Auto Scaling of Key Value stores

Raghukul Raman

IIT Kanpur

April 27, 2019

Outline

Outline

Aim

- Design a system to collect real-time statistics on cluster-based system.
- Collect statistics for different queries on different scaling configurations and try to find possible bottlenecks in the system.
- Providing an auto-scaling solution for key value stores.

Outline

Key Value Stores

- Data is organized in just two columns - a **key**, and a **value**.
- Actual data is value - can be object, keys are used to index these data objects.
- Satisfy **BASE** property.
  - Basically Available
  - Soft state
  - Eventually consistent
- **CAP** theorem
  - Consistency
  - Availability
  - Partition tolerance
- Fall into CP category.
- Mostly In-memory - extremely fast compared to traditional DBs.

Scalability

- *Def:* Property of a system to handle a growing amount of work by adding resources to the system[Bondi, Andre - 2000].
- Two variants:
  - **Vertical Scaling:** Increasing the resources in the server which we are currently using, i.e increase the amount of memory, CPU, etc.
  - **Horizontal Scaling:** Increasing the number of servers (instances).
- Benefits of horizontal scaling:
  - Make the system fault tolerant.
  - Downtime.
- Several load balancing schemes are used in horizontal scaling.

Auto Scaling

- Implemented in four phases of the MAPE loop:
  - Monitoring
  - Analysis
  - Planning
  - Execution
- In this project, we aim to work on the monitor and analysis phase.
- Why AutoScaling?
  - To prevent overprovisioning.

Redis

- Developed by Salvatore Sanfilippo(*antirez*).
- Open source, in-memory data structure store, used as a database, cache and message broker.
- Supports other data structures like strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs.
- Built-in support for replication, on-disk persistence and automated partitioning with Redis cluster.
- Master-slave asynchronous replication.
- Transaction, expiration time (BigTable).

Redis Cluster

- Collection of redis nodes
  - Able to communicate among themselves.
  - Able to respond to requests collectively.
- Data is automatically sharded across multiple Redis nodes.
- Every redis node require 2 ports:
  - The lower one is used to serve clients.
  - Other used for Cluster Bus (node-to-node communication).
- CRC16 hashing system - 16384 hash slots.
- Each cluster node is responsible for a subset of hash slots.
- Hash tags.
- After cluster meet, every node contains node - hash slot mapping.

Resharding

- Redis Cluster supports the ability to add and remove nodes while the cluster is running.
- Same basic mechanism can be used in order to rebalance the cluster, add or remove nodes.
- Moving hash slots through Cluster Bus.
- No down time - with the help of MOVED Error.

Redirection

- Client knows nothing - can send request to any node.
- On receiving a request:
  - Return the value if serves that hash slot.
  - MOVED error, if not served.
- MOVED error response also contain IP:PORT of node serving that hash slot.
- Still is an overhead.

Partitioning

- The way how we shard data among different nodes of redis cluster.
- **Client side partitioning:** redis clients select the node to which read/write request need to be made.
- **Proxy assisted partitioning:** client sends request to a proxy, which analyzes the request and forwards it to the correct node.
- **Query Routing:** can send request to any node, and the node will forward our request to the desired correct node.
- Examples: Jedis, Twemproxy, Redis Cluster.

Twemproxy

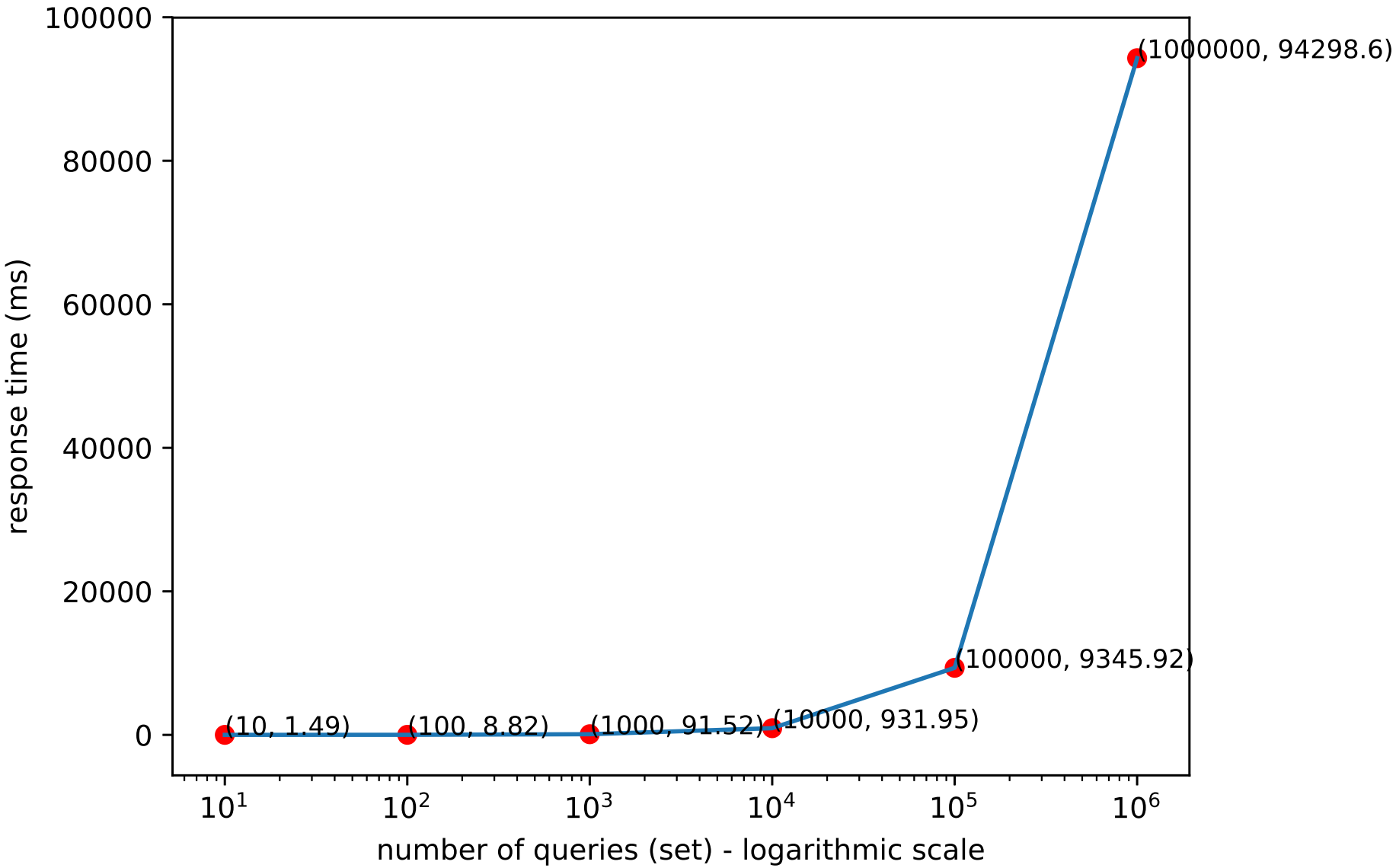
- Proxy assisted partitioning implementation.
- It maintains **persistent** server connections.
- Shard data automatically across multiple servers, keeps copy of node configurations.
- Not a single point of failure.
- Stats monitoring port.

Outline

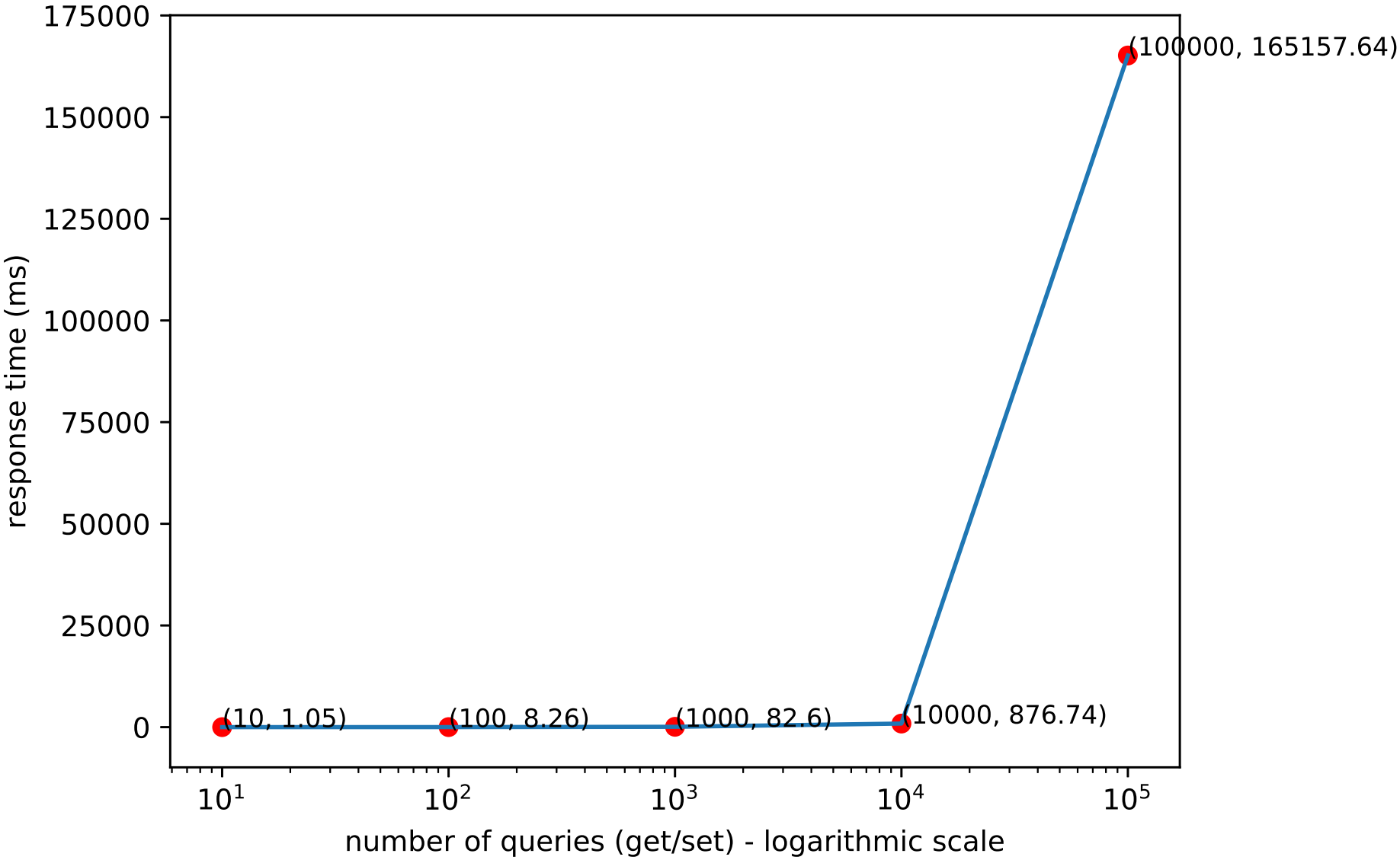
Experiments

- Collecting stats - by limiting resources.
- Generate random strings of length 64 - mimic SHA2.
- Tests based on get/set commands.
- Resource limitation enforced using Docker.

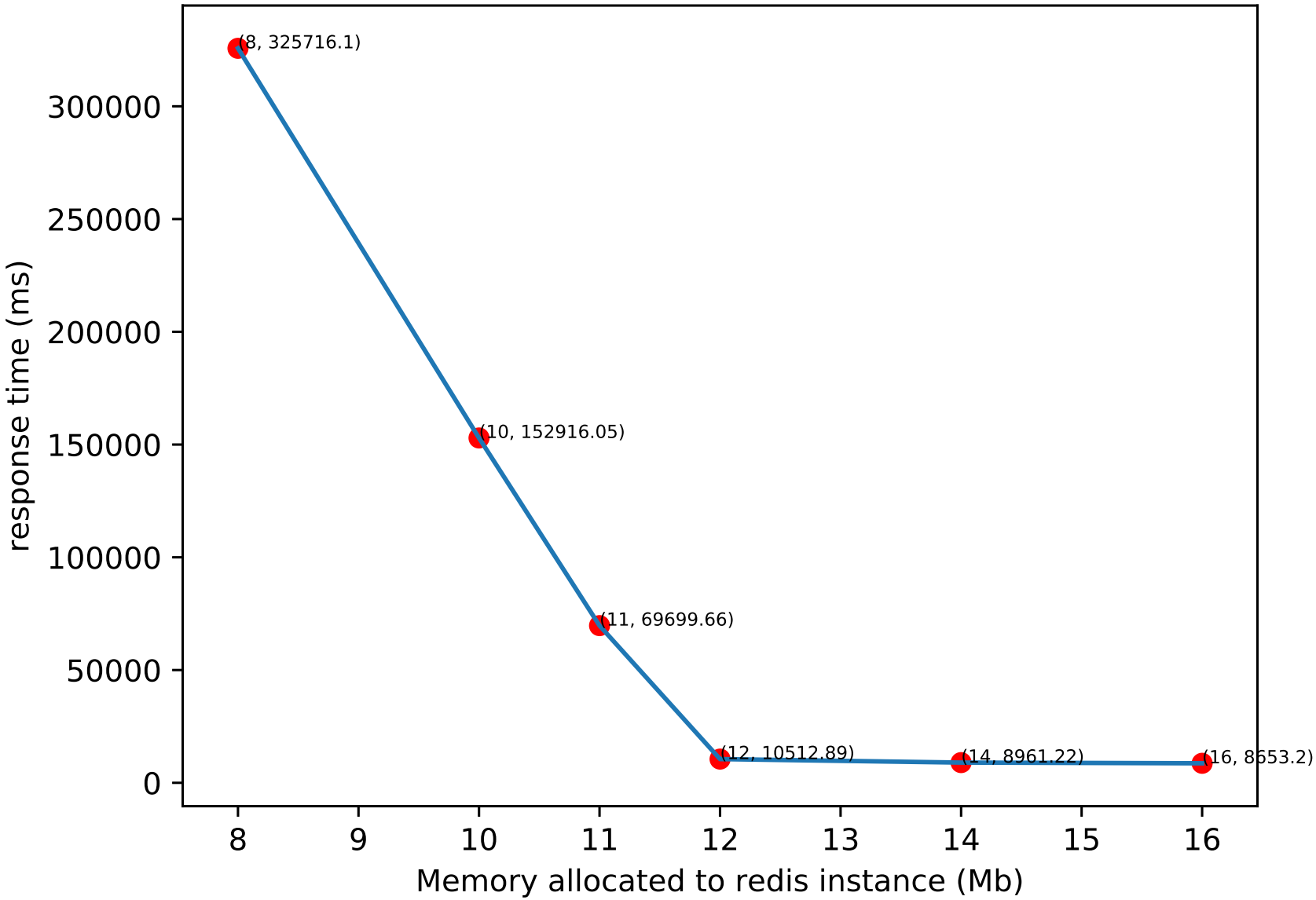
Single redis instance, with no memory/cpu restriction



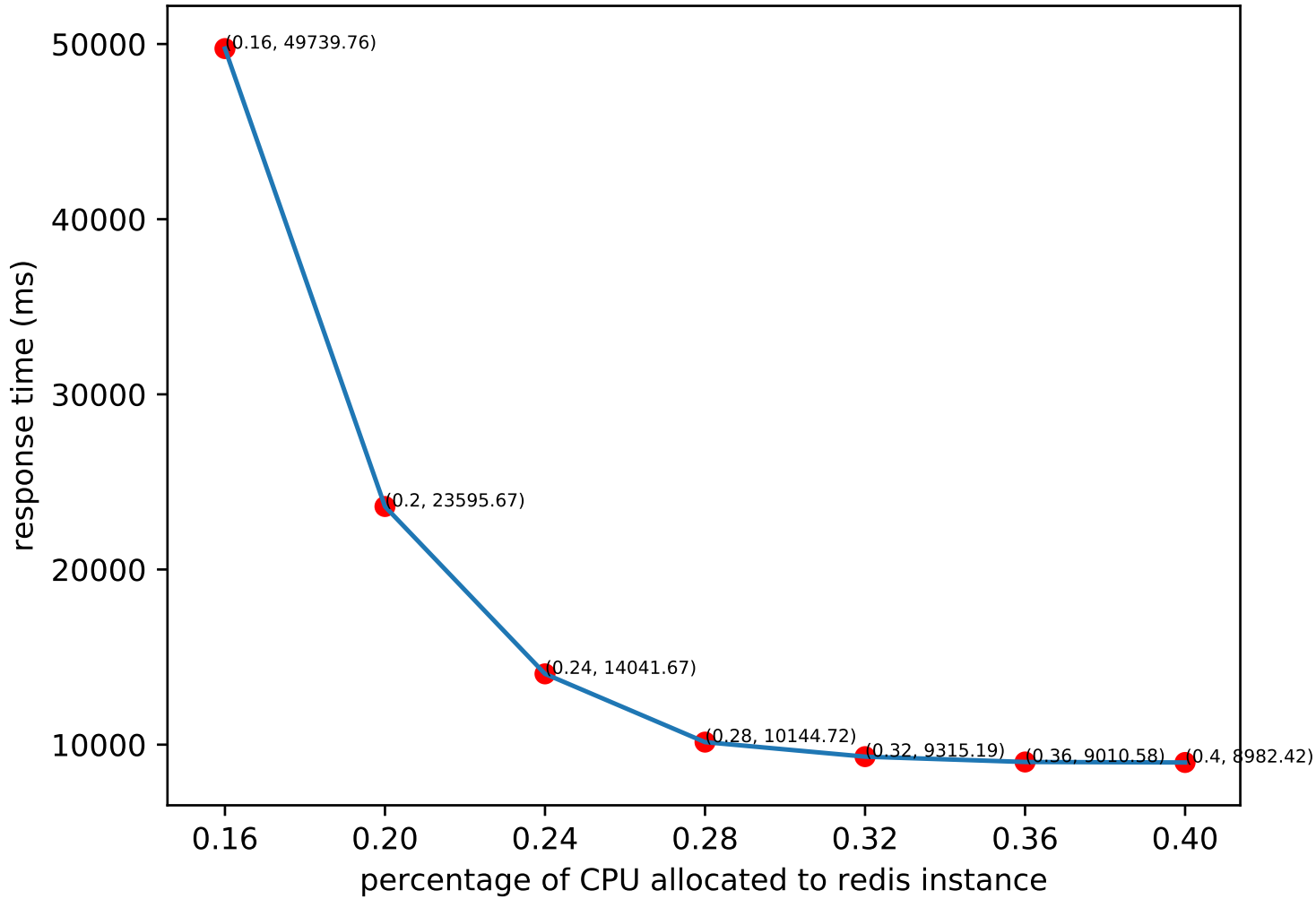
Single redis instance, with max main memory = 10Mb



10^5 queries on redis instance, with varying main memory

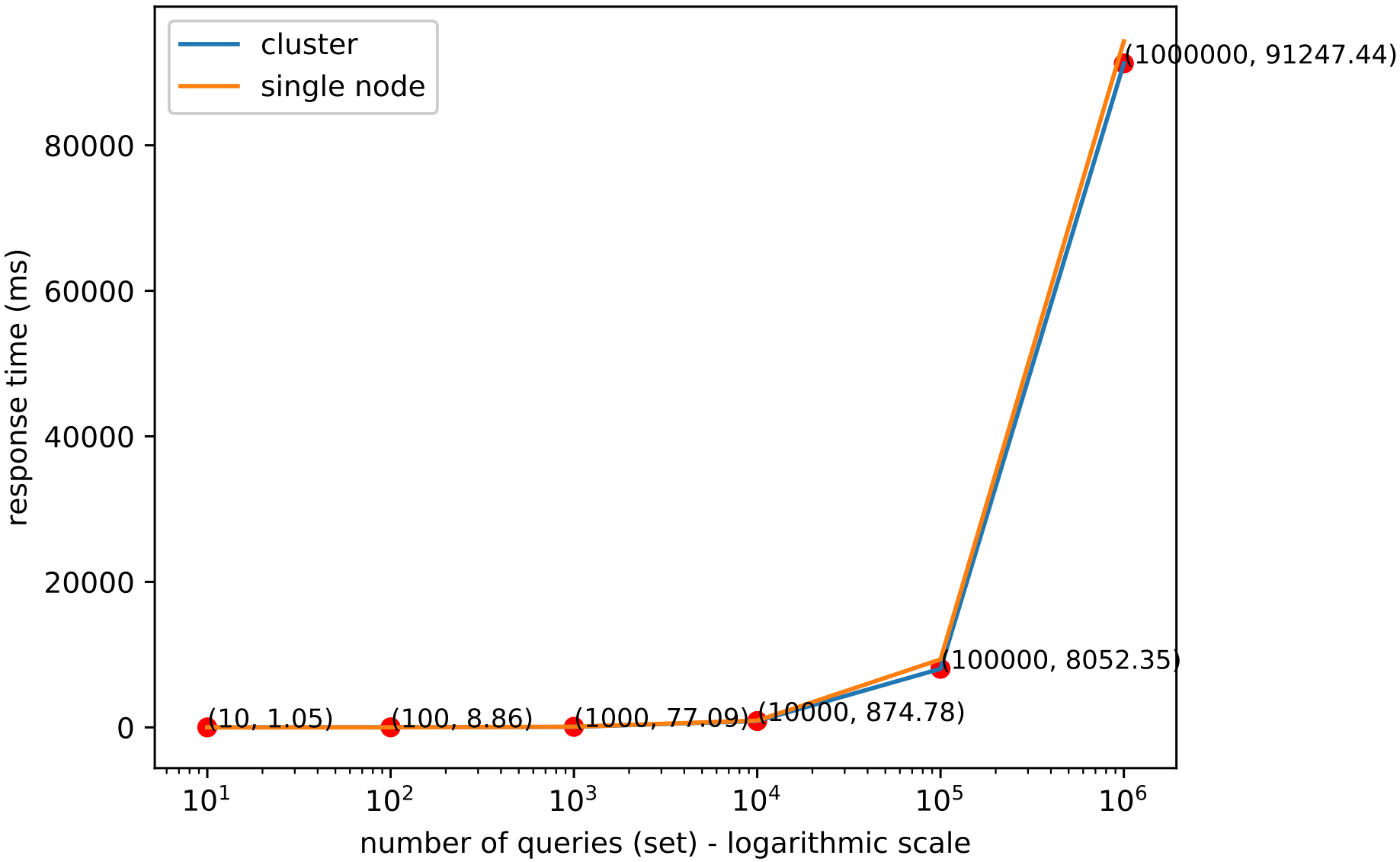


10^5 queries on redis instance, with varying CPU

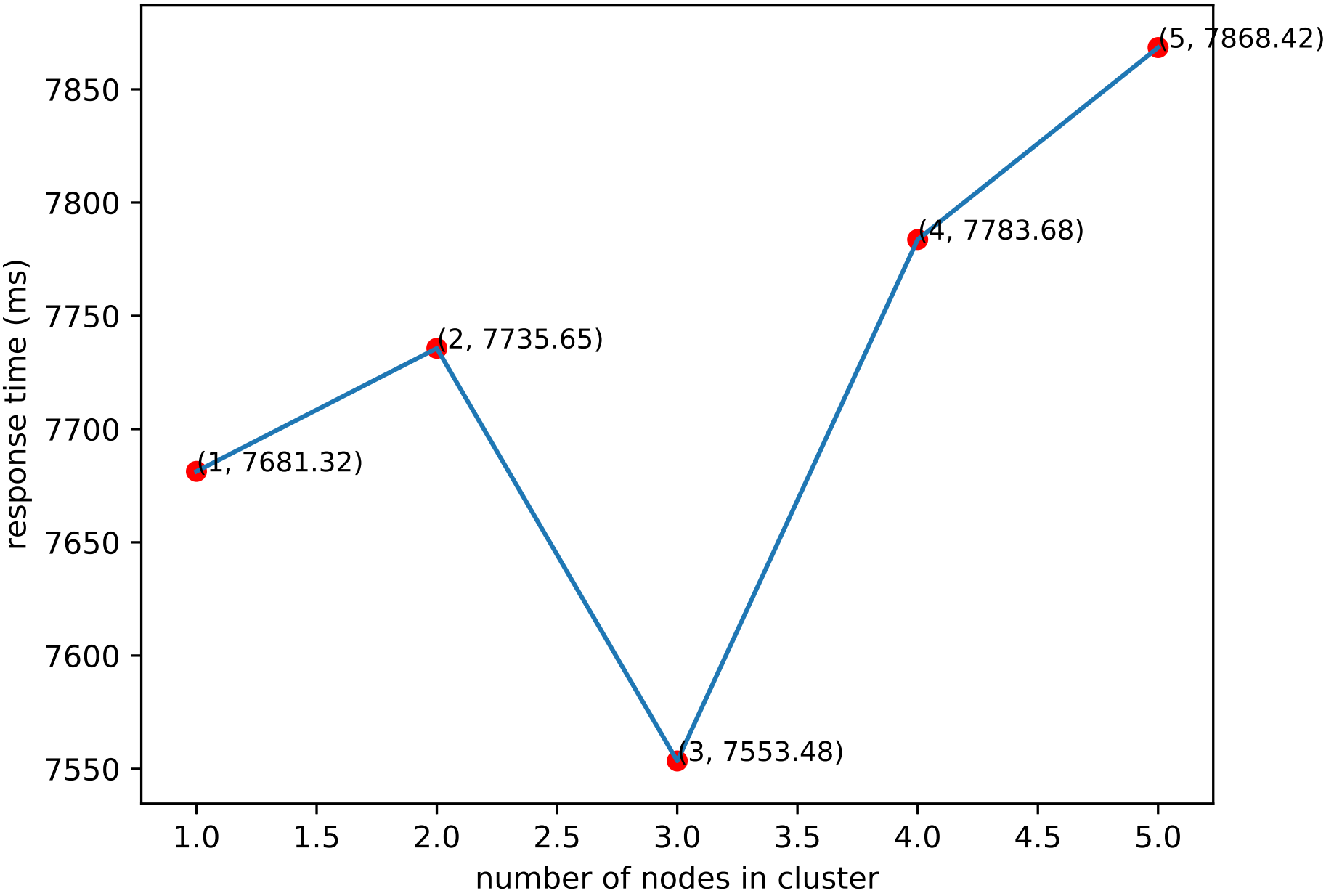


- fractional CPUs?

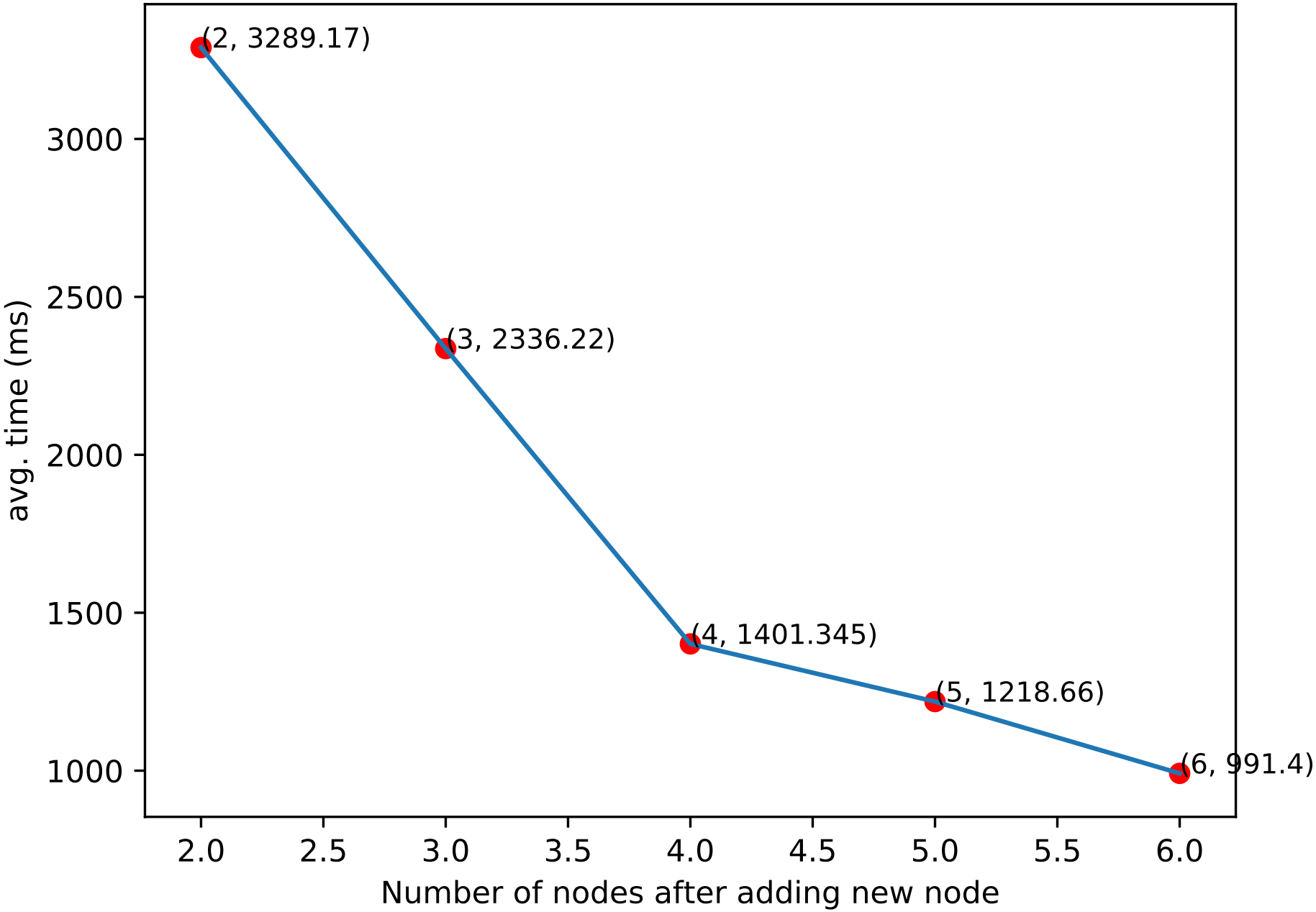
Single redis instance, compared with 6 Node cluster



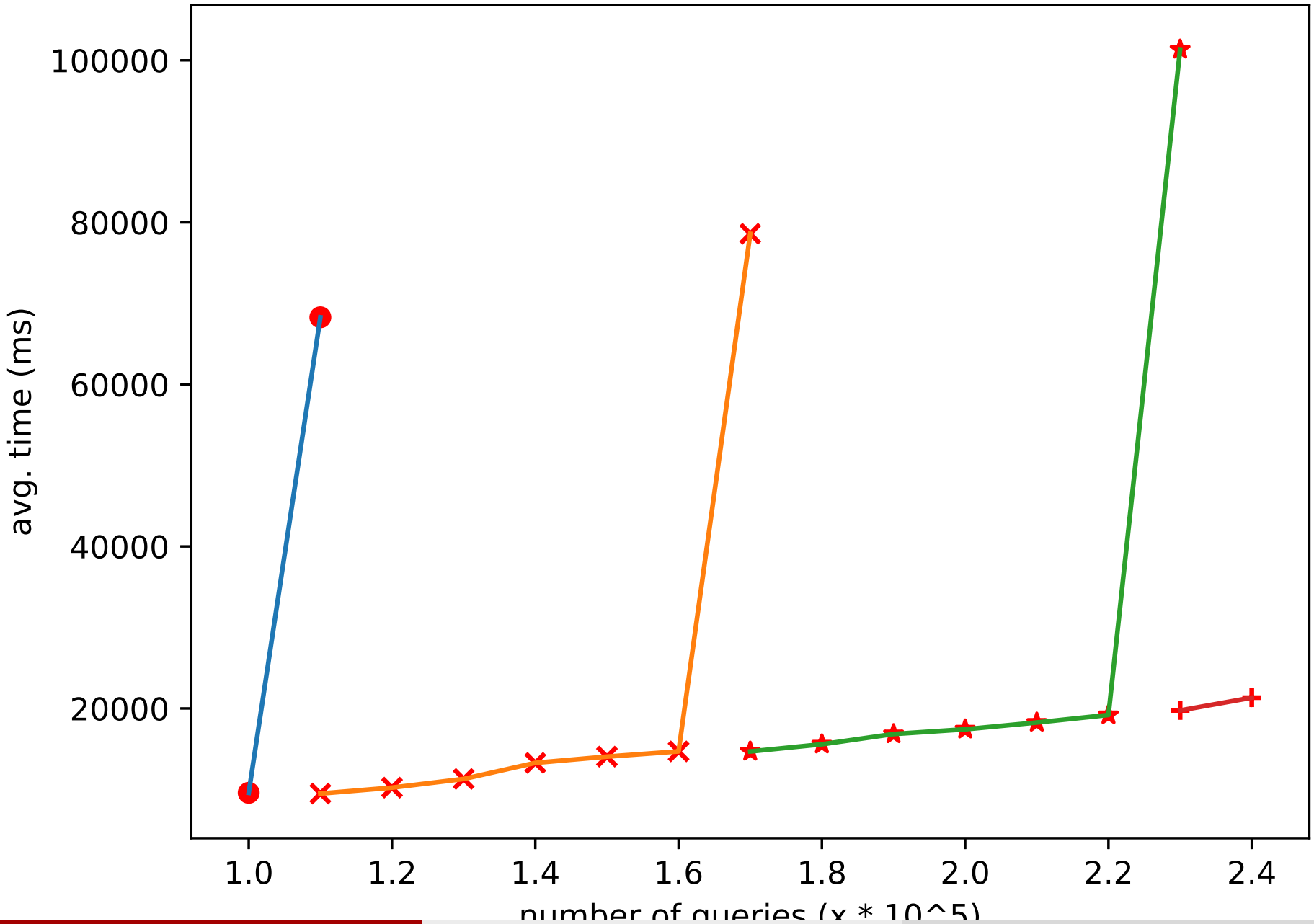
10^5 queries on different node configurations



10^5 keys present, setup time for adding new node



Varying queries and number of nodes



Outline

Machine Learning based solutions

- *Auto-Scaling Network Resources using Machine Learning to Improve QoS and Reduce Cost* arxiv-1808.02975
- Measured network traffic load to dynamically react to traffic changes.
- A classification problem.
- This ML classifier learns from past network scaling decisions and seasonal/spatial behavior of network traffic load to generate scaling decisions ahead of time.
- Models used:
  - Reduced Error Pruning Tree.
  - Decision Table.
  - Multi-Layer Perceptron.
  - DNNs.
- Features used: startup time, resource utilization factor, etc.

Static metric based approach

- *Containers Orchestration with Cost-Efficient Autoscaling in Cloud Computing Environments - arxiv:1812.00300*
- Simple Autoscaler.
  - Elapsed time since last instance launched >= provisioning_interval
- Single Instance Binding Autoscaler.
  - Multiple requests to autoscale triggered by the same unschedulable pod may be invoked.

Performace models

- *Automatic configuration and scaling of stream processing pipelines - arxiv:1812.09442*
- Precise performance models can be automatically learned for distributed stream processing systems that can predict the execution performance of a job even before deployment.
- These models can be used to optimally schedule logically specified jobs onto available physical hardware.
- These models and the derived execution schedules can be refined online to dynamically adapt to unpredictable changes in the runtime environment or auto-scale with variations in job load.

Outline

Further Works

- Implement dynamic scaling feature in Twemproxy.
- Add a service on top of Twemproxy to apply scaling logic (based on statistics received from Twem).
- Test some state of the art machine learning algorithms like Meta learning, Siamese nets, etc.

Thank You!