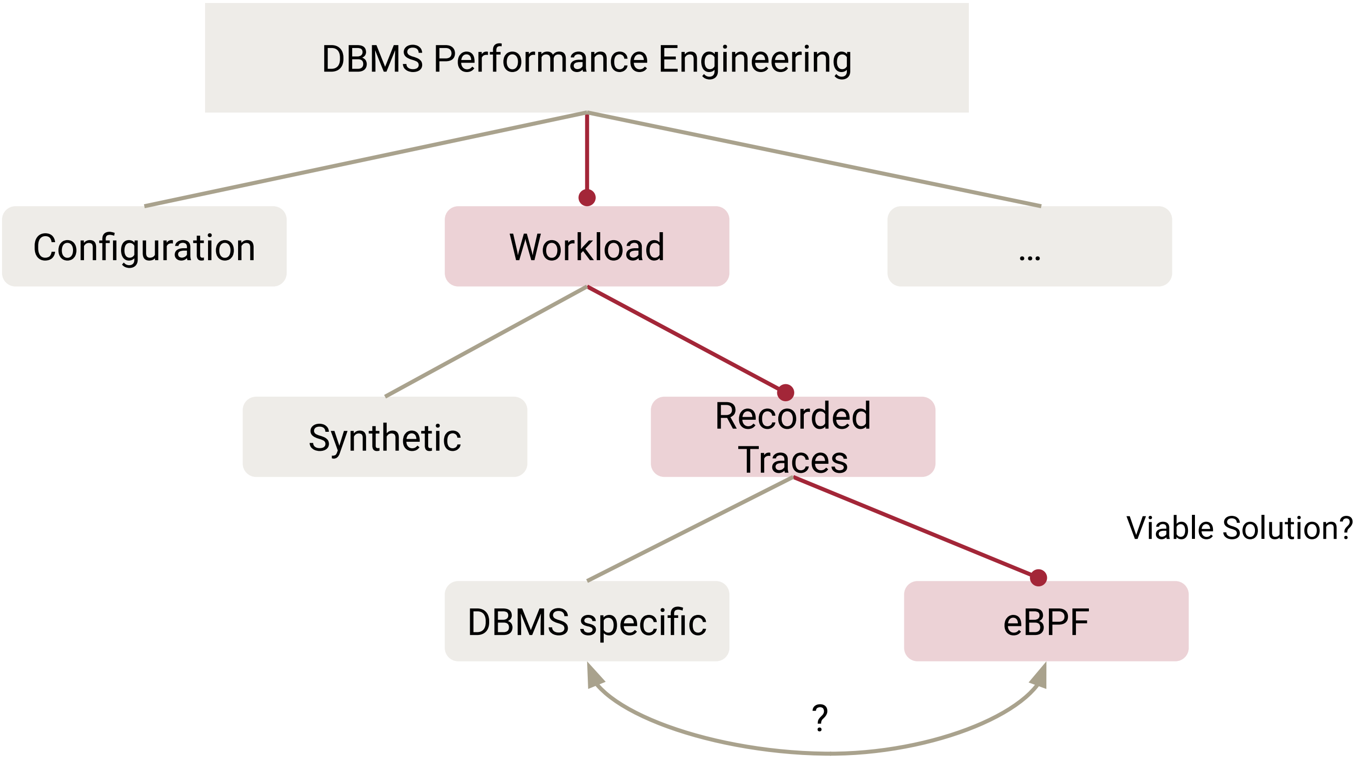
Using eBPF for Database Workload Tracing: An Explorative Study

benchANT

Georg Eisenhart

Jörg Domaschka, Simon Volpert, Kevin Maier, Georg Eisenhart, Daniel Seybold

Motivation

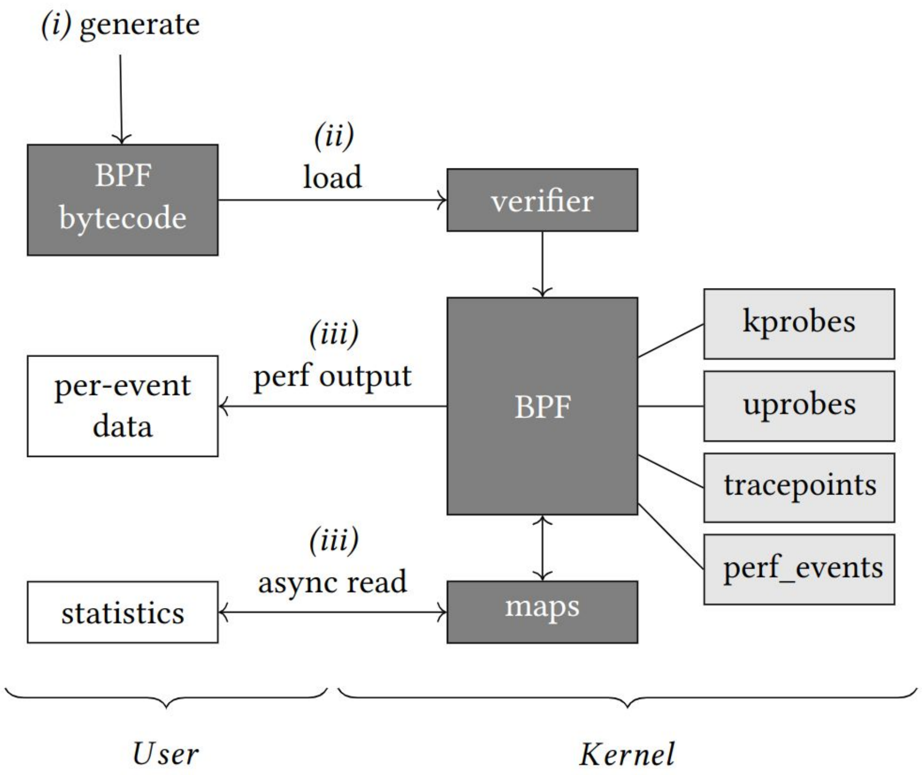

DBMS Performance Engineering
Configuration
Workload
…
Synthetic
Recorded Traces
Viable Solution?
DBMS specific
eBPF
?

Research Questions

eBPF aims to enable low-overhead observability & tracing

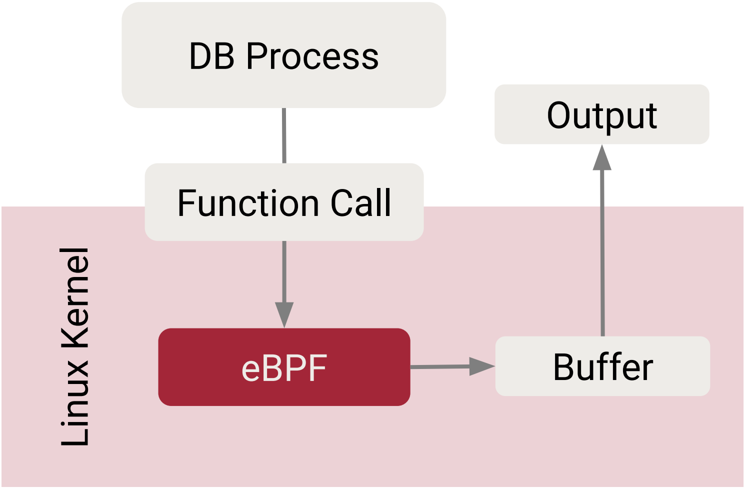
- **RQ1** Can DBMS be instrumented (using eBPF) in order to trace occurring workload?
- **RQ2** How big is the impact of such an (eBPF) instrumentation on the overall performance?
- **RQ3** How does the eBPF impact compare to native DBMS tracing?

eBPF Overview



According to https://www.brendangregg.com/ebpf.html

eBPF Example



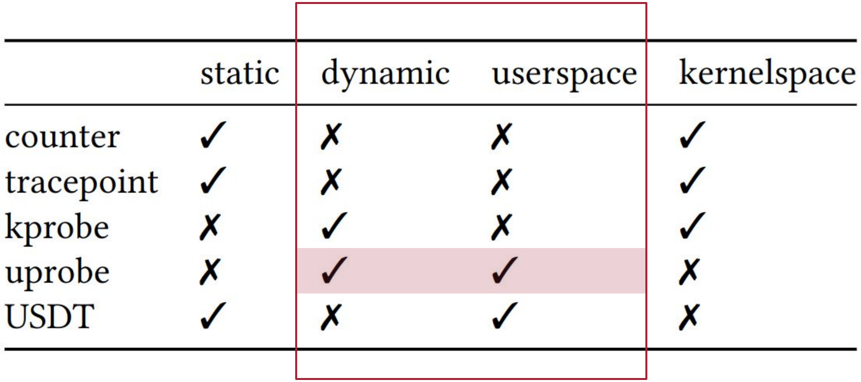
```
bpf.attach_uprobe(name=args.path,
sym="\\w+dispatch_command\\w+", fn_name="query_start")
```

```
int query_start(struct pt_regs *ctx) {
    int zero = 0;
    struct temp_t *tmp = temp_data_buffer.lookup(&zero);
    if (!tmp)
        return 0;
    tmp->timestamp = bpf_ktime_get_ns();

#if defined(MYSQL56)
    bpf_probe_read_user(&tmp->query, sizeof(tmp->query),
(void*) PT_REGS_PARM3(ctx));
. . .
#endif

    u64 pid = bpf_get_current_pid_tgid();
    temp_data_buffer.update(&zero, tmp);
    return 0;
}
```

Excerpt of
https://github.com/benchANT/dbms-tracing-overhead

Probe Selection

- User space tracing
  - USDT and uprobes allow capturing function invocations including parameters and execution time.
- We focus on **uprobes**
- Attach uprobe to the query start
- Attach uretprobe to query end



	static	dynamic	userspace	kernelspace
counter	✓	✗	✗	✓
tracepoint	✓	✗	✗	✓
kprobe	✗	✓	✗	✓
uprobe	✗	✓	✓	✗
USDT	✓	✗	✓	✗

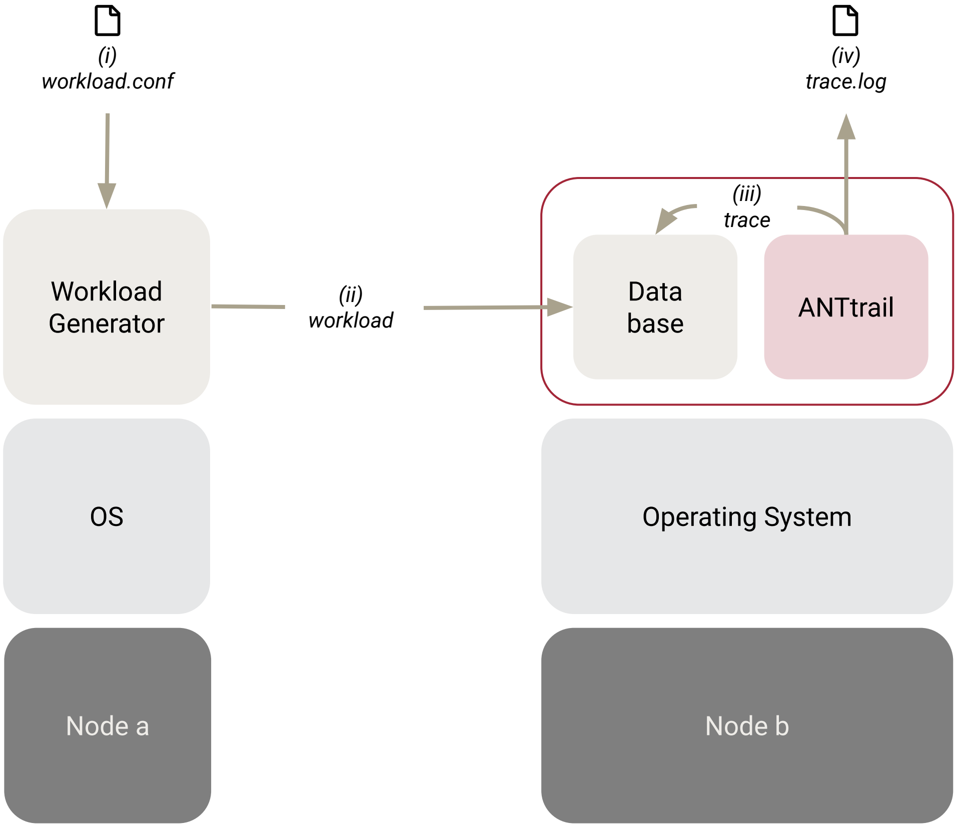
Toolset

- Baseline tool
  - Dbslower[1]
  - On function call detection collect timestamp and arguments
  - On end detection calculate execution time
- Tool Extensions
  - Extended char array to hold full queries
  - Added support for PostgreSQL uprobes
  - Tracing and logging af any query can be written to files
  - Ability to replay later for production workload analysis
  - Remove filters to be able to trace fast queries

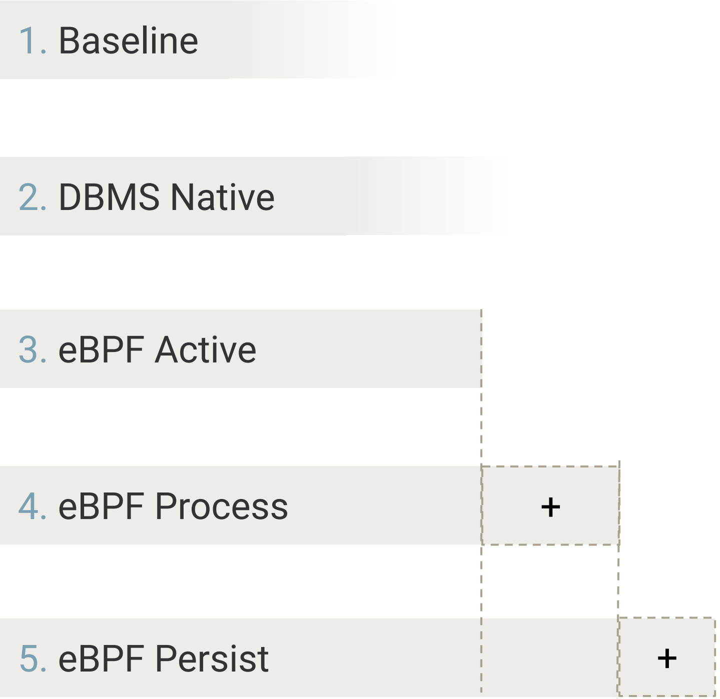
ANTrail

[1] https://github.com/iovisor/bcc/blob/master/tools/dbslower.py

Experiment Workflow

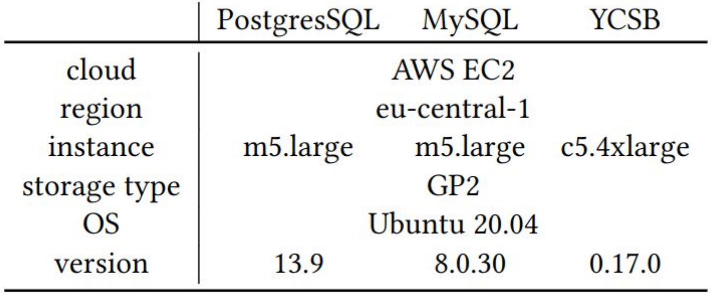


Evaluation Scenarios

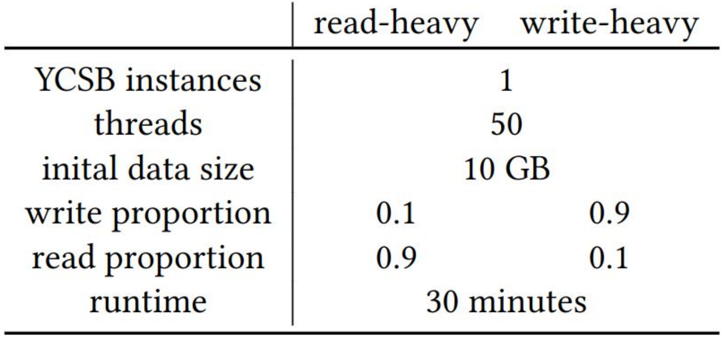


1. Baseline	DBMS Performance without any tracing in place
2. DBMS Native	Native DBMS specific tracing capabilities enabled
3. eBPF Active	Detect queries with ANTtrail, without processing
4. eBPF Process	Process queries with ANTtrail, but not persisting
5. eBPF Persist	Process and persist queries to file with ANTtrail

Evaluation Setup

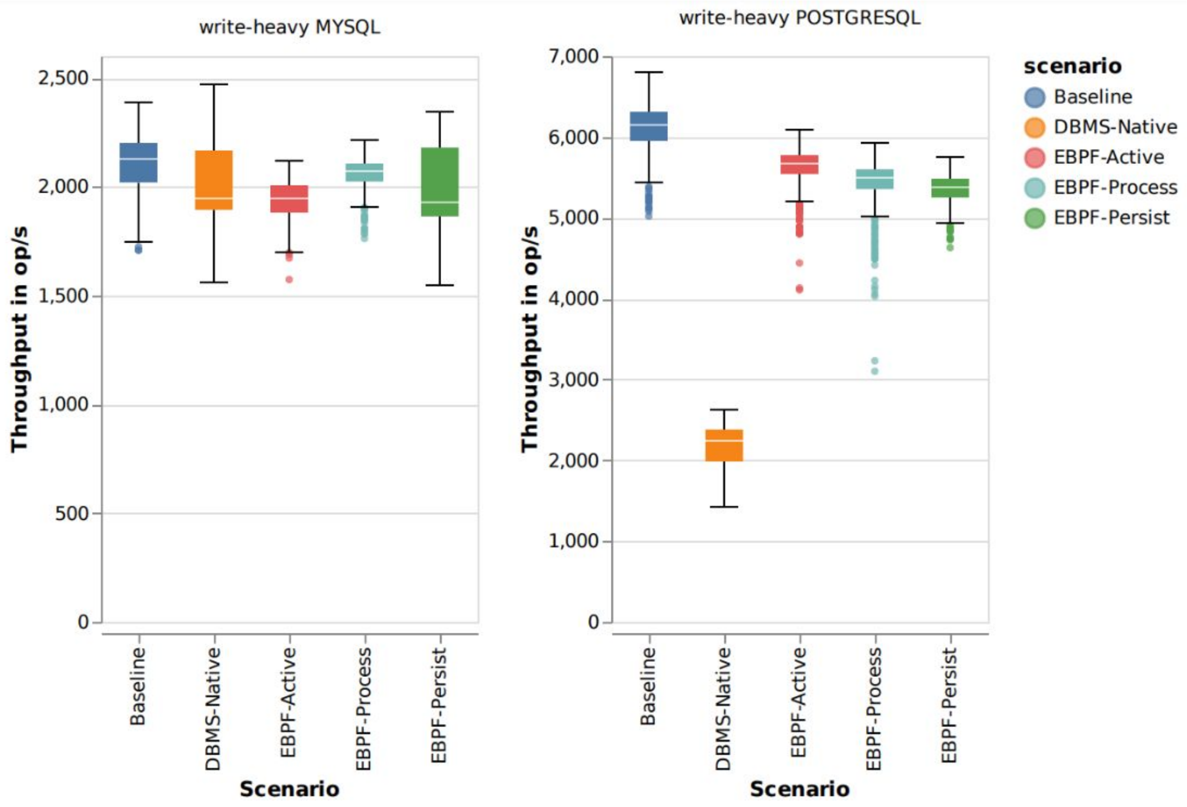


	PostgresSQL	MySQL	YCSB
cloud	AWS EC2		
region	eu-central-1		
instance	m5.large	m5.large	c5.4xlarge
storage type	GP2		
OS	Ubuntu 20.04		
version	13.9	8.0.30	0.17.0

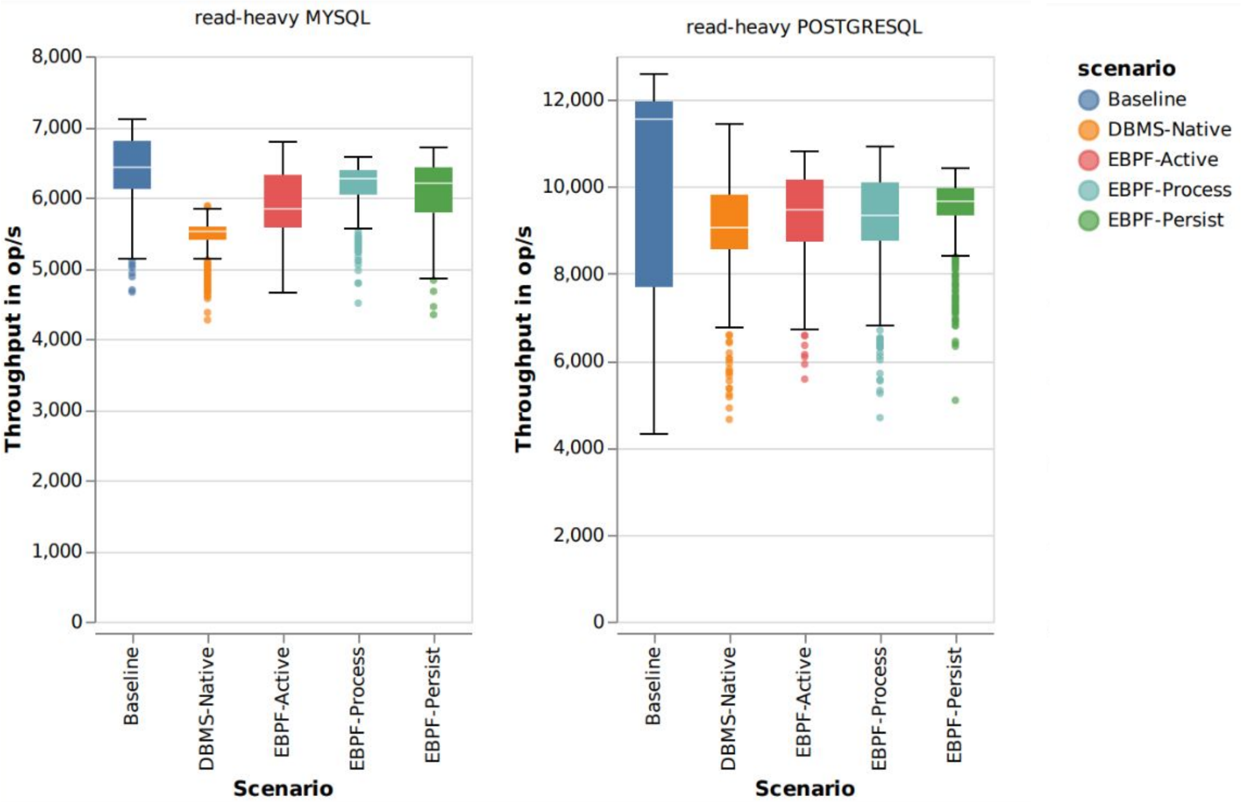


	read-heavy	write-heavy
YCSB instances	1	
threads	50	
inital data size	10 GB	
write proportion	0.1	0.9
read proportion	0.9	0.1
runtime	30 minutes	

Results – Write-Heavy Workload



Results – Read-Heavy Workload



Result Evaluation

- **RQ1** Instrumentation with eBPF using uprobes works.
  No technical constraints to use with other DBMS.

- **RQ2** Performance impact with eBPF-based workload tracing is not stable across different workloads.
  Overhead depends on DBMS technology under test.

- **RQ3** eBPF-based approach competes with DBMS-native tracing or outperforms it.

Summary

- eBPF has similar or lower impact on database performance than native DBMS tracing
- Impact highly depends on applied workload
- Very specific to database implementation
- DBMS independent traces via eBPF possible
- Generation of real world database traces helps in operations
- Our approach provides a non-intrusive method
- May be usable in production for selected cases

Outlook

- Add more DBMS
  - Like MongoDB and Redis
- Improvements and optimizations of the eBPF program
- Take various probes into account e.g. kprobes
  - Retrieve query on signal path
- Different benchmarks and workloads
  - E.g. TPC

Thank you for attention!

Questions?

Source code and data available at:
https://github.com/benchANT/dbms-tracing-overhead

Results

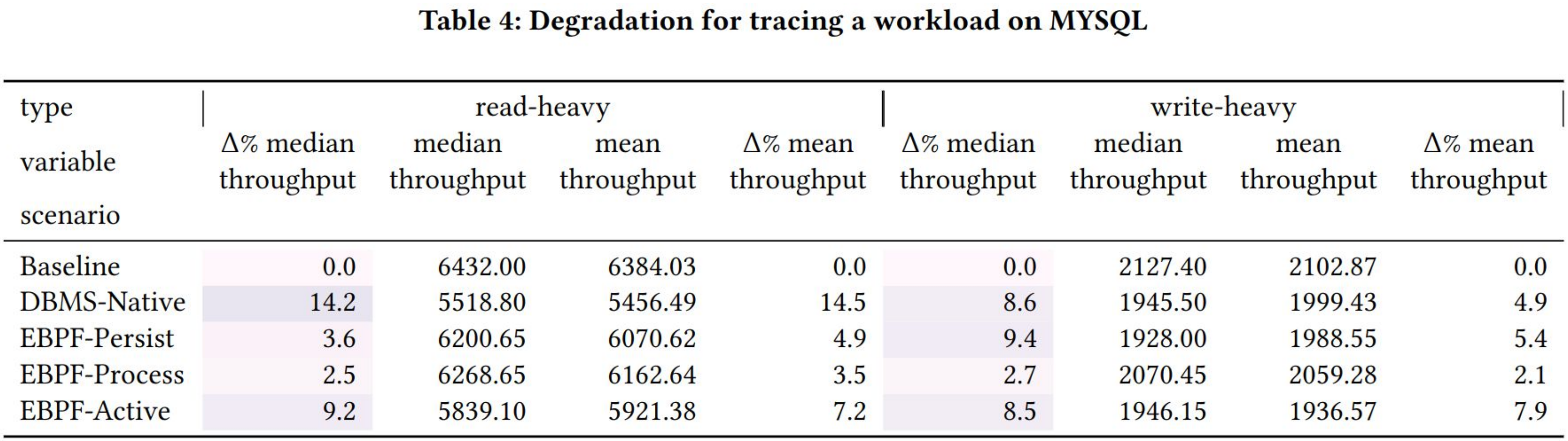
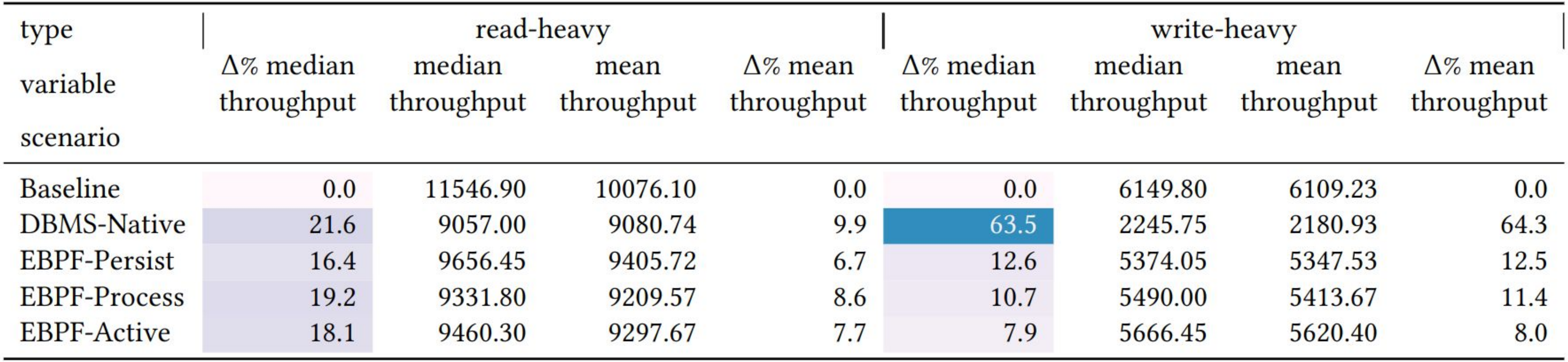


Table 4: Degradation for tracing a workload on MYSQL

type	read-heavy				write-heavy			
variable	Δ% median throughput	median throughput	mean throughput	Δ% mean throughput	Δ% median throughput	median throughput	mean throughput	Δ% mean throughput
scenario								
Baseline	0.0	6432.00	6384.03	0.0	0.0	2127.40	2102.87	0.0
DBMS-Native	14.2	5518.80	5456.49	14.5	8.6	1945.50	1999.43	4.9
EBPF-Persist	3.6	6200.65	6070.62	4.9	9.4	1928.00	1988.55	5.4
EBPF-Process	2.5	6268.65	6162.64	3.5	2.7	2070.45	2059.28	2.1
EBPF-Active	9.2	5839.10	5921.38	7.2	8.5	1946.15	1936.57	7.9

Results

Table 5: Degradation for tracing a workload on POSTGRESQL



type	read-heavy				write-heavy			
variable	Δ% median throughput	median throughput	mean throughput	Δ% mean throughput	Δ% median throughput	median throughput	mean throughput	Δ% mean throughput
scenario								
Baseline	0.0	11546.90	10076.10	0.0	0.0	6149.80	6109.23	0.0
DBMS-Native	21.6	9057.00	9080.74	9.9	63.5	2245.75	2180.93	64.3
EBPF-Persist	16.4	9656.45	9405.72	6.7	12.6	5374.05	5347.53	12.5
EBPF-Process	19.2	9331.80	9209.57	8.6	10.7	5490.00	5413.67	11.4
EBPF-Active	18.1	9460.30	9297.67	7.7	7.9	5666.45	5620.40	8.0