Automatic Database Management System Tuning Through Large-scale Machine Learning (May 2017)

Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, Bohan Zhang

Presented By

Garrett Manaster, Dhruval Trivedi, Yushi She, Richard So, Chengqi Luo

Introduction to DBMS Tuning

- DBMS performance depends significantly on optimal configuration
  - Impacts throughput and latency
  - Misconfiguration may lead to data loss
- Configuration consists of hundreds of “knobs”
  - Buffer Pool Size, Checkpoint Frequency, Thread Pool Size, Replication Delay, … etc.
- Configuration factors are complex
  - Complexity of interaction is “beyond what humans can reason”
- Yet, DBMSs traditionally are manually tuned

DBMS Manual Tuning

- Manually tuning is performed by Database Admin (DBA)
- Common for DBA to use “*trial and error*” approach
  - Measure performance of a sample workload
  - Update knob configurations (usually 1 knob at a time)
  - Repeat until satisfactory performance
- **Expensive**
  - Nearly 50% of DBMS ownership cost comes from personnel (salary, training, etc)
  - DBAs spend ~25% of time on database tuning
- **Challenging**
  - Many knobs are not independent
  - Value for some knobs are continuous
  - Successful configurations are application dependant
  - New knobs are introduced with DBMS updates

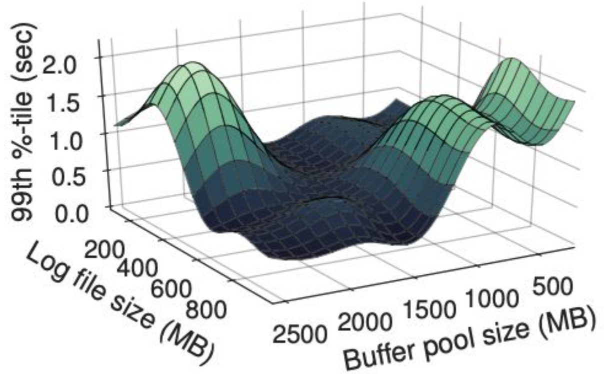
DBMS Tuning Challenges

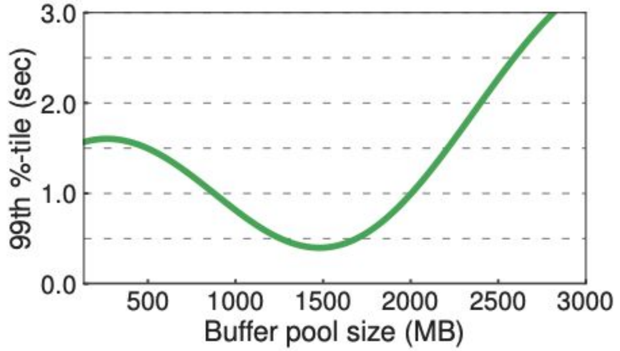
Knob Dependencies

- Changing one knob may affect the benefits of another
- Finding the optimal configuration is NP-hard

Continuous Knob Settings

- Many possible settings for a knob
- Irregular performance impact





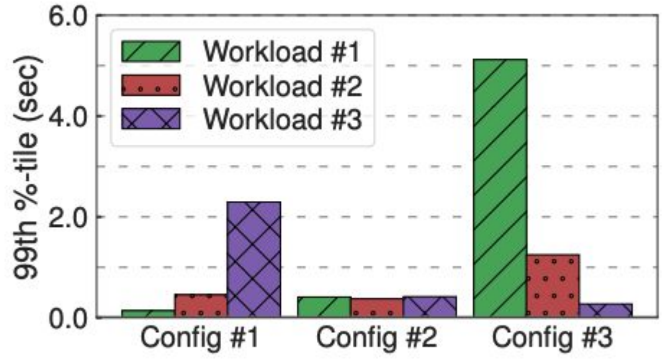
DBMS Tuning Challenges

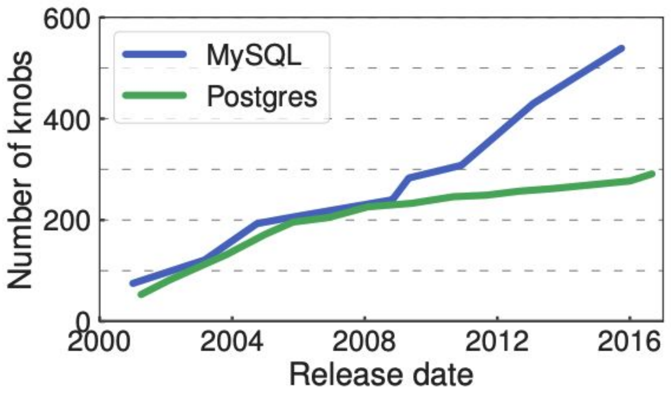
Workload dependant configuration success

- Best configuration for workload can be the worst for another
- Learnings from tuning one DBMS may not apply to others

New knobs introduced with DBMS updates

- DBAs must stay up to date with new releases
- 3x Postgres and 6x MySQL knobs since original release





Automatic DBMS Configuration Tools

- Automatic DBMS tuning tools have deficiencies
  - Many limited to specific DBMS systems
  - Require manual setups from DBA
    - Database Copying
    - Knob Mapping
    - Training Process Guidance
- OtterTune resolves these limitations
  - DBMS agnostic
  - Fully automated: No required DBA interaction during tuning

OtterTune: Applying Machine Learning to DBMS Tuning

- OtterTune
  - Selects the **most important knobs**
  - Maps new workloads to known workloads, allowing **experience transfer**
  - **Recommends knob settings** to improve a target objective
- Achieves *58–94%* lower latency compared to default settings or configurations generated by other tuning advisors
- Generates configurations within *94%* of expert DBAs in under 60 minutes

System Overview

- OtterTune has two main parts: A controller (client-side) and a tuning manager.
    a. The tuning manager stores data in OtterTune’s repository.
- OtterTune begins its tuning process by observing the current DBMS configuration in time blocks called “Observation Periods”. During such periods, the efficiency of the DBMS is recorded.
    a. Observation periods can be fixed length (better for OLTP queries) or variable length (better for OLAP).
- After an observation period ends, OtterTune’s controller stores the DMBS metrics. Unlike other systems, this process doesn’t need the DBA’s input.
- OtterTune then attempts to understand the significance of the metrics and begins step 2: recommending ideal knob configurations.
- OtterTune has another trick up its sleeve: the controller gives the DBA an estimate of how much better the new configuration is compared to the current one.

Assumptions and Limitations

- We are assuming that OtterTune has administrative permission to shut down or modify the DBMS.
    a. Restarting the DBMS may be necessary to employ new knob configurations.
- There is a workaround. Many DBMS have dynamic knob changing features that allows knobs to be changed without restarting the system.
- If certain knobs are off-limits, the DBA can manually choose which knobs can be modified with a blacklist. OtterTune will keep its hands away from these blacklisted knobs.

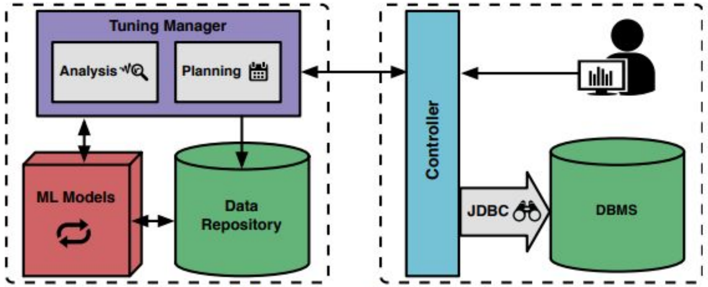
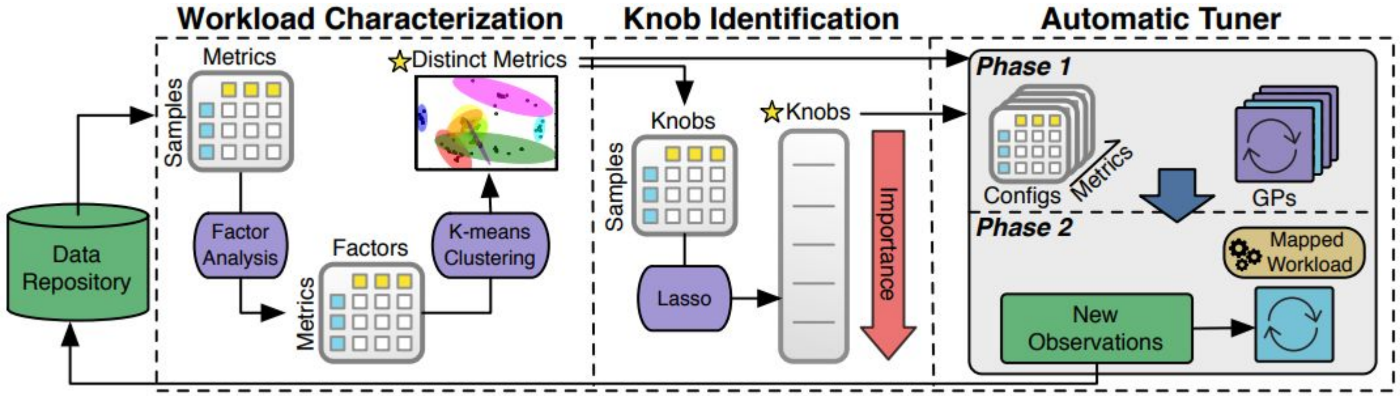


Figure 2: OtterTune Architecture – An overview of the components in the OtterTune system. The controller connects to the DBMS and collects information about the performance of the system. This information is then sent to the tuning manager where it is stored in its repository. It then builds models that are used to select an optimal configuration for the DBMS.

Workload Characterization

- Does a configuration perform well? Internal runtime metrics DBMS might help.
- Problem: OtterTune collects all metrics and stores them in key/value pairs because it doesn't know which are useful.
  a. A good naming scheming can help alleviate this problem, but distinguishing metric types (e.g. Postgres) is hard. A potential solution is to only use sum-scalar values to make OtterTune DBMS-agnostic.
- To get to the important details, OtterTune needs to prune the metric data to reduce the amount of processing needed for knob optimization.



How do we Prune?

- There are too many metrics. Pruning is needed. Use factor analysis (FA) and k-means clustering.
- FA reduces the number of necessary metric/knob combinations.
- In K-means, each cluster represents on metric. Similar metrics are clustered.
- With FA and k-means, the MySQL and Postgres metrics are reduced by 93% and 82% respectively.

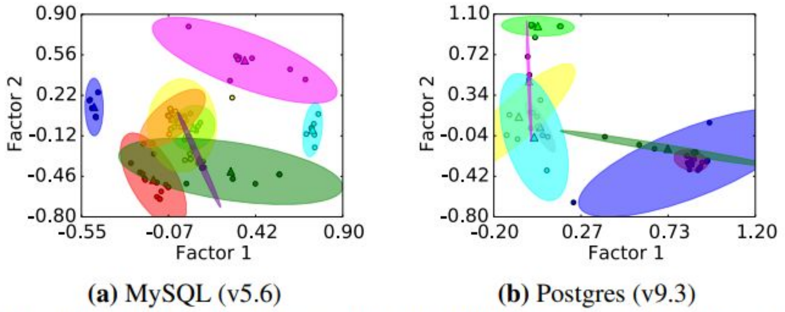


Figure 4: Metric Clustering – Grouping DBMS metrics using k-means based on how similar they are to each other as identified by Factor Analysis and plotted by their (`f1`, `f2`) coordinates. The color of each metric shows its cluster membership. The triangles represent the cluster centers.

Importance of Identifying Key Configuration Knobs

Purpose of Knob Selection:

- Enhance DBA's target objective function
- Reduce the configuration space by eliminating non-impactful knobs

Impact of Knobs on Performance:

- Discover both positive and negative correlations
- Example: Memory allocation for buffer pool affects latency

Benefits of Pruning Knobs:

- Limits the number of DBMS configurations to consider
- Improves tuning efficiency and effectiveness

Feature Selection with Lasso Regression

Why Lasso Over OLS:

- Addresses high variance and overfitting in high-dimensional settings
- Performs feature selection by shrinking irrelevant coefficients to zero

How Lasso Works:

- Applies L1 penalty to regression coefficients
- Encourages sparsity, selecting only the most relevant knobs

Implementation in OtterTune:

- Incorporates polynomial features to detect nonlinear dependencies
- Continuously runs in the background, handling large datasets efficiently

Handling Dependencies and Incremental Knob Selection

Detecting Knob Dependencies:

- Use polynomial features to identify interactions between knobs
- Example: Buffer pool memory allocation interacts with log buffer size

Incremental Knob Selection:

- Dynamically increase the number of knobs used over time
- Balances optimization time with configuration space complexity

Benefits of Incremental Approach:

- Ensures comprehensive search without excessive computational overhead
- Proven to yield better configurations compared to static knob counts

Overview of Automated Tuning Process

Core Components of OtterTune:

- Non-redundant metrics set
- Most impactful configuration knobs
- Repository of historical tuning data

Iterative Tuning Process:

- Continuous data analysis
- Configuration recommendation loop

Two-Step Analysis Framework:

- Step 1: Workload Mapping
- Step 2: Configuration Recommendation

Step 1 – Workload Mapping

Objective:

- Match target workload with similar repository workload

Process:

- Construct set S of N matrices for non-redundant metrics
- Compute Euclidean distances across metrics
- Calculate average distance scores to identify best match

Data Normalization:

- Apply decile binning to ensure metrics are comparable

Step 2 – Configuration Recommendation

Gaussian Process (GP) Regression:

- Models configuration space with confidence intervals
- Balances exploration and exploitation

Handling Uncertainty:

- Increase variance for untested configurations
- Add ridge terms to manage noise

Optimization Strategy:

- Use gradient descent to find local optima
- Initialization set: top-performing and random configurations
- Quick convergence (10–20 seconds per period)

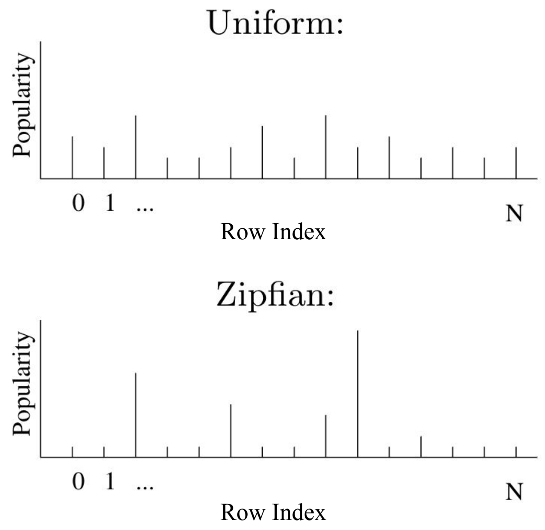
Evaluating OtterTune – *A Breakdown*

1. Defining **Experimental Workloads**
    - *Diverse, controlled scenarios to test OtterTune's effectiveness*
2. Bootstrapping with **initial Training Data**
    - *Prime Ottertune with a bit of the "knob-tuning"/feature space*
3. **Empirical Performance** of OtterTune
    - *DBMSs for evaluation: MySQL, PostgreSQL, and Actian Vector*
      (OLTP) *(OLAP)*
    - *All algorithms & experiments executed with TensorFlow and scikit-learn*

Evaluating OtterTune – *Experimental Workloads (1)*

- **YCSB: Y**ahoo! **C**loud **S**erving **B**enchmark
  - 6 simple transactions involving random access of tuples from a Zipfian distribution
  - Single table w/ 10 columns & 18 million rows (~18 GB of data)

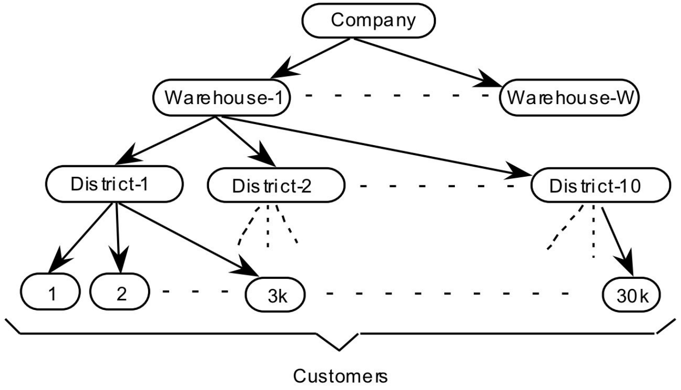
- Primary measurements (OLTP): **Throughput** and **P99** latency over a 5-minute workload runtime



*****Zipfian distribution concentrates randomized access to a handful of tuples/rows*****

Evaluating OtterTune – *Experimental Workloads (1)*

- **TPC-C:** simulation of a wholesale supply chain comprised of warehouses
  - Encapsulates real-world hierarchical and customer-centric data
  - OtterTune sets ***W*** = 200 ; ~18 GB of data ; 5 OLTP Transactions



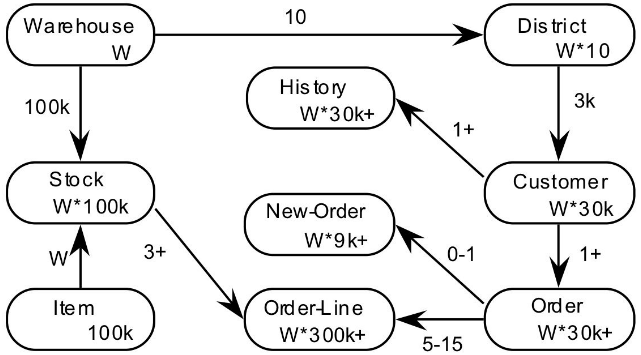
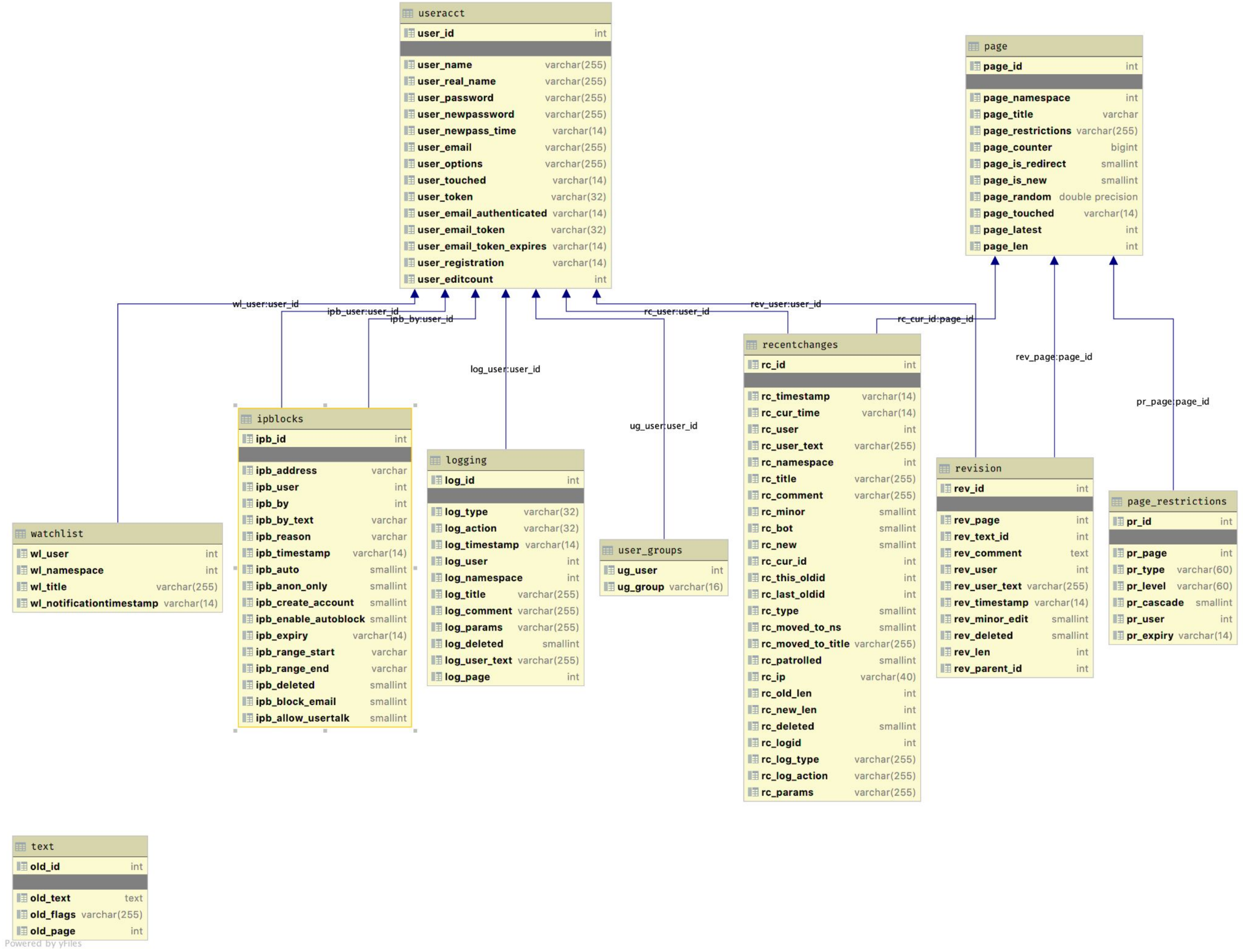


Table Schema (9 tables)

Evaluating OtterTune – *Experimental Workloads (1)*

- **Wikipedia (OLTP Benchmark):** Mocks the website's MediaWiki backend
    - Consists 8 transactions involving articles, revisions, and user watchlists
    - Contains 11 tables w/ large, complex secondary indexes and foreign keys
    - Database of 100k articles; ~ 20 GB in size



useracct	
user_id	int
user_name	varchar(255)
user_real_name	varchar(255)
user_password	varchar(255)
user_newpassword	varchar(255)
user_newpass_time	varchar(14)
user_email	varchar(255)
user_options	varchar(255)
user_touched	varchar(14)
user_token	varchar(32)
user_email_authenticated	varchar(14)
user_email_token	varchar(32)
user_email_token_expires	varchar(14)
user_registration	varchar(14)
user_editcount	int

page	
page_id	int
page_namespace	int
page_title	varchar
page_restrictions	varchar(255)
page_counter	bigint
page_is_redirect	smallint
page_is_new	smallint
page_random	double precision
page_touched	varchar(14)
page_latest	int
page_len	int

watchlist	
wl_user	int
wl_namespace	int
wl_title	varchar(255)
wl_notificationtimestamp	varchar(14)

ipblocks	
ipb_id	int
ipb_address	varchar
ipb_user	int
ipb_by	int
ipb_by_text	varchar
ipb_reason	varchar
ipb_timestamp	varchar(14)
ipb_auto	smallint
ipb_anon_only	smallint
ipb_create_account	smallint
ipb_enable_autoblock	smallint
ipb_expiry	varchar(14)
ipb_range_start	varchar
ipb_range_end	varchar
ipb_deleted	smallint
ipb_block_email	smallint
ipb_allow_usertalk	smallint

logging	
log_id	int
log_type	varchar(32)
log_action	varchar(32)
log_timestamp	varchar(14)
log_user	int
log_namespace	int
log_title	varchar(255)
log_comment	varchar(255)
log_params	varchar(255)
log_deleted	smallint
log_user_text	varchar(255)
log_page	int

user_groups	
ug_user	int
ug_group	varchar(16)

recentchanges	
rc_id	int
rc_timestamp	varchar(14)
rc_cur_time	varchar(14)
rc_user	int
rc_user_text	varchar(255)
rc_namespace	int
rc_title	varchar(255)
rc_comment	varchar(255)
rc_minor	smallint
rc_bot	smallint
rc_new	smallint
rc_cur_id	int
rc_this_oldid	int
rc_last_oldid	int
rc_type	smallint
rc_moved_to_ns	smallint
rc_moved_to_title	varchar(255)
rc_patrolled	smallint
rc_ip	varchar(40)
rc_old_len	int
rc_new_len	int
rc_deleted	smallint
rc_logid	int
rc_log_type	varchar(255)
rc_log_action	varchar(255)
rc_params	varchar(255)

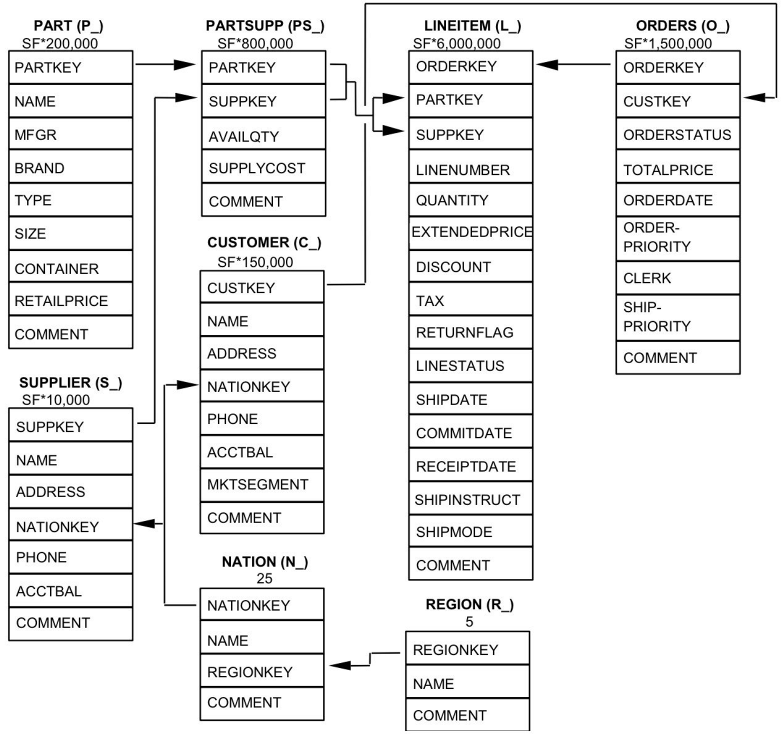
revision	
rev_id	int
rev_page	int
rev_text_id	int
rev_comment	text
rev_user	int
rev_user_text	varchar(255)
rev_timestamp	varchar(14)
rev_minor_edit	smallint
rev_deleted	smallint
rev_len	int
rev_parent_id	int

page_restrictions	
pr_id	int
pr_page	int
pr_type	varchar(60)
pr_level	varchar(60)
pr_cascade	smallint
pr_user	int
pr_expiry	varchar(14)

text	
old_id	int
old_text	text
old_flags	varchar(255)
old_page	int

Evaluating OtterTune – *Experimental Workloads (1)*

- **TPC-H:** Models a decision support system, OLAP-like environment
  - 22 queries of varying complexity
    - *Mocks situation where little is known about the queries*
  - 8 tables in 3NF schema
- Primary measurement (OLAP): Total **workload execution time**

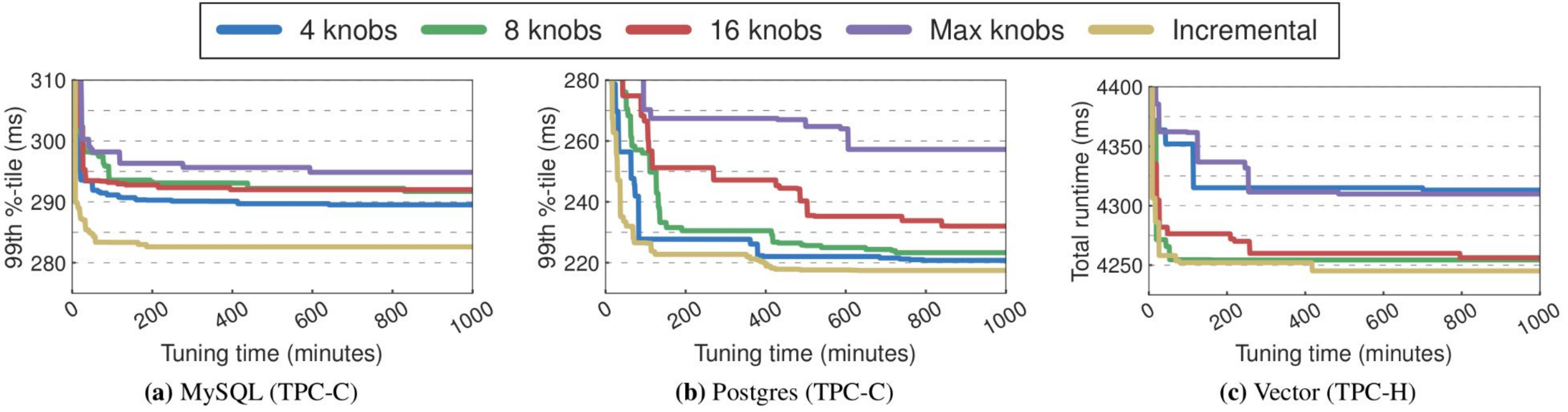


Evaluating OtterTune – *Initial Training Data (2)*

- *Recall: OtterTune requires previous DB tuning data to operate properly*
  - This initial data should not align exactly with the tested workloads
- **Solution:** Curate different permutations of the workloads
  - 15 variants of YCSB w/ different workload mixtures
  - 4 groups of queries from TPC-H which encapsulate the overall workload
- Parameter sweep over all knobs and possible values; ~ 30k trials per DBMS
  - OtterTune tracks performance metrics of the DBMS over each setting
  - *Initial training data is reset after each experiment*

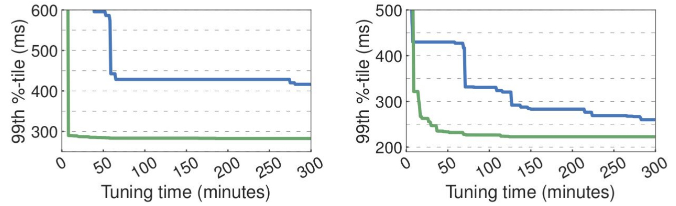
Evaluating OtterTune – *Empirical Performance (3)*

- **Incremental Knob Selection performs best <u>across all DBMS</u>**
    - It is best to begin with a few features to optimize first, then gradually expand the feature space for even more optimization



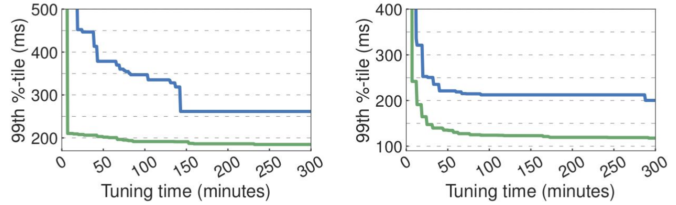
Evaluating OtterTune – *Empirical Performance (3)*

- **OtterTune outperforms existing tuning tool, iTuned**
  - *iTuned samples on the current workload for initialization, rather than requiring previous tuning sessions*



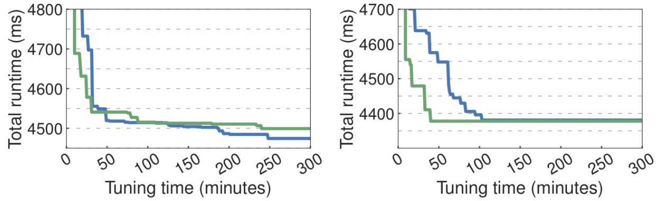
(a) MySQL **(b)** Postgres

Figure 6: Tuning Evaluation (TPC-C) – A comparison of the OLTP DBMSs for the TPC-C workload when using configurations generated by OtterTune and iTuned.



(a) MySQL **(b)** Postgres

Figure 7: Tuning Evaluation (Wikipedia) – A comparison of the OLTP DBMSs for the Wikipedia workload when using configurations generated by OtterTune and iTuned.



(a) Vector (TPC-H #1) **(b)** Vector (TPC-H #2)

Figure 8: Tuning Evaluation (TPC-H) – Performance measurements for Vector running two sub-sets of the TPC-H workload using configurations generated by OtterTune and iTuned.

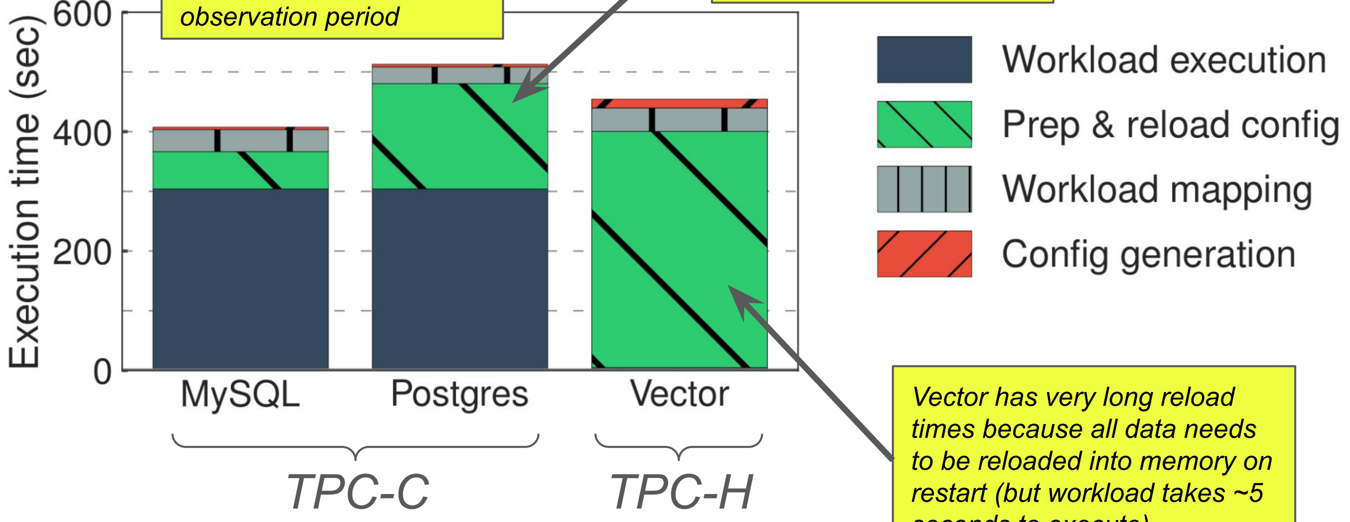
🟦 iTuned 🟩 OtterTune

Evaluating OtterTune – *Empirical Performance (3)*

- **Execution Time Breakdown**

OLTP workloads were expected to be 5 minutes to align with OtterTune's observation period

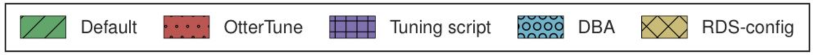
Increased preparation time for Postgres due to vacuum command to reclaim stale storage between runs

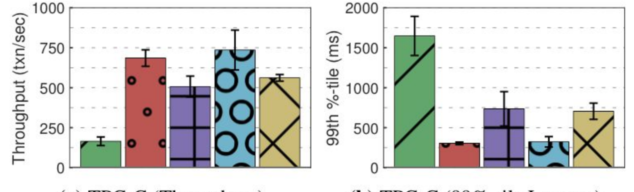


Vector has very long reload times because all data needs to be reloaded into memory on restart (but workload takes ~5 seconds to execute)

Evaluating OtterTune – *Empirical Performance (3)*

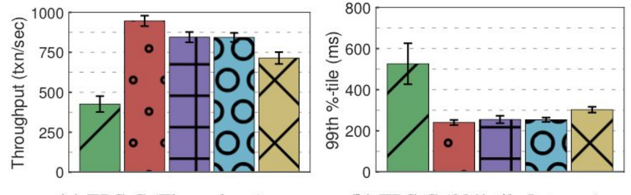
- **OtterTune is on par with–if not, better than–Human DBAs**
    - Effectiveness seen on both MySQL and PostgreSQL
    - However, OtterTune optimizes purely for latency reduction and throughput
        - *e.g. DBA sets the max # of Postgres log files to be 16-64, but OtterTune sets this to 540, fully neglecting the cost of recovery time in blind favor of I/O performance*





(a) TPC-C (Throughput) **(b)** TPC-C (99%-tile Latency)

Figure 10: Efficacy Comparison (MySQL) – Throughput and latency measurements for the TPC-C benchmark using the (1) default configuration, (2) OtterTune configuration, (3) tuning script configuration, (4) Lithuanian DBA configuration, and (5) Amazon RDS configuration.



(a) TPC-C (Throughput) **(b)** TPC-C (99%-tile Latency)

Figure 11: Efficacy Comparison (Postgres) – Throughput and latency measurements for the TPC-C benchmark using the (1) default configuration, (2) OtterTune configuration, (3) tuning script configuration, (4) expert DBA configuration, and (5) Amazon RDS configuration.

Related Work: Existing Solutions

Physical design: indexing, partitioning schemas, views, etc.

Auto tune DBMSs’ config knobs

1) Rule-based approaches
2) ML-based approaches

Related Work: Rule-based Approaches

Predefined set of rules/ heuristics

- System specific
- Limited scope (only target a specific group of knobs)
- Requires manual intervention

IBM DB2, Oracle, BestConfig, etc.

Related work: ML-based approaches

iTuned

- **Pro**: improving both latency or throughput
- **Con**: cannot transfer learned insights from previous tuning sessions
- Requires significantly more time for training

Uses Gaussian Process (GP) models to optimize configurations but starts each tuning session with stochastic sampling (e.g., Latin Hypercube Sampling), requiring substantial time to populate its models.

-> non-reusable config -> no transfer learning -> inefficient when applied to new workloads

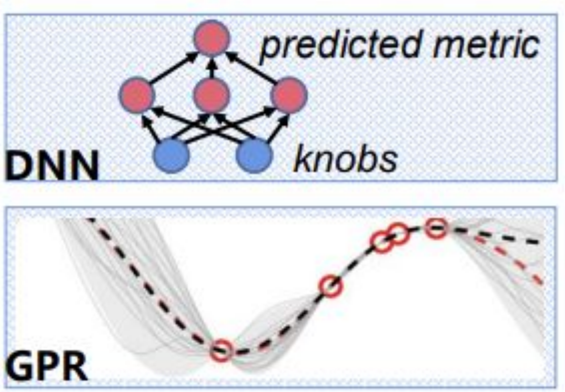
Follow-up Work: ML-based approaches

DNN - Ottertune (2019)

Gaussian Process models do not perform well on **larger data sets** and **high-dimensional** feature vectors.

-> use deep neural network (DNN)

instead of the Gaussian models



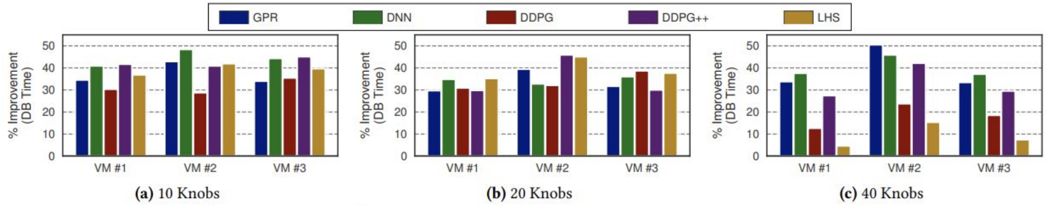
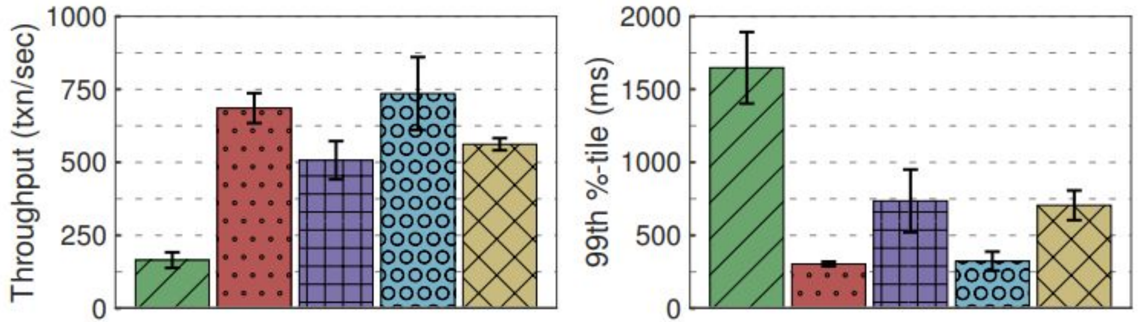


Figure 10: Tuning Knobs Selected by DBA (Per VM) – The performance improvement of the best configuration per algorithm running on separate VMs relative to the performance of the SG default configuration measured at the beginning of the tuning session.

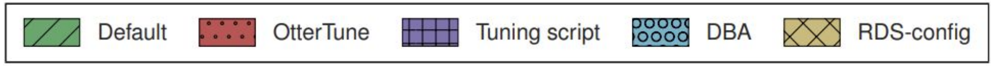
Conclusion

improves both latency and throughput without comprising either.



(a) TPC-C (Throughput) **(b)** TPC-C (99%-tile Latency)

Figure 10: Efficacy Comparison (MySQL) – Throughput and latency measurements for the TPC-C benchmark using the (1) default configuration, (2) OtterTune configuration, (3) tuning script configuration, (4) Lithuanian DBA configuration, and (5) Amazon RDS configuration.



Conclusion

Unlike previous tuning tools, OtterTune

- reuses training data gathered from previous tuning sessions.
- uses a combination of supervised and unsupervised machine learning methods to

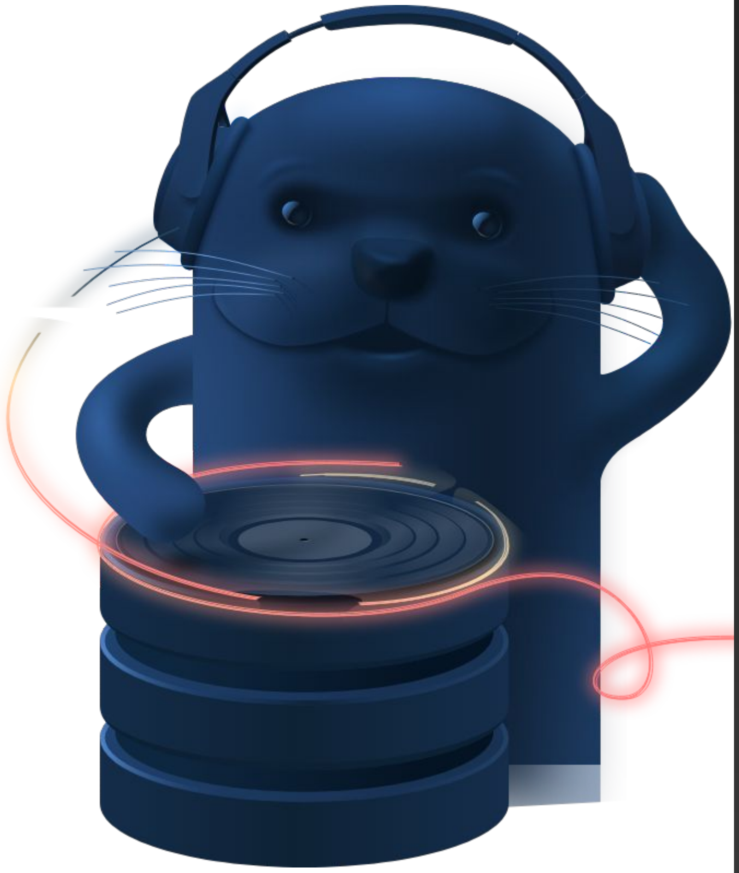
    (1) select the most impactful knobs,

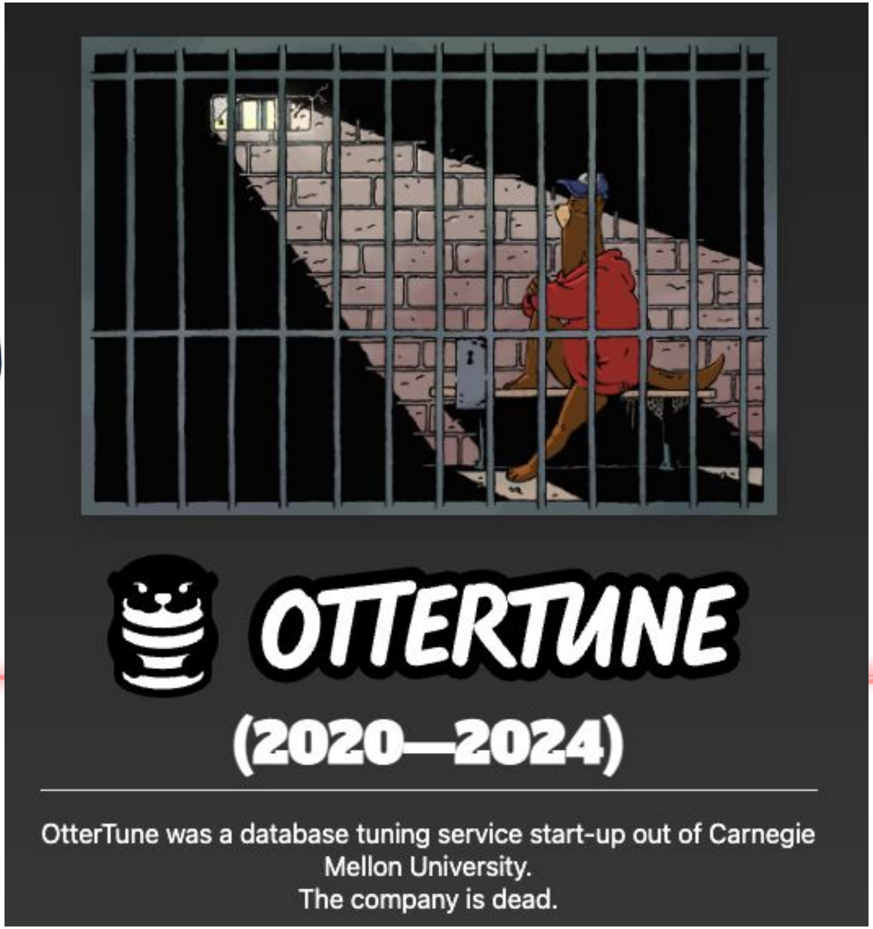
    (2) map previously unseen database workloads to known workloads, and

    (3) recommend knob settings.

Questions

- **Stopping/starting** criteria?
- As workload shifts over time, how to detect when it has changed to the point where tuning is needed?
- What about **multi-objective optimization**? (for example, tradeoff between latency and recovery)
- How does OtterTune's performance compare to **DB-specific tuning tools** such as PgTune for PostgreSQL or MyTune for MySQL? Are there any benchmarks or evaluations available that highlight OtterTune's advantages or limitations in comparison to these tools?





(2020—2024)

OtterTune was a database tuning service start-up out of Carnegie Mellon University.
The company is dead.

https://ottertune.com/