an Incremental Online Graph Partitioning algorithm for distributed graph databases

Dong Dai*, Wei Zhang, Yong Chen
Workflow of The Presentation

A Use Case

OLTP vs. OLAP

Rethink Graph Partitioning

IOGP Details

IOGP Design Idea

Modeling and Analysis

Evaluation Setup

Partition Quality

OLTP Performance
A Use Case

• Meet Daniel from “All-About-HPC” LLC
  • He wants to build a provenance system for HPC
    • He traces all provenance-relevant events
    • He models and stores them into a provenance graph
    • He offers users graph interfaces to query provenance

New events inserted continuously
Build large graphs with properties
Thousands of clients query it

Typical OLTP Workloads
OLTP vs. OLAP

- We have two sets of tools to handle graphs
  - Graph Databases
    - short, finish in milliseconds
    - touch a small part of the graph
    - measured by time cost of each transaction
  - Graph Processing Engines
    - longer, finish in seconds, minutes
    - touch a large part of or the whole graph
    - measured by system throughput

Daniel needs **distributed graph databases** for his OLTP ops

But, how he partitions the graphs?
Rethink the Graph Partitioning

• Many graph partitioning algorithms
  • Multi-level scheme
    • METIS, Chaco, PMRSB, Scotch, ParMetis, Pt-Scotch, etc.
  • Heuristic single-pass methods
    • LDG, Fennel, 2D-Grid, Vertex-Cut, etc.
  • Online partitioning algorithms
    • Hashing, Leopard, etc.

Not Designed for OLTP Workloads
Rethink the Graph Partitioning

• What does OLTP need?

• Response quickly when insertion happens
  • It needs to finish fast (in $ms$)
  • No time to wait for multi-level schemes or even re-partitioning

• Response while knowing nothing about the vertex
  • The insertion may be “random”
  • Do not assume knowing connectivity of inserted vertex

• Measurement of a good partitioning
  • OLTP cares response time of each operation
  • Only minimizing edge-cuts/maximizing the balance is not enough
  • Think about a graph with $n$ equal-size subgraphs
Modeling and Analysis

• Generic distributed graph database model
  • directed/undirected graphs
  • bi-direction accesses on the graphs
  • queryable properties on vertices and edges
Factors of OLTP Performance

• Single-Point Access Performance
  • One-hop: if clients know the location of queried data
  • It saves time for querying location information

• Traversal Performance (Edge-Cut and Vertex-Cut)

a) Graph travel example from u  b) Graph with 3 partitions (P0,P1,P2)
IOGP Core Idea

• Leverage deterministic hashing to place new vertices
  • fast and easy to calculate
  • enable one-hop access
  • need no info to conduct partitioning

• Incrementally reassign vertices to better location
  • continuously get better locality with increasing knowledge

• For vertex is too large, change from edge-cut to vertex-cut
  • increase the parallelism
  • improve OLTP response time
IOGP - Quiet Stage

- Quiet Stage is the default stage
  - a vertex $v$ is inserted, it will be placed based on Hashing
  - an edge $u \rightarrow v$ is inserted, it will be placed together with its connected vertices
Vertex Reassign Stage

- When we knew enough connectivity of a vertex $v$
  - move it to the partition that stores the most of its neighbors
  - A heuristic function (derived from Fennel)

$$
max\{|N(v) \cap P_i| - \alpha \frac{Y}{2} (|P_i|)^{Y-1}\}
$$
Edge Splitting Stage

• When a vertex becomes extremely large
  • Edge-cut leads to significant performance degradation
  • Split all edges to increase the parallelism of data accesses
IOGP Details

- We need per-vertex metadata
  - To record vertex location \(\text{loc}(v)\)
  - To know vertex degree or it has been split \(\text{split}(v)\)
  - To efficiently calculate \(\text{counters}(v)\)

\[
\max\{|N(v) \cap P_i| - \alpha \frac{Y}{2} (|P_i|)^{Y-1}\}
\]

- Counters

```
split(v)
loc(v)
alo(v)
ali(v)
plo(v)
pli(v)
```

Original Location

Scale Well

Not That Bad Actually
Update Counters after Reassignment

1. Update $\text{loc}(v)$ in the original server

2. In server $S_u$ ($v$ is moving out of it)
   - $v$’s actual locality turns into potential locality
   - $v$’s local neighbors
     - reduce actual locality by 1 because $v$ is not local anymore

3. In server $S_k$ ($v$ is moving into it)
   - $v$’s potential locality turns into actual locality
   - $v$’s local neighbors
     - increase their actual locality by 1 because $v$ is local to them now
Optimization: Timing of Vertex Reassignment

• Vertex reassignment is time-consuming

• Observation
  • more edges, more stable connectivity;
  • less reassignment is needed

• Design
  • deferring vertex reassignment until its connectivity stabilizes
  • reducing vertex reassignment frequency with more edges

• Implementation
  • Start reassignment until its degree is higher than $k$
  • Check for reassignment when its degree reaches $[2^1 k, 4^1 k, ..., 2^l k...]$
Optimization: Asynchronous Data Movement

• Vertex reassignment and edge splitting
  • sync data movement may stale OLTP operations
  • optimization: asynchronous data movement

• Edge splitting example
  • Update in-memory counters
  • add vertex in pending movement queue
  • Server starts to reject new edges

• Data may in different locations during movement
  • original server, new server, and maybe both
  • clients need to read multiple copies and aggregate them
    • Prefer the one from “new server”
Evaluation Setup

• All evaluations were conducted on the CloudLab cluster
  • 32 servers for back-end storage

• We chose various datasets from SNAP for evaluations
  • scales from 200K edges to over 100M edges
  • from various domains, forming different shapes

• We evaluated IOGP on a distributed graph database prototype, namely SimpleGDB (https://github.com/daidong/simplegdb-Java)
  • A prototype system has been used in several research projects
  • For fair comparison and controllable optimizations
Partitioning Quality

- METIS runs directly on the final graph
- Fennel runs in random order

METIS is still the best
IOGP achieves very well

METIS is not always balanced
Others (including IOGP) are well balanced
Continuous Refinement of IOGP

- Using similar heuristic, IOGP performs better than Fennel due to continuous refinement
Key Tunable Parameters

Reassignment Threshold

somewhere around average degree of the graph

Split Threshold

Relevant with
1) hardware;
2) scale of the database cluster;
3) vertex degree and property size
Memory Footprint

• How many memory is used in IOGP to keep the in-memory counters?
OLTP Performance

Graph Insertion
less than 10% overhead

Graph Travel
As much as 2x benefits
Conclusion & Future Work

• OLTP workloads of distributed graph databases require new set of graph partitioning algorithms.

• IOGP achieves less than 10% overheads on insertion, and as much as 2x performance improvement on queries.

• There are still lots of things can be done
  • Reduce the overheads of vertex reassignments
  • Partitioning the graphs based on the exact workload patterns
  • ...
Thanks and Questions