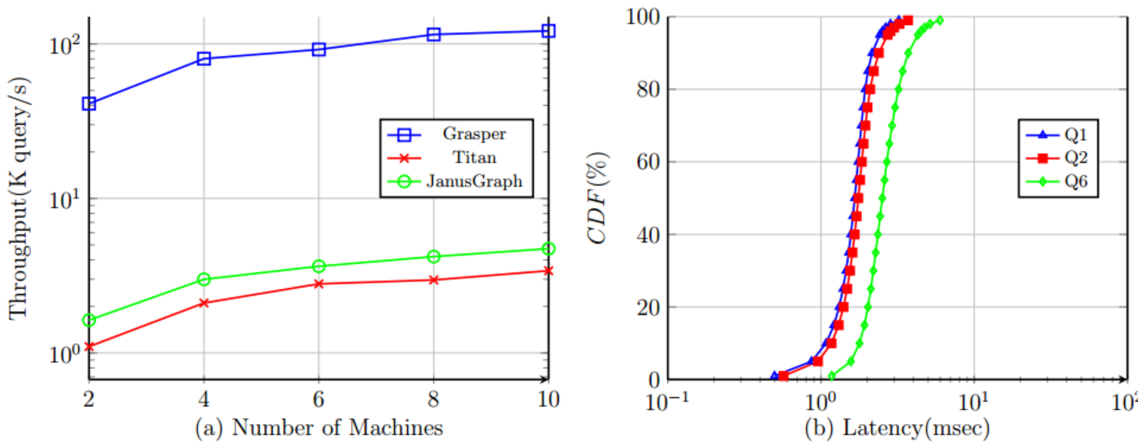


Figure 12. (a) Throughput on AMiner for {Q1, Q2, Q6}; (b) CDFs of Grasper's query latency for {Q1, Q2, Q6} (using 10 machines)

Evaluation

Effects of System Designs & Opts

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

LDBC	IC1	IC2	IC3	IC4	IS1	IS2	IS3	IS4
Grasper	271	16.7	388	77.3	0.30	2.19	0.91	0.32
w/o APC	469	24.8	666	131	0.51	3.63	1.43	0.54
AMiner	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**
Grasper	0.17	0.42	17.3	45.2	104	28.8	2.32	4.41
w/o APC	0.20	0.62	23.7	59.6	111	35.4	4.50	6.15

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

LDBC	IC1	IC2	IC3	IC4	IS1	IS2	IS3	IS4
Grasper	271	16.7	388	77.3	0.30	2.19	0.91	0.32
-RDMA	1349	17.97	1253	260	1.04	2.57	2.06	1.26
-Q.Opts	374	19.39	558	81.26	0.31	2.38	0.93	0.32
-Steal	488	24.68	671	127	0.57	3.25	1.31	0.54
AMiner	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**
Grasper	0.17	0.42	17.3	45.2	104	28.8	2.32	4.41
-RDMA	0.54	1.18	21.54	70.47	222	30.69	9.09	6.23
-Q.Opts	0.17	4254	22.84	417	131	35.49	2.89	4.34
-Steal	0.23	0.61	20.91	57.62	111	33.44	4.01	6.07

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

Husky Data Lab, CSE
The Chinese University of Hong Kong