anagodb: Offering Massive Parallelism for Database Engine

Yuto Hayamizu
haya@tkl.iis.u-tokyo.ac.jp
The University of Tokyo
Meguro-ku, Tokyo, Japan

Ryoji Kawamichi
Tsuyoshi Ozawa
The University of Tokyo
Meguro-ku, Tokyo, Japan

Masaru Kitsuregawa
The University of Tokyo / Research Organization of Information and Systems
Meguro-ku, Tokyo, Japan

Kazuo Goda
The University of Tokyo
Meguro-ku, Tokyo, Japan

Abstract

Parallelism is a primary factor in exploiting the potential capability of modern computer systems having a number of processing cores and storage chips/spindles. The database community has exerted much effort to achieve good parallelism in query processing to shorten latency and/or improve throughput. One approach is *out-of-order database execution* (OoODE), which dynamically and recursively decomposes the query work into tasks; it divides a parent task into child tasks if the parent task is dividable, and eventually, it divides the query work into a massive number of tasks. Hence, this mechanism can exploit the parallelism of the underlying computer infrastructure, specifically leveraging the potential computational capacity and/or the potential IO bandwidth. In particular, OoODE offers significant speedups for selective analytical queries. This paper presents *anagodb*, a research prototype database engine that we have developed for exploring the implementation of OoODE. anagodb has the capability of processing a wide variety of analytical queries according to the OoODE mechanism. Our demonstration will present how anagodb runs analytical queries on a cloud server. The visual client tool reports the query progress and the hardware performance metrics at runtime, clarifying that anagodb exploits the potential hardware power effectively.

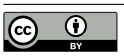
CCS Concepts

• **Information systems → Database management system engines**.

Keywords

Database engine; massively parallel execution

ACM Reference Format:
Yuto Hayamizu, Ryoji Kawamichi, Tsuyoshi Ozawa, Masaru Kitsuregawa, and Kazuo Goda. 2025. anagodb: Offering Massive Parallelism for Database Engine. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3722212.3725079



SIGMOD-Companion '25, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3725079

1 Introduction

Parallelism has long been an important focus for database engines to offer responsive query processing over large-scale database. Modern computer systems have been incorporating a larger number of processing cores and storage chips/spindles in a single system. In order to exploit their computational capacity and IO bandwidth more effectively and more efficiently, the endeavor for parallel query processing is still actively continuing [9, 16, 13, 3, 2].

Out-of-order database execution (OoODE) [6, 17] is a solution that we have developed in the view of this technology trend. OoODE allows the database engine to dynamically and recursively decompose the query work into tasks. It divides a parent task into child tasks if the parent task can be divided, and applies this decomposition to their descendants. Eventually, the OoODE mechanism can divide the query work into a massive number of tasks during the query execution, thus achieving the potential parallelism of the underlying computer infrastructure. Specifically speaking, the database engine is allowed to leverage the potential computational capacity and/or the potential IO bandwidth. In particular, OoODE potentially offers significant speedups for selective analytical queries, whereas existing solutions have only provided conservative parallelism for such queries [17].

OoODE offers massive parallelism for a wide variety of database queries. Yet, fully transforming its parallelism into query speedups is also technically challenging. OoODE potentially generates more than tens of thousands of concurrent tasks and IOs at runtime. Poor software implementation would yield non-negligible overheads for managing those tasks, thus failing to achieve possible speedups.

This paper briefly presents *anagodb*, a research prototype database engine that we have developed for exploring implementation aspects of OoODE [7]. Currently, anagodb has the capability of processing analytical queries according to the OoODE mechanism. This paper also presents our demonstration plan to show how efficiently anagodb runs analytical queries. We plan to execute different analytical queries on anagodb running on a cloud server, and present our visual client tool which reports the query progress and the hardware performance metrics at runtime, demonstrating that anagodb exploits the potential hardware power effectively.

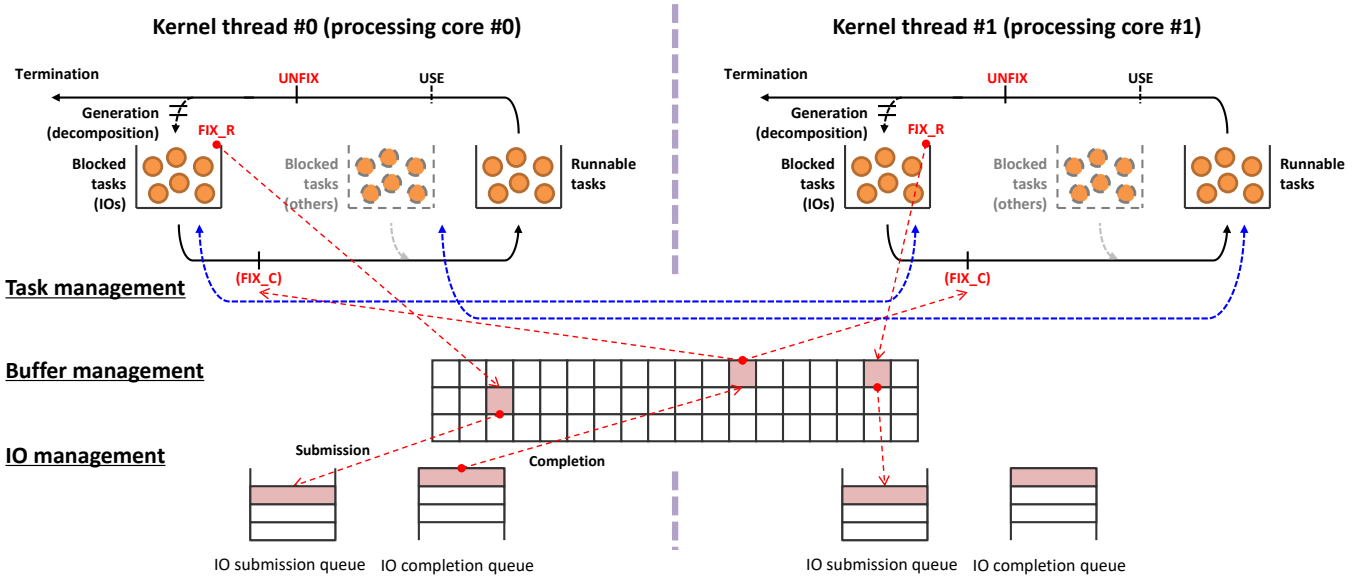


Figure 1: Task and IO management inside anagodb.

2 anagodb

anagodb is a research prototype database engine developed to explore the implementation aspects of OoODE. Similar to other relational database engines, anagodb supports the following capabilities:

- managing relations and their indices in the database organized in secondary storage (i.e., files or block devices);
- accepting relational queries and administration commands from clients via a proprietary protocol and the JDBC protocol;
- compiling a SQL-written relational query to obtain a tuned query execution plan;
- processing a query in any of different execution modes (e.g, serial execution and OoODE);
- loading/unloading data into/from the database in a bulky manner; and
- reorganizing the database to recover the structural efficiency.

Currently, anagodb does not support the transaction capabilities since it focuses on analytical database workloads. The anagodb database server can work on a single server running Linux and on a shared-nothing cluster of network-connected servers. For simplicity, this paper only assumes the single-server environment.

anagodb processes a relational query fully according to the OoODE mechanism. Upon accepting a new query, anagodb compiles the query to derive a tuned query execution plan and then starts executing the query by invoking a single task to trigger the first query operator. In executing each task, when anagodb finds an opportunity to decompose the existing job (assigned to each task) into multiple independent jobs, it invokes a separate task to execute each job. The decomposing opportunity can be defined for every database operator. For example, a *scan* operator (retrieving all the records from a relation and applying the selection filter) can be evaluated for different disjoint sets of records. A typical practice is to decompose the scan job for each extent (i.e., the fixed number of consecutive database pages); this strategy is often successful to intensively utilize the available IO bandwidth and the computational capacity. Another example is a B^+*tree search* operator (searching records matching the selection condition from a relation or an index structured in a B^+tree form). During B^+tree traversal process, the search work can be decomposed at each node (i.e., database page) into separate tasks for visiting child nodes that may contain matching records. Thus, an practice of invoking a new task for each child node works effectively to utilize the available IO bandwidth and computational resources. These practices can be easily extended to *hash join* (build and probe) operators and *nested-loop join* operators.

The fine-grained decomposition strategy noted above allows anagodb to execute a massive number of concurrent tasks and IOs. Their efficient execution is crucial to achieving the possible speedups. Figure 1 presents the task and IO management scheme that have been employed in anagodb; this scheme is specially tuned to enable anagodb to operate such a massive number of concurrent tasks and IOs so as to exploit the parallelism with smaller processing overheads.

Task management. anagodb assigns a separate kernel thread for each processing core and manages its tasks as coroutines [12, 4] in each thread [10, 14]. Usually, the coroutine runs inside its assigned kernel thread according to the *local binding* policy [8]; its child coroutines also run inside the same kernel thread. This policy works effectively to reduce costly context switches and inter-core communications.

Memory management. anagodb applies the local binding policy for memory management. When initialized, the kernel thread secures local memory pool from its NUMA node, and upon a memory allocation request from a coroutine, it allocates claimed memory space from the pool [15, 1]. This policy also helps to reduce costly context switches and inter-node memory accesses [11].

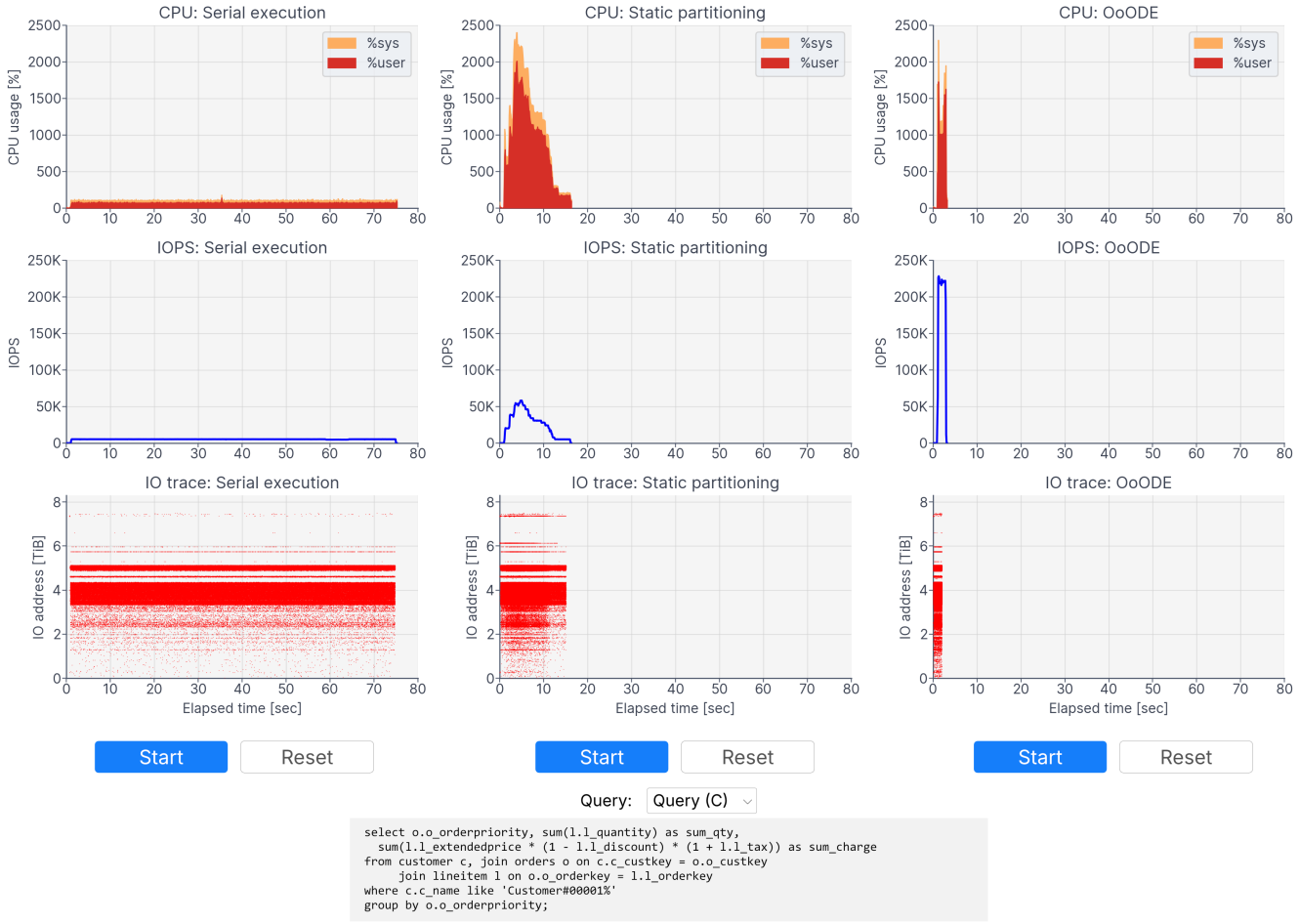


Figure 2: An example of the visual interface to be presented in the demonstration.

IO management. anagodb processes IOs in an asynchronous manner. When a coroutine needs to retrieve a database page from secondary storage to execute a database operation, it tries to fix a buffer page in the database buffer. If the page is already ready (i.e., fixed by preceding coroutines), the coroutine continues to run the operation on the buffer page. Otherwise, the coroutine is suspended and the buffer manager issues an asynchronous IO request to the IO manager [5]. Upon the IO completion, the IO manager transfers the IO content to the buffer page, and raises a signal to resume the suspended coroutine to execute the operation on the buffer page. Similarly, anagodb processes other events including network communication in an asynchronous manner.

Load balancing. Load balancing is also crucial in effectively utilizing all the available processing cores. anagodb invokes task migration between kernel threads conservatively. Specifically, anagodb periodically monitors the number of coroutines of all the kernel threads, and if it finds that a specific kernel thread is likely to have no more runnable coroutines, it triggers the task migration to move several tasks from other busy kernel threads. As the dynamic decomposition mechanism keeps anagodb to have a massive number of fine-grained tasks, this passive strategy effectively works to feed tasks to all the kernel threads throughout the query processing. Rather, this design allows each kernel thread to work with very few mutual exclusions by avoiding unnecessary task migrations.

Our experience is that this implementation strategy effectively utilizes the computational and IO resources in modern data-intensive server configurations, which may combine tens of processing cores and tens of magnetic disk or flash-memory devices.

3 Demonstration

The demonstration plans to execute analytical database queries with anagodb running on a cloud server and show the query progress and the hardware performance metrics at runtime on the visual interface, allowing the audience to experience how effectively anagodb exploits the potential capability of underlying hardware resources for the query processing.

3.1 anagodb database server

We have set up anagodb on an EC2 instance (i3en.6xlarge) on the AWS cloud. The database volume is organized from two NVMe

SSDs attached to the EC2 instance with `mdadm` in the striping mode, the TPC-H dataset (scale factor: 1,000) is loaded into relations, and separate indexes are built on attributes used in selection predicates or join conditions of the standard TPC-H queries. anagodb is configured so that it can execute a given query in one of different execution methods (e.g., serial execution and OoODE) and with related configuration parameters.

3.2 Visual client tool

We have developed a visual client tool to allow the user to interact with the database server. The visual client tool accepts a SQL-written query from the user, and issues the query to the database server. During the query execution, the visual client tool displays the query progress and hardware performance metrics at runtime.

Figure 2 presents an example of the visual interface. The bottom text box accepts an SQL-written query. The "Start" button sends the query to the database server to start executing the query processing in a designated execution scenario (indicated by each column). Three runtime graphs (in each column) shows the CPU utilization, the IO throughput, and the IO trace, respectively, which are lively updated as the query processing progresses.

3.3 Observations

The example case presented in Figure 2 compares three execution scenarios of an identical analytical query. Serial execution (in the left-most column) indicates a scenario in which the query is non-parallelized, being executed with a single thread. Static partitioning (in the center column) indicates that the query is pre-partitioned into 24 query parts [1]; these parts are executed with the same number of threads and then their results are merged. OoODE (in the right column) indicates the query is dynamically decomposed according to the OoODE mechanism in order to intensively parallelize its work.

While static partitioning is 4.89 times faster than serial execution, it suffers from load imbalances and fails to fully utilize the hardware capabilities. This is because the processing time for each partition varies widely, ranging from 0.2 seconds to 8.0 seconds. On the other hand, OoODE achieves further performance improvement, performing 36.2 times faster than serial execution and 7.40 times faster than static partitioning in this case. It is worth mentioning that OoODE exploits the available processing capacity at up to 95.6% and the available IO bandwidth at up to 92.2%.

4 Conclusion

anagodb is a research prototype database engine that fully implements OoODE. It has the capability of dynamically and recursively decomposing the query work into a number of tasks, thus achieving massive parallelism and offering significant speedups, in particular, for selective analytical queries. The live demonstration shows that how effectively anagodb exploits the potential hardware power.

As a demonstration paper, it only outlines the early implementation work of anagodb. We would like to deeply dive into its further technical details in a separate paper.

[1]We configured 24 query partitions since preliminary experiments showed this to be the optimal number of partitions for our environment.

Acknowledgments

This work has been supported in part by FIRST and ImPACT research funding programs (Cabinet Office, Japan), Cross-cutting Technology Development for IoT Promotion program (NEDO, Japan), Collaborative Research on Big Data Platform Architecture (KDDI and UTokyo-IIS), and Cross-ministerial Strategic Innovation Promotion Program on Integrated Health Care System (Cabinet Office, Japan).

References

[1] Joseph Antony, Pete P. Janes, and Alistair P. Rendell. 2006. Exploring thread and memory placement on NUMA architectures: solaris and linux, ultrasparc/fireplane and opteron/hypertransport. In *Proc. HiPC 2006* (LNCS). Vol. 4297, 338–352.

[2] Jianjun Chen et al. 2023. Krypton: real-time serving and analytical SQL engine at bytedance. *Proc. VLDB Endow.*, 16, 12, 3528–3542.

[3] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hetexchange: encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endow.*, 12, 5, 544–556.

[4] Ana Lúcia de Moura and Roberto Ierusalimschy. 2009. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.*, 31, 2, 6:1–6:31.

[5] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding modern storage apis: a systematic study of libaio, spdk, and io_uring. In *Proc. SYSTOR*, 120–127.

[6] Kazuo Goda, Yuto Hayamizu, Hiroyuki Yamada, and Masaru Kitsuregawa. 2020. Out-of-order execution of database queries. *Proc. VLDB Endow.*, 13, 12, 3489–3501.

[7] Yuto Hayamizu, Ryoji Kawamichi, Tsuyoshi Ozawa, and Kazuo Goda. 2020. anagodb: a research test-bed database engine by and for performance enthusiasts. Retrieved Jan. 25, 2025 from https://anagodb.io/.

[8] Genki Kimura, Yuto Hayamizu, Rage Uday Kiran, Masaru Kitsuregawa, and Kazuo Goda. 2023. Efficient parallel mining of high-utility itemsets on multicore processors. In *Proc. ICDE*, 638–652.

[9] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *Proc. SIGMOD*, 743–754.

[10] Bil Lewis and Daniel J. Berg. 1998. *Multithreaded programming with Pthreads*. Prentice-Hall.

[11] Zoltan Majó and Thomas R. Gross. 2011. Memory system performance in a NUMA multicore multiprocessor. In *Proc. SYSTOR*, 12.

[12] Chris D. Marlin. 1980. *Coroutines: A Programming Methodology, a Language Design and an Implementation. LNCS*. Vol. 95.

[13] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. 2017. Monotasks: architecting for performance clarity in data analytics frameworks. In *Proc. SOSP*, 184–200.

[14] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. 2008. *Operating System Concepts*. Wiley.

[15] Jaehyun Song, Minwoo Ahn, Gyusun Lee, Euiseong Seo, and Jinkyu Jeong. 2021. A performance-stable NUMA management scheme for linux-based HPC systems. *IEEE Access*, 9, 52987–53002.

[16] Li Wang, Minqi Zhou, Zhenjie Zhang, Yin Yang, Aoying Zhou, and Dina Bitton. 2016. Elastic pipelining in an in-memory database cluster. In *Proc. SIGMOD*, 1279–1294.

[17] Hiroyuki Yamada, Kazuo Goda, and Masaru Kitsuregawa. 2023. Nested loops revisited again. In *Proc. ICDE*, 3708–3717.