Tuning Databases Towards Reduced Energy Consumption

Jiun-Ting, Shing Ng, Matt Blakely, group 11

1 Introduction

In recent years, tuning configuration parameters to maximize database system performance has been a very popular research topic and has substantially practical values. A large amount of research[3][5][17] has found that a significant increase in performance in terms of throughput, CPU and memory usage, and other performance related metrics. More importantly, research has found that tuning only a few important knobs would lead to virtually the same performance increase as configuring a wide range of parameters[5]. However, the direction of most of the research has been focusing on increasing performance. They have neglected another important aspects that can also have a significant impact especially for data centers.

These neglected aspects are non-performance metrics that could potentially affect the cost and reliability of database systems. One way that can affect the cost[13] and reliability is the power consumption of the database. For instance, even though the tuning of the database system increased performance, the power consumption could potentially increase significantly. The corresponding issues such as cooling[11][20] and integrity of the hardware might be compromised. The cooling and hardware integrity issues could cause catastrophic consequences such as systems failing to output the same performance consistently or system breakdown. With the database systems down, the increased performance will not matter because the system is not operational anymore. In addition, high power consumption naturally will cost more in electricity. The increased cost for either electricity or purchasing more powerful cooling hardware could negate the benefits and potential revenues from increased performance.

Therefore, measuring the power consumption while tuning database systems is critical in order to keep cost-to-performance in check. This project focuses on this area to achieve an equilibrium between high performance and reasonable power consumption.

2 Background

This project is based on a previous research paper that focused on accelerating auto-tuners by tuning a small number of knobs in database systems for maximizing performance[5].

Originally, it is time-consuming if we try to evaluate different settings from the entire parameter configuration space since modern database systems has at least hundreds of knobs and the combinations these knobs can produce are astronomical. As a result, auto-tuners are used to streamline and facilitate the exploration of the space of parameter configuration in database systems.

In this paper, we addressed one of the neglected areas about auto-tuning for non-performance metrics (power consumption) while keeping the same performance. We built this paper based on a previous auto-tuning and bench-marking framework Nautilus[5] and made modifications to it in order to measure power consumption and performance metrics such as CPU and memory usage. Also, as part of the work, we kept tuning the five most influential knobs for different combinations to maintain high performance as found in previous research and measured its power consumption.

The main hypothesis we want to test through this paper is: we can tune the database system in a way that maintains high performance and relatively low power consumption. To test this hypothesis, we conduct a detailed and systematic measurement-oriented experiments. Specifically, we measure the performance and power consumption of two different database systems (PostgreSQL [1] (Postgres for short) and Cassandra [2]) with different configurations and compare and analyze the result measurements. Our results confirmed that we can achieve high performance with relatively low power consumption in both Postgres and Cassandra. Based on the accelerated auto-tuning approaches used by Nautilus we can quickly find important knobs that dictate both performance and power consumption. In the end, we will mention some of the future directions for research in different database systems and hardware.

3 Design

In this section we will describe the software and hardware components we worked with and modified, our approach to creating configurations, and, how the evaluations were run in Cloudlab [12]. Section 3.1 will discuss the tools we used for measuring power usage, section

3.2 will discuss the Nautilus framework and our modifications to it, section 3.3 will discuss our process for generating database configurations, and section 3.4 will discuss how the database configurations were run in Nautilus on Cloudlab.

3.1 Power Measurement

This subsection will describe the two tools we used to capture power usage and the two primary units for discussing energy and power. The first tool, IPMItool, leverages the openIPMI framework to collect various physical sensor data from the server. We collected data from a few sensors, but we will focus on the PSU1_PIN which captures watts. The second tool, Intel Running Average Power Limit (RAPL), tracks a running counter of Joules consumed by various sub components of the node. Combining the tools we were able to determine the energy usage for: the overall system, CPU, RAM, and combined "other" components including cooling and storage.

3.1.1 Energy Units

There a few units which are used semi-interchangeably, but should be disambiguated. The three units are Joules, Watts, and Kilowatt Hours. The main unit we will use throughput the paper is Watt which is a rate, Joules per second, and is generally referred to as power. Joules is a unit of energy and is similar to the commonly used unit in data center literature of kilowatt hours (kWh). Joules and kWh have the same unit signature but kWh is the number of thousand watts used over an hour span. Since our tests were short we will not use kWh. We will use Watts for a rate of energy consumption and when describing total energy consumed it will be in Joules.

3.1.2 IPMItool

IPMItool is a user facing wrapper to interact with the openIPMI framework which is an implementation of the Intelligent Platform Management Interface standard. IPMItool allows us to interact with kernel drivers and modules that can directly access different sensors physically attached to the server. The openIPMI framework maintains a repository of hundreds of Sensor Device Records (SDR) for sensors ranging from temperature, fan speeds, voltages, and critically for us, watts. The Wisconsin cluster of Cloudlab has two power sensors, POWER USAGE and PSU1_PIN. In order to collect the data, the sensor must exist on the server. The Utah and Clemson clusters did not have any power sensors so we could not use any instances from that cluster. Both sensors capture Watts but the POWER USAGE sensor had a higher error range (plus or minus 4 Watts) so we ignored that sensor and used only the PSU1_PIN since it claims to be exact. The SDR repository returns a large amount of metadata about the sensor reading so we used a grep command to filter out the unnecessary information. See Listing 1 for an example of the command to run IPMItool.

Listing 1: IPMItool command and result

```
sudo ipmitool sdr get PSU1_PIN | grep "Reading"
Sensor Reading: 112 (+/- 0) Watts
```

There is little documentation about openIPMI or IPMItool so even though the example from Listing 1 is quite simple, we spent a lot of time discovering how to use the tool and how to interpret the results. Additionally IPMItool pulls the sensor information through the /dev/ipmi0 device and requires various supporting kernel modules. This caused quite a bit of contention in Nautilus because all of the worker processes run inside of containers. We will discuss this more in the Nautilus section.

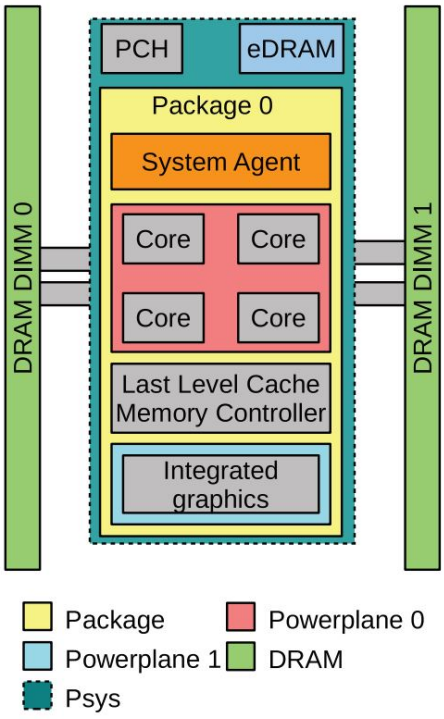

Figure 1: Intel RAPL domain structure described in K. Kahn et al. [9]

3.1.3 Intel RAPL

Intel RAPL is a processor specific framework developed by Intel which keeps a running counter of the Joules consumed by different domains of the processor architecture. The framework was intended to be used in laptops

to help manage battery usage but in recent years has been used in data centers to help monitor and manage energy usage. Intel RAPL is able to capture Joules used at the overall processor level, as well as per CPU socket, per GPU, and per DRAM associated with each socket. Figure 1 shows an example Intel RAPL breakdown of a single package domain. The information that Intel RAPL provides is closely tied with the architecture of the processor. For our tests using C220g5 instances the processor was broken down into these 4 domains: package_0, package_1, package_0_DRAM, and package_1_DRAM, each package represented 10 cores.

We polled the Intel RAPL counters directly in Python using a simple and elegant API created by William Katsak called py-rapl [7]. The API handles determining the available domains, then captures the start and end Joules from the different counters for each processor domain. Finally, it computes the Watts by finding the difference of those counters (the number of Joules consumed) and dividing by the number of seconds between the start and end events, thus producing Joules per second.

There is some debate about the accuracy of software tools at measuring energy consumed however both J. Kelley et al. [8] and K. Kahn et al. [9] concluded that Intel RAPL is accurate enough to be used as a power monitoring tool. Before we turned to the py-rapl API we attempted to use released tools turbostat and powertop but we could not make them function properly inside of a container due to permission and Linux device contention. However, they were not a complete waste of time because we used them to confirm that we accurately pulled Watt readings from the Intel RAPL framework using py-rapl.

3.2 Nautilus

Nautilus is a framework created by K. Kanellis et al. [5] to gather large amounts of performance data from various databases while using different database configurations. The framework is still nascent but is flexible and well designed, allowing us to extend it to capture power usage through IPMItool and Intel RAPL.

Nautilus is a distributed framework built upon Celery [15] that receives tasks created by a driver program and distributes those tasks to worker processes on the same or other hosts. The worker process initializes a database instance of a container and loads required records for a given benchmark workload and stores that database information in persistent storage through docker volumes. The worker process then initializes a new database instance with specified configuration but to save time loads the cached database record files and runs the workload while capturing metrics. Once the workload is complete, the results are returned to a MongoDB and the database instance with the specified configuration is destroyed. Since the workload can include insertions, updates, or deletions, the configuration specific database records cannot be used in future runs so when the configuration specific database instance is destroyed, its updates are not made to the cached version. Because the cached version is not modified, it can be used repeatedly for new tasks. The cache mechanic makes clever use of containers to save a significant amount of time per run by not creating the same initial dataset every time. See Figure 9 in Appendix A for a diagram of the Nautilus architecture. The diagram shows four servers because that was our configuration, Nautilus could use a different number of servers.

The modifications we made fell into four categories: deployment generalization, worker permissions, creating samplers to collect information, and creating a driver. The driver and deployment changes were fairly straight forward and allowed us to use different numbers and types of hardware, and to queue up tasks for the different database configurations we wanted to measure. Collecting the information was anything but straight forward due to the workers running in containers but needing to access kernel information. The crux of the problem stemmed from the fact that containers are virtualized inside of the host system.

Even though we could successfully run our scripts to collect power information on the host machine, the containers lacked both access and the correct devices to run those commands on. To resolve the permission issue we discovered a docker setting called privileged which when set to true in a service gives the container process root permissions. This setting is discouraged and not publicised because it nullifies the sandbox premise of the container but it allowed us to access kernel space. Even with the permissions we were not able to run IPMItool in the container because the devices IPMItool depends on did not find any information in the virtualized operating system. To resolve this we discovered that containers can also map devices. The setting is intended to map storage devices into a container. Regardless, once we mapped the OpenIPMI devices the container was able to read the sensor data.

Once we had access to the information we needed to implement a mechanism within Nautilus to run arbitrary stateless commands on an interval. Nautilus allowed for a sampler abstraction, however it had only one sampler to collect Postgres database performance metrics at one second intervals. To support our disparate collection needs we built out a generic sampler that is initialized with an interval to run commands on and handlers to register python functions against. Once the workload finishes the sampler returns a JSON object with each timestamp where a sampler ran and the output of each regis-

tered python function. This allowed us to associate each sampler output to the same interval and thus compare different metrics like CPU utilization, RAM utilization, system Watt usage, and processor watt usage at the same timestamp. The sampler then serialized the results and passed them off to Nautilus to store in the result MongoDB. Reference Listing 3 for an example output of the sampler.

Using Nautilus was difficult because it was not yet in a stable version. We worked through many bugs and idiosyncrasies carried over from the initial implementation. Investigating and troubleshooting Nautilus was challenging because it is both distributed and highly containerized causing us to invest a significant amount of time into getting our experiments running. However, this was time well spent. We learned about docker and how docker interacts with the storage and network systems, and how docker interacts with the kernel. Additionally, we had no experience working with distributed framework, to get our experiments running we learned a lot about Celery and how to approach troubleshooting a distributed system. While not directly related to our project, learning this content feels within the ethos of the Big Data course and we enjoyed learning about it.

3.3 Configuration Generation

In this section we will describe our configuration generation strategies and how Nautilus compiles configurations. Nautilus expects that each task will be associated with one database configuration to evaluate. Each database has a pre-configured default configuration which it will use if no configuration is specified. If a configuration is specified, Nautilus will treat all values in the specified configuration as overrides over the default values, if no value is specified for a knob, it will fallback to the default value. Each configuration is a dictionary where the key is the knob and the value is the value the knob should be configured with. All together this means that for each experiment we wanted to run, we needed to generate one JSON file with a single list of dictionaries containing the knobs we wanted to change from their default values per task.

To generate the configurations we created a script that read in a JSON file with metadata about each knob to include in the override configuration and created sets of configurations in a sweep or random pattern. We categorized each knob we wanted to test for Postgres and Cassandra as enumeration, or range (integer or float). For enumeration knobs we listed out the possible values and for range we listed the min and max values.

To generate configurations in a sweep pattern, the script would load in the metadata file, the number of configurations to generate and then would output one configuration per knob per value. For instance, if the knob was a boolean, it would generate two configurations, one set to true, the other set to false. Nautilus would then load the default values for all other knobs required by the database. If the knob was a range, the script would compute a step size by dividing the range of values by the number of requested configurations then output a configuration for each multiple of step size between the min and max values. We used the sweep pattern configurations to investigate how a single knob affects the power consumption of the database.

To generate configurations in a random pattern, the script would load the metadata file and the number of configurations and output configurations with each knob in the metadata file set to a random valid value. For instance if we ran the script on a metadata file with [*concurrent_readers*, *concurrent_writers*, *memtable_heap_size*] then each configuration would have those three values each set to a different random value in the different configurations. We used the random configurations to try to identify power consumption trends across knobs.

Some of the knobs had dependencies between them. For instance, in Postgres if *fsync* was set to *false*, and *wal_sync_method* was set to *fsync* or *fsync_writethrough* then the database would fail to initialize. Since we could not determine all of these dependencies we decided not to try to account for it during the configuration generation. Instead, since Nautilus is fault tolerant, when a task with an invalid configuration failed, Nautilus gracefully cleaned up the task and moved on to the next one. Deferring the dependency identification to run time wasted a bit of time while running experiments but saved us time overall by not trying to prevent these failures from happening.

3.4 Experiment setup

Now that we have background on the various components of our design, we can discuss our experimental setup. We conducted all of the tests on a network of four C220g5 instances of Cloudlab. Each node used 10-core Intel Xeon Silver 4114 CPU with 64s GB of memory and 480-GB SSD for storage. We used one node to host the driver and three nodes ran worker processes. This under utilized the node with just the driver on it, but because the power consumption cannot be determine at a process level, we needed to keep the processes other than the worker as minimal as possible. We ran the driver process using the webui profile which launched processes to track the task queue, the state of the result MongoDB, and in-process tasks (refer to Figure 9 for a diagram of the architecture). Listing 2 shows an example command to launch the driver

for Nautilus, to launch the worker process on a different node, we would ssh into the other node and run the same command but replace ”start driver” with ”start worker”.

Listing 2: Example Nautilus command

```
python3 deploy.py start driver
//profile=master-worker-webui
//driver=async-parallel
//driver.args.samples=./nautilus/configs.json
//driver.args.num_workers=3
//driver.args.dbms=postgres
//driver.args.initial_wait=60
//driver.args.workload=workloada
```

For each configuration we ran YCSB workload A and B for 330 seconds, just shy of 5 minutes. Each task required about 5 minutes of overhead to initialize the database and to tear it down, all totaled, each task took about 10 minutes to run. From the YCSB framework, Nautilus returns throughput in operations per second broken down by update throughput and read throughput. During the YCSB workload we ran a sampler which collected measurements every 6 seconds. We would have preferred to collect measurements at a faster interval, but IPMItool required about 5 seconds to return the PSU sensor information, fortunately the Intel RAPL measurements were nearly instantaneous. To ensure the Intel RAPL measurements were comparable to the IPMItool measurements we measured the Intel RAPL Joules counter before and after the IPMItool measurement and divided the Joule difference by the time it took for the IPMItool to run. This ensured that we found the average Watts measured by Intel RAPL and average Watts found by IPMItool for the same time span. For each 6 second interval we captured Watt usage information like in listing 3.

Listing 3: Example interval data from sampler, the value in each field is a Watt

```
"2020-11-27T15:45:14":{
        "psu_watt":168,
        "total_compute":78.25,
        "package0":41.27,
        "package0_ram":6.37,
        "package1":36.98,
        "package1_ram":4.95
}
```

For each experiment we collected around 55 Watt usage data points like above.

4 Evaluation

In this section we will discuss some findings from our experiment. We first compared the performance and the energy consumption of two workloads and databases. Then, we observed the effect of key knobs and the variation of energy consumption with some visualizations. In the end, we analyzed the composition of energy consumption in different settings.

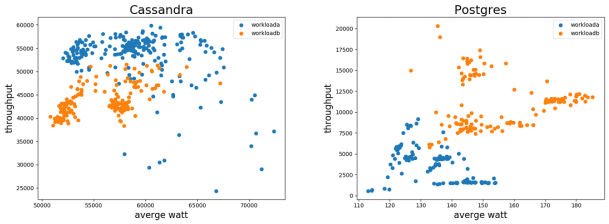

Figure 2: Distribution of Energy Consumption and Performance. Y-axis is Throughput (operations per second), X-axis is Watts. See Figure 10 in Appendix B for zoomed in view

4.1 Workload and Database

To understand the relationship of energy consumption and performance metrics of different workloads and databases, we ran about 220 tasks per each type of workload (A, B) and database (Cassandra, Postgres). Figure 2 shows the distribution of average Watt and throughput of both databases. We see that Cassandra tasks have higher throughput in general. Besides, since Cassandra is optimized for heavy write workloads, it's not surprised to see that it performs well on workload A.

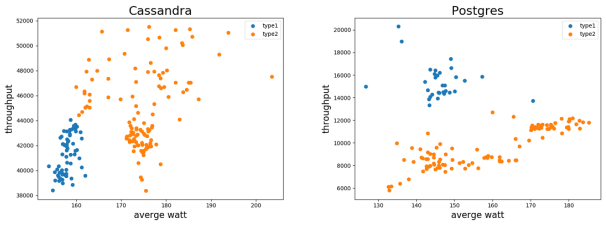

Figure 3: Clusters of Read Heavy Workloads. Y-axis is Throughput (operations per second), X-axis is Watts. See Figure 11 in Appendix B for zoomed in view

Moreover, it is worth noting that workload B (read heavy) in both database seems to concentrate in two groups. Figure 3 shows the result of agglomerative clustering that we applied to the read heavy workloads.[1] [2] For Cassandra, we see that one type has lower energy

[1] Both throughput (operations/sec) and average watt are mean value of a workload.

[2] The clustering are based on throughput, mean of energy consumption, and the standard variation of energy consumption. As the pattern shown in Figure 2, we set the number of cluster to 2. We have also tried K-means, spectral clustering and some other density based clustering methods without the requirement of specifying cluster number. However, the results are not consistent and it seems that the agglomerative clustering one is the most intuitive.

consumption and throughput, while the other one has higher performance and energy consumption. [3] Note that some tasks in type 2 are not energy efficient, which consume more watt given a certain throughput. For Postgres, one group has higher throughput and relative low energy consumption with small variation, while the other type has lower throughput and unstable energy consumption.

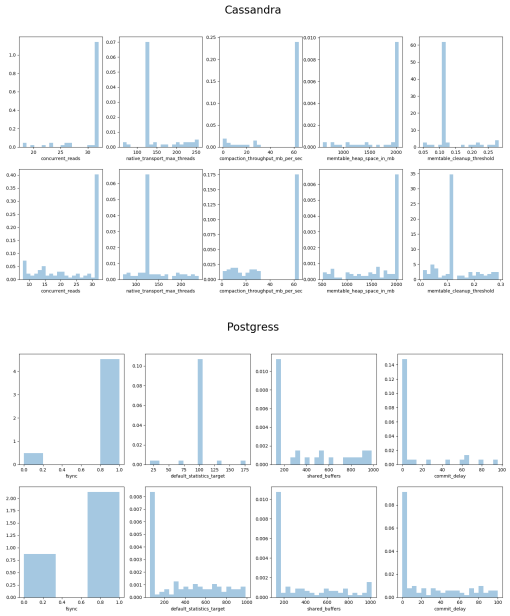

Figure 4: Distribution of the Key Knobs of Read Heavy Workload (Upper subplots in each database are type 1, and the lower ones are type 2). X-axis is the value of knob. Y-axis is the density of each knob value derived from Kernel Density Estimation. See Figure 12 in Appendix B for zoomed in view

4.2 Key Knobs

To explore the properties of clusters that we categorized for read heavy workloads, Figure 4 shows the distribution of their configurations.The x-axes and y-axes are the value of each knobs and their density derived from Kernel Density Estimation.[4] Since most of our samples are generated by changing only one features, a large portion of knobs are set to default values.

In the graphs for Cassandra, it seems that there doesn't exist a decisive difference between two types of read workloads. However, the density of knob values that is not equal to default values in type 2 are a bit higher in general, which implies that the default values may be better choices. For the read heavy workloads of Postgres, the inefficient group appears to have higher *default_statistics_target*. Based on the documentation, raising *default_statistics_target* might allow more accurate planner estimates to be made, particularly for columns with irregular data distributions, at the price of consuming more space and slightly more time to compute the estimates. However, a lower limit might be sufficient for columns with simple data distributions.

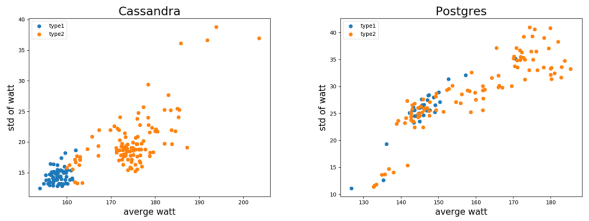

Figure 5: Clusters of Read Heavy Workloads-Average and Standard Deviation of Energy Consumption. Y-axis is Watts and X-axis is Watts. See Figure 14 in Appendix B for zoomed in view

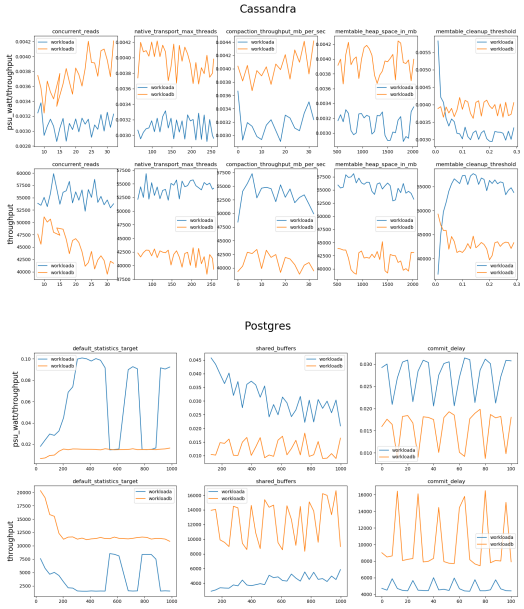

Figure 6: Throughput and Energy Efficiency with Different Knob Values. Y-axis Throughput (operations per second). X-axis is knob values. See Figure 13 in Appendix B for zoomed in view

To identify the affect of a given knob, we generated workloads by changing the value of the knob while keeping others the same. In figure 6, we see that the throughput of workload B decreases as *default_statistics_target*

[3] For simplicity, we'll call them type 1 and type 2 later.

[4] See the documentation of Cassandra and Postgres for more details about their configurations [1][2]

grows to 200, which is consistent to our above finding. We can also see that the throughput of workload A improved as the value of *memtable_cleanup_threshold* increased to 0.11. Moreover, some features appear to have an obvious wave pattern (eg. *commit_delay*, *memtable_heap_space_in_mb* and *shared buffers*). Based on these results, we may want to notice such property when setting configurations.

4.3 Variation

For database users, it would be easier to schedule tasks and to more fully utilize servers if workloads are predictable. Hence, we care about the variance of energy consumption. Figure 5 shows the relationship of energy consumption and variation of two types of read workload. In the graphs for Cassandra, there's a low consumption/low variance group and a high consumption/high variance one. Regrading Postgres, we see a group with relatively low consumption with low variance and a group with high consumption and high variance. Moreover, we also found that the value of knobs matters in some case. As the Figure 7 shows, [5]the variance of workload B increases as *default_statistics_target* grows to 200. And the variance of Cassandra's energy consumption also decline as *memtable_heap_space_in_mb* and *memtable_cleanup_threshold* fall to certain level.

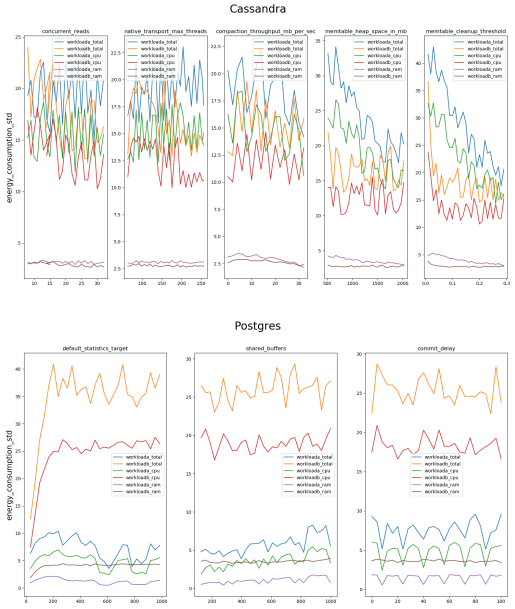

Figure 7: Standard Deviation of Energy Consumption with Different Knob Values. Y-axis is Watts. X-axis is knob values. See Figure 15 in Appendix B for zoomed in view

[5]Energy consumption

4.4 Composition

After decomposing the contribution of energy consumption, our result in Figure 8 shows that CPU takes the most power, and confirmed the work from A. Karyakin et al. [6] that claimed RAM requires consistent power usage regardless of utilization of the RAM.

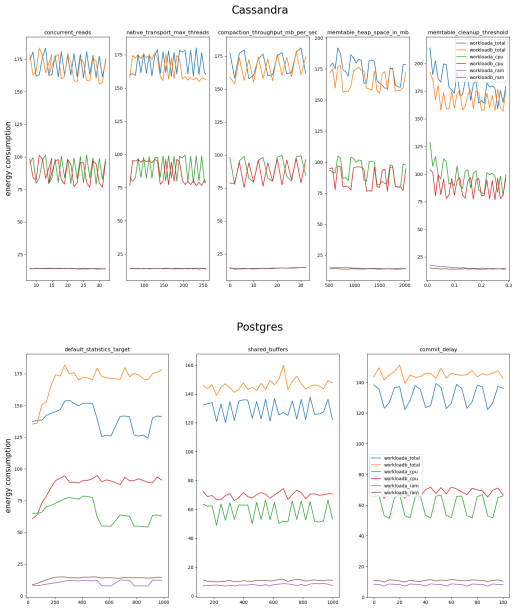

Figure 8: Composition of Energy Consumption with Different Settings. Y-axis is Watts. X-axis is knob values. See Figure 16 in Appendix B for zoomed in view

5 Related Work

5.1 Hardware/Platform level power consumption research

M. Poess et al.[10] compares power consumption and performance among various hardware configurations, which includes options of CPU (processor clock speed adjustments), memory (low/high), storage (low/high end), and data compression. A. Karyakin et al. [6] analysed how memory usage affects energy consumption and determined that since memory usage is evenly distributed across the memory modules, power consumption is not proportional to load. S. Götz et al. [4] analysed CPU energy consumption while changing frequencies and thread count while running different database workloads. They concluded that limiting the processor frequency and thread count could have a large impact on power consumption without significantly impacting performance.

5.2 Database level power consumption research

Z. Xu et al. [18] and Rodríguez et al. [14] shows that optimization mechanisms in traditional DBMSs could easily miss query plans that are highly power-efficient and yet lead to little degradation of performance. Their result indicates that designing power aware query optimizers is a promising direction to enable power conservation. Interestingly, this is in contrast to Tsirogiannis et al. [16] which found that the fastest query path is the most energy efficient because it reduces the amount of time the database is processing the query.

5.3 Parameter tuning research

For ML mode-based tuning, K. Kanellis et al. [5] and Van Aken et al. [17] use linear and nonlinear models to capture the important attributes first, and then auto-tune them based on the selected knobs. Some recent studies by Zhang et al. [19] adopt a deep reinforcement learning models to solve this optimization problem. All of these studies focus strictly on performance and do not take power consumption into consideration.

6 Future work

We have shown that database configurations can have a significant impact on the energy consumption of a server running that database while keeping the high level throughput (operations per second) consistent. We will describe some future work ranging from hardware ideas to different analysis techniques

6.1 Hardware

We tested with only one hardware configuration using Intel processors but there are a number of variations which could yield interesting results. AMD recently created a patch which enables RAPL on AMD, named AMD RAPL. It would be interesting to procure two servers with the same hardware but use comparable AMD and Intel processors. Perhaps one processor brand has a better throughput per watt ratio. In a similar vein, it would be interesting to compare typical DRAM with new persistent memory like Intel Optane, or to compare solid state hard drives between brands or between PCIe versions 3.0 and 4.0.

6.2 Databases

We compared Cassandra and Postgres and they had different energy consumption and throughput profiles. In general, Cassandra had higher throughput and higher energy consumption. Where would other databases fall? It would be particularly interesting to gather different types of databases, relational, graph, key-value, timeseries, etc. and compare the energy profiles between them to determine if a certain style of database is more energy efficient. This experiment could be extended to analyze which knobs are the most influential for energy consumption. In different databases of the same style, are similar knobs similarly influential to energy consumption?

Another interesting path of investigation are distributed databases. Does energy consumption scale linearly as more database nodes are added to the distributed cluster? What server components consume the most energy? Perhaps the network becomes the main consumer but how can we measure that?

Another line of research may be comparing different energy saving techniques in databases. From our related work we see that different researchers have tried adding energy use into query optimizer cost calculation or to design new databases from the ground up to reduce energy consumption while minimally sacrificing performance. How do these different energy aware approaches compare when running the same workload on the same hardware? How do the bespoke energy saving techniques compare to adjusting comparable databases through configuration?

6.3 Workloads

We evaluated only throughput (operations per second) for the YCSB workflow but other workflows may care more about latency or handling complex analytics. It would be interesting to capture more detailed metrics (using tools like pcm) from the database like page misses, or idle time from IO or locks, or system metrics throughput in MB/sec for the RAM or storage. This may give insight into if a particular component is using energy efficiently relative to the performance it is producing.

The YCSB workloads only include operations like read and update and may not be representative of how that database is normally used. For instance, Postgres is a relational database so supports more advanced analytics like joining and aggregation but the YCSB workloads do not include those operations. It would be interesting to evaluate TPC benchmarks or other data warehouse-centric workloads to see if the same knobs influence the energy consumption in the same manner. Also, in general, synthetic workloads may not be representative of workloads for a database in industry. Would using traces obtained from the real life workloads provide more accurate or insightful results?

Could we tie it all together and create a predic-

tion model which evaluates the configuration, hardware, database, and workload to predict the energy consumption and throughput? This could be useful to cloud database providers like Snowflake or Amazon to meet the required performance but minimize the energy consumed and thus minimize operational costs.

6.4 Statistical Analysis

Due to the limited resources, our analysis are based on some relative few samples, which are generated from the importance knob found by K. Kanellis et al. [5]. Hence, our research is more like showing some idea to evaluate the energy consumption and performance rather than drawing a solid conclusion. If more resources are available, we expect to generate more random samples with more knobs to observe their properties. Moreover, it would be possible to use methods like random forest and SVM to capture the high dimensional interaction between knobs.

References

[1] *Postgres*, 1996 (accessed November 11, 2020). (https://www.postgresql.org/docs/9.5/.

[2] Apache. *Cassandra*, accessed November 11, 2020. http://cassandra.apache.org/.

[3] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[4] S. Götz, T. Ilsche, J. Cardoso, J. Spillner, T. Kissinger, U. Aßmann, W. Lehner, W. E. Nagel, and A. Schill. Energy-efficient databases using sweet spot frequencies. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 871–876, 2014.

[5] K. Kanellis, R. Alagappan, and S. Venkataraman. Too many knobs to tune? towards faster database tuning by pre-selecting important knobs. *usenix*, 2020.

[6] A. Karyakin and K. Salem. An analysis of memory power consumption in database systems. In *Proceedings of the 13th International Workshop on Data Management on New Hardware*, DAMON '17, New York, NY, USA, 2017. Association for Computing Machinery.

[7] W. Katsak. *py-rapl*, 2020 (accessed November 11, 2020). https://github.com/wkatsak/py-rapl.

[8] J. Kelley, C. Stewart, D. Tiwari, and S. Gupta. Adaptive power profiling for many-core hpc architectures. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 179–188, 2016.

[9] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(2), Mar. 2018.

[10] M. Poess and R. O. Nambiar. Tuning servers, storage and database for energy efficient data warehouses. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 1006–1017, 2010.

[11] R. Pries, M. Jarschel, D. Schlosser, M. Klopf, and P. Tran-Gia. Power consumption analysis of data center architectures. In *International Conference on Green Communications and Networking*, pages 114–124. Springer, 2011.

[12] R. Ricci, E. Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), Dec. 2014.

[13] M. Rodriguez-Martinez, H. Valdivia, J. Seguel, and M. Greer. Estimating power/energy consumption in database servers. *Procedia Computer Science*, 6:112–117, 2011.

[14] M. Rodríguez, D. Jabba, E. E. Zurek, A. Salazar, P. Wightmam, A. Barros, and W. Nieto. Analyzing power and energy consumption of large join queries in database systems. In *2013 IEEE Symposium on Industrial Electronics Applications*, pages 148–153, 2013.

[15] A. Solem. *Celery*, 2009 (accessed November 11, 2020). https://docs.celeryproject.org/en/stable/.

[16] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. Analyzing the energy efficiency of a database server. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 231–242, New York, NY, USA, 2010. Association for Computing Machinery.

[17] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning

through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1009–1024, New York, NY, USA, 2017. Association for Computing Machinery.

[18] Z. Xu, Y. Tu, and X. Wang. Exploring power-performance tradeoffs in database systems. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 485–496, 2010.

[19] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, M. Ran, and Z. Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 415–432, New York, NY, USA, 2019. Association for Computing Machinery.

[20] X. Zhang, T. Lindberg, N. Xiong, V. Vyatkin, and A. Mousavi. Cooling energy consumption investigation of data center it room with vertical placed server. *Energy procedia*, 105:2047–2052, 2017.

Appendix A Nautilus

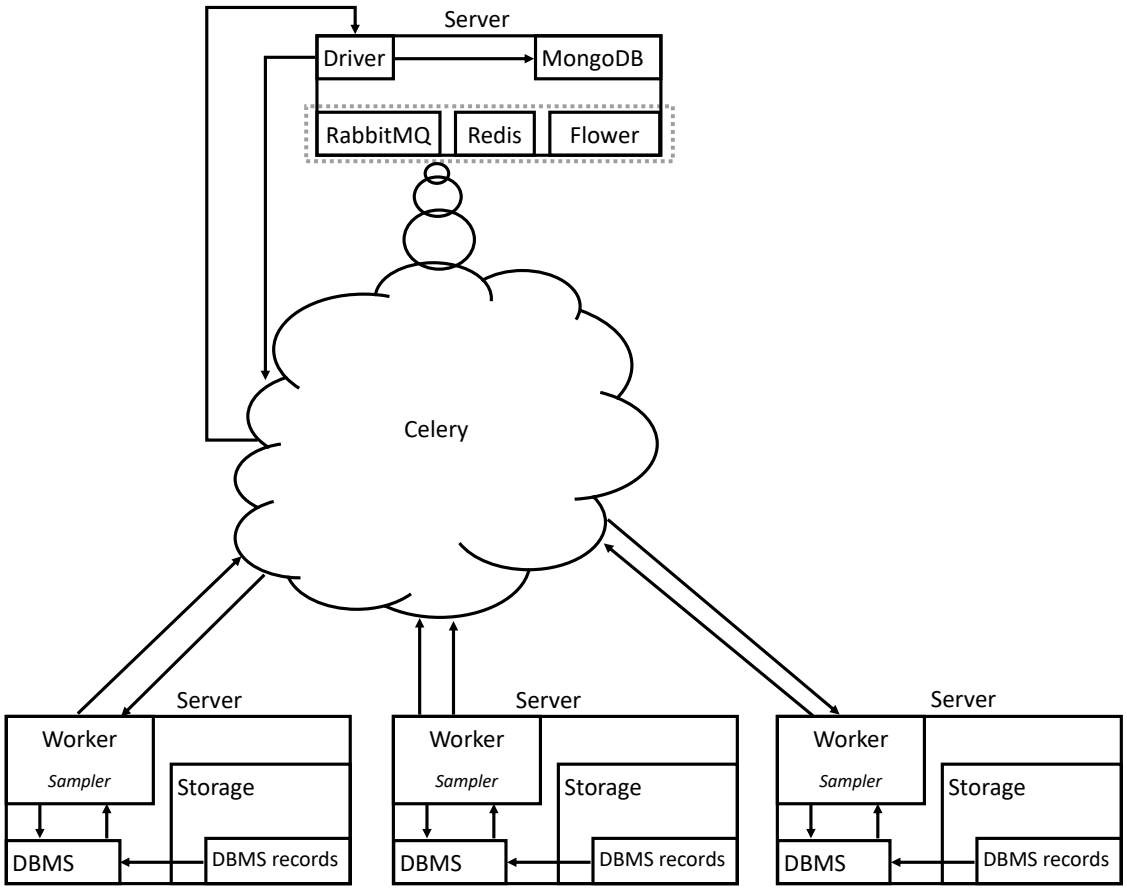

Figure 9: Nautilus Containerized Architecture. Each box inside of a server is a container. The driver communicates with Celery to run tasks on Workers. Workers control the DBMS by initializing it from cached workload records and run the workload in the DBMS. While the DBMS runs, the Worker runs samplers to collect information. When the workload is complete, the worker returns the results to Celery. Celery gives the results to the Driver which stores them in MongoDB. Jump to Nautilus section 3.2

Appendix B Graphs

B.1 Workload and Database

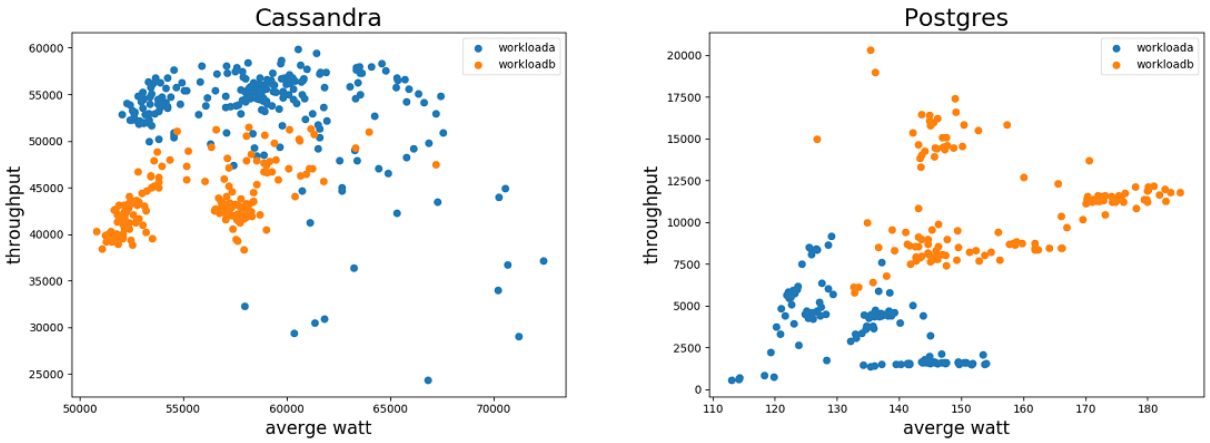

Figure 10: Distribution of Energy Consumption and Performance. Y-axis is Throughput (operations per second), X-axis is Watts. Jump to inline Figure 2

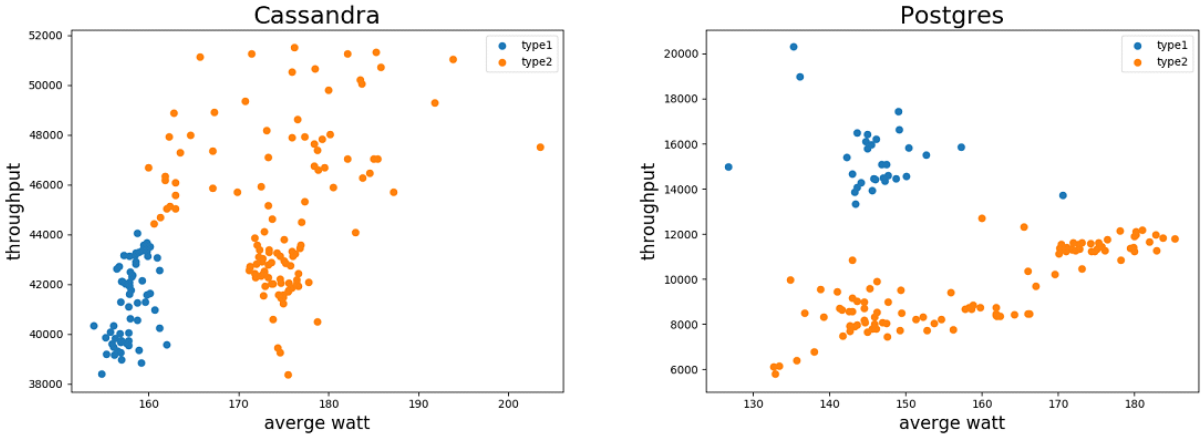

Figure 11: Clusters of Read Heavy Workloads. Y-axis is Throughput (operations per second), X-axis is Watts. Jump to inline Figure 3

B.2 Key Knobs

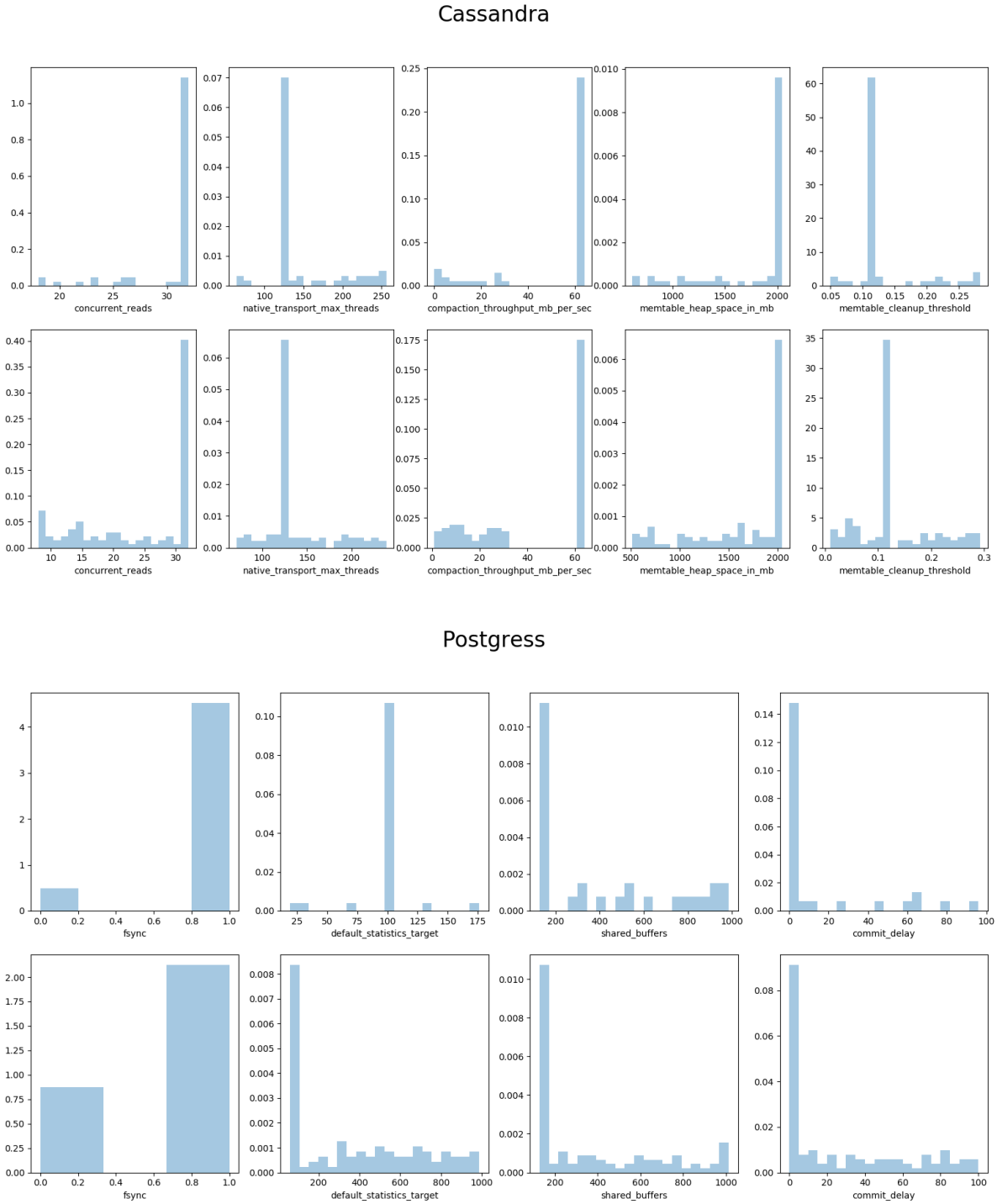

Figure 12: Distribution of the Key Knobs of Read Heavy Workload (Upper subplots in each database are type 1, and the lower ones are type 2). X-axis is the value of knob. Y-axis is the density of each knob value derived from Kernel Density Estimation, Jump to inline Figure 4

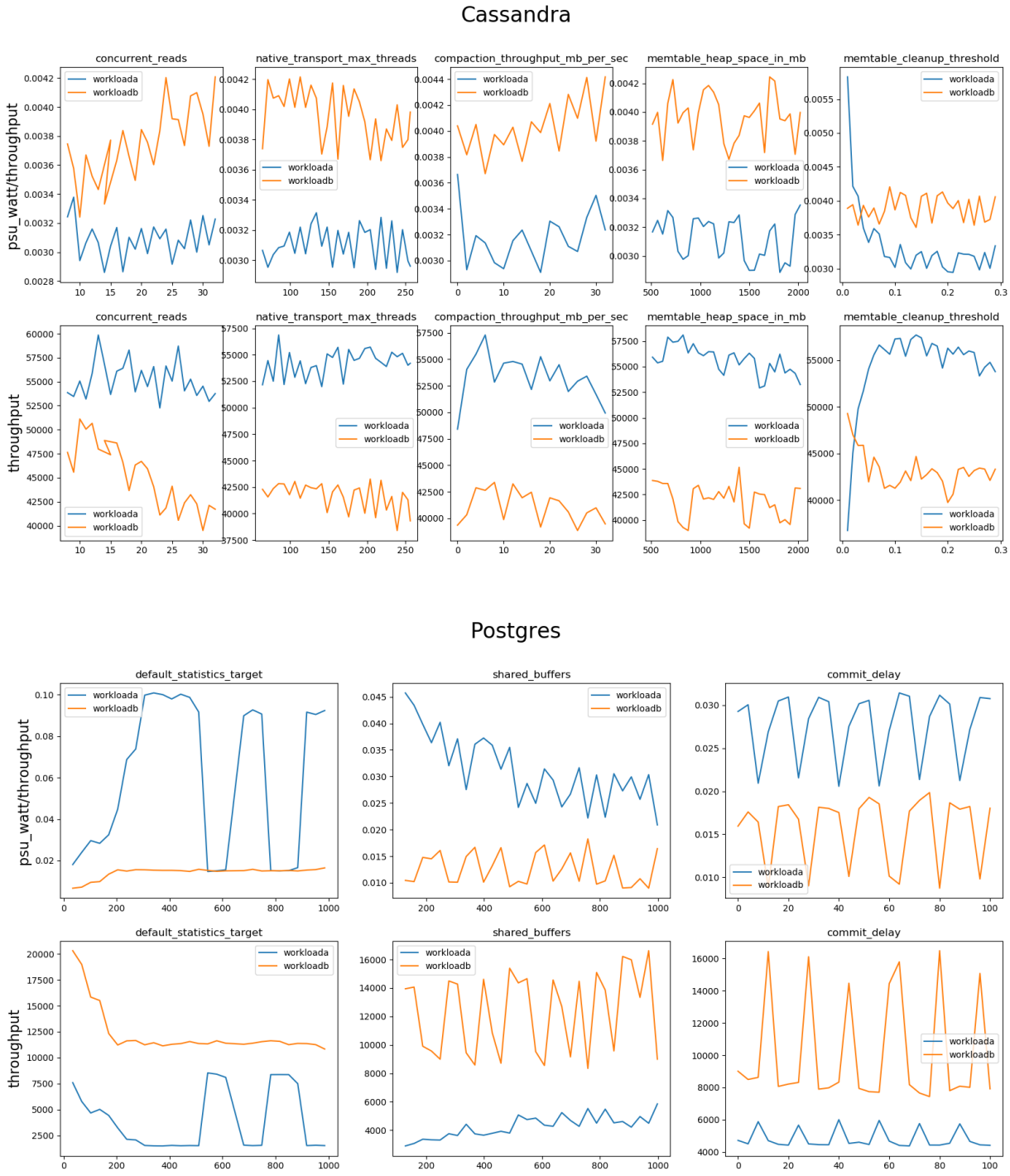

Figure 13: Throughput and Energy Efficiency with Different Knob Values. Y-axis Throughput (operations per second). X-axis knob values. Jump to inline Figure 6

B.3 Variation

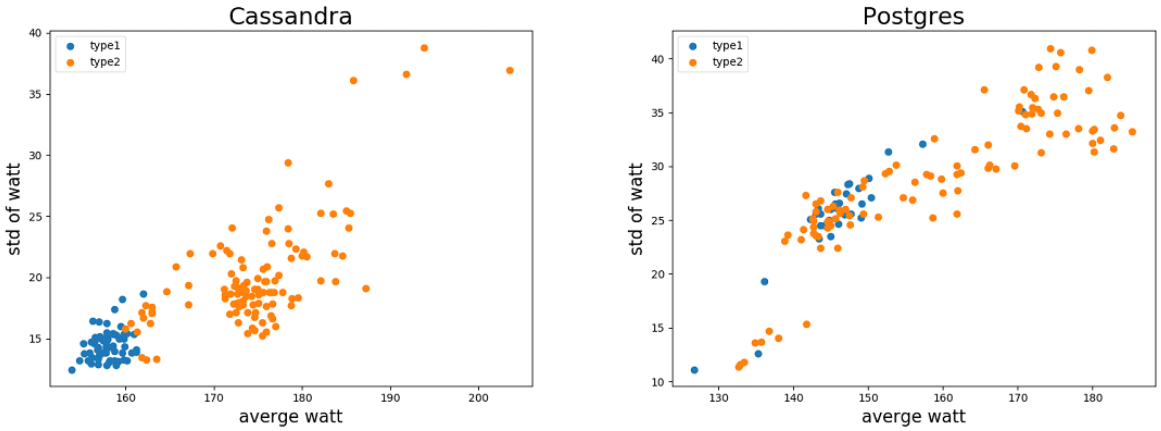

Figure 14: Clusters of Read Heavy Workloads-Average and Standard Deviation of Energy Consumption. Y-axis is Watts and X-axis is Watts. Jump to inline Figure 5

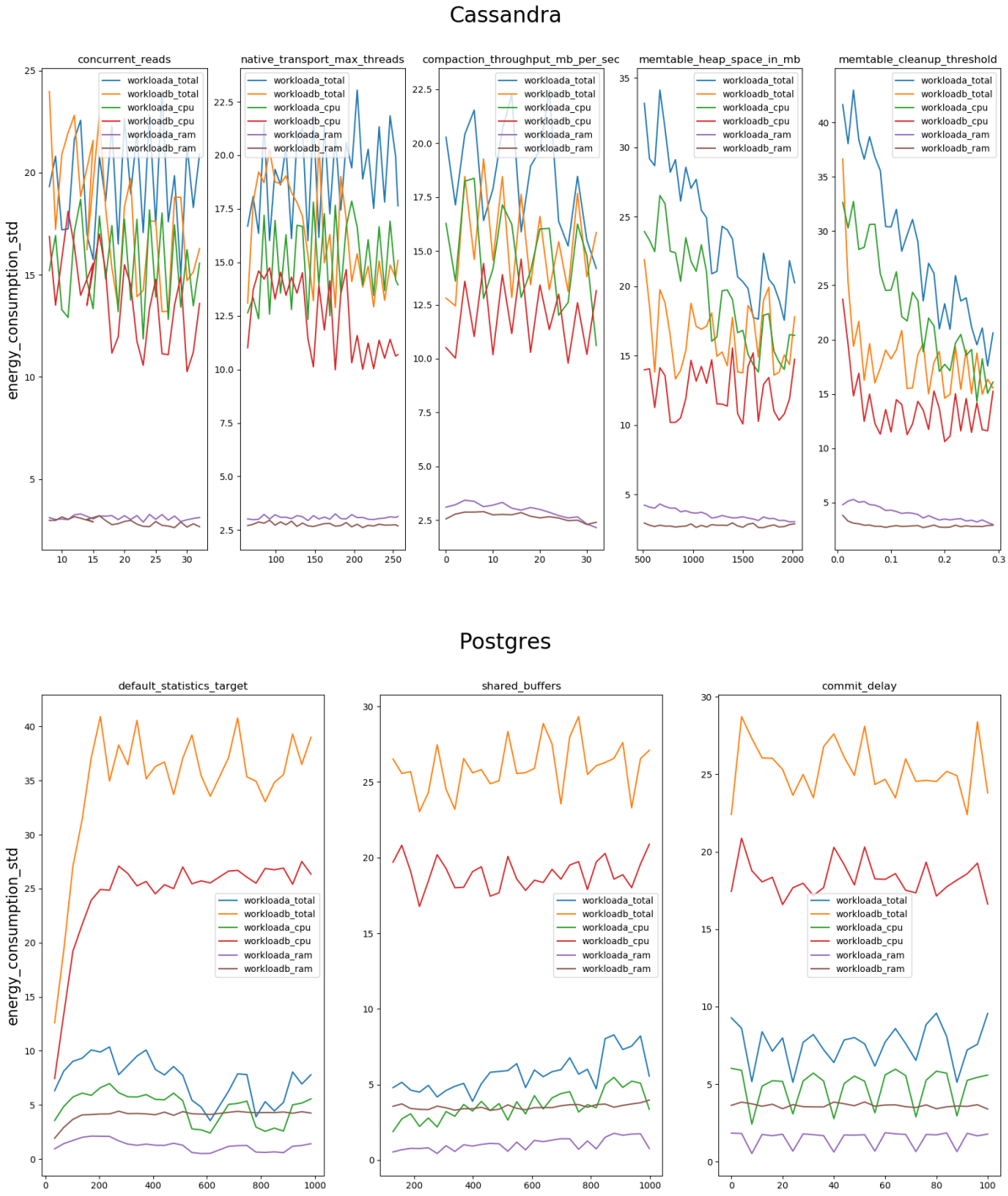

Figure 15: Standard Deviation of Energy Consumption with Different Knob Values. Y-axis is Watts. X-axis knob is values. Jump to inline Figure 7

B.4 Composition

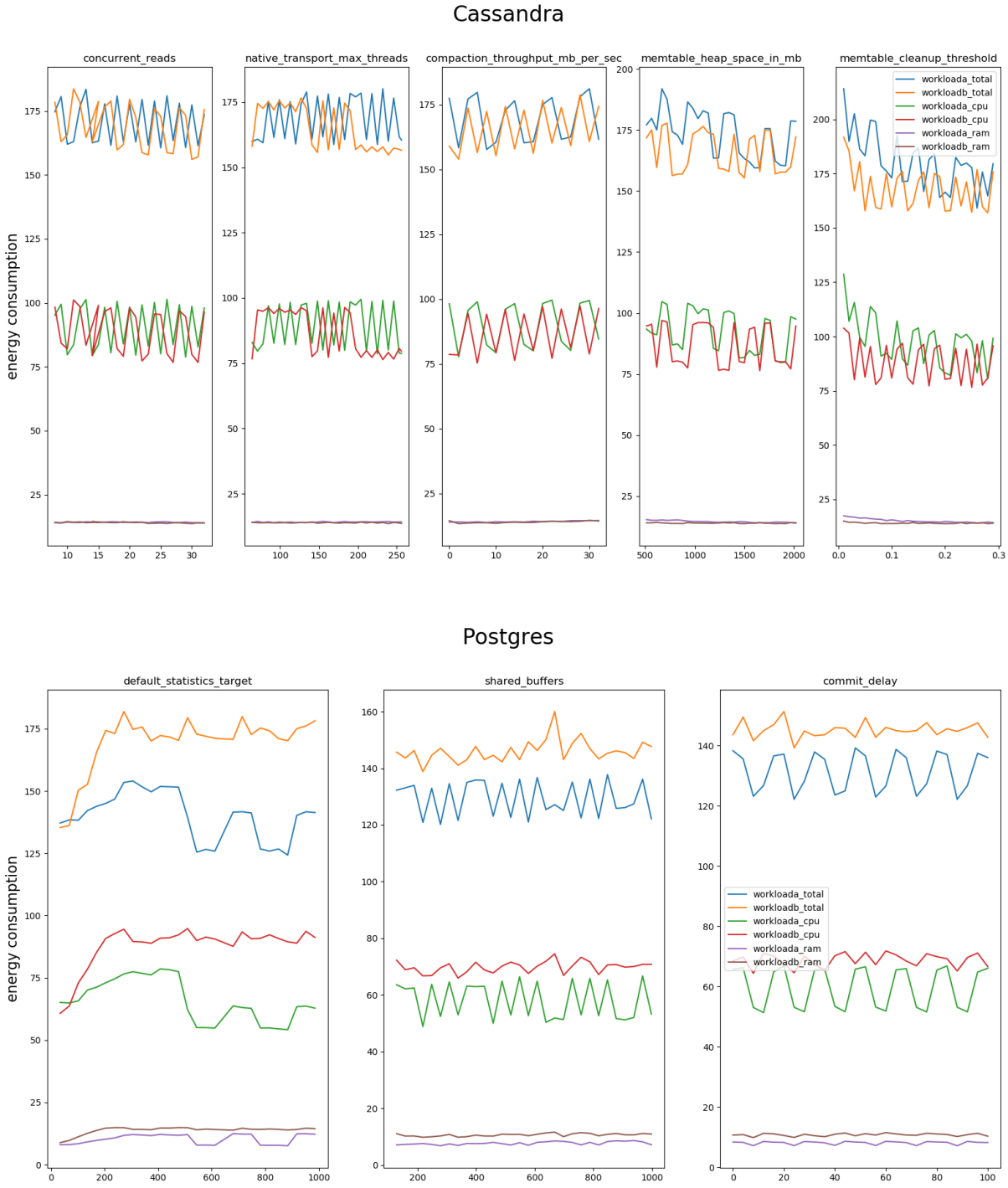

Figure 16: Composition of Energy Consumption with Different Settings. Y-axis is Watts. X-axis is knob values. Jump to inline Figure 8