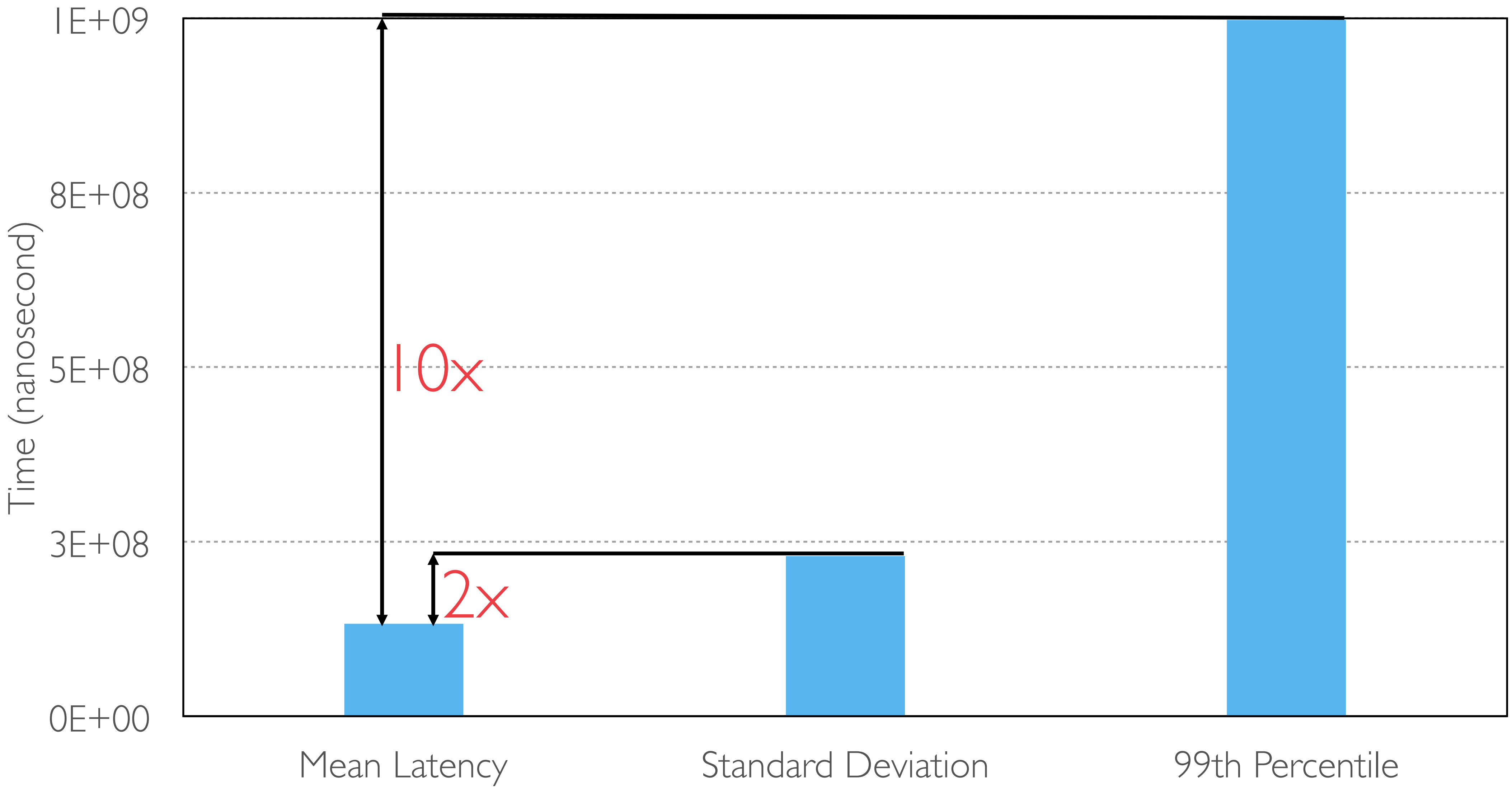
A Top-Down Approach to Achieving Performance Predictability in Database Systems

Jiamin Huang, Barzan Mozafari, Grant Schoenebeck and Thomas F. Wenisch
University of Michigan

Performance Predictability in Today's DBMS

By focusing too much on raw performance
we have neglected predictability

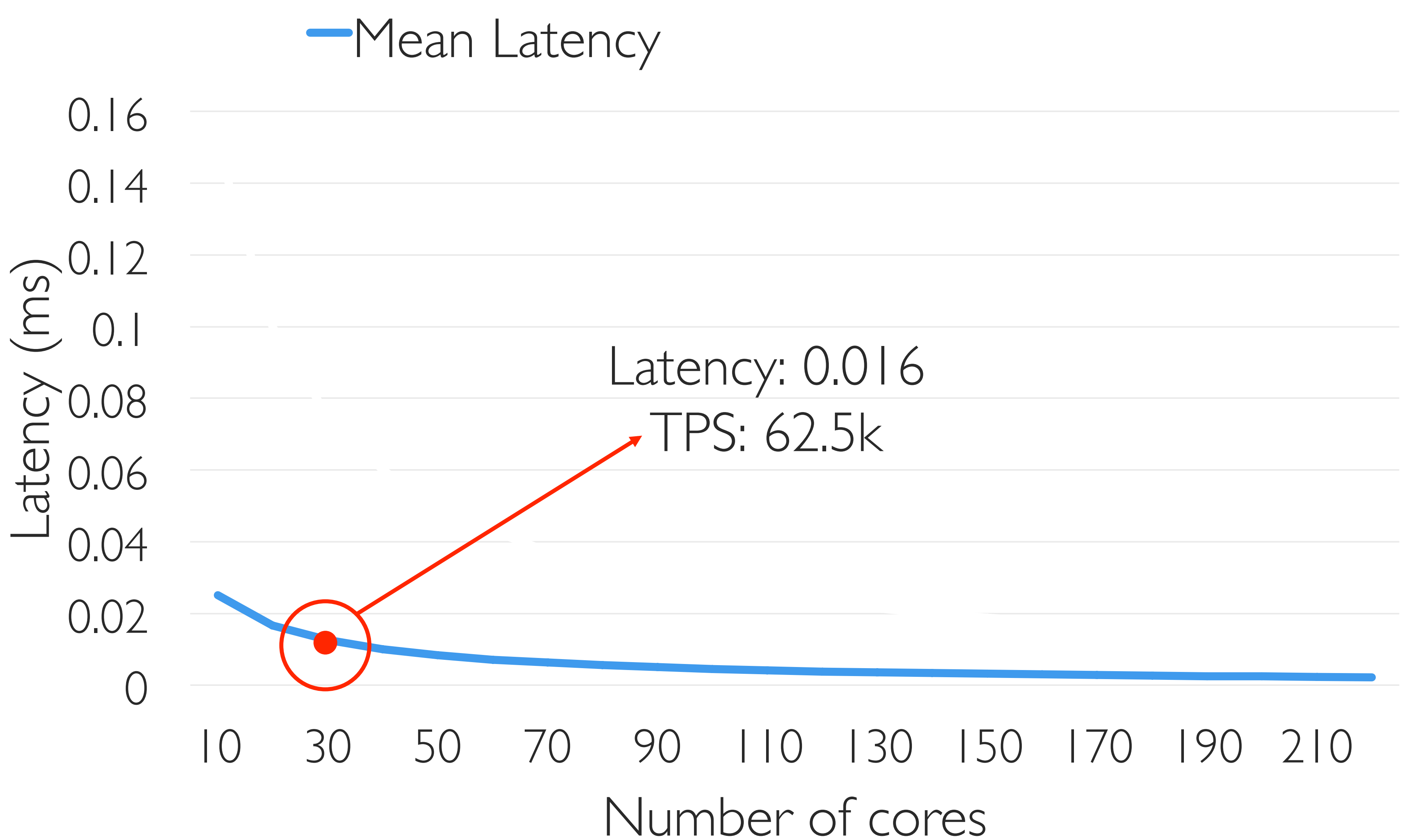


MySQL running TPC-C benchmark at a fixed rate

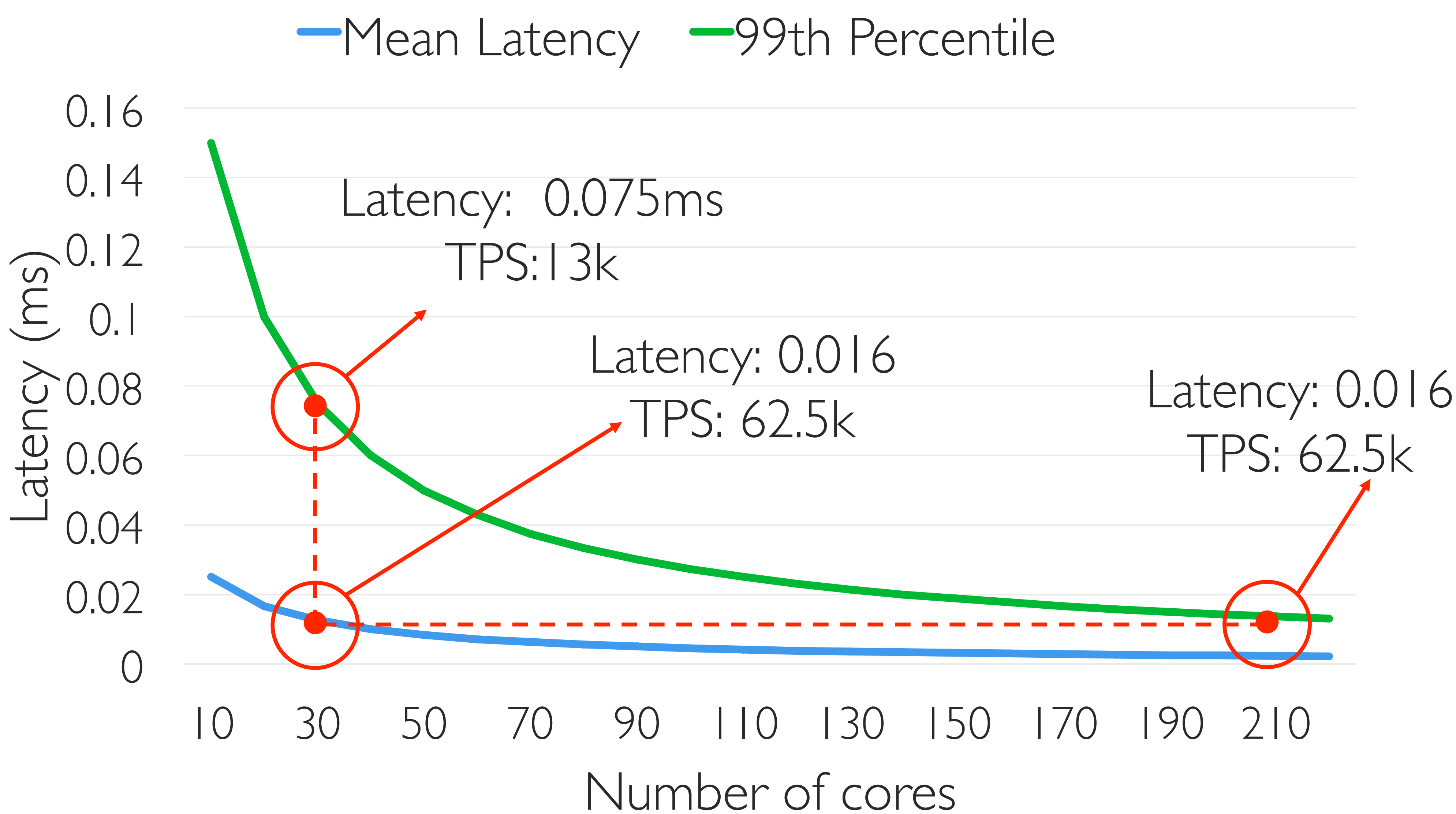
Why Does Predictability Matter?

- Latency-sensitive applications
  - Provisioning
  - SLA guarantees
  - Tuning
- Interactive applications
  - User-experience

Example: Provisioning & SLAs



Example: Provisioning & SLAs



What is Performance Predictability?

- Performance Variance:

  1. Inherent (External): varying amounts of work, network problems, …

  2. Avoidable (Internal): due to internal artifacts of the DBMS (algorithms, data structures, …)

Two Approaches to Achieve Predictability

- Bottom-up: build a new DBMS from scratch
  - Once an academic prototype, always an academic prototype
  - Sacrifice performance for predictability
- Top-down: identify root causes of unpredictability and mitigate them
  - **Goal**: *do not* compromise performance
  - **Benefit**: adoption is "no-brainer"
  - **Challenge**: today's DBMSs are extremely complex

Key Questions

1. How to identify sources of variance?
2. What makes today's DBMSs unpredictable?
3. How to achieve perf. predictability?
4. How effective are our techniques?

Key Questions

1. How to identify sources of variance?
2. What makes today's DBMSs unpredictable?
3. How to achieve perf. predictability?
4. How effective are our techniques?

Identifying Root Causes of Performance Variance

- Profiling tools: critical for diagnosing perf. problems in modern software
- Existing profilers focus on average performance
  - DTrace, gprof, perf, etc.
- Breakdown of avg. performance of DBs done before
  - **"OLTP through the looking glass, and what we found there" [SIGMOD'08]**
- Need a new profiler capable of breaking down perf. variance → TProfiler

TProfiler

- **Goal:** Pinpoint root causes of performance variance in large and complex codebases of today's DBMS

PostgreSQL

770K lines of code

MySQL®

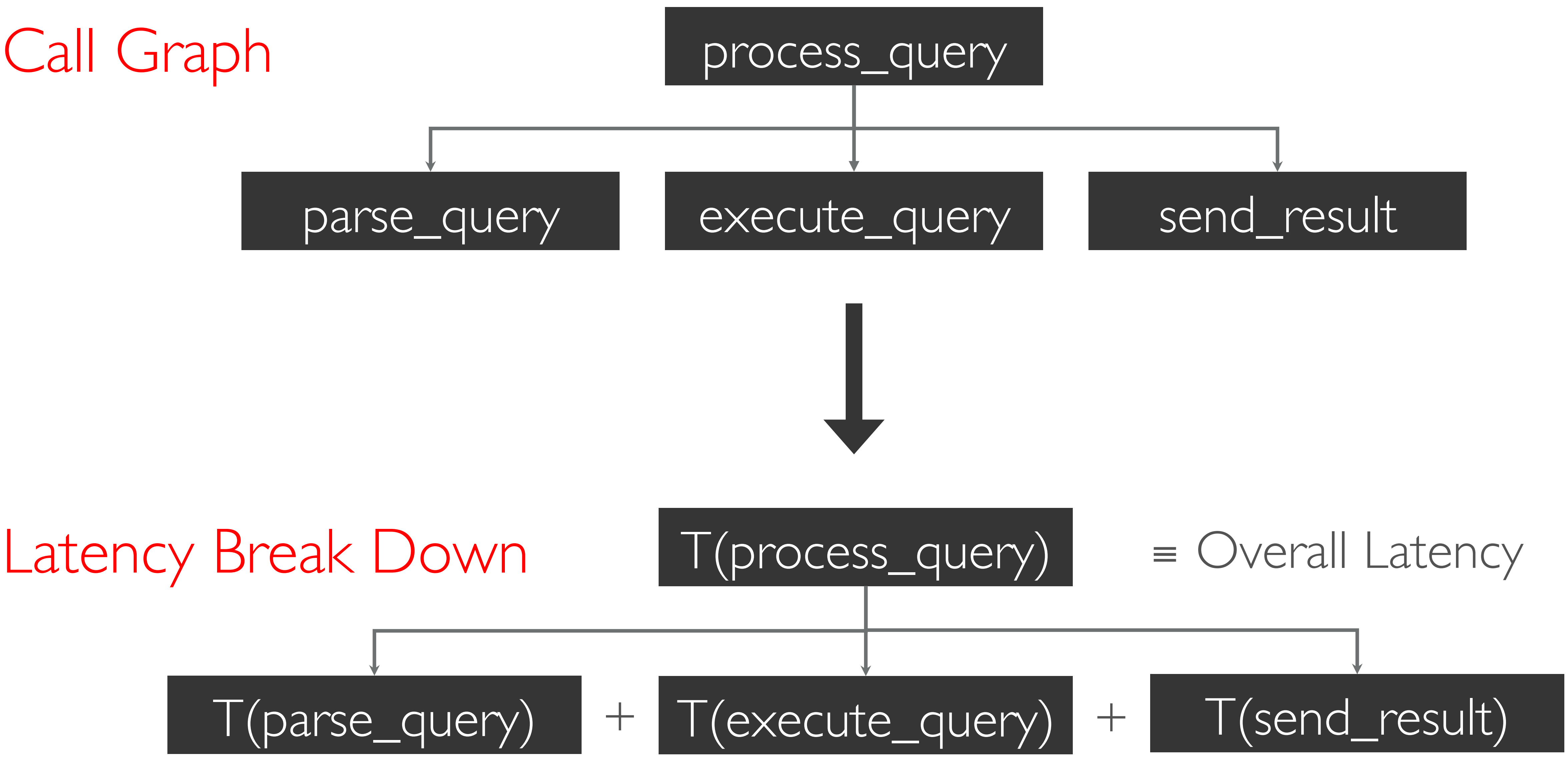
1.5M lines of code

VOLTDB

1.9M lines of code

Q: How to find the root causes of performance variance efficiently and accurately?

Our Solution: Variance Trees



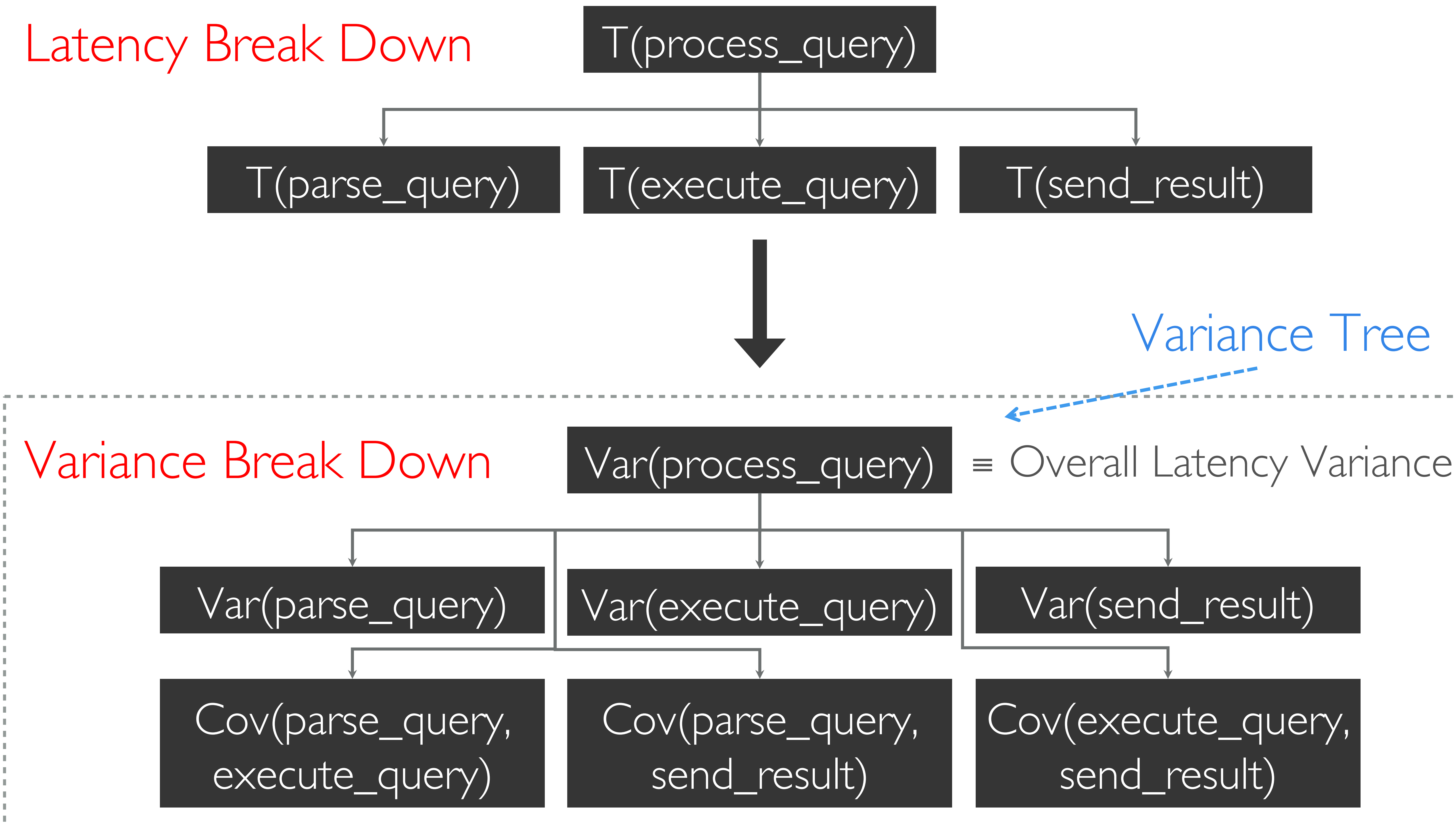
T(*f*): Execution time of function *f*

Our Solution: Variance Trees

- *If* $T = \sum_i T_i$ *, then:*

$$Var(T) = \sum_i Var(T_i) + \sum_{i \neq j} Cov(T_i, T_j)$$

Our Solution: Variance Trees



Efficiency

- Observation: most nodes are actually insignificant
  - Do **not** build a complete variance tree!
- Build variance tree iteratively and selectively

1. **Tree expansion**: break down variance of selected functions (process_query at the beginning)
2. **Node selection**: select significant* nodes from the tree
3. **User inspection**: users inspect selected functions, and decide whether to further investigate

* See paper for details

Key Questions

1. How to identify sources of variance?
2. What makes today's DBMSs unpredictable?
3. How to achieve perf. predictability?
4. How effective are our techniques?

Case Studies

- Used TProfiler to analyze 3 popular (both traditional and modern) DBMSs

MySQL

PostgreSQL

VOLTDB

Setup

- Application: MySQL 5.6.23
- Hardware: Intel Xeon E5 2.1GHz
- Workload: TPC-C
- 128 Warehouses, 30GB Buffer Pool

Root Causes of Performance Unpredictability in MySQL

- With 37 iterations, 6 mins manual inspection time each, out of 30K functions

Function Name	Contribution to Overall Latency Variance
os_event_wait[A]	37.5%
os_event_wait[B]	21.7%
buf_pool_mutex_enter	32.92%

os_event_wait[A] and os_event_wait[B]: Transactions waiting for locks on data objects. Same function, different call sites

buf_pool_mutex_enter → Waiting for lock on the buffer pool before updating the list of buffer pages

Key Questions

1. How to identify sources of variance?
2. What makes today's DBMSs unpredictable?
3. How to achieve perf. predictability?
4. How effective are our techniques?

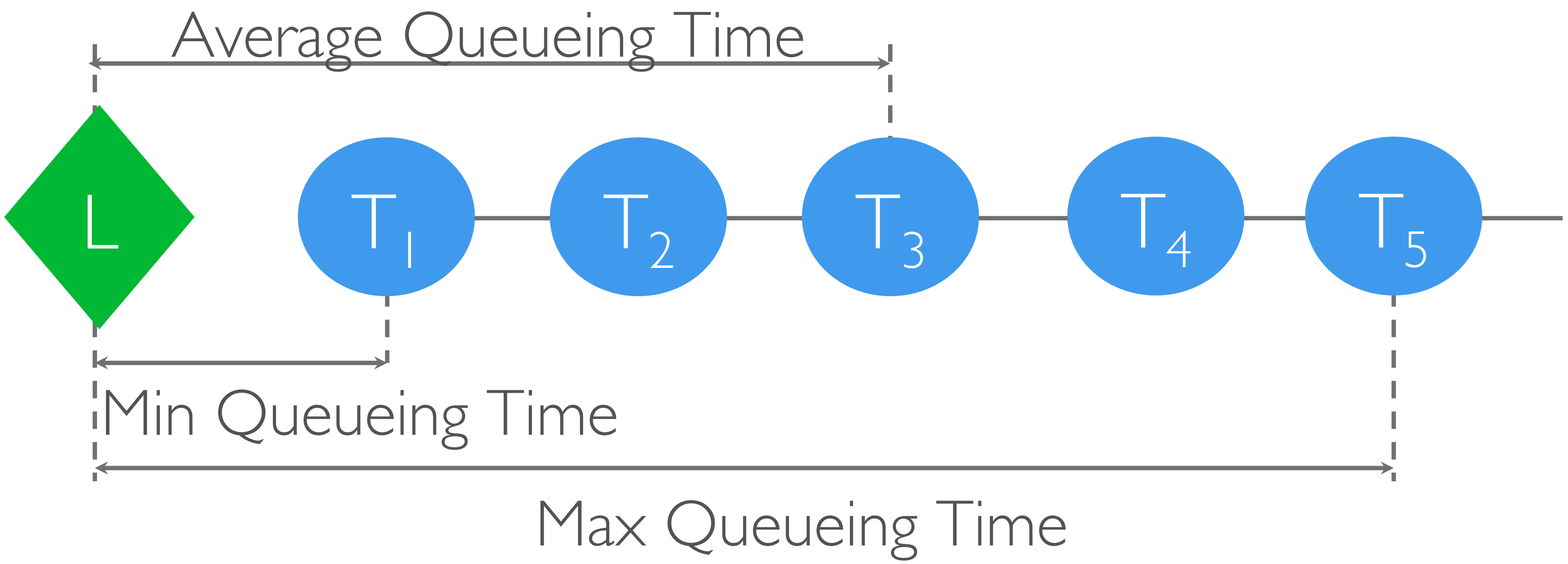
Mitigating Performance Variance

1. Changing the implementation
   - Parallel Logging
2. Changing the algorithm
   - VATS, LLU
3. Changing the tuning parameters
   - Buffer pool size, redo log flush policy, etc.

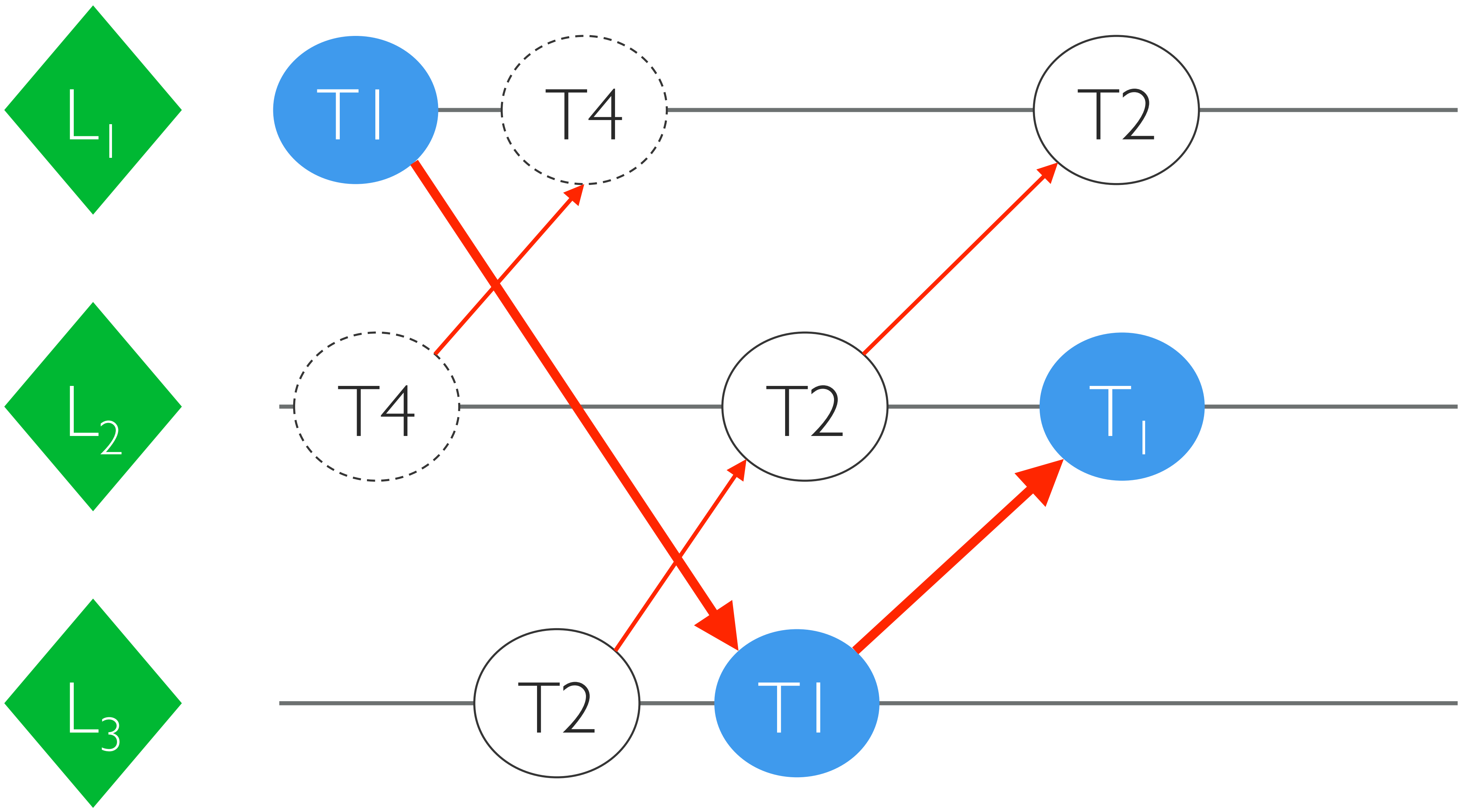
Mitigating Performance Variance

1. Changing the implementation
   - Parallel Logging
2. Changing the algorithm
   - VATS, LLU
3. Changing the tuning parameters
   - Buffer pool size, redo log flush policy, etc.

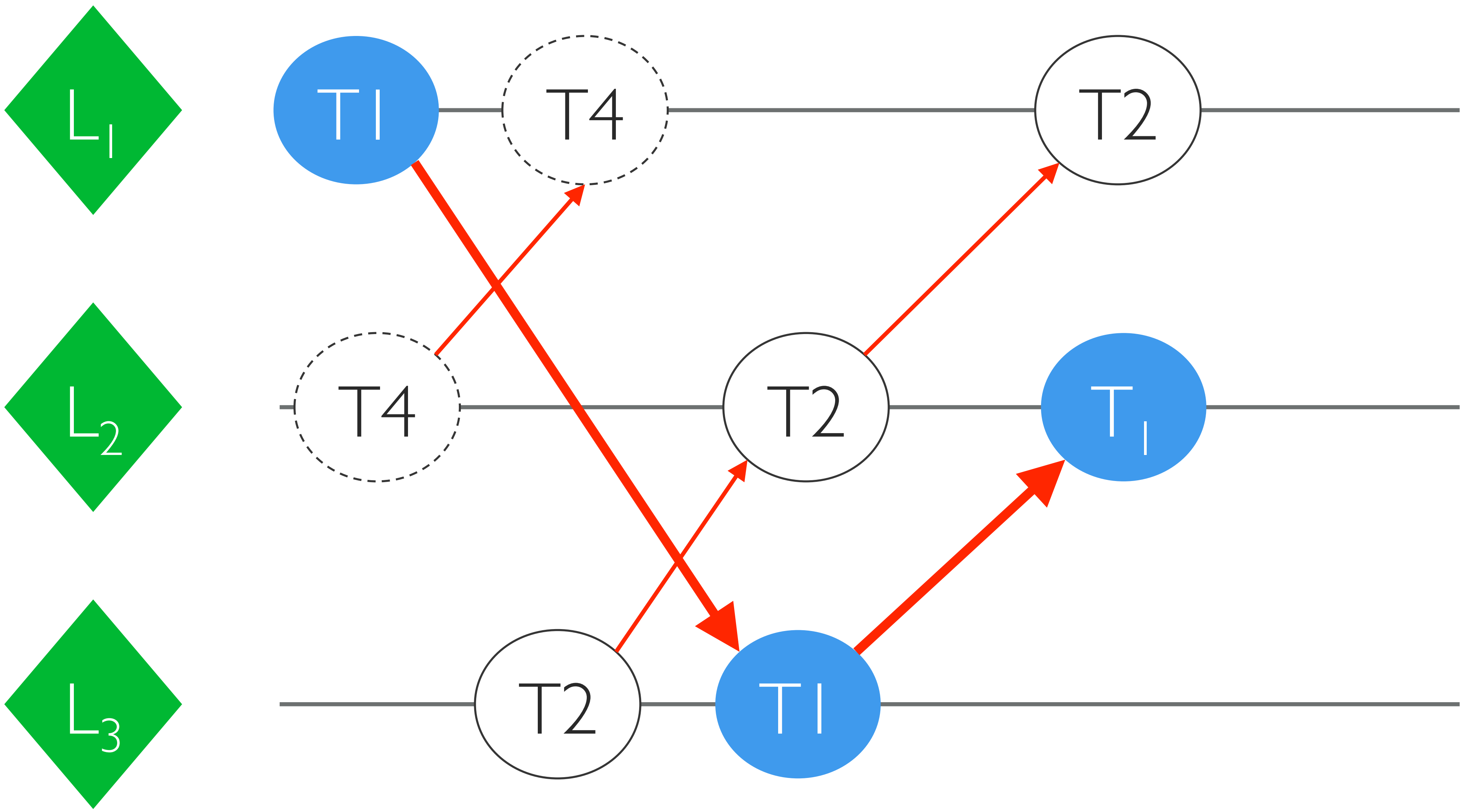
Latency Variance Caused by Queuing



Our Insight: Look at the Big Picture

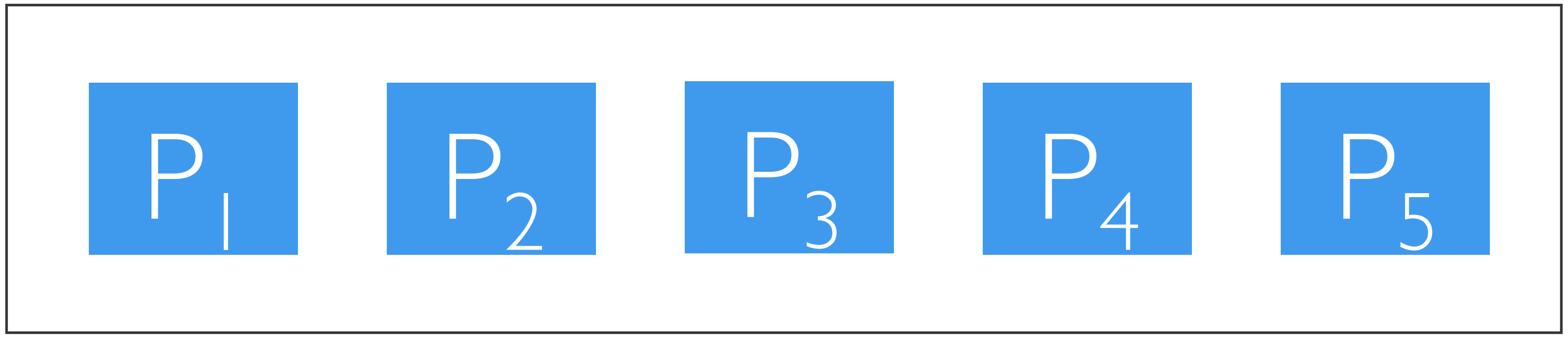


VATS: Variance Aware Transaction Scheduling Algorithm



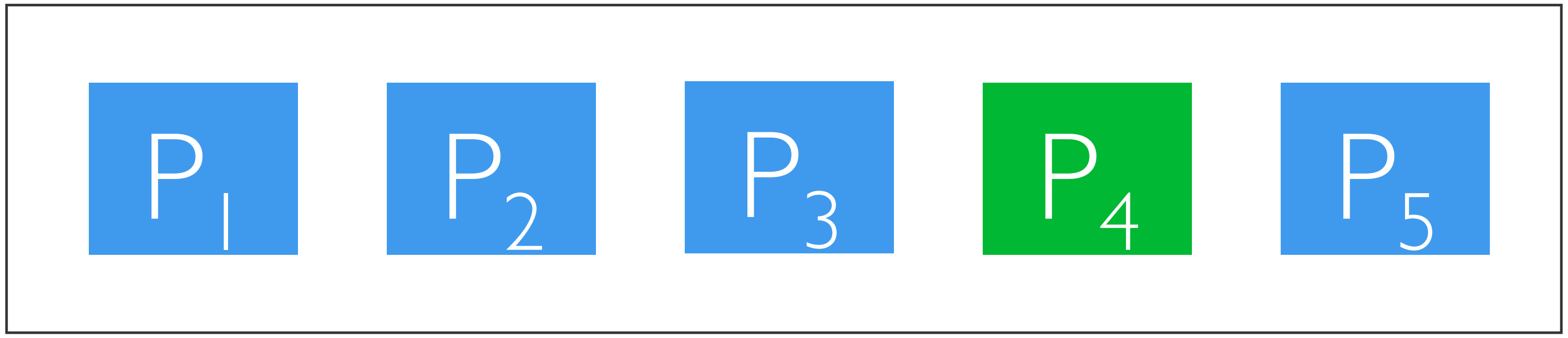
VATS grants locks according to transactions' arrival time in the system, not in the queue (earliest first)

LRU Ordering of Buffer Pages



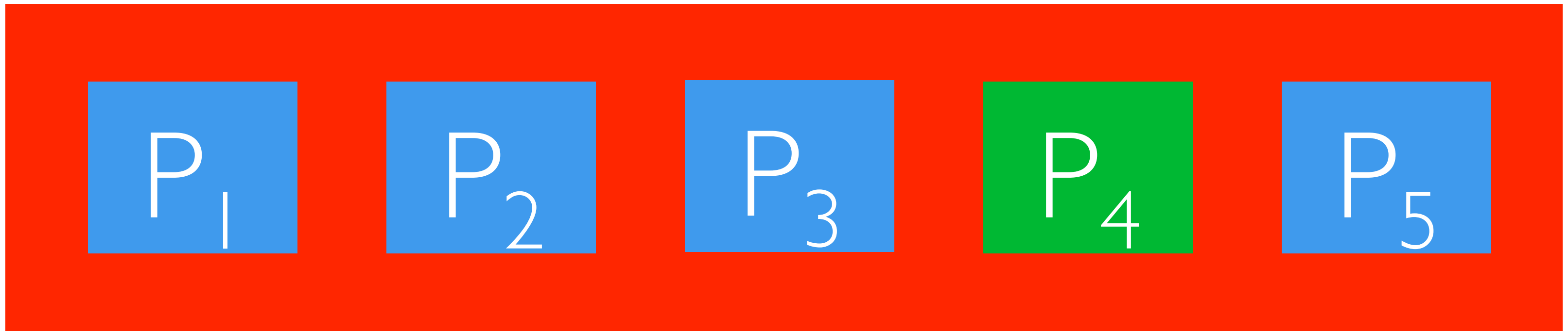
List of buffer pages

LRU Ordering of Buffer Pages



P_4 is accessed

LRU Ordering of Buffer Pages



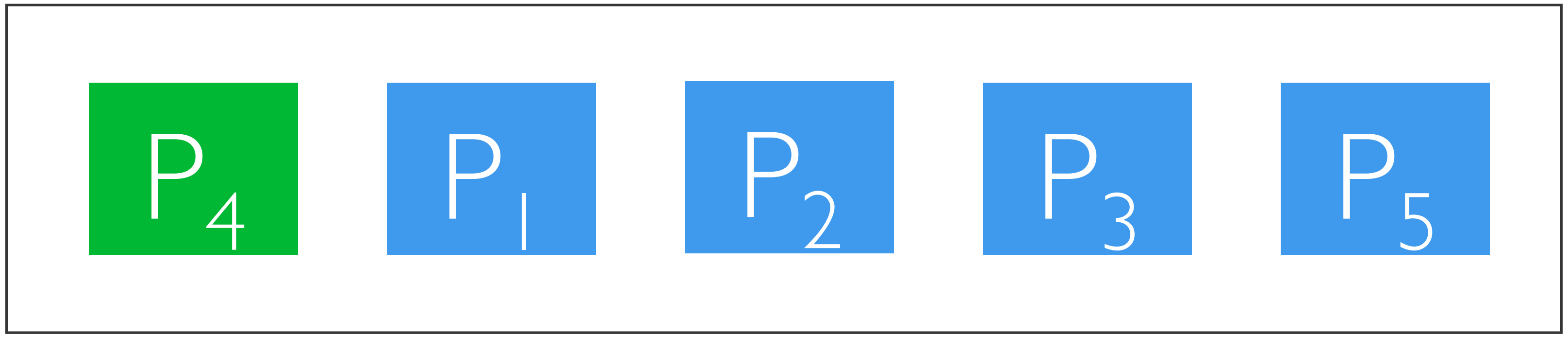
The whole list is locked

Place where variance occurs

LRU Ordering of Buffer Pages

P_4 is moved to the head

LRU Ordering of Buffer Pages



Lock is released

Solution: Use a lazy page update algorithm (LLU)

Variance-aware Tuning

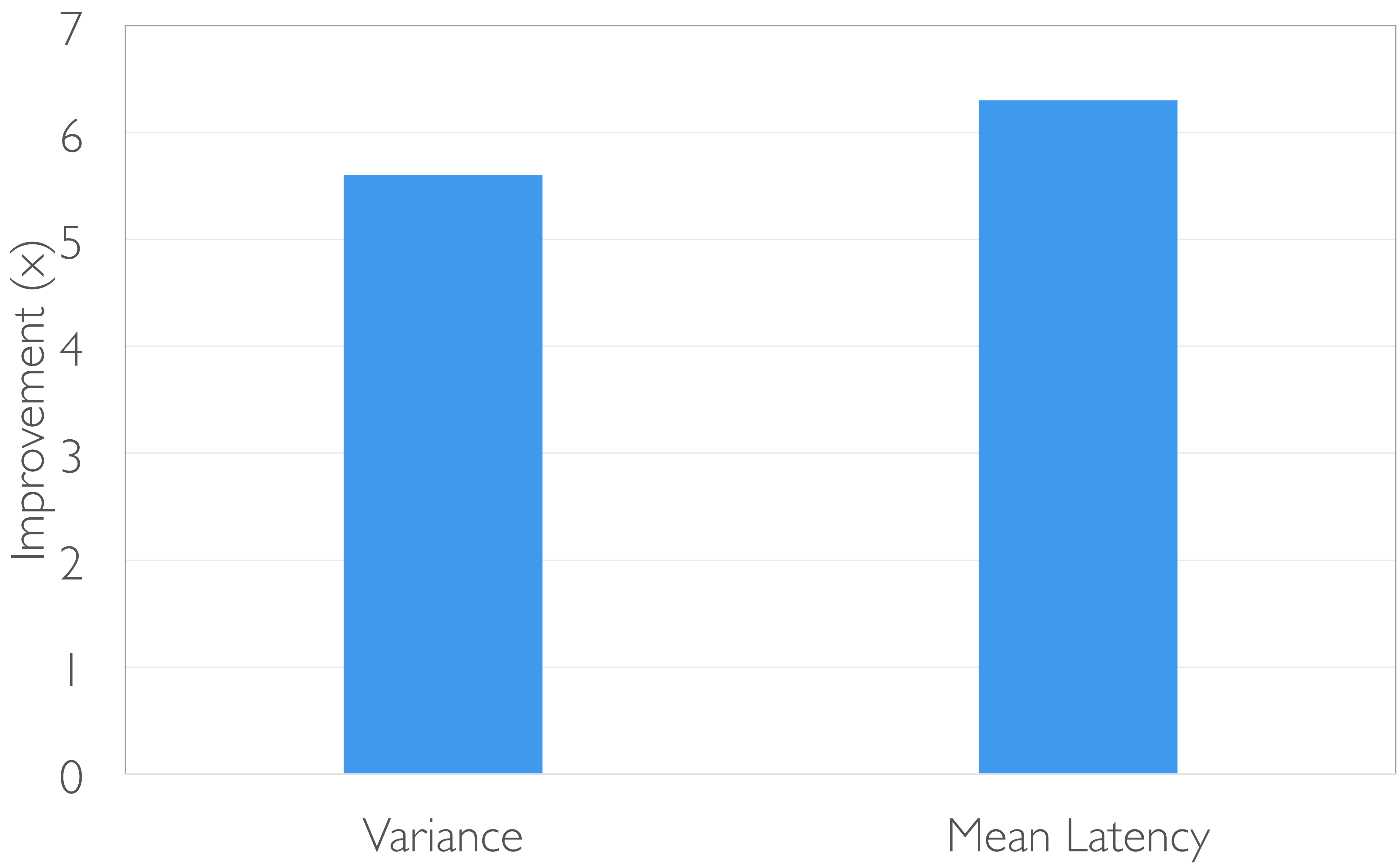
- buf_pool_mutex_enter – buffer pool size
- 33%
- 66%
- 100%

Key Questions

1. How to identify sources of variance?
2. What makes today's DBMSs unpredictable?
3. How to achieve perf. predictability?
4. How effective are our techniques?

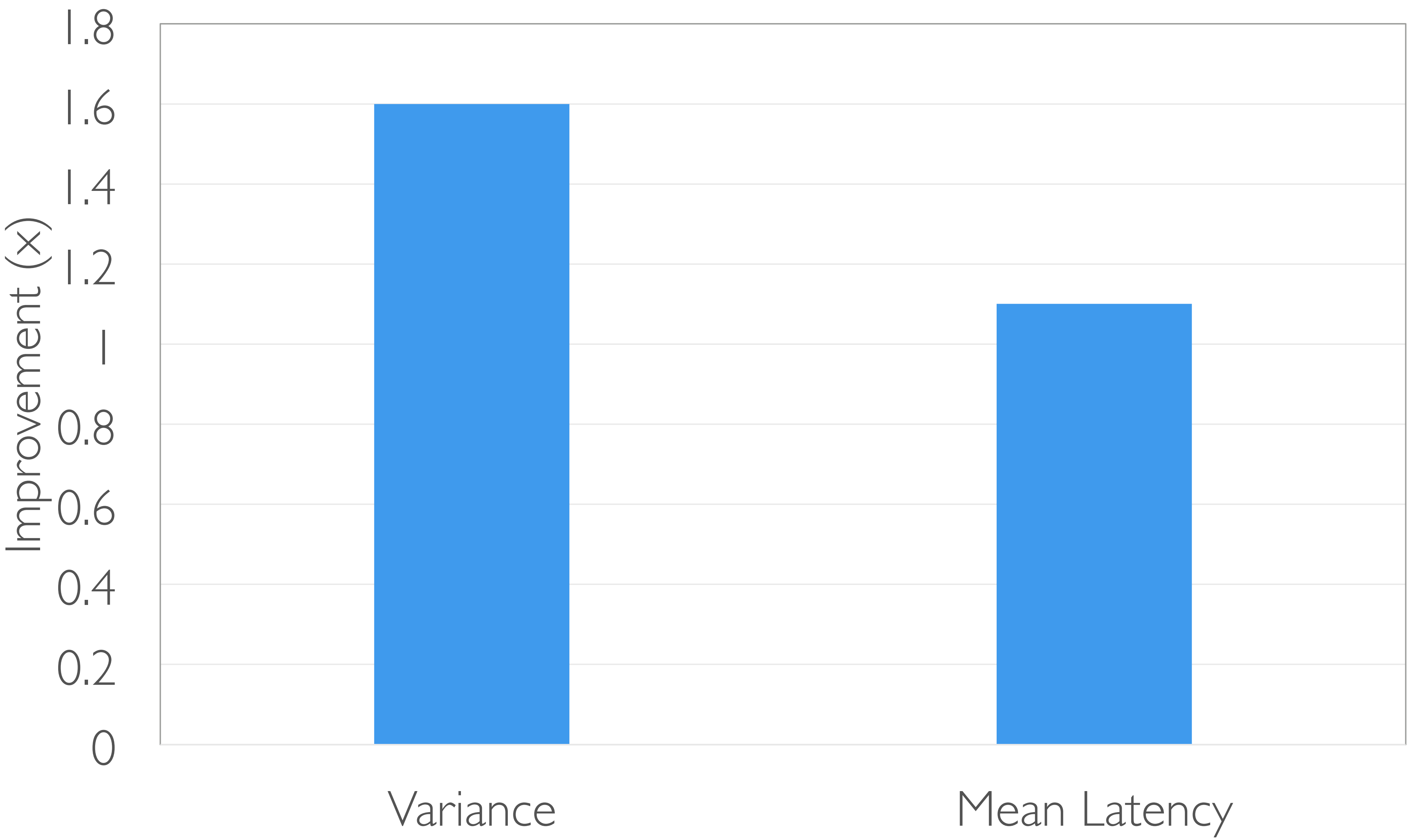
VATS Improvement

- 189 lines of code changed in MySQL

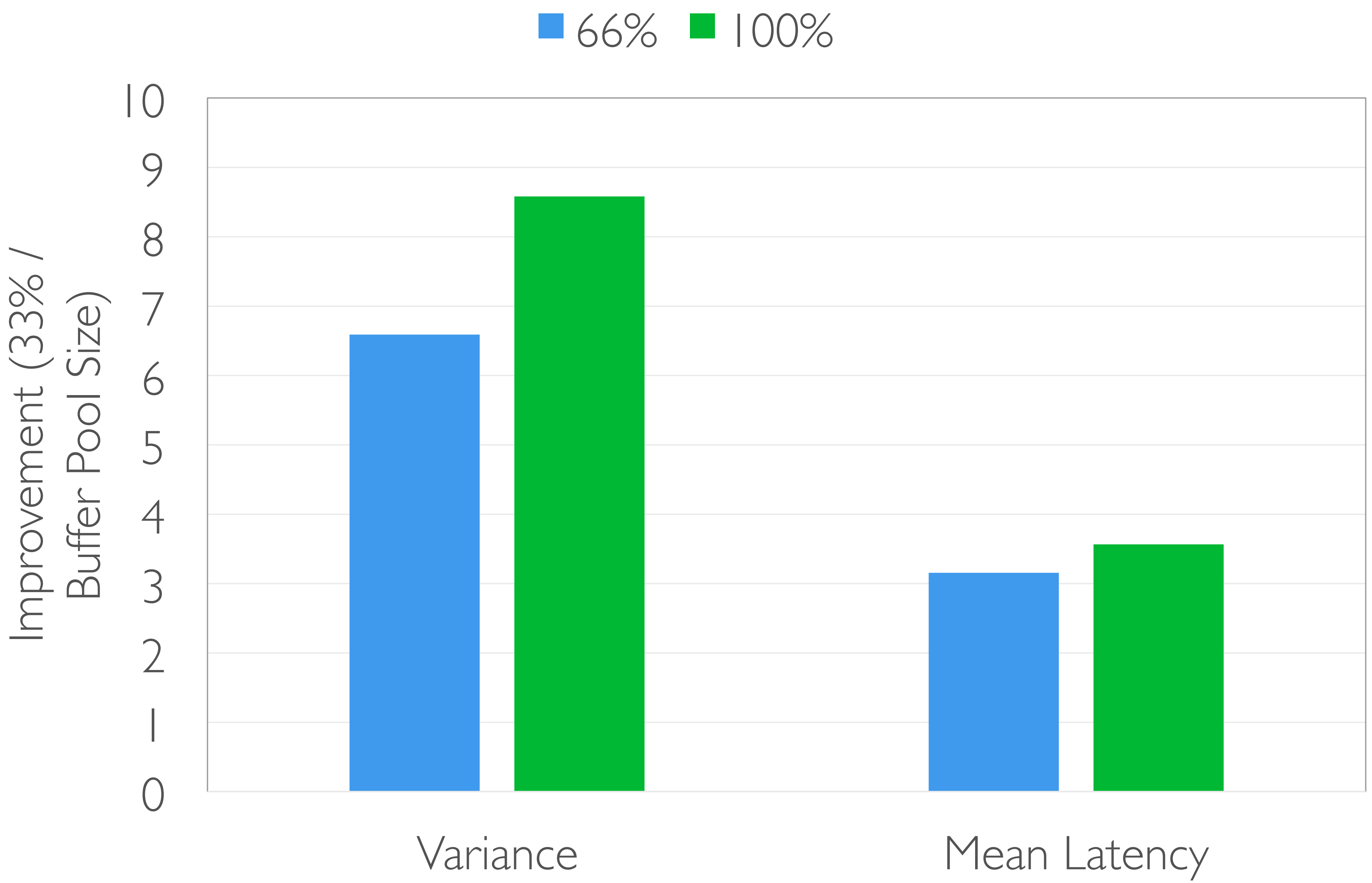


LLU Improvement

- 46 lines of code changed in MySQL



Buffer Pool Size Tuning



Real-world Adoption

- TProfiler open-sourced
- VATS has been merged into MySQL distributions (default in MariaDB and staged in Oracle MySQL)
  - 2M+ installations in the world
- Our buffer pool problem independently discovered and fixed in MySQL 5.8.0

Conclusion

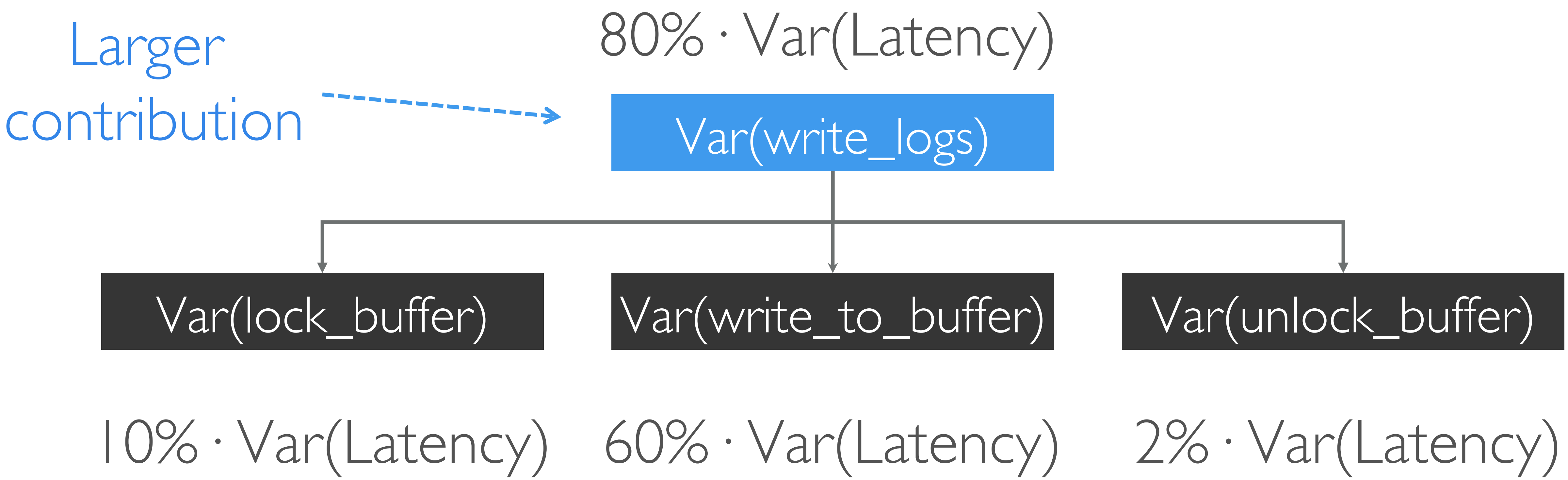
- Predictability is an increasingly critical dimension of modern software overlooked in today's DBMSs
- TProfiler identifies root causes of perf. variance in a principled fashion
  - Enable local and surgical changes to complex DBMS codebases
- Lock waiting is major source of perf. variance in today's DBMSs
- Variance-aware scheduling, lazy optimizations, and tuning strategies dramatically improve predictability w/o sacrificing raw performance

Backup Slides

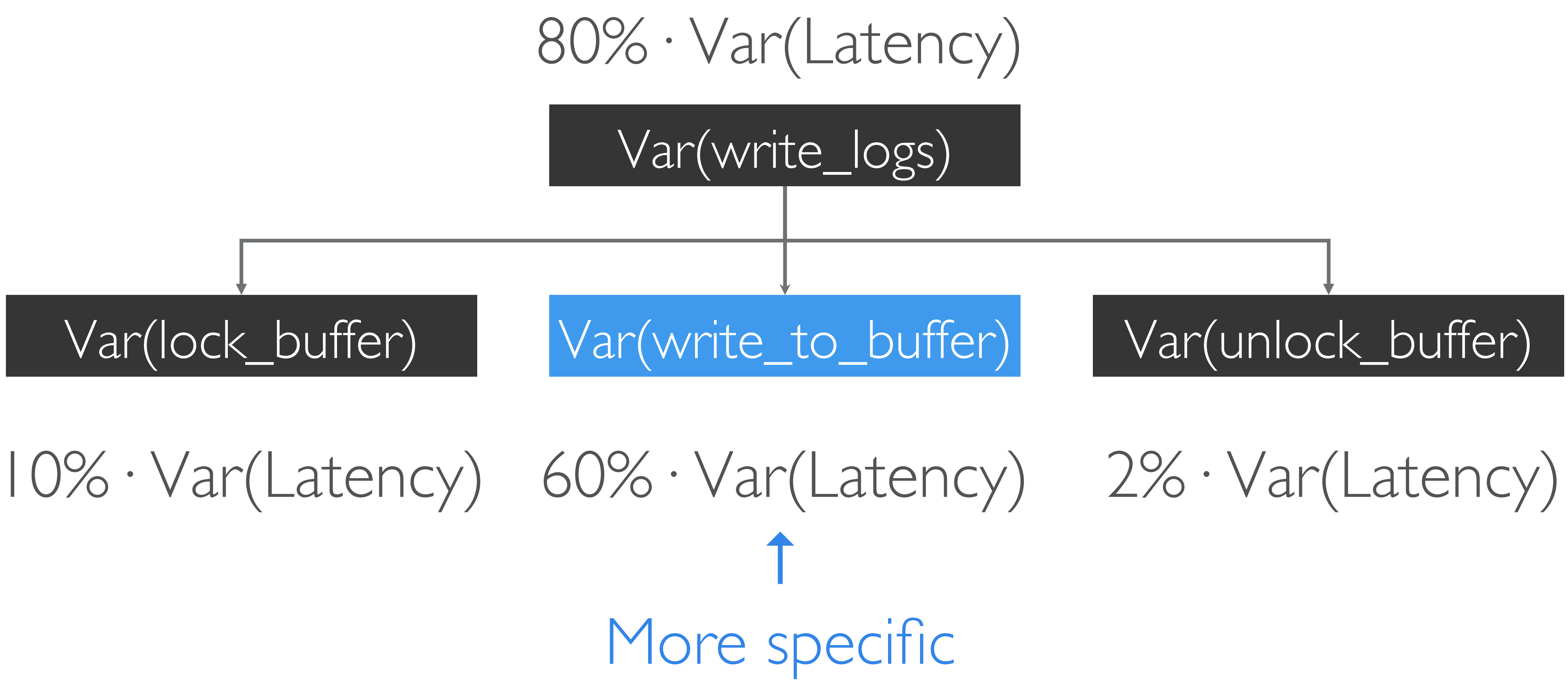
Definition of Predictability

- Many ways to capture *perf. predictability*
  - Minimize latency variance or tail latencies
  - Bound latency variance or tail latencies
  - Minimize the (*stdev* / *mean*) ratio
- Our focus: identifying source of latency variance
  - Reducing variance without sacrificing mean latency

Node Selection Example



Node Selection Example



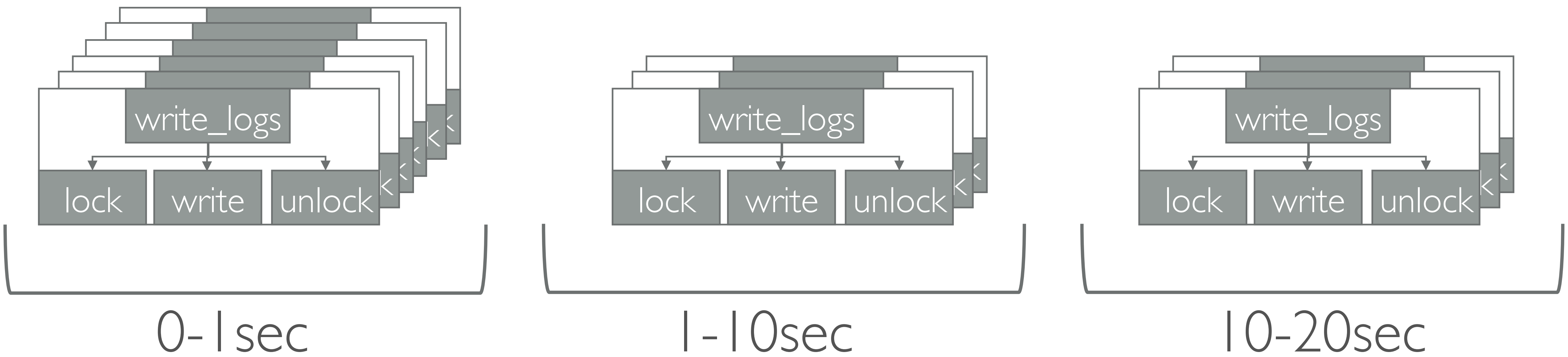
The lower in the variance tree, the more specific

Manual Efforts

Application	Semantic Interval Annotation	# of TProfiler Runs	Avg. Manual Inspection Time per Run	Modified Lines of Code
MySQL	9 lines of code	37	6 minutes	235
Postgres	7 lines of code	16	10 minutes	355
Httpd	4 lines of code	17	12 minutes	45

Related Work: DARC

- Uses multiple runs to produce latency histograms



- Can find man contributors of latency in each execution time range

≠ Main contributors of latency variance in a semantic interval

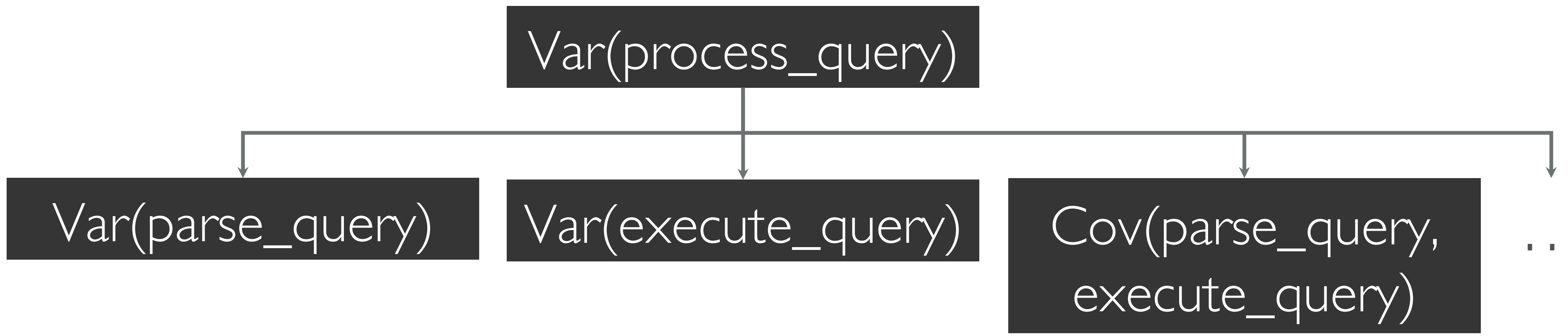
1. Tree Expansion

Root Creation

Var(process_query)

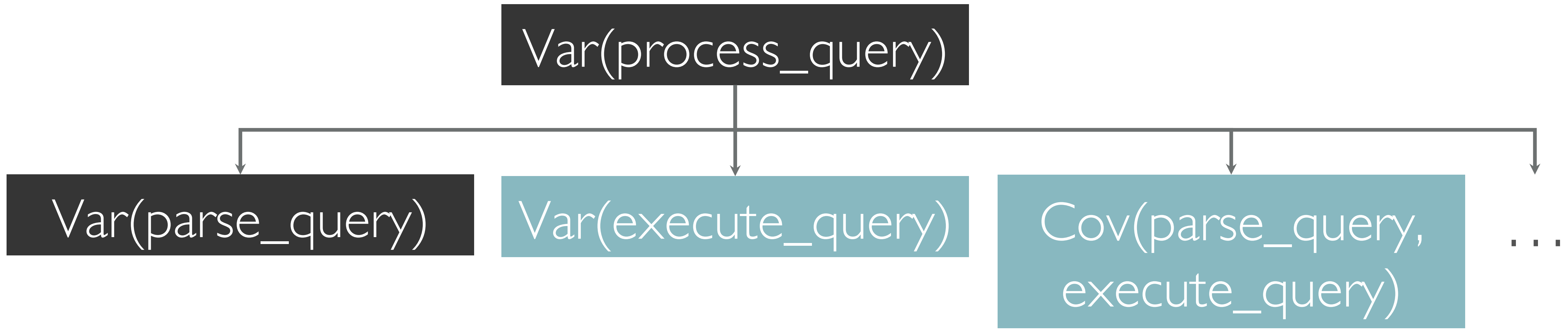
Set the root to the variance of the top level function for query processing

1. Tree Expansion



Break down the root and expand the variance tree

2. Node Selection



Select the most "informative" nodes from the tree

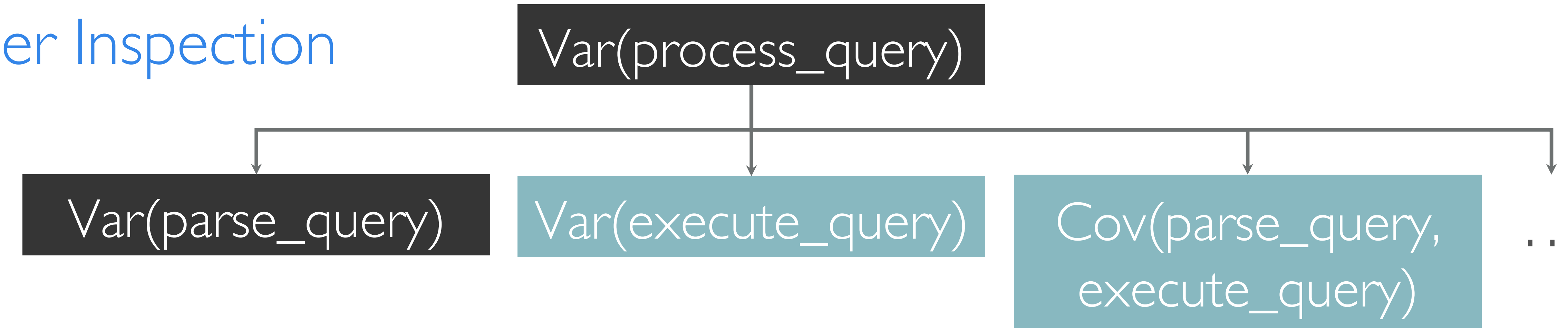
informative = large-enough value + deep-enough in the tree

Variance Contribution

Specificity

3. User Inspection

User Inspection



- Ask for user inspection when:

1. *Cov* terms are large
    - Study how to de-correlate the two functions
2. *Var* terms are both large and deep
    - If cause is still unclear, repeat the expand-select-inspect process