Analytics Are Heavy. The DBMS Is Busy. When Will My Mission-Critical Transaction Start Running?

Jiatang Zhou
Simon Fraser University
jiatangz@sfu.ca

Kaisong Huang
Simon Fraser University
kha85@sfu.ca

Zhuoyue Zhao
University at Buffalo
zzhao35@buffalo.edu

Dong Xie
The Pennsylvania State University
dongx@psu.edu

Tianzheng Wang
Simon Fraser University
tzwang@sfu.ca

ABSTRACT

Conventional non-preemptive scheduling strategies struggle to meet the latency requirements of mixed workloads: low-priority, long-running analytics can dominate CPU cores while short, high-priority transactions wait a long time to be scheduled. Although preemptive scheduling appears to be a natural solution, it has long been discouraged in DBMSs by conventional wisdom due to concerns about deadlocks and interrupt-handling overheads. In this demonstration, we highlight that this is no longer the case with PreemptDB, a modern memory-optimized DBMS that we built around (1) optimistic concurrency and (2) userspace interrupts that recently became available in x86 CPUs. PreemptDB proposes user-interrupt-assisted context switching to renew preemptive scheduling in modern DBMSs. Through a set of demonstration scenarios, we show that preemptive scheduling is practical and prioritizes high-priority transactions while preserving throughput and fairness.

PVLDB Reference Format:
Jiatang Zhou, Kaisong Huang, Zhuoyue Zhao, Dong Xie, and Tianzheng Wang. Analytics Are Heavy. The DBMS Is Busy. When Will My Mission-Critical Transaction Start Running?. PVLDB, 18(12): 5299 - 5302, 2025.
doi:10.14778/3750601.3750656

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/sfu-dis/preemptdb.

1 INTRODUCTION

As database applications become increasingly heterogeneous [5, 8], modern database systems need to handle workloads where transactions of vastly different lengths and priorities coexist. In particular, short, high-priority (e.g., mission-critical) transactions may co-exist with long, low-priority analytics. For example, in e-commerce scenarios, heavyweight operational reporting can run concurrently with short, latency-sensitive sales transactions. The former can identify sales trends using the most up-to-date sales data for better decision making in real time, while the latter is the core of the application.

Scheduling these two types of transactions in one system, however, is non-trivial as they have conflicting desiderata. On the one hand, users prefer for short, high-priority transactions to be admitted and executed immediately by the DBMS. On the other hand, long analytics read much data and therefore prefer to distribute work across all available CPU cores for faster execution. As a result, without special care—for example by simply employing a first-in-first-out (FIFO) scheduling policy as most systems do—once a long-running analytics query has started to run, it can monopolize the CPU (to accelerate itself) and subsequently delay short but high-priority transactions.

Beyond simple FIFO scheduling, it is natural for DBMS users to pause (i.e., preempt) currently running long transactions to free up CPU resources for higher-priority transactions. However, preemption-based scheduling in DBMSs was discouraged [1] due to issues such as deadlock handling and scheduling overheads. For example, in traditional DBMSs that use locking-based concurrency control, long-running transactions may hold many and/or coarse-grained locks, preempting which may not allow conflicting short transactions to proceed anyway. Preemption was also typically done using traditional interrupts which cross user-kernel space boundaries, adding more overhead. Some systems therefore settle on cooperative scheduling [2, 3, 7] that modifies the workload or engine code to introduce specific yield points. This way, a long-running transaction can voluntarily give up the CPU so that other (high-priority) transactions may have a chance to be scheduled. The drawback of this approach is that such yield points have to be hard coded and for the best performance, one must co-design the application and the DBMS, which is often impractical.

Advances in recent memory-optimized DBMSs and hardware have made preemption practical in DBMS. First, we note that these systems rely on optimistic and/or multi-versioned concurrency control, instead of traditional pessimistic locking. This means long-running analytics will not be holding many locks for extended periods, and preempting them would not result in as many aborts as in traditional DBMSs. Second, recent x86 CPUs (e.g., since Intel 4th Generation Xeon Scalable, formerly known as Sapphire Rapids) offer userspace interrupts (user interrupt or `uintr`) [6] which allow interrupts to be sent, delivered and handled purely in the userspace. These two trends collectively mitigate the previous concerns on preemptive scheduling in DBMSs. Following these observations, our recent work PreemptDB [4] proposed new mechanisms and policies

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750656

for preemptive scheduling. PreemptDB uses lightweight user interrupt to handle high-priority jobs with low latency, even when the CPU is heavily occupied with OLAP tasks. PreemptDB implements multiple transaction contexts for each thread, where each context has its execution states (register states, call stack) and transaction metadata (e.g., current transaction ID), and can perform lightweight context switches among contexts. When a high-priority transaction arrives, it issues a user interrupt to perform a lightweight context switch and to achieve efficient task pause/resume without crossing kernel and userspace boundaries.

In this demonstration, we highlight the design and effectiveness of PreemptDB (and more generally, advocate the idea of preemptive scheduling) by comparing it with the aforementioned traditional approaches and baselines. We do so under a typical hybrid transactional and analytical processing (HTAP) scenario with regular, long-running, low-priority reporting queries and ad-hoc short but high-priority transactions. Next, we first lay out the necessary background on transaction scheduling and user interrupt. We then introduce PreemptDB, followed by our demonstration scenarios.

2 TRANSACTION SCHEDULING POLICIES

In many traditional database systems, the non-preemptive FIFO policy (denoted as **wait**) is the default. It is easy to implement but requires high-priority transactions to wait until all tasks arrived earlier have finished. Thus, the user could experience high latency and high variance in end-to-end transaction latencies. For example, in Figure 1, a high-priority transaction (lasting 1 time unit) that arrives while a long-running transaction (12 time units) is running (at $time = 4$) can experience very high end-to-end latency (9 time units) as it cannot start until time unit 12 when the long transaction finishes.

One way to reduce such high latency is to implement a cooperative **yield** scheduling policy, i.e., to have threads voluntarily give up CPU and switch between tasks. This can be done by strategically inserting yield points where the system checks whether it should switch to a waiting high-priority transaction. This opportunistically improves the end-to-end latencies of high-priority transactions—the more frequently the system checks for yield opportunities, the sooner a high-priority transaction can be scheduled to run. However, it does not fundamentally solve the problem because yielding is limited to specific yield points, and it is not possible to switch before reaching them. As a result, there could still be high variance and long tails in end-to-end latencies. Continuing the example in Figure 1, suppose the system yields every 6 time units in a long transaction. Depending on the arrival time, a short, high-priority transaction can still experience delays of 0 to 6 time units. The first short transaction still needs to wait for 4 time units. A possible mitigation is to increase yield frequency. However, as the number of yield points increases, each yield is an overhead that takes additional CPU cycles, thus decreasing overall throughput. The locations of such yield points are also highly workload dependent. That is, one needs to know precisely the execution time of a transaction and strategically insert yield points inside transaction logic or the DBMS, requiring co-designing the DBMS and application. It is then becomes challenging (if not impossible) for such yield

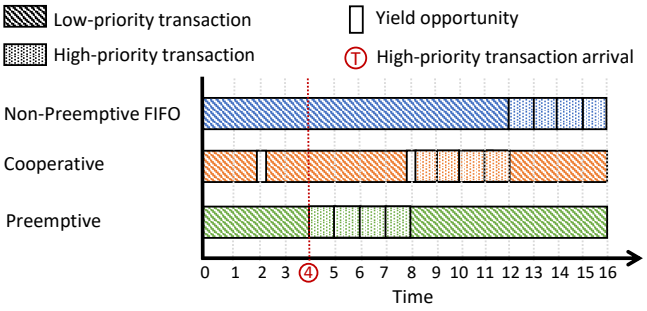

Figure 1: Compared to FIFO/cooperative scheduling, preemption can prioritize and maintain high overall throughput [4].

policies to achieve both high throughput and low latency for high priority.

The ideal policy is **preemptive**, which allows preempting low-priority transactions as high-priority transactions on demand. As shown in Figure 1, a high-priority transaction can almost immediately scheduled and executed when it arrives at time 4, leading to an end-to-end latency of 1. However, as Section 1 states, traditionally it is avoided as the overhead of kernel/user-space switch in regular interrupt mechanisms could lead to significant drop in throughput, and the switch overhead can also be significant for in-memory transactions with μs-level execution latencies. Preemptive execution can also incur deadlocks that would cancel out the benefits. Therefore, most DBMSs have avoided preemption but opted for cooperative scheduling [1].

The emergence of lightweight user interrupt on recent x86 CPUs and changes in the design principles of modern DBMSs have made it possible to (re)introduce preemption into DBMSs. In the remaining sections, we describe our recent work PreemptDB [4] that realizes this idea and demonstration scenarios to showcase its effects.

3 PREEMPTDB

PreemptDB is a memory-optimized OLTP engine that leverages user interrupt for preemptive scheduling. Figure 2 shows an overview of PreemptDB. In addition to worker threads which are responsible for executing transaction logic and performing commit processing, PreemptDB additionally employs a scheduling thread that implements the preemptive scheduling policies (described below), including transaction admission, making scheduling decisions and sending preemption interrupts. What is unique in PreemptDB is that each worker can have multiple interruptible transaction contexts, allowing independent transactions to time-share the same thread or CPU core. While the CPU core is executing in a context, it can be preempted by a user interrupt (sent from the scheduling thread upon receiving a high-priority transaction) and passively switch to another context, or actively switch back to the preempted context (performed at the end of a high-priority transaction to resume execution of the previously interrupted transaction).

PreemptDB currently employs two contexts and two transaction queues per worker thread, one for low-priority transactions and the other for high-priority transactions. When the scheduling thread receives a low-priority transaction, it will try to find an empty slot in one of the worker thread's low-priority transaction queue

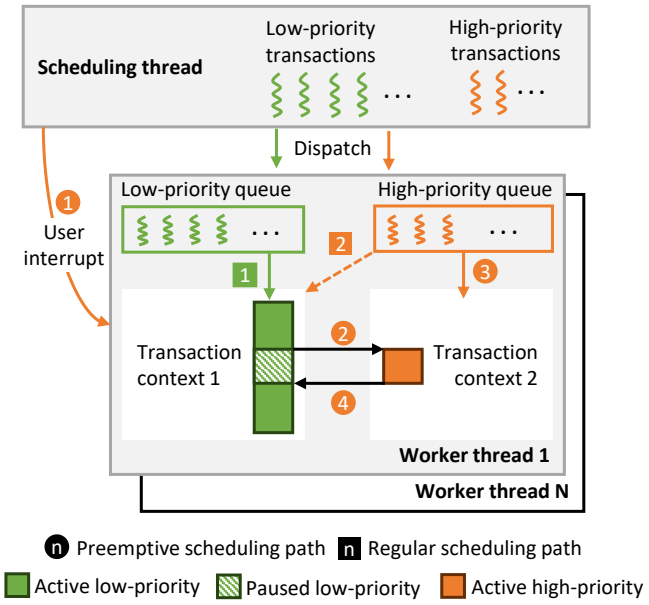

Figure 2: PreemptDB Overview. Upon arrival of a high-priority transaction T, a scheduling thread ❶ issues a user interrupt to preempt and pause an in-progress low-priority transaction S. The worker thread then ❷ switches to another context to ❸ run T and ❹ resumes S after T finishes.

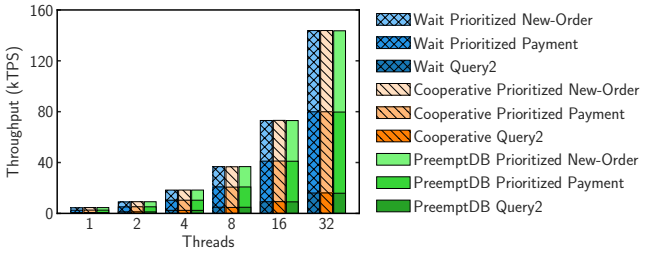

Figure 3: Scalability under different scheduling policies.

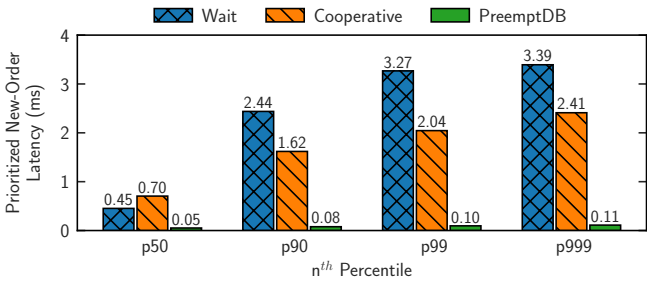

Figure 4: End-to-end latency of high-priority New-Order transaction with 16 worker threads.

in a round-robin fashion. When the scheduling thread receives a high-priority transaction, it will try to find an empty slot in one of the high-priority transaction queues, again in round-robin fashion. Then it will push the transaction to the chosen queue, but *may* immediately send a user interrupt to the chosen thread. This effectively preempts and pauses a low-priority transaction if it is running on that thread, and causes the thread to switch to the high-priority context. As a result, a high-priority transaction may be scheduled as soon as a user interrupt is delivered which takes only ~1 μs [4]. Once scheduled for execution, the transaction will execute on its assigned context; our current policy does not further interrupt a high-priority transaction. Note that it is configurable whether to immediately send a user interrupt for every high-priority transaction (or for a batch periodically), or temporarily disable preemption for starvation prevention purposes under high-priority transaction surges in PreemptDB.

While the idea of preemption using user interrupt is straightforward, realizing it in a full-fledged DBMS engine is challenging. Below we highlight two key challenges and key performance results; interested readers may refer to elsewhere [4] for more details.

Userspace Context Switch. PreemptDB supports both passive (through interrupt) and active (through a function call) context switching among contexts without needing to enter kernel space. Different from OS kernel context switches, which by default only needs to save general and flag register states (as typically Linux OS kernel and modules are not allowed to use X87 floating point registers and other extension registers such as SSE, AVX-2, etc.), our context object needs additionally save the extended processor states using `xsave` and `xrstor` instructions. Also, due to several nuances such as difference between the userspace interrupt handler and regular function call in terms of call stack layout and return instructions, PreemptDB has to specially design a unified stack layout that works for both cases (as a preempted context can actively switch back to the other and vice versa).

Single-Threaded Deadlocks. One pitfall of performing preemption to switch contexts within the same thread is that they are prone to single-threaded deadlocks (akin deadlocks in a single-core multi-threaded system). If the current context holding some locks and been preempted and switch to another context, and the new context tries to acquire a lock that is held by the original context, a single-threaded deadlock will result. To tackle this challenge, PreemptDB relies on both optimistic and multi-version concurrency control (MVCC) to minimize the lock usages, and a lightweight critical section mechanism to temporarily disable context switches for deadlock prevention in the scenarios when we cannot avoid locks (e.g., during transaction commit processing, or inside I/O syscalls).

Performance. We used a mixed TPC-H and TPC-C workload to evaluated PreemptDB. We use TPC-H Query 2 (Q2) as the long, low-priority transaction and TPC-C New-Order as short, high-priority transaction. As shown in Figures 3–4, preemption in PreemptDB provides similar throughput and scalability to non-preemptive policies (wait and yield) while providing substantially lower end-to-end latency for high-priority transactions.

4 DEMONSTRATION PLAN

Our demo highlights PreemptDB's ability to provide low end-to-end latency for short, high-priority transactions under a high load of long-running, low-priority transactions, as well as the effects of different scheduling policies. This is done by (1) a poster introduction and (2) a live demo session.

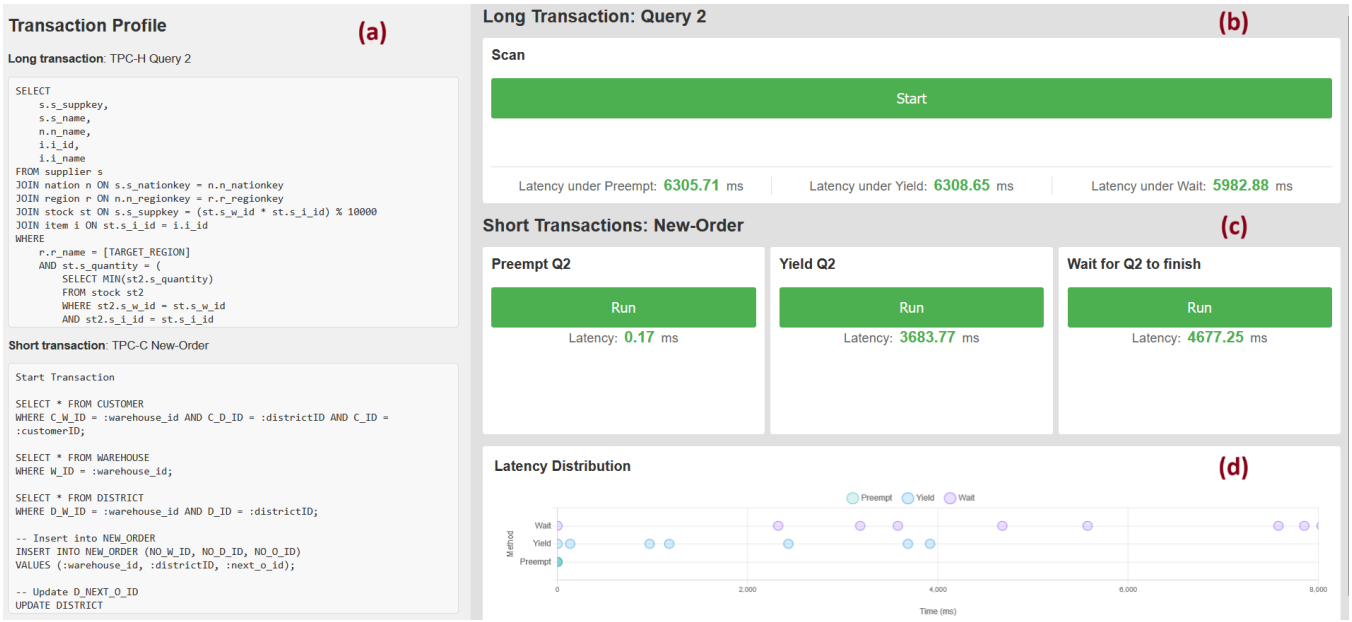

Figure 5: Demonstration UI and workflow. After reviewing the workload setup (a), the user first starts the long TPC-H Q2 query (b), and then issues the short, high-priority transaction on demand using different scheduling policies (c). In addition to individual run latency, we also show historical latency statistics (d).

4.1 Poster: Introduction

To prepare the audience for the necessary background, we will set up a poster on the side that introduces our target scheduling problem and the idea of PreemptDB. It will also give an overview of our demonstration scenario, including the workload description and hardware/software setup. The poster will also highlight key results from PreemptDB presented earlier.

4.2 Live: Demo Session

We have set up a backend server to simulate an online warehouse database based on a mix of TPC-C and TPC-H benchmarks. In this workload, the user issues short, high-priority New-Order transactions on demand as the system is running a long, low-priority Q2. The ideal scenario is to maintain high throughput or stable execution time of Q2, while New-Order transactions should be scheduled as soon as possible to ensure low end-to-end latency.

During the demo, we will run three instances of PreemptDB, configured with the preempt, yield and wait policies for high-priority transactions, respectively. These instances are hosted on our server with two 32-core Intel Xeon Gold 6448H CPUs and 512 GB DDR5 main memory, running Ubuntu 22.04.4 LTS.

To make it easy for audience to experience the demonstration scenario, we created a web-based UI (Figure 5) that connects to the server backend. The UI explains the workload (a) and allows the audience to issue different types of transactions under varying scheduling policies (b–d). The user starts by enabling the heavy, long-running Q2 query in Figure 5(b). This will quickly take up all the CPU resources, i.e., making the DBMS busy. In the middle part of the interface, the user can then choose to issue a short, high-priority transaction under the specified scheduling policy. To facilitate this, on the backend we start three different instances of PreemptDB, each using a different policy (wait, yield or preempt). The system then performs the transaction, and reports back the latency of the high-priority transaction. The user can repeat these operations and the UI further maintains cumulative statistics as shown in Figure 5(d). Throughout the demonstration, we will refer back to the poster setup earlier to explain the results obtained under different scheduling policies.

REFERENCES

[1] M. J. Carey, R. Jauhari, and M. Livny. 1989. Priority in DBMS resource scheduling. In *Proceedings of the 15th International Conference on Very Large Data Bases (VLDB '89)*. 397–410.

[2] K. Delaney and C. Freeman. 2013. *Microsoft SQL Server 2012 Internals*. Pearson Education. https://books.google.ca/books?id=wK1CAwAAQBAJ

[3] Kaisong Huang, Tianzheng Wang, Qingqing Zhou, and Qingzhong Meng. 2023. The Art of Latency Hiding in Modern Database Engines. *Proceedings of the VLDB Endowment* 17, 3 (2023), 577–590.

[4] Kaisong Huang, Jiatang Zhou, Zhuoyue Zhao, Dong Xie, and Tianzheng Wang. 2025. Low-Latency Transaction Scheduling via Userspace Interrupts: Why Wait or Yield When You Can Preempt? *Proc. ACM Manag. Data* 3, 6, Article 182 (2025).

[5] Guoliang Li and Chao Zhang. 2022. HTAP Databases: What is New and What is Next. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. 2483–2488.

[6] Sohil Mehta. 2021. User Interrupts – A faster way to signal. https://lpc.events/event/11/contributions/985/attachments/756/1417/User_Interrupts_LPC_2021.pdf

[7] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2017. Interleaving with coroutines: a practical approach for robust index joins. *Proc. VLDB Endow.* 11, 2 (Oct. 2017), 230–242.

[8] Haoze Song, Wenchao Zhou, Heming Cui, Xiang Peng, and Feifei Li. 2024. A survey on hybrid transactional and analytical processing. *The VLDB Journal* 33, 5 (2024), 1485–1515.