IBM Digital Business Group

# On-boarding with JanusGraph Performance

***June 17, 2017***

Chin Huang, chhuang@us.ibm.com;github:chinhuang007
Yi-Hong Wang, yh.wang@us.ibm.com;github:yhwang
Ted Chang, htchang@ibm.com;github:tedhtchang

**JanusGraph:@JanusGraph**

IBM

# Agenda

Overview – Onboarding with graph performance

JanusGraph performance evaluation scenarios

- Bulk loader performance
- Data import performance
- Query performance

Lessons learned

Q&A

# Onboarding with graph performance

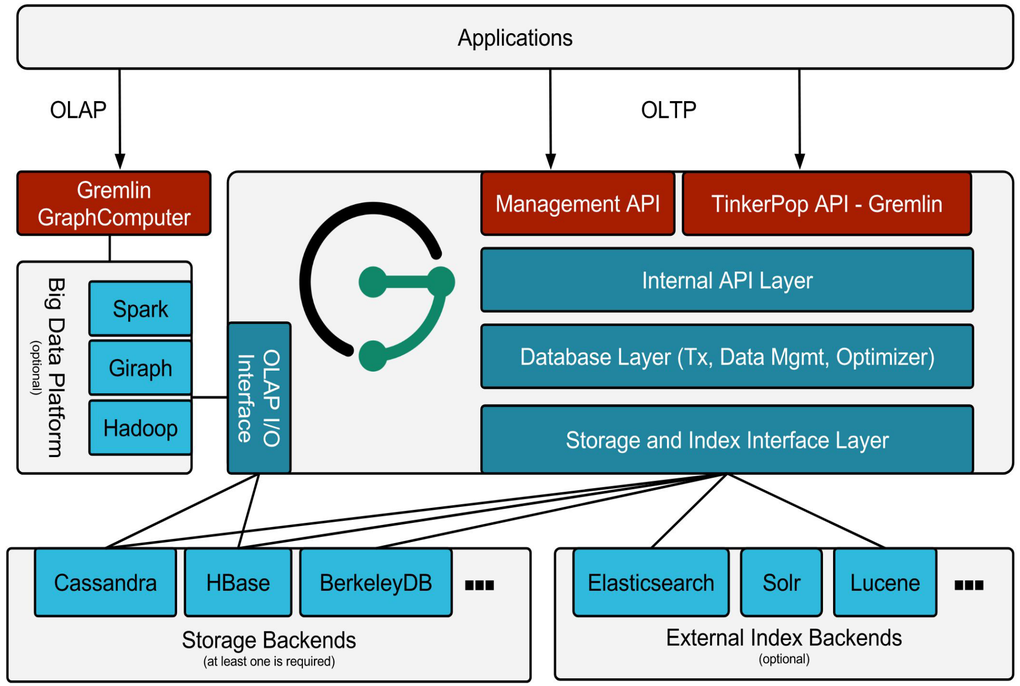
Exciting era with many new technologies!!

Onboarding users/developers to graph databases

- Typical focus areas: features and benefits, ease of use, suitability, extensibility, APIs…
- Performance is one of the most important differentiators for any application
- Is performance just for system testing?!
- Performance and scalability are key considerations for design, development, and operations

Journey to JanusGraph with a performance mind!

- Check out graph structures and traversals
- Evaluate reads and writes in high volume
- Can JanusGraph scale out for future data/user growth?
- Look for bottlenecks and provide improvements



# JanusGraph performance test environment

## Server spec

- Physical servers: x3650 M5, 2 sockets x 14 cores, 384 GB (12 x 32G) memory
- CPU: Intel Xeon Processor E5-2690 v4 14C 2.6GHz 35MB Cache 2400MHz
- Network interface: Emulex VFA5.2 ML2 Dual Port 10GbE SFP+ Adapter
- Disk: 720 GB SSD, RAID 5
- Operating system: Ubuntu 16.04.2 LTS

## Existing tools

- jMeter - load testing tool
- nmon, nmon analyser - system performance monitor and analyze tool
- VisualVM - all-in-one Java troubleshooting/profiling tool
- GCeasy - garbage collection log analysis tool

## Home grown tools

- Graph schema loader, data generator, batch importer, batch requester

# JanusGraph performance tool - Graph schema loader

Enable the graph model creation via the gremlin console or embedded in java

- Use JSON to describe your graph model
- Support:
  - Property
  - Vertex
  - Edge
  - Index

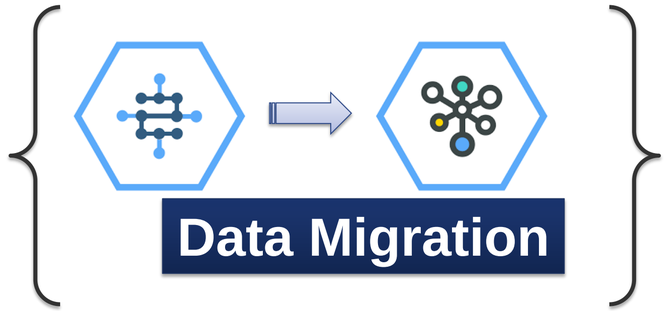
Benefit: Create schema on-the-fly without single line of code!

https://github.com/yhwang/janusgraph-utils

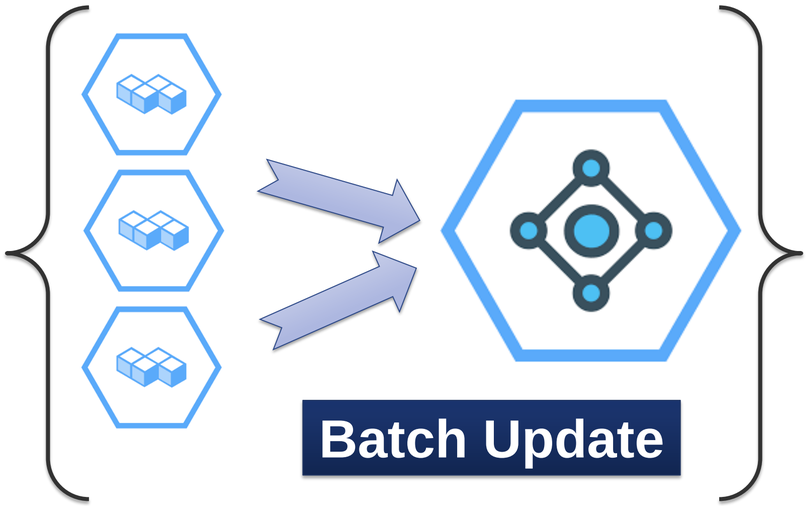
```
{
  "propertyKeys": [
    { "name": "name", "dataType": "String", "cardinality": "SINGLE" }
  ],
  "vertexLabels": [
    { "name": "person" }
  ],
  "edgeLabels": [
    { "name": "favorites", "multiplicity": "MULTI" }
  ],
  "vertexIndexes": [
    { "name": "vByName", "propertyKeys": ["name"],
      "composite": true, "unique": true }
  ],
  "edgeIndexes" :[
    { "name": "eByTime", "propertyKeys": ["time"],
      "composite": true }
  ],
  "vertexCentricIndexes" :[
    { "name": "vcByTime",
      "propertyKeys": ["time"], "edge": "mentions",
      "direction": "BOTH", "order": "incr" }
  ]
}
```

# Bulk load performance – Use case and data

**OneTimeBulkLoader**




**IncrementalBulkLoader**




**Supported Formats**

**Gryo:** `01111010010110010l`

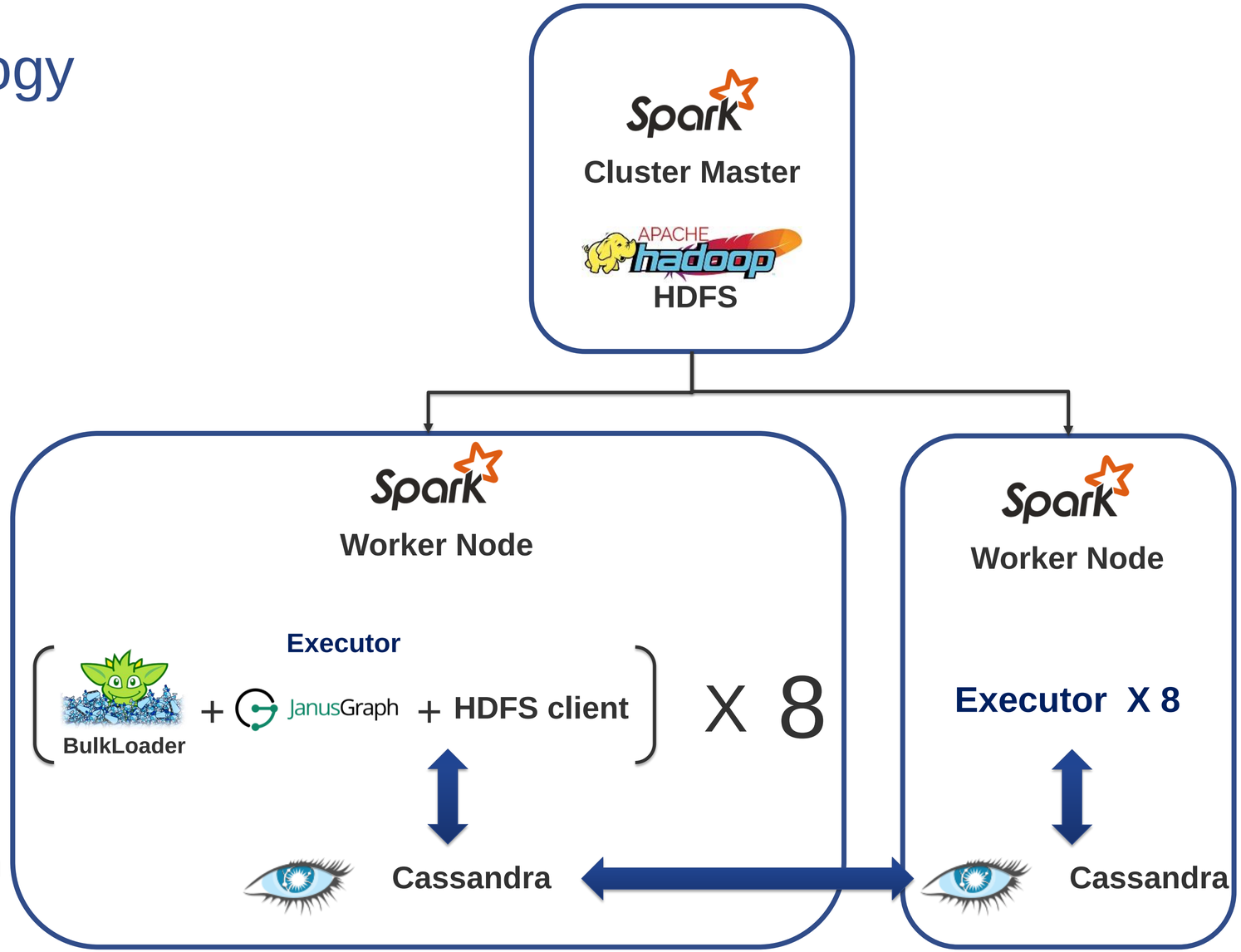
**GraphSON:** `{"id": 1, "label":…}`

**Script:** `1:person:marko:29`

- OneTimeBulkLoader
- 128GB GraphSON file
- 31 million vertices
- 38 million edges
- 3277 propertyKeys
- 5 vertex labels
- 3 edge labels
- 78.9 properties per edge
- 18.7 properties per vertex

# Bulk load performance – Topology

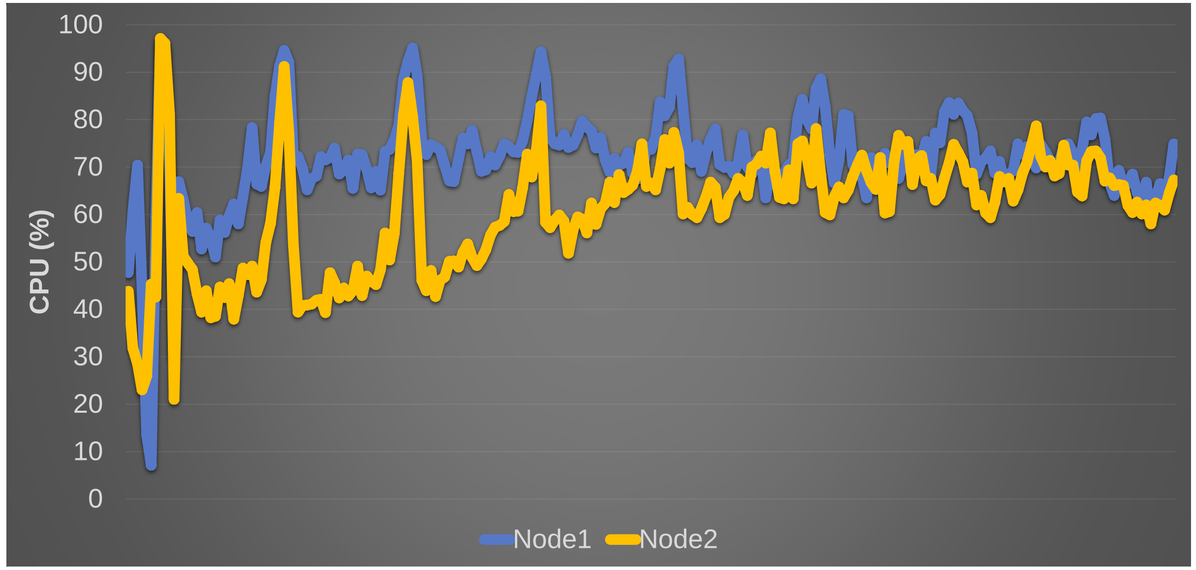
- Spark - 1.6.1
  - Standalone Cluster
  - 2 worker nodes
  - 8 executors per node
  - 8 cores per executor
  - 2GB per executor
- Hadoop - 2.7.2
  - Use HDFS to store the GraphSON file
- Cassandra - 2.1.17
  - 2-node cluster
- Tinkerpop3 – 3.2.3
  - GraphComputer
- JanusGraph – 0.1.1
  - JanusGraphBulkLoaderVertexProgram
  - Astyanax persistence provider

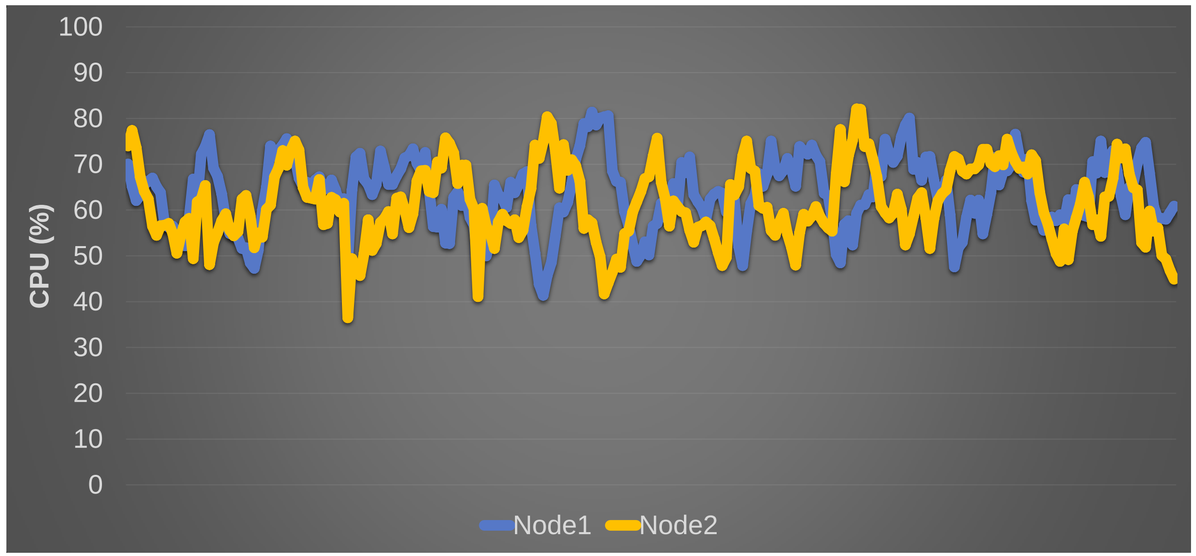


# Bulk load performance – results

- Vertex:
  - 31,594,277
  - 19 mins
  - 495 records/sec per core

- Edge:
  - 38,322,731
  - 24.8 mins
  - 461.8 records/sec per core



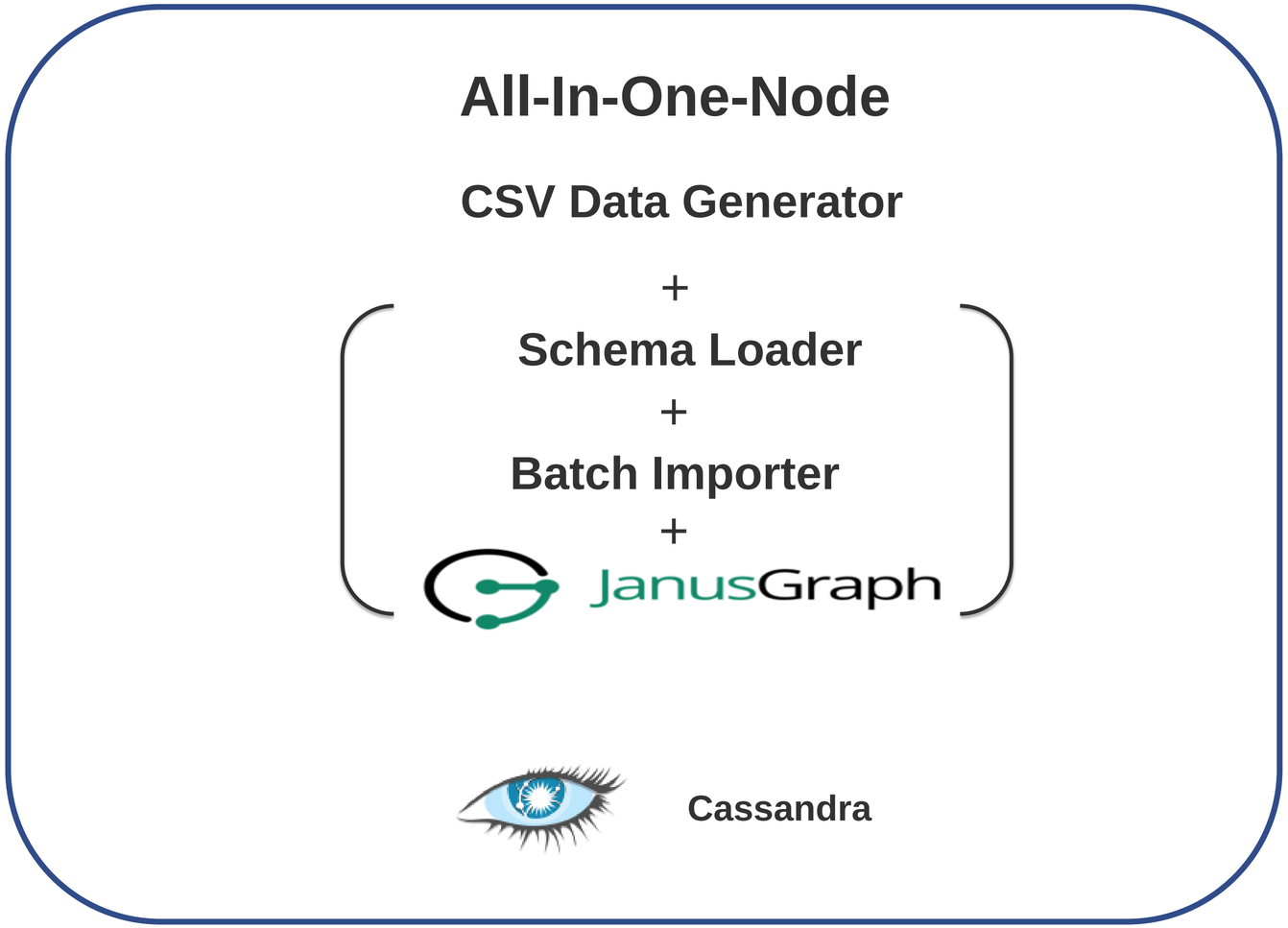



# Data import performance – use case and data

| Public Data | | | |
|---|---|---|---|
| | **Wikimedia votes** | **Higgs Twitter** | **Panama Papers** |
| Vertices(Million) | 0.007 | 0.456 | 1.04 |
| Edges (Million) | 0.1 | 16 | 1.53 |
| PropertyKeys | 0 | 2 | 22 |
| Vertex labels | 1 | 1 | 5 |
| Edge labels | 1 | 4 | 261 |

| Synthetic Data | | | | | | | |
|---|---|---|---|---|---|---|---|
| | **Small** | **Medium** | **Large** | **XLarge** | **10x Properties** | **50x Properties** | **100x Properties** |
| Vertices(Million) | **0.3** | **3** | **30** | 30 | 3 | 3 | 3 |
| Edges (Million) | **0.3** | **3** | **30** | **300** | 3 | 3 | 3 |
| PropertyKeys | 7 | 7 | 7 | 7 | **70** | **350** | **700** |
| Vertex labels | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| Edge labels | 2 | 2 | 2 | 2 | 2 | 2 | 2 |

# Data import performance – topology and configuration




JanusGraph configuration:
storage.backend=astyanax
ids.block-size = 500000
storage.buffer-size = 2560
storage.batch-loading = true
schema.default = none

BatchImporter configuration:
commit size = 100
worker target size = 10000

# Data import performance tooling- Graph data generator

A Java application

- Vertices and edges labels
- Number of vertices and edges
- Number of properties and data types
- Native and mixed index
- Relations patterns
- Super-nodes
- Generate graph-db schema in JSON
- Generate datamap JSON for BatchImporter

https://github.ibm.com/htchang/JanusGraphBench

```
{
  "VertexTypes": [
    { "name": "T1",
      "columns": {
        "T1-P1": {"dataType": "String", "composit": true} },
      "row": 10 },
    { "name": "T2",
      "columns": {
        "T2-P1": {"dataType": "String", "composit": true},
        "T2-P2": {"dataType": "Integer", "composit": true} },
      "row": 10 }
  ],
  "EdgeTypes": [
    { "name": "E1",
      "columns": {
        "E1-P1": {"dataType": "Date", "composit": true} },
      "relations": [
        {"left": "T1", "right": "T2", "row": 10, "supernode":{"vertices":5, "edges":1000} },
        {"left": "T2", "right": "T1", "row": 10 }
      ]
    }
  ]
}
```

# Data import performance tooling - Graph data batch importer

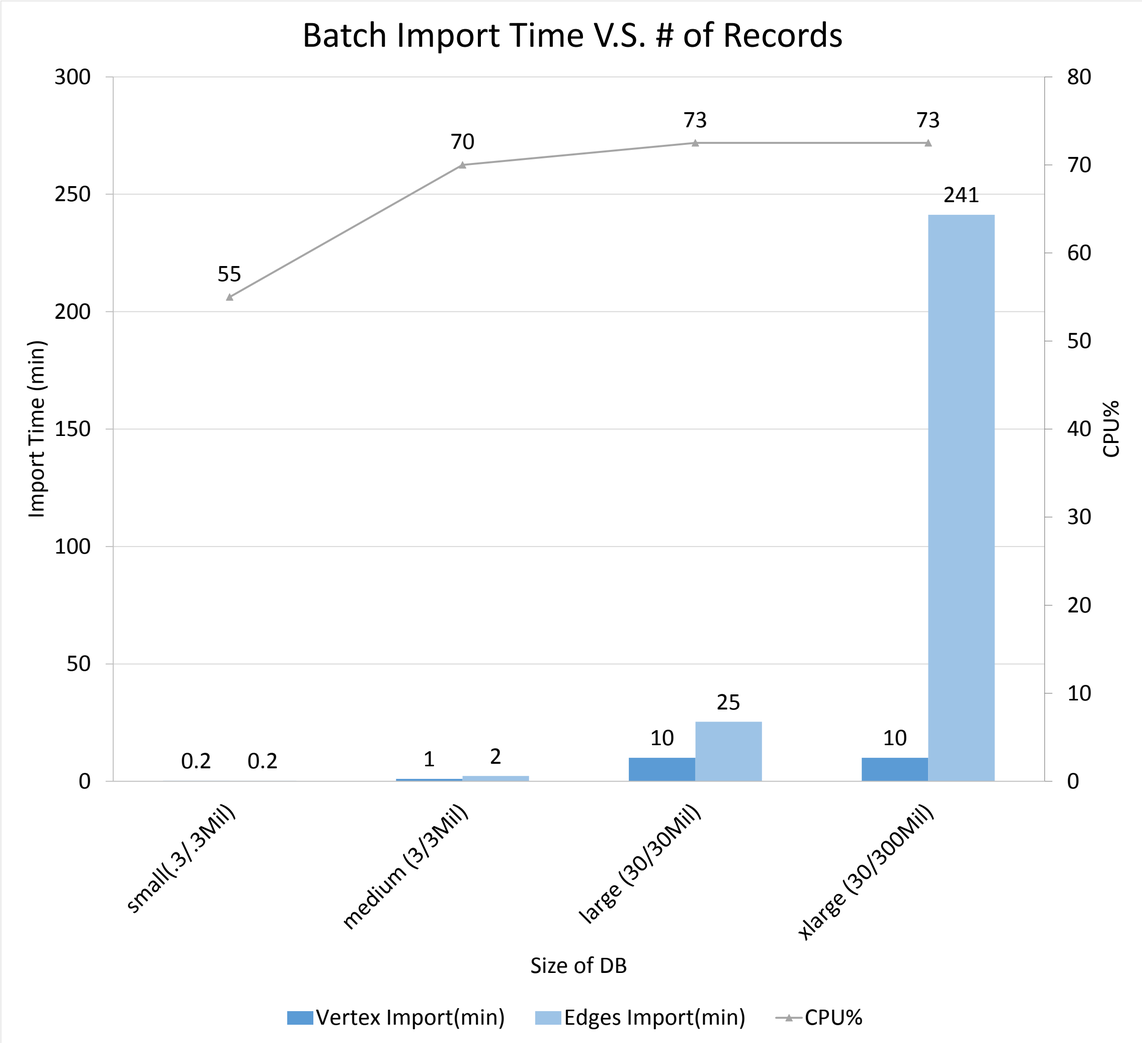
Java application to Import CSV data into JanusGraph

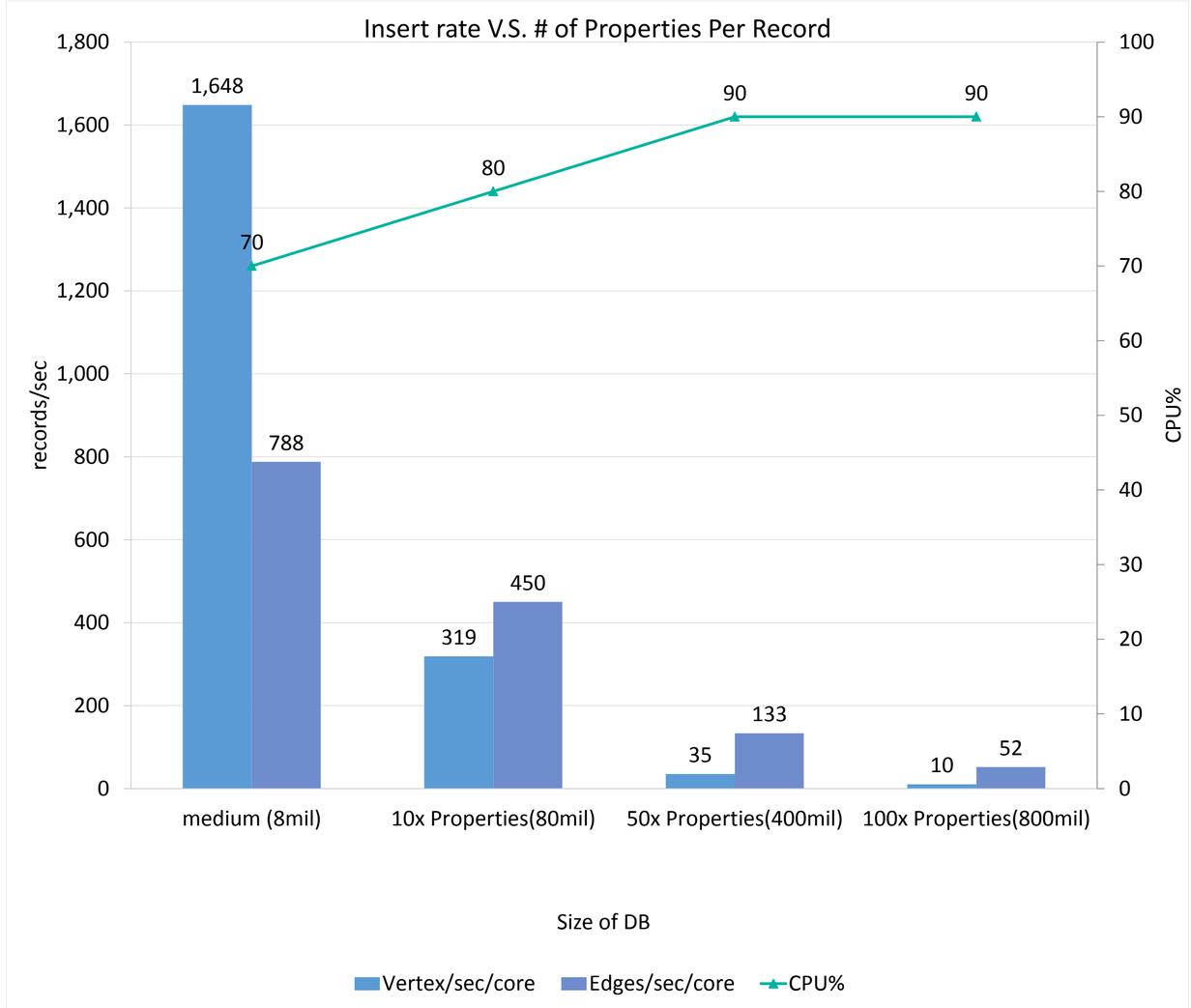
Features:

- Multiple Threads
- Worker record size
- Commit size
- Import schema
- Import CSV to JanusGraph with configurable data mapping

https://github.com/sdmonov/JanusGraphBatchImporter

# Data import performance – results



# Data query performance – use case and data

## Flight search

- All flights from airport A to airport B on a given date and time
- # of stops: non-stop, one-stop, two-stop…

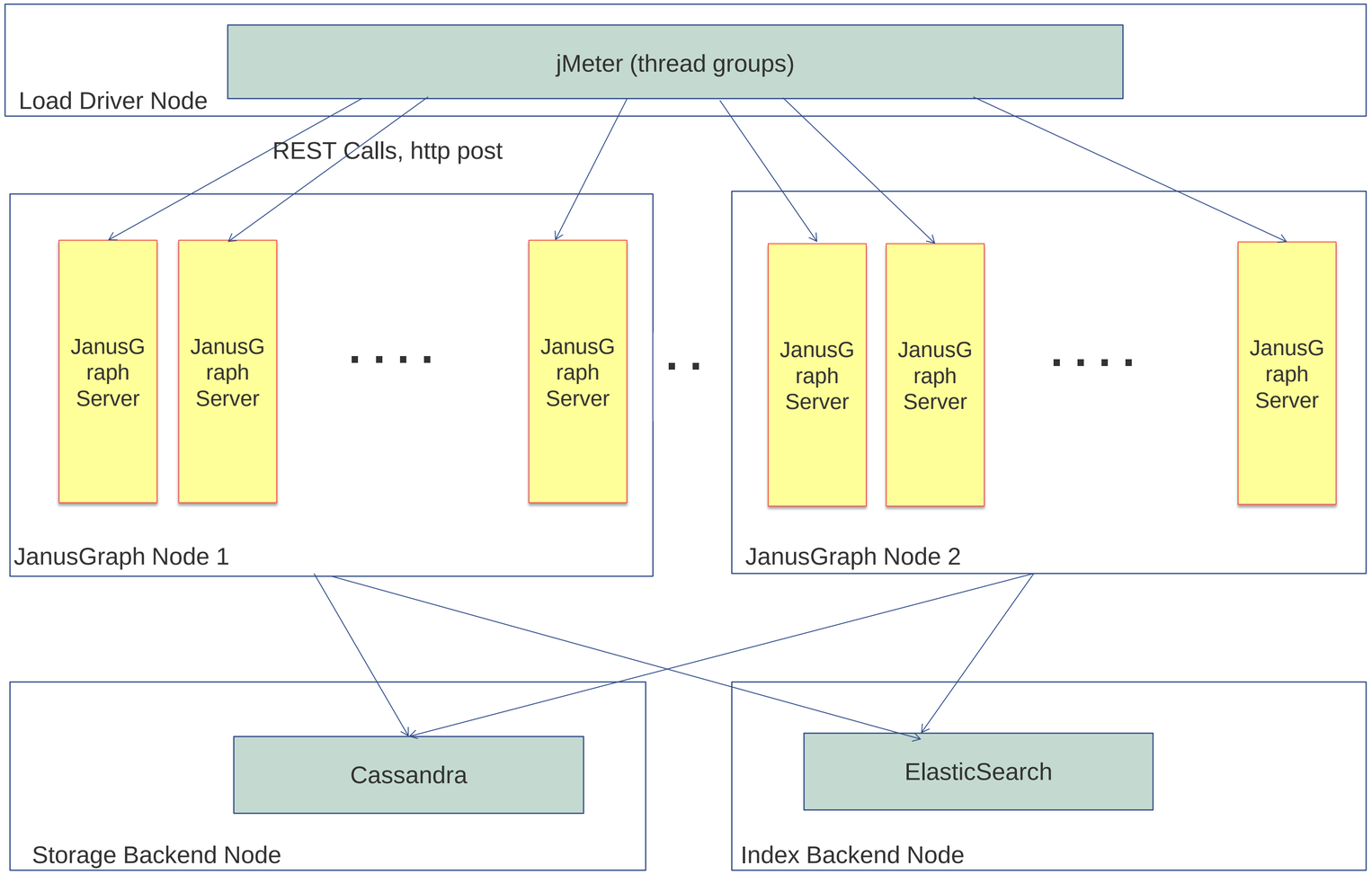
## Data spec

- 600+ airports, 350K+ flight schedules

## Performance analysis

- How many requests per second can JanusGraph handle?
- Can JanusGraph scale with future volume growth?

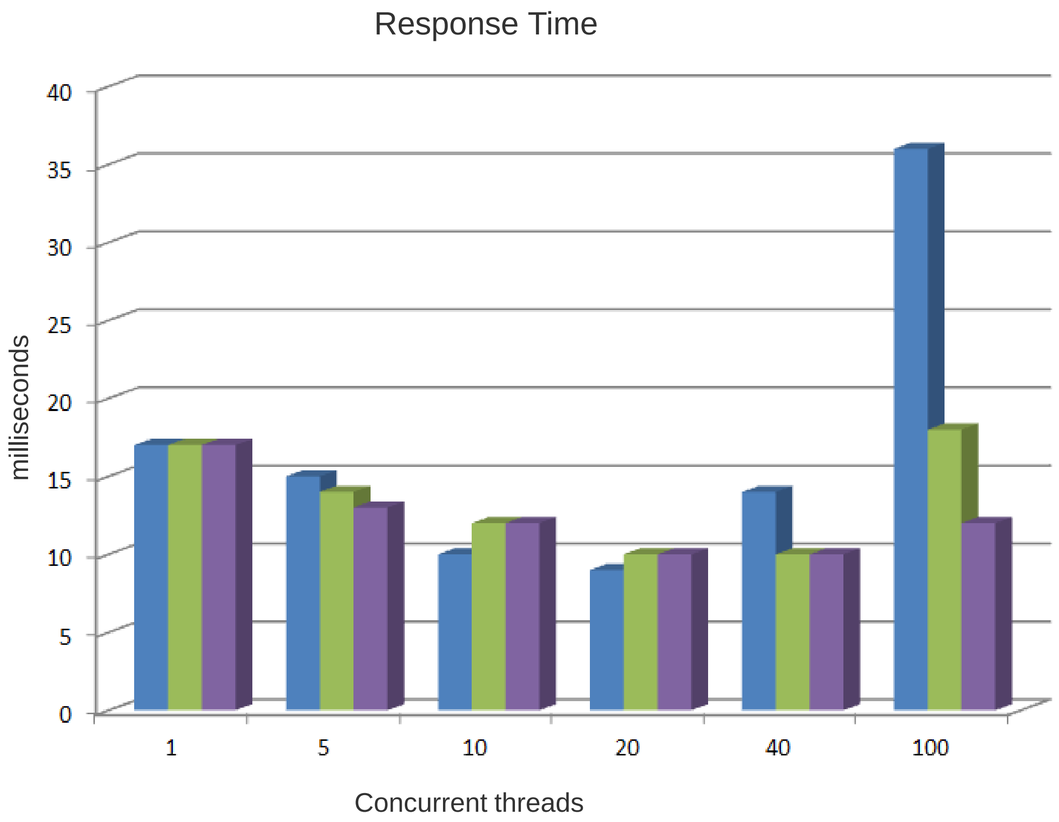
# Data query performance - Topology and configuration

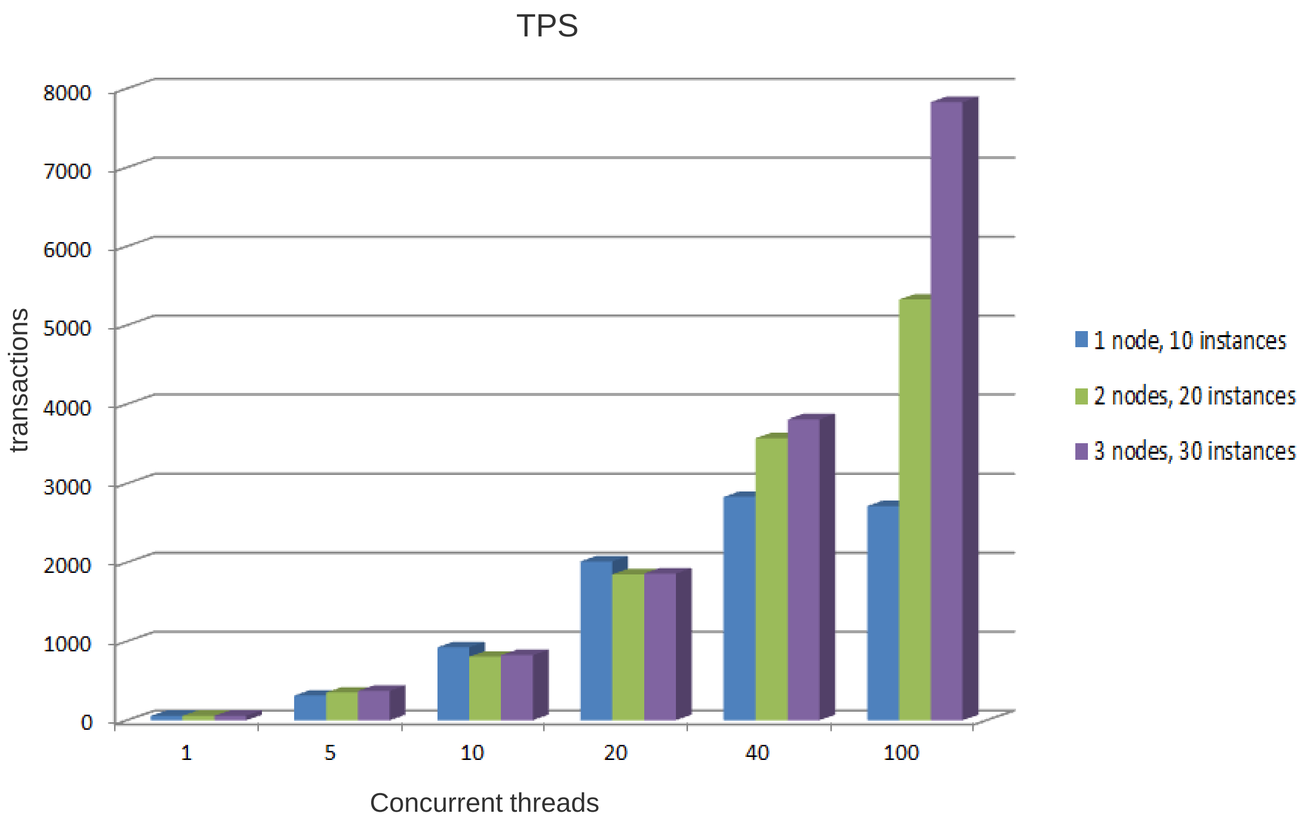



- JanusGraph server with REST
- 1 or 10 instances per server
- Astyanax persistence provider
- threadPoolBoss: 2
- threadPoolWorker: 20
- Java heap: -Xms512m -Xmx8G
- Concurrent threads (users): 1, 5, 10, 20, 40, 100, 200
- Think time: 0 ms
- Run duration: 5 minutes
- Multiple test configurations
  - 10 instances on 1 node
  - 20 instances on 2 nodes
  - 30 instances on 3 nodes

# Data query performance – Non-stop flights (one level deep traversals)

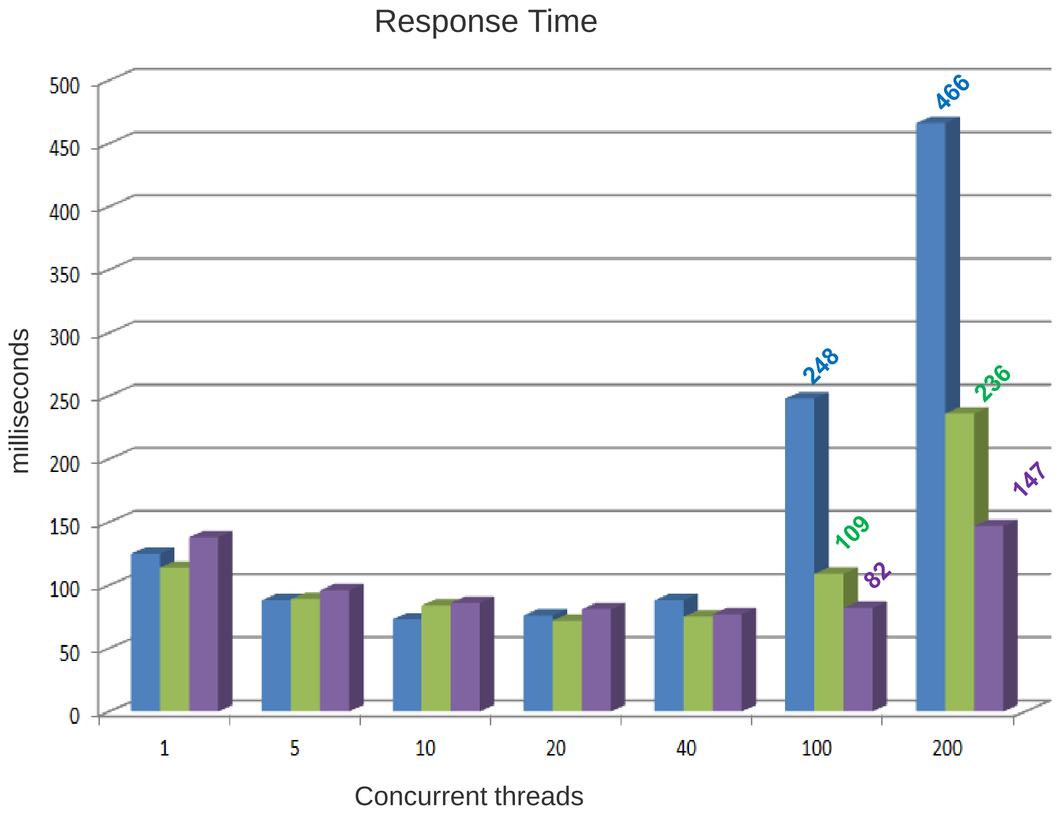
## Performs well regardless number of instances and nodes

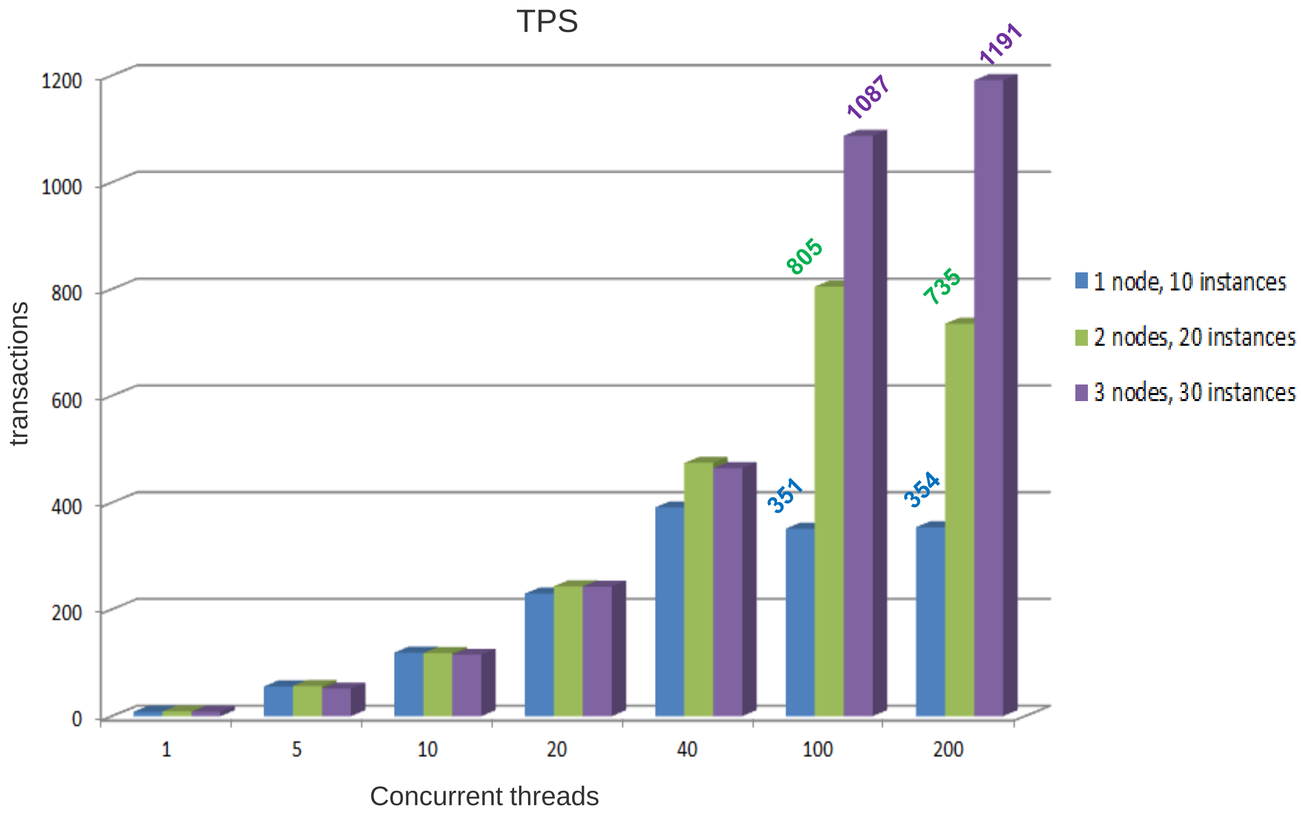


# Data query performance – One-stop flights (two levels deep traversals)

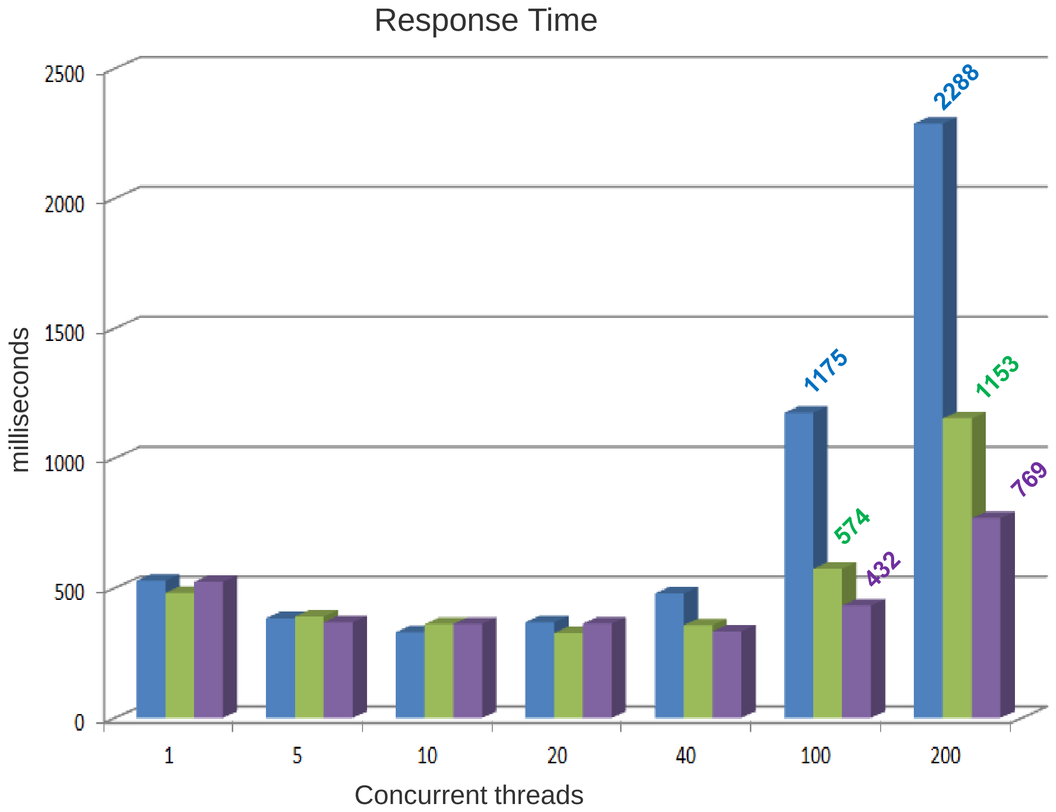
People would like to see more than just non stop flights…

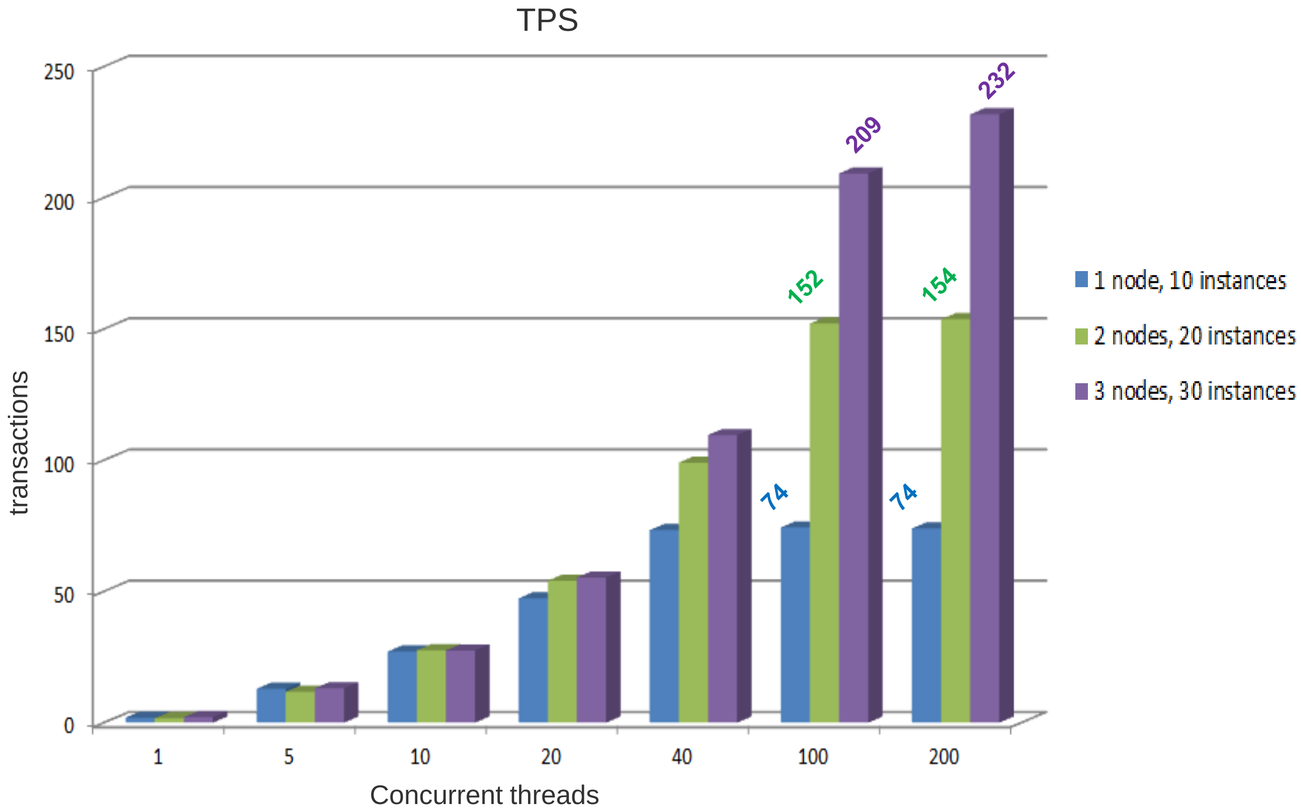


# Data query performance – Two-stop flights (three levels deep traversals)

The query gets complicated because we need to operate and filter on multiple vertices and edges.



# Lessons Learned

## Model your graph database for performance

- Data is yours. Design the data model for your use cases!
- What kind of queries you want to support? How many levels deep into a traversal?
- Consider denormalization…
- Design and use indexes, graph indexes and vertex-centric indexes in JanusGraph, for better performance, but not over-use indexes
- It is recommended to create the complete data model before inserting content

## Use batch commits with caution

- Batch commits allow multiple transactions to be committed together. The batch size affects performance and the optimal size depends on the characteristics of data.
- Need to handle conflicts for inserts and updates in a multi-threads/multi-clients implementation
- Make sure the commit is completed and closed

# Lessons Learned

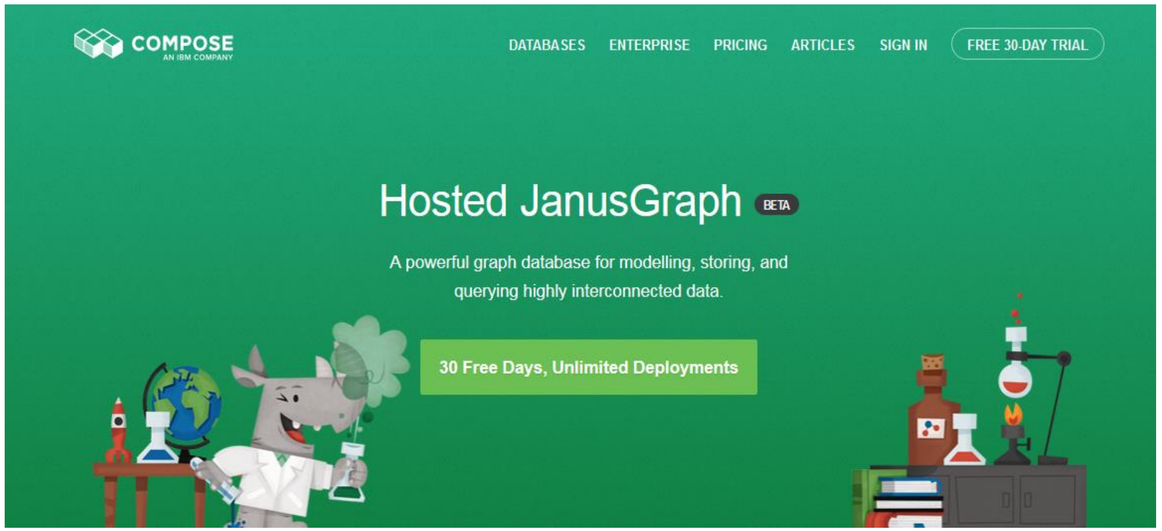
Fine-tune for your workloads and systems

- JanusGraph supports storage and index backends therefore tune your backends!
- JanusGraph server configurations, such as threadPoolBoss and threadPoolWorker
- JVM configurations, such as Xms (initial and minimum Java heap size) and Xmx (maximum Java heap size) You don’t want to see the annoying java.lang.OutOfMemoryError exceptions ☺ But at the same time an oversized Xmx has negative impact on performance due to long and slower GCs.
- Use multiple threads and/or instances to your system’s capacity
- Next step… consider cloud and auto-scaling
- Be thorough and be patient because it will take a few iterations
- Just like a fine-tuned instrument, you will enjoy the beautiful music for a long time!

# Compose for JanusGraph

## What is it?

- Compose is an open-source database hosting provider
- Supports backups, monitoring, performance tuning, and a full-suite of deployment management tools backed by a 24x7 support and operations team
- Offers JanusGraph technology with Scylla database
- https://www.compose.com/janusgraph



# What's next?

The journey continues…

- Find ways to improve JanusGraph performance
- Join us if you are interested in graph performance
- Work with us if you have graph datasets
- Talk to us if you have any comments or suggestions

## Thank you for keeping performance in mind !!

Chin Huang, chhuang@us.ibm.com;github:chinhuang007
Yi-Hong Wang, yh.wang@ibm.com;github:yhwang
Ted Chang, htchang@ibm.com;github:tedhtchang