Out-of-order Execution of Database Queries

Kazuo Goda† Yuto Hayamizu† Hiroyuki Yamada† Masaru Kitsuregawa†,‡
† The University of Tokyo, Japan ‡ National Institute of Informatics, Japan
{kgoda,haya,hiroyuki,kitsure}@tkl.iis.u-tokyo.ac.jp

ABSTRACT

Intra-query parallelism is a key for database software to offer acceptable responsiveness for data-intensive queries. Many researchers have studied how to achieve greater execution parallelism for database queries. *Partitioning* is a representative approach, which divides a query into multiple subtasks and executes them in parallel. However, given a new query, optimal division is not necessarily obvious. Database software utilizes heuristic rules or statistical information to decide how to divide the query before execution. As yet another approach to achieve execution parallelism, this paper presents *out-of-order database execution* (OoODE), a massively-parallel query execution method to offer significant speedup for database queries consistently. OoODE dynamically decomposes query work by making the best use of the exact knowledge of the potential execution parallelism for each operation ready to be performed during query execution. With OoODE, the database software is allowed to automatically squeeze out the execution parallelism that the query inherently holds. Hence, for a wide spectrum of queries, OoODE performs significantly faster than the serial (non-parallelized) execution, while it performs better than or comparably with alternative parallelizing methods without the need for dividing the query before execution. This paper presents the experiments that we conducted using the prototyped database software and demonstrates that OoODE is two to three orders of magnitude faster than the serial execution, whereas it is substantially (up to 2.07 times) faster than the best achievable case of partitioning. Besides, OoODE performs two to four orders of magnitude faster than major DBMSs.

PVLDB Reference Format:
Kazuo Goda, Yuto Hayamizu, Hiroyuki Yamada and Masaru Kitsuregawa. Out-of-order Execution of Database Queries. *PVLDB*, 13(12): 3489-3501, 2020.
DOI: https://doi.org/10.14778/3415478.3415571

1. INTRODUCTION

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415571

Intra-query execution parallelism is a key for database software to answer data-intensive queries within an acceptable time. So far, many researchers have studied how to achieve greater execution parallelism for database queries. A mainstream approach is *partitioning* [4,7,13,18,19,25,39,49], which divides a single query into multiple subtasks and then executes the subtasks simultaneously, the results of which are then merged. This solution is widely deployed in major DBMSs [5,20,60].

Partitioning is technically challenging. Optimal partitioning is nontrivial. The performance of partitioning varies significantly depending on how to divide a query and it degrades if a non-optimal number of partitions is chosen. However, no known algorithm can achieve optimal partitioning.

In this paper, as yet another approach to achieve execution parallelism for database queries, we present *out-of-order database execution* (OoODE), a massively-parallel execution method for dynamically decomposing query work by making the best use of the exact knowledge of the potential execution parallelism for each operation ready to be performed during query execution; database software is enabled to spawn a new fine-grained task in charge of a disjoint part of the query work every time it is determined to be possible to execute the task independently of the others. Eventually, the query is divided into a massive number of tasks to be executed in parallel such that the database software can automatically squeeze out the execution parallelism that the query inherently holds. Hence, OoODE significantly increases the IO throughput and speeds up the query execution by fully exploiting the potential parallelism if the underlying infrastructure is sufficiently powerful.

The contribution of this paper is summarized as follows.

- This paper describes OoODE that can automatically squeeze out the execution parallelism that the query inherently holds by dynamically decomposing relational query work with the exact knowledge of the potential execution parallelism at runtime.
- This paper presents that, for a wide spectrum of queries, OoODE performs significantly faster than the serial execution, and it performs better than or comparably with alternative parallelizing methods without the need for dividing the query before submission or at compile time.
- This paper reports the performance experiments to demonstrate that OoODE is two to three orders of magnitude faster than the serial execution, whereas it is substantially (up to 2.07 times) faster even than the

Algorithm 1 Execute a database operation $f(x, y)$ in the serial execution method.

```
procedure EXECUTE_SERIAL($f(x, y)$)
    $R \leftarrow \emptyset$
    if $f$ is unary then                                   ▷ $x = \emptyset$
        while $s \leftarrow$ FETCH($y$) do               ▷ Continue if $s \neq \emptyset$
            $r \leftarrow$ Apply $f$ to $s$
            $R \leftarrow R \cup r$
    else                                                   ▷ $f$ is binary
        while $s \leftarrow$ FETCH($x$) do               ▷ Continue if $s \neq \emptyset$
            $r \leftarrow$ EXECUTE_SERIAL($f_s(\emptyset, y)$)
            $R \leftarrow R \cup r$
    return $R$
procedure FETCH($\alpha$)
    if $\alpha$ is a relation stored in the database then
        if no unfetched tuple remains in $\alpha$ then
            $\beta \leftarrow \emptyset$
        else
            $\beta \leftarrow$ Fetch one or more tuples from $\alpha$
    else                              ▷ $\alpha$ is another operation ($g(v, w)$)
        $\beta \leftarrow$ EXECUTE_SERIAL($g(v, w)$)
    return $\beta$
```

best achievable case of partitioning. Besides, OoODE performs two to four orders of magnitude faster than major DBMSs.

The remainder of this paper is organized as follows. Section 2 presents the motivation and vision for OoODE, and Section 3 introduces the OoODE algorithm for database queries, presents the performance advantage and explains our implementation. Section 4 reports our performance experiments that clarify the performance advantage. Section 5 reviews related work and finally Section 6 concludes the paper.

2. MOTIVATION AND VISION

The design principles employed for executing database queries can be roughly divided into two groups, fetch-all and fetch-necessary, in terms of IOs to be exchanged between processors and database storage.

The fetch-all principle scans the entire relation regardless of whether each tuple in the relation is necessary or not for a given query. A typical example is a hash join, which usually performs a linear scan for both relations to be joined. The necessity is not taken into consideration for each tuple. Thus, excessive IOs may be produced; however, IOs to be requested are trivial since one can know that the whole relations will be scanned once a query arrives. Existing query parallelization techniques (e.g. parallel hash joins) have utilized this nature to algorithmically divide a known linear scan into a set of multiple scans to achieve intensive parallelism. This solution has been incorporated into modern DBMSs and has formed a backbone of recent parallel programming frameworks such as MapReduce [16].

The other group is fetch-necessary, which fetches only tuples being necessary for a query. A typical example is a nested-loop join, which picks up only necessary tuples from relations to be joined. The necessity decision is made by the combination of query logic (e.g. a join condition predicate) and a data structure (e.g. a B+tree index) for each tuple. Accordingly, the total IO amount is automatically

Algorithm 2 Execute a database operation $f(x, y)$ in the OoODE (out-of-order database execution) method.

```
procedure EXECUTE_OoODE($f(x, y)$)
    $R \leftarrow \emptyset$
    if $f$ is unary then                                   ▷ $x = \emptyset$
        if $y$ is a relation stored in the database then
            while $\hat{y} \leftarrow$ Assign from $y$ do        ▷ If $\hat{y} \neq \emptyset$
                $t \leftarrow$ Generate a new task
                TASK($\hat{y}$, $f$, $\emptyset$; $R$) on $t$
        else
            TASK($y$, $f$, $\emptyset$; $R$) on the self task
    else                                                   ▷ $f$ is binary
        if $y$ is a relation stored in the database then
            while $\hat{x} \leftarrow$ Assign from $x$ do        ▷ If $\hat{x} \neq \emptyset$
                $t \leftarrow$ Generate a new task
                TASK($\hat{x}$, $f$, $y$; $R$) on $t$
        else
            TASK($x$, $f$, $y$; $R$) on the self task
    Wait until all the children tasks complete
    return $R$
procedure TASK($\chi$, $\phi$, $\upsilon$; $\rho$)
    if $\phi$ is unary then
        while $s \leftarrow$ FETCH($\chi$) do                  ▷ If $s \neq \emptyset$
            $r \leftarrow$ Apply $\phi$ to $s$.
            $\rho \leftarrow \rho \cup r$            ▷ $\rho$ is shared between tasks
    else                                                   ▷ $\phi$ is binary
        while $s \leftarrow$ FETCH($\chi$) do                  ▷ If $s \neq \emptyset$
            $r \leftarrow$ EXECUTE_OoODE($\phi_s(\emptyset, \upsilon)$)
            $\rho \leftarrow \rho \cup r$            ▷ $\rho$ is shared between tasks
```

FETCH() is the same as in Algorithm 1 except it calls EXECUTE_OoODE() instead of EXECUTE_SERIAL() (line 19.)

minimized. However, due to the complexity of the decision process, IOs are unknown in advance of the query execution. No known algorithms provide the best query division to achieve the potential parallelism for a given query. In practice, query parallelization techniques (e.g. query partitioning) involve deliberate case-specific tuning work that requires the expert knowledge of query logic and data structures. Despite the efforts, the tuning may not be successful to achieve the best performance.

OoODE (out-of-order database execution) is a simplified and unified execution formula to offer the best-in-practice IO parallelism for a wide spectrum of database queries consistently, where "best-in-practice" means better than or comparable with existing query execution techniques in the same physical environments and for the same data sets. A primitive step of OoODE is to divide a query task every time the task is dividable [35]. Its recursive iteration eventually converts a given query into the execution code that exploits the execution parallelism that the query inherently holds. The execution order of the divided tasks is non-deterministic. We have come to use the term *out-of-order execution* after the processor technology of the same name [34]. Existing parallelization techniques have involved intensive specific efforts such as algorithmic refinements and expert tuning. OoODE removes this complication and offers a universal solution for both fetch-all and fetch-necessary to achieve the best-in-practice parallelism and to fully exploit the IO bandwidth.

We are witnessing that an increasing amount of data is being accommodated into database, and business and science applications are making the intensive use of the data. In our experience, analysts often start a knowledge discovery job by initiating a linear-scan query to grasp a broad overview of the available data, and then they deep-dive into the data by iteratively and interactively invoking an ad-hoc focused query until reaching a conclusion. As decision-making speed matters, parallelization throughout the entire process is a prime concern. Despite the demand, modern DBMSs and parallel programming frameworks are wholly devoted to fetch-all.

OoODE is introduced to offer the best-in-practice parallelism for database queries. In comparison with existing parallelization techniques, OoODE potentially performs comparably fast for fetch-all, while it achieves significant performance improvement for fetch-necessary, for which the IO complexity has hindered intensive query parallelization. Later sections explain OoODE with a focus on the fetch-necessary query execution even though OoODE works for fetch-all.

3. OUT-OF-ORDER EXECUTION OF DATABASE QUERIES

3.1 Out-of-order database execution (OoODE)

This section introduces an algorithm of OoODE, which dynamically decomposes a relational query during execution by spawning a task for a database operation every time it is possible to execute the task independently of the others.

Let $f(x, y)$ be a database operation, where f is a relational database operator such as σ (selection) and $\bowtie$ (join) and x and y are inputs to the operator. x and y may be relations (sets of tuples) stored in the database or outputs of other database operations. We presume that, if f is unary, x is invalid ($\emptyset$) and only y is valid; otherwise, both are valid. For example, $\sigma(\emptyset, x)$ is a single-relation selection query and $\bowtie(\bowtie(x, y), z)$ is a three-way join query.

As a preparation prior to the introduction of the OoODE algorithm, we firstly present a conventional algorithm, in Algorithm 1, for executing a database operation $f(x, y)$ based on the serial execution method [21, 61], which is basic and straightforward, but still popular for many database software. For simplicity, we only consider additive database operators[1] and omit some implementation techniques such as predicate pushdown [31, 55]. For a unary database operator f, the database software iteratively fetches one or more tuples from y and applies f until evaluating all the tuples of y (lines 4–6). In fetching tuples, if y is a relation stored in the database, the database software issues an IO command to a storage system storing the database to fetch tuples (lines 13–17). In contrast, if y is an output of another database operation, the database software recursively calls the operation and fetches its result in the same manner (lines 18–19). For a binary operator f, the database software performs the same steps in a nested-loop fashion (lines 8–10); it iteratively fetches tuples from x (in an outer loop) and, for each of the tuples fetched from x, further iteratively fetches tuples from y and apply f (in an inner loop). Note that the notation $f_s(\emptyset, y)$ represents a curried function of $f(s, y)$ (line 9); when the outer loop recursively calls $f_s(\emptyset, y)$, the inner loop recognizes f_s to be unary (line 3). According to the serial execution method, every time the database software needs to fetch tuples from the database, it requests the tuple fetch, waits until the fetch completes and then applies the operator. It is difficult to fully exploit the potential parallelism of a given query.

[1]The algorithms can be easily extended for a query containing a non-additive database operator such as aggregation.

CUSTOMER

CID	*NAME*	*CITY*
101	AAA	NEW YORK
102	BBB	TOKYO
103	CCC	LONDON
104	DDD	TOKYO
105	EEE	BEIJIN
106	FFF	NEW ORK
107	GGG	TOKYO
108	HHH	PARIS

ITEM

IID	*PRICE*
1001	$31.22
1002	$18.15
1003	$1.25
1004	$102.98
1005	$5.96
1006	$6.71
1007	$9.90

ORDER

OID	*CID*	*DATE*
1	104	2017-12-01
---	---	---
4	102	2017-12-14
---	---	---
8	107	2017-12-18
9	104	2017-12-18
---	---	---
13	104	2017-12-19
---	---	---
17	102	2017-12-21
---	---	---
24	104	2018-01-03
---	---	---

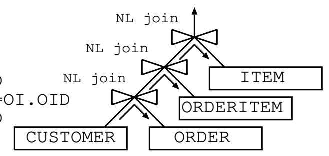
ORDERITEM

OID	*O#*	*IID*
1	1	1007
1	2	1005
---	---	---
4	1	1003
---	---	---
8	1	1002
9	1	1002
9	2	1003
9	3	1001
---	---	---
13	1	1001
---	---	---
17	1	1006
17	2	1002
---	---	---
24	1	1007

Figure 1: Example database.

```
SELECT
 C.CID, I.NAME, I.PRICE
FROM CUSTOMER C
 JOIN ORDER O ON C.CID=O.CID
 JOIN ORDERITEM OI ON O.OID=OI.OID
 JOIN ITEM I ON OI.IID=I.IID
WHERE C.CITY='TOKYO'
```

(a) Query



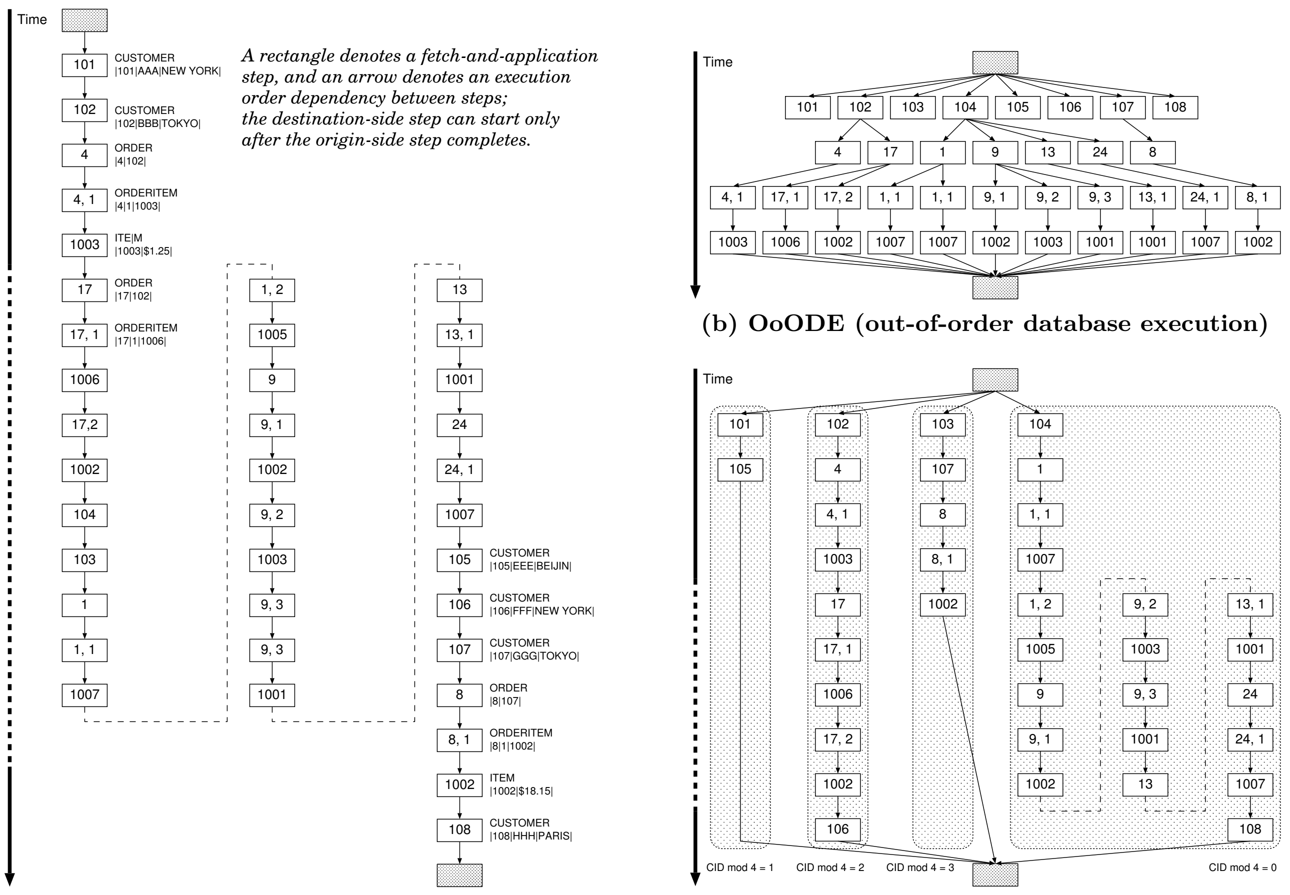
(b) Execution plan

Figure 2: Example query and execution plan.

Next, we introduce the OoODE algorithm in Algorithm 2. The point of difference is that OoODE enables database software to dynamically decompose query execution into many tasks at runtime and execute them in parallel [34,35]. Every time the database software needs to fetch new tuples from a relation stored in the database, it invokes a new task for each part (one or more tuples) of the relation and delegates its concerned work to the task; the new task fetches tuples from the assigned part and then executes the database operation (lines 5–7, 12–14). The decision steps (lines 5,12) internally check the necessity of tuple fetch; only if the decision is affirmative, OoODE invokes a new task and assigns a part of the unfetched (not yet fetched) tuples. This decision is helped by the runtime data structure information. The detail is described in Section 3.4. In contrast to the serial execution, OoODE is not blocked by tuple fetch. Instead, by making the best use of the exact knowledge regarding the necessity of tuple fetch for each database operator, OoODE dynamically invokes a new task, in which another tuple fetch is executed in parallel. Eventually, OoODE decomposes the query work into a massive number of parallel tasks so that OoODE can achieve the potential IO parallelism that the query inherently holds.

3.2 Examples of query execution

Before entering into details on OoODE, this section presents example query execution to demonstrate performance superiority of OoODE over the serial execution and the next (Section 3.3) extends the analysis to comparison with an


Time
A rectangle denotes a fetch-and-application step, and an arrow denotes an execution order dependency between steps; the destination-side step can start only after the origin-side step completes.
CUSTOMER |101|AAA|NEW YORK|
CUSTOMER |102|BBB|TOKYO|
ORDER |4|102|
ORDERITEM |4|1|1003|
ITEM |1003|$1.25|
ORDER |17|102|
ORDERITEM |17|1|1006|
CUSTOMER |105|EEE|BEIJIN|
CUSTOMER |106|FFF|NEW YORK|
CUSTOMER |107|GGG|TOKYO|
ORDER |8|107|
ORDERITEM |8|1|1002|
ITEM |1002|$18.15|
CUSTOMER |108|HHH|PARIS|
CID mod 4 = 1
CID mod 4 = 2
CID mod 4 = 3
CID mod 4 = 0

(a) Serial execution

(b) OoODE (out-of-order database execution)

(c) Partitioning (four subtasks)

Figure 3: Execution sequences of fetch-and-application steps.

alternative parallelizing technique.

Assume a dataset, as illustrated in Figure 1, which mimics sales record database, where four relations (`CUSTOMER`, `ITEM`, `ORDER` and `ORDERITEM`) respectively store customer profiles, product items, orders and order details. Let us suppose that database software executes a query, illustrated in Figure 2, for listing customers who lived in Tokyo and names and prices of the items which they purchased in accordance with the nested-loop join plan.

Figure 3(a) illustrates a logical execution sequence for a case where the query is executed in Algorithm 1 (the serial execution method.) Database software iterates tuple fetch and operator application by traveling from the outer-most relation (`CUSTOMER`) into the inner-most relation (`ITEM`) in the depth-first manner according to the execution plan. The tuple fetch and the operator application are performed step by step. The query execution is a sequence of 38 fetch-and-application steps, which are serially executed; only after a step completes, its next starts. The total execution time is the accumulation of response time of all the steps.

By contrast, Figure 3(b) illustrates an execution sequence that Algorithm 2 (OoODE) provides for the same query. Note that the query work decomposition is assumed to be done at a single tuple level. The query execution is composed of the same 38 fetch-and-application steps. However, OoODE allows logically independent steps to be executed in parallel. Figure 3(b) indicates that, out of the 38 fetch-and-application steps, many steps can be executed in parallel. Assuming that the underlying infrastructure is powerful enough to support this execution parallelism, the execution time can be significantly shortened because latencies of the steps to be executed in parallel are not serially accumulated.

The recent hardware is accommodating more processor cores and storage devices. In view of the entry to mid-range data-intensive market, recent Linux servers normally have tens of processing cores, each core being capable of running tens to hundreds of kernel-managed threads [8,36], whereas many recent storage systems such as disk arrays and flash arrays have tens to hundreds of storage devices, each device being capable of servicing more than tens of concurrent IO commands. In this way, today's hardware technology holds a great potential capability to support high execution parallelism. With the assistance of such hardware technology, OoODE appears to speed up the query execution significantly and automatically up to the limitation determined by the query execution parallelism that the query inherently holds and the hardware execution parallelism that the underlying infrastructure supports.

3.3 Comparison of OoODE and partitioning

Partitioning is a representative technique for parallelizing query execution [4, 7, 13, 18, 19, 25, 39, 49]. It divides a given query into multiple subtasks at compile time and then executes them in parallel; the results of the subtasks are fi-

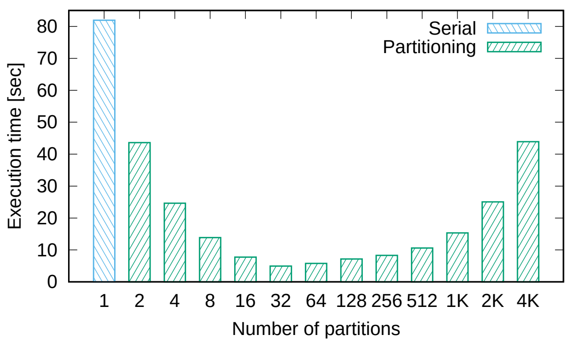


Figure 4: Execution time of partitioning varies significantly with different numbers of partitions. *Query A on the moderately skewed dataset in the small flash machine is plotted.*

nally merged. Each subtask is often assigned to a part of the database that has a certain disjoint property. Many DBMSs have employed this approach [5, 20, 60].

When applying partitioning to a query, one needs to decide how to divide the query. However, optimal query division is nontrivial. The potential execution parallelism of a database operator depends on the input to the operator. The input is unknown when the query execution starts. Without any knowledge of the input, one cannot know the potential execution parallelism of the query. Actually, partitioning achieves some execution parallelism by using heuristic rules or statistical information, but it is difficult to guarantee to leverage the potential parallelism at full.

Let us see the example query again. A popular partitioning practice is to virtually append a selection predicate (or an equivalent operator) on an outer relation at compile time. For example, an additional predicate, `C_CID mod 4 = i (i = 0, ..., 3)`, divides the example query into four subtasks. As illustrated in Figure 3(c), this partitioning is too moderate. Each subtask still contains several logically independent fetch-and-application steps, each of which can be potentially executed in parallel. Worse, the workloads are not well balanced among subtasks. Executing these subtasks in parallel seems to take shorter time than executing the original query, but takes longer than OoODE. Suppose another case where the query is divided into too many partitions (not illustrated); processing overheads seem to be rather significant. Figure 4 presents our experimental result, which demonstrates that the query execution time widely varies with different partitioning numbers[2]. It is difficult to figure out the optimal partitioning number before actually executing a query.

By contrast, OoODE is capable of dynamically decomposing query work into independent tasks every time a new independent task can be generated. Hence, OoODE can fully leverage the parallelization opportunity that the query inherently holds, performing better than or comparably with partitioning. Actually, as Figures 6 to 10 later present, our experiment confirmed that OoODE performed two to three orders of magnitude faster than the serial execution and substantially faster even than the best achievable case of partitioning without the need for dividing the query beforehand.

[2]We will report a further study on the partitioning technique in a separate paper.

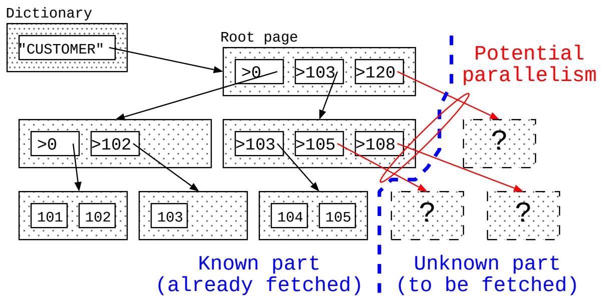


Figure 5: Data structure offers exact knowledge of potential parallelism. *Red references (known but unfollowed) can be followed in parallel.*

3.4 Exploiting data structure knowledge

As we have presented in Section 3.1, OoODE decides the task invocation based on the knowledge regarding the necessity of tuple fetch before actually performing the tuple fetch. This section enters into details on how OoODE makes the decision.

Suppose scanning a single relation `CUSTOMER` (in Figure 1) by iteratively calling a tuple fetch routine. Now you have just finished fetching the first five tuples (`|101|` to `|105|`). Obviously, one cannot know whether this relation holds any more *unfetched* tuples (`|106|` and more) before actually trying another tuple fetch. The necessity of tuple fetch is unknown. Fortunately, a relation is usually organized in a certain data structure. The database storage typically contains a dictionary that maps a relation name (e.g., `CUSTOMER`) and a reference to a data structure (e.g., a tree form illustrated in Figure 5), where tuples of the relation are packed into pages and the pages contain inter-page references. When trying to fetch a tuple from the database, the database software first checks the dictionary to identify a reference to a data structure, and then reads pages by following the reference. In this process, the database software incrementally acquires the knowledge of how tuples are stored in the data structure. For example, Figure 5 illustrates the knowledge that the database software holds just after fetching the `|105|` tuple. Red references (originating `|>105|`, `|>108|` and `|>120|`) indicate that the relation stores more tuples that have not been fetched yet. Trying to fetch tuples by following these references simultaneously is beneficial to achieve execution parallelism. Every time discovering new references stored in the known part of the data structure, OoODE *unfolds* the references during execution so that an individual task can be invoked for each reference.

Interestingly, inter-page references are often chained and nested in a tree form in many data structures. The idea of OoODE can be applied to unfold such references. Algorithm 2 only describes tuple-level parallelism for simplicity, but, by generalizing a tuple to any entry contained in a data structure (e.g., a leaf entry and an internal-node entry in a B+tree), OoODE is allowed to invoke a respective task for each of chained and nested references. Hence, OoODE fully exploits the potential IO parallelism that is inherent to the data structure.

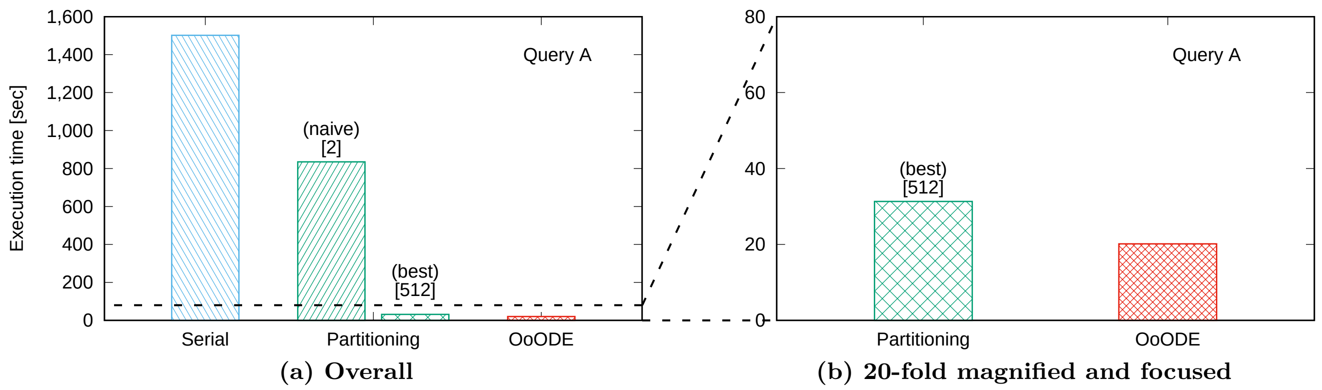


Figure 6: (a) OoODE is 75 times faster than Serial for Query A and (b) OoODE is 55% faster even than the best achievable case of Partitioning. *Execution time of Query A on the moderately skewed dataset in the small disk machine is plotted. A number embraced in square brackets denotes the number of partitions. Table 1 summarizes the speedup ratios.*

Besides, a data structure in the database may have a reference to another data structure. Typically, a secondary index entry contains a reference (e.g. a primary key or a physical record identifier) to a relation tuple. OoODE works to unfold such inter-structure references at runtime. Thus, OoODE exploits the potential IO parallelism among different data structures.

In this way, OoODE dynamically unfolds unfollowed references identified in the known part of the data structure to invoke a respective task for each reference during query execution. Therefore, OoODE squeezes out the IO parallelism inherent to data structures to speed up the query execution.

3.5 Prototype implementation

We have implemented OoODE in our database software prototype to investigate the performance characteristics experimentally. The prototype can be roughly divided into two components, a query processor and a storage engine.

The query processor executes a given query in the OoODE method. OoODE invokes a massive number of parallel tasks; the query processor needs to manage an execution state of each task. There are a variety of implementation options, but after an intensive exploration, we have finally come to utilize two threading mechanisms together; a single kernel thread (e.g., pthread in Linux) is assigned to each processor core[3] and a number of user threads are assigned to each kernel thread [38,54]. The execution state of an individual task is managed in a user thread associated to the task. In issuing a fetch command, the query processor tags an identifier of the concerned user thread. When the fetch completion is signaled, the query processor checks the identifier and resumes the user thread to perform the associated database operation with the fetched data. The combination of the two threading mechanism works effectively in today's multicore architecture. First, kernel threads are mandatory for spreading codes among multiple processor cores, but interthread context switching is costly because every switch imposes the kernel interaction. Binding only a single kernel thread to each processor core and placing many user threads in each kernel thread is beneficial to utilize all the available cores while reducing the context switching overhead. Second, recent processors employ the NUMA architecture, where remote memory access is costly [41]. The combination strategy allows the database software to explicitly store an execution state in local memory of each processor core, helping minimize remote memory accesses. In our experience, this implementation successfully runs tens of thousands of concurrent tasks.

[3]Assigning multiple kernel threads to each core may be effective when hardware multithreading [43,57] is enabled.

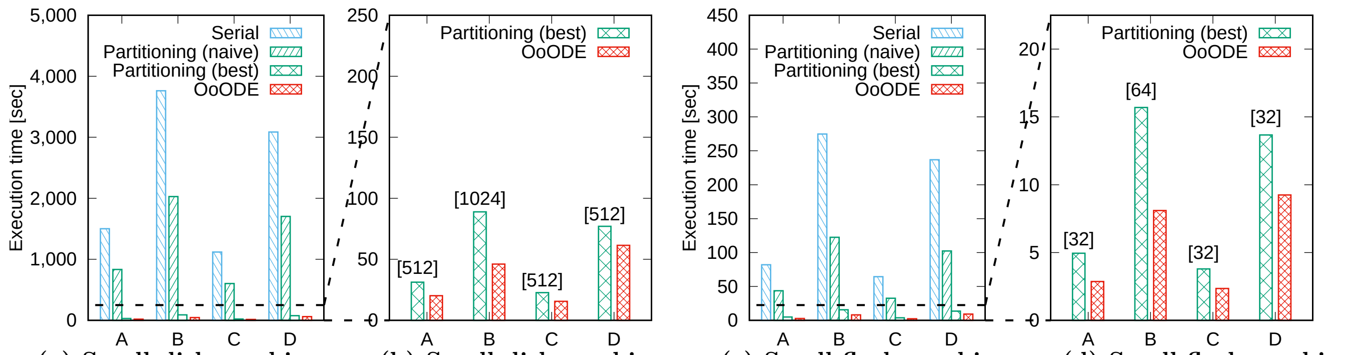
We have discussed the tuple/entry-level query decomposition so far. Our prototype has basically employed this strategy to fully utilize the parallelization opportunity. However, a certain database operator (e.g., a linear scan operator) tries to fetch multiple tuples stored in an identical page in a bulky manner. For such operations, it doesn't look a good solution to invoke an independent task for each of the tuples. We have tuned the prototype to be able to fetch multiple tuples stored in the same page at once for such operators to avoid unnecessary overheads.

The storage engine manages IOs from/to a storage system storing the database. In order to operate many concurrent IOs efficiently, we have chosen to utilize asynchronous IO mechanism (e.g., libaio in Linux); the storage engine issues an asynchronous IO request and, upon the completion, notifies back to the query processor to resume the concerned task with the fetched data. The storage format has been made compatible with InnoDB storage engine [46] such that the prototype can mount an InnoDB volume generated by MySQL [46]. This allows us to directly compare the prototype with MySQL (presented in Section 4.3) while helping us to focus on the query execution part.

Note that, for the purpose of experimental comparison presented in Section 3, we have additionally implemented alternative execution methods, the serial execution method (Algorithm 1) and the partitioning technique (dividing query work before execution.)

4. EXPERIMENTAL STUDY

This section presents an introductory part of the experiments that we intensively and extensively conducted to clarify performance benefits of OoODE (out-of-order database execution) in comparison with alternative execution methods by using our prototype database software and other DBMSs. Throughout the study, we utilized revised datasets



(a) Small disk machine (overall) **(b) Small disk machine (20-fold magnified)** **(c) Small flash machine (overall)** **(d) Small flash machine (20-fold magnified)**

Figure 7: (a) For the small disk machine, OoODE achieves two orders of magnitude speedup over Serial and Partitioning (naive) for Queries A, B, C and D, (b) OoODE is 25% to 93% faster even than the best achievable case of Partitioning, (c) and (d) OoODE achieves similar speedup for the small flash machine. *Execution time of Queries A, B, C and D on the moderately skewed dataset in the small disk and small flash machines is plotted. Table 1 summarizes the speedup ratios.*

and simplified queries of TPC-H [56], the de facto standard decision-support database benchmark.

4.1 Query speedup in small machines

First of all, we present a comparative performance study of multiple execution methods. We tested a decision-support query searching a single relation (Query A) on the prototype database software in a small disk-based server having twenty-four internal disk drives (hereinafter called "small disk machine.") The database was organized with the regular TPC-H schema in a raw storage volume made of the twenty-four disk drives. A test dataset was generated at a scale factor of 400. Note that the original dataset generation in TPC-H assumes uniform distribution, but in reality, data distribution is mostly skewed. In order to follow this property, we revised the regular dataset generator `dbgen` to be able to generate a dataset having moderate skewness (moderately skewed); specifically, 80% of the orders were made associated to 20% of the customers and vice versa, and the similar skewed association was applied between the order items and the sales parts. Indices were built for attributes appearing in condition predicates of the test query. Appendix summarizes the supplementary information on the machine configuration and the query definition.

As a metric of execution performance, query execution time was measured for each of the following execution methods. First, **Serial** executes a query based on the serial execution (non-parallelized) method. Second, **Partitioning** executes a query based on the serial execution method with the partitioning technique as described in Section 3.3 (dividing the query into subtasks and then executing them in parallel.) We tested twelve partitioning options; dividing the query into two to 4,096 partitions. The execution time varied according to the partitioning options as is presented in Figure 4. The paper reports a naive case, **(naive)** dividing the query into merely two partitions, and the best case, **(best)** yielding the shortest execution time. Third, **OoODE** executes a query in the OoODE method. In all the execution methods, the query was executed according to the same execution plan (e.g. an index-based search for Query A.) We performed every measurement strictly in a

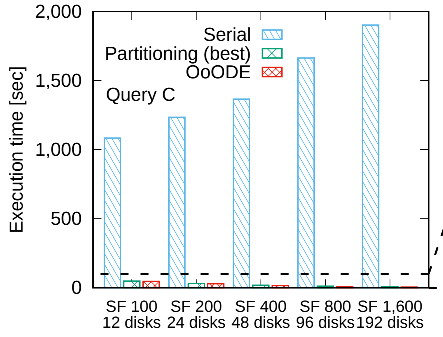
Table 1: Speedup by OoODE. *Summary of Figures 6–7.*

Query	A	B	C	D
Small disk machine				
over **Serial**	74.5	81.7	71.6	50.2
over **Partitioning (best)**	1.55	1.93	1.45	1.25
Small flash machine				
over **Serial**	28.7	33.9	27.4	25.6
over **Partitioning (best)**	1.73	1.94	1.62	1.48

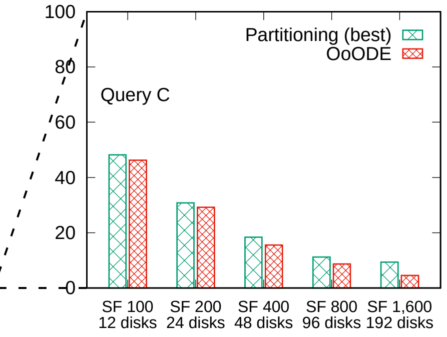
cold state in order to eliminate caching effects. The paper will report an average value of five trials.

Figure 6 shows that **OoODE** performed 74.5 times faster than **Serial** and 55% faster even than the best achievable case of **Partitioning**. The left-hand graph (a) summarizes the query execution time; the four bars indicate **Serial**, **Partitioning (naive)**, **Partitioning (best)** and **OoODE** respectively. For reference, a number embraced in square brackets denotes the number of partitions that we chose for **Partitioning**. **Serial** took approximately 1,500 seconds for executing the query, and **Partitioning (naive)** reduced the execution time to 835 seconds, which is slightly longer than half of the execution time of **Serial**. In contrast, **Partitioning (best)** and **OoODE** reduced it by two orders of magnitude; these two bars are too short to visually distinguish. Thus, we present the right-hand graph (b), which focuses on **Partitioning (best)** and **OoODE** with 20-fold magnification along the vertical axis. **Partitioning (best)** shortened the execution time to 31.3 seconds, whereas **OoODE** further reduced it down to 20.2 seconds (74.5 times faster than **Serial** and 55% faster than **Partitioning (best)**.)

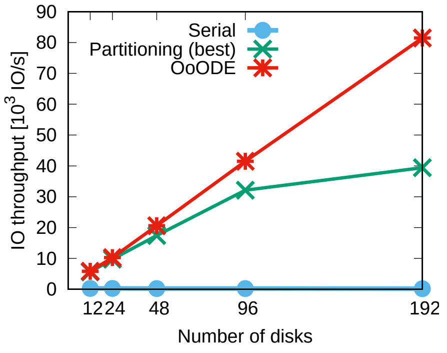
Next, we stretched the study to other decision-support queries, Queries B (joining two relations), C (joining three) and D (joining six), at the same test configuration. Figure 7 presents that **OoODE** consistently achieved significant speedup for all the queries. The two graphs (a) and (b) on the left summarize the execution time of the four queries, presenting an overall chart and its 20-fold magnified chart similarly to the previous figure. For convenience, we also present Table 1 to summarize the speedup ratios. **OoODE** performed two orders of magnitude (50.2 to 81.7



(a) Execution time (overall)



(b) Execution time (20-fold magnified)



(c) IO throughput

Figure 8: (a) OoODE is up to three orders of magnitude faster than Serial for Query C for the large disk machine, (b) OoODE achieves up to 107% speedup even over the best achievable case of Partitioning and (c) OoODE achieves linear increase in IO throughput. *Execution time and IO throughput of Query C on the moderately skewed dataset at different scale configurations in the large disk machine are plotted. Table 2 summarizes the speedup ratios.*

times) faster than **Serial** and substantially (25% to 93%) faster even than **Partitioning (best)**.

Furthermore, we performed the same test in another machine, a small flash-based server having eight internal flash drives ("small flash machine.") The two graphs (c) and (d) on the right at Figure 7 summarize the execution time and Table 1 again summarizes the speedup ratios. Even in the flash machine, the performance trend was similar to the observation that we obtained in the disk machine. **OoODE** speeded up the queries by two orders of magnitude (25.6 to 33.9 times) over **Serial** and substantially (48% to 94%) even over **Partitioning (best)**.

These experimental results verify that **OoODE** achieves significant speedup over the other execution methods and it consistently works for all the test queries both in the disk and flash machines.

4.2 Performance scalability in large machines

We then present a performance scalability study that we conducted by extending the previous experiment to a large disk-based system having 160 disk drives ("large disk machine") at different configurations of storage scales and dataset scales. Specifically, we tested five different scale configurations by changing the storage scales (the numbers of disks to be utilized) from 16 disks to 192 disks and simultaneously the dataset scales (the scale factors) from 100 to 1,600 proportionally. Note that a dataset scale per a single storage device was kept constant. At each of the scale configurations, the execution performance was measured for the different execution methods in the similar way. In this section, we show results of Query C due to the space limitation. Other queries presented similar performance properties.

Figure 8 shows that **OoODE** achieved greater speedup at the higher scales. The graph (a) summarizes the query execution time at different scale configurations and the graph (b) presents its 20-fold magnified chart focusing on **Partitioning (best)** and **OoODE**. Note that we omit **Partitioning (naive)** hereafter because it consistently took roughly half time of **Serial**. Table 2 summarizes the speedup ratios. The observation was two-fold. Firstly, at each of the scale configurations, we witnessed a similar performance trend to the observation on the small machines (presented in Section 4.1.) **OoODE** consistently performed more than two orders of magnitude faster than **Serial** and even faster than **Partitioning (best)**. Secondly, we in particular observed that larger scale configurations allowed **OoODE** to provide greater speedup. At the largest configuration (192 disks), the speedup reached a factor of 419 over **Serial** and a factor of 2.07 even over **Partitioning (best)**.

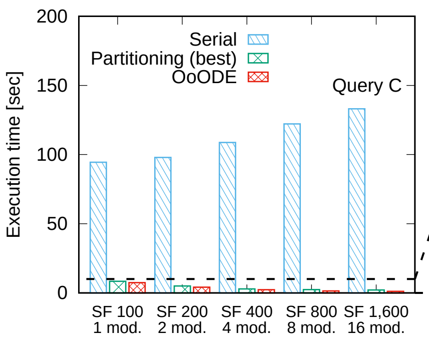
Table 2: Speedup by OoODE. *Summary of Figure 8.*

Scale (number of disks)	12	24	48	96	192
over **Serial**	23.4	42.2	87.9	192	419
over **Partitioning (best)**	1.04	1.05	1.18	1.29	2.07

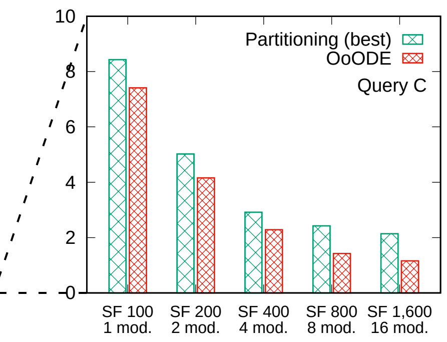
In order to further investigate this phenomenon, we measured average aggregate IO throughput during query execution. Figure 8 (c) presents that **OoODE** gained a linear increase in the IO throughput thoroughly up to 192 disks. **Serial** could hardly increase the IO throughput (remaining at 194 to 250 IO/s), even if more storage devices became available. Because **Serial** had no capability of parallelizing the query execution, it failed to leverage the IO bandwidth provided by many disks. Meanwhile, **Partitioning (best)** gained much better execution parallelism. The IO throughput increased in a near-linear fashion from 5,610 IO/s (12 disks) to 32,100 (96 disks.) But, it became almost saturated at the scale configuration exceeding 96 disks, and finally limited at 39,400 IO/s (192 disks.) This implies that **Partitioning (best)** failed to gain sufficient parallelism to fully utilize the IO bandwidth of more than 96 disks. On the contrary, **OoODE** succeeded to linearly increase the IO throughput, reaching 81,500 IO/s for 192 disks. Resultingly, **OoODE** achieved further better speedup at the largest scale.

We also performed the same test in a large flash-based machine composed of a two-socket server and an external all-flash array having sixteen flash modules ("large flash machine.") The storage scales (the numbers of modules) were changed from one to sixteen modules, while the dataset size per module was kept constant similarly to the previous test.

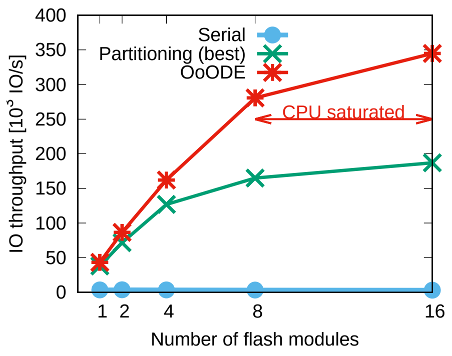
Figure 9 shows that **OoODE** achieved greater speedup at the higher scales even in the flash machine. Again, we observed a similar performance trend. **OoODE** always per-



(a) Execution time (overall)



(b) Execution time (20-fold magnified)



(c) IO throughput

Figure 9: (a) OoODE is up to 85.7 times faster than Serial for Query C on the large flash machine and, (b) OoODE achieves up to 84% speedup even over the best achievable case of Partitioning, and (c) OoODE achieves linear increase in IO throughput by OoODE up to 8 flash modules. *Execution time and IO throughput of Query C on the moderately skewed dataset at different scale configurations in the large flash machine are plotted. Table 3 summarizes the speedup ratios.*

formed faster than the other. As Table 3 denotes, the speedup increased as the scales grew and finally reached a factor of 115 over **Serial** and a factor of 1.84 over **Partitioning (best)** at the largest configuration. Figure 9 (c) presents a noteworthy observation that the IO throughput of **OoODE** consistently increased in a linear fashion up to eight flash modules allowing 280,000 IO/s, but it became almost saturated at larger scales and finally became limited to 345,000 IO/s at the scale configuration of sixteen modules. Because the potential IO bandwidth was an order of magnitude higher in the large flash machine than in the disk-based machine, all the server processors became fully busy[4]. Although **OoODE** might have provided further greater execution parallelism potentially, the hardware capacity of our experimental environment limited the throughput, interfering the further performance improvement. Regardless of such limitation, **OoODE** successfully gained significant speedup.

In summary, even in the large flash machine, **OoODE** consistently achieved better performance than the other. Besides, as the configuration scale became larger, **OoODE** offered greater speedup until it reached the potential parallelism that the query inherently held or the scale limitation that the hardware configuration imposed.

4.3 Comparison of OoODE and other DBMSs

Finally, we present a comparative performance study of **OoODE** and other DBMSs. We tested the following major DBMSs that were popular in the market and often reported in literature: **MySQL** (a representative open-source DBMS [46], Version 5.7), **MariaDB** (a fork of MySQL, allowing non-blocking index-based search/join operations [42], Version 10.2), and **PostgreSQL** (another representative open-source DBMS [50], Version 9.6.) In each case, we honestly tuned performance with known state-of-the-art skills.

Figure 10 demonstrates that **OoODE** reduced query execution time by two to four orders of magnitude from the other DBMSs for all the tested queries. The graphs (a) and (b) on the left summarize the execution time that **OoODE** and the alternatives took for executing the four test queries in the largest scale configurations in the large disk machine, whereas the graphs (c) and (d) on the right present that in the large flash machine. We again witnessed that **OoODE** consistently achieved significantly better performance in comparison with the alternatives. As Table 4 indicates, the speedup ranged two to four orders of magnitude for the large disk machine and for the large flash machine. The DBMSs that we tested for comparison have different query execution magics[5]. **MySQL** basically executes a query in a way similar to **Serial**. Meanwhile, **MariaDB** and **PostgreSQL** have alternative parallelizing techniques. The result implies that these techniques worked in part to speed up the query execution in specific cases. In contrast, **OoODE** consistently performed two to four orders of magnitude faster than these DBMSs. This significant speedup demonstrates the benefit of the unique parallelization capability of **OoODE**.

[4]In the other cases presented so far, the query execution was basically IO bound.

Table 3: Speedup by OoODE. *Summary of Figure 9.*

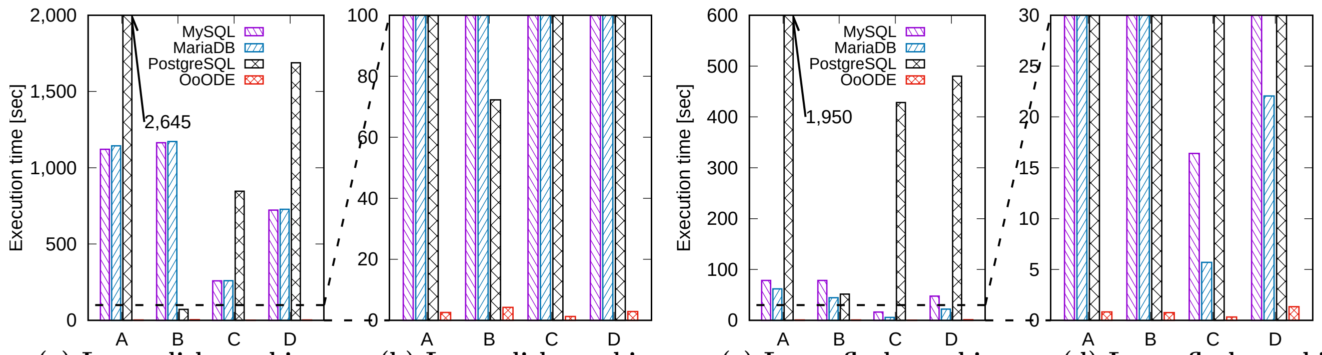
Scale (number of modules)	1	2	4	8	16
over **Serial**	12.7	23.5	47.6	85.7	115
over **Partitioning (best)**	1.14	1.21	1.28	1.70	1.84

5. RELATED WORK

A key of OoODE (out-of-order database execution) is that, by exploiting the exact knowledge of the potential execution parallelism for each operation ready to be performed during query execution, it dynamically invokes tasks and executes them in parallel, such that it can automatically squeeze out the execution parallelism inherent to the query at maximum. Accordingly, OoODE significantly speeds up the query execution by making the best use of the IO bandwidth.

Static query parallelization. Partitioning [13, 19, 39] and pipelining [6, 11, 40, 45, 48] are popular techniques to divide a given query into multiple subtasks to exploit partitioned parallelism and pipilined parallelism respectively. Both may be utilized together [4,7,12,18,25,49]. These techniques have been deployed into commercial DBMSs [5,20,60]

[5]Our separate paper will report the detailed investigation.



(a) **Large disk machine (overall)** (b) **Large disk machine (20-fold magnified)** (c) **Large flash machine (overall)** (d) **Large flash machine (20-fold magnified)**

Figure 10: (a), (b) OoODE is two to four orders of magnitude faster than other DBMSs for Queries A, B, C and D on the large disk machine, (c) and (d) achieves similar speedup on the large flash machine. *Execution time of Queries A, B, C and D on the moderately skewed dataset in the large disk and flash machines at the largest scale configuration is plotted. Table 4 summarizes the speedup ratios.*

and they have also formed the basis of the recent parallel data processing systems such as MapReduce [9,16,30,44,62]. Partitioning and pipelining statically determine the way to split query work at compile time, whereas OoODE dynamically determines the query decomposition during execution.

Query optimization. How to tune the static parallelization has been explored in query optimization studies [1, 10, 23, 33, 51, 53, 63]. They utilize heuristic rules and/or statistical information to optimize the parallelism at query compile time. Researchers have extended query optimization to runtime tuning [2, 14, 26, 29, 32, 52]. They gather statistical execution information during query execution and optimize the execution plan again. By contrast, OoODE utilizes the exact knowledge of the potential execution parallelism for each operation during query execution with the assistance of runtime data structure information.

Dynamic query parallelization. Recent work has introduced techniques to parallelize given query work during execution. Leis et al. has proposed morsel-driven query processing [37], which divides input data into small fragments (morsels) to achieve better load balancing among all the processor cores. Elastic iterator model, proposed by Wang et al. for in-memory database clusters [59], has introduced a dynamic scheduler in order to improve the processor utilization efficiency. A very recent study [12] plans to integrate such techniques to control parallelism dynamically. In contrast to these work, OoODE makes the best use of the IO bandwidth by dynamically parallelizing a query with the exact knowledge of its potential parallelism.

Ousterhout et al. has proposed an execution framework, named Monotasks, for Spark [47]. Their focus is on allowing good performance clarity among different resources, whereas OoODE focuses on optimizing IO parallelism at runtime.

Dataflow machines. Dataflow machines [3,15,17,22,24, 27,58] exploit instruction parallelism inherent to a given program, while OoODE exploits IO parallelism for a relational query.

6. CONCLUSION

This paper has presented OoODE (out-of-order database execution), which is a simplified and unified execution formula to offer significant speedup for database queries consistently. OoODE dynamically decomposes query execution at runtime by using the exact knowledge of the potential parallelism of each database operator with the assistance of runtime data structure information. Owing to this capability, OoODE achieves better or comparable performance with respect to conventional parallelizing methods, while reducing the underlying complexity. The introductory experiments have verified that OoODE performs (1) two to three orders of magnitude faster than serial execution; (2) substantially (up to 2.07 times) faster than the best achievable case of partitioning; and (3) two to four orders of magnitude faster than major DBMSs. OoODE has been in part employed in commercial DBMS [28].

We have just opened the gate for a new horizon enabled by OoODE. Several open topics can be raised for further exploration. As an early study, this paper has focused on selective decision support queries which are mainly composed of additive operators; however, potentially OoODE can be applied or extended to other types of database queries and manipulations. We have been working on more comprehensive design and extensive performance experiments. In addition, extending OoODE to other database models also remains to be solved. This paper has started the discussion with the relational database model. The standard query language, SQL, is declarative, thereby providing database software with an opportunity to maneuver an execution procedure. Recently, procedural languages such as Java or Python are often utilized for analytics work. We plan to investigate a new compilation technology for automatically embedding OoODE into procedural programs.

Table 4: Speedup by OoODE. *Summary of Figure 10.*

Query	A	B	C	D
Large disk machine				
over **MySQL**	432	274	205	249
over **MariaDB**	440	276	206	251
over **PostgreSQL**	1018	17.0	670	581
Large flash machine				
over **MySQL**	94.9	103	49.1	35.3
over **MariaDB**	74.7	58.4	17.1	16.4
over **PostgreSQL**	2355	67.6	1282	356

Table 5: Test machines.

	Small disk machine	Large disk machine	Small flash machine	Large flash machine
Subsystem(s)	Dell PowerEdge 740xd	IBM SystemX x3750M4 IBM Storwize V7000	Lenovo x3850 X6	Hitachi HA8000 Hitachi Unified Storage
Processors / Memory	2x Xeon 2.60GHz (28 cores) / 96 GB	4x Xeon 2.70 GHz (24 cores) / 192 GB	4x Xeon 2.10GHz (64 cores) / 2,048 GB	2x Xeon 3.0GHz (20 cores) / 64 GB
Storage	24x 900GB 10Krpm NL-SAS HDDs RAID6 (22D+2P)	192x 900GB 10Krpm NL-SAS HDDs 16x RAID6 (10D+2P)	8x 1.2TB flash SSDs RAID6 (6D+2P)	16x 1.6TB flash modules 2x RAID5 (7D+1P)
OS	CentOS Linux 7.4	RedHat Ent. Linux 6.9	CentOS Linux 7.5	CentOS Linux 6.9
Database	Page size: 16KB, Database buffer: 16GB			

7. ACKNOWLEDGMENTS

We are grateful to Calton Pu of Georgia Institute of Technology for his helpful comments and suggestions on this paper. This work has been supported in part by multiple research funds: Novel Systems Software Project (Ministry of Education, Culture, Sports, Science and Technology of Japan), Funding Program for World-Leading Innovative R&D on Science and Technology (Cabinet Office, Japan), Impulsing Paradigm Change through Disruptive Technologies Program (Cabinet Office, Japan), and Cross-sectoral Technologies for IoT (New Energy and Industrial Technology Development Organization, Japan.)

8. APPENDIX

Test machines. Table 5 summarizes four machine configurations that we utilized in the experiments presented in Section 4. Storage resources for operating system and experiment data management are omitted for simplicity. If a storage controller (RAID-5/6) exposed multiple storage volumes, they were merged into a single database volume by the Linux logical volume manager (striping). The volume was utilized as a *raw* device except for PostgreSQL, for which a Linux standard filesystem ext4fs was utilized since PostgreSQL did not allow the direct use of raw devices. Hyper-threading was disabled.

Test queries. The test query set was composed of a single-relation search query (Query A), a two-way join query (Query B), a three-way join query (Query C) and a six-way join query (Query D). SQLs of the test queries are listed below.

```
-- Query A
SELECT SUM(L_QUANTITY),
 SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)*(1+L_TAX))
FROM LINEITEM WHERE L_PARTKEY BETWEEN 1 AND 8000;
-- Query B
SELECT P_BRAND, SUM(L_QUANTITY),
 SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)*(1+L_TAX))
FROM PART JOIN LINEITEM
 ON PART.P_PARTKEY = LINEITEM.L_PARTKEY
WHERE P_NAME LIKE 'goldenrod lavender spring%'
GROUP BY P_BRAND;
-- Query C
SELECT O_ORDERPRIORITY, SUM(L_QUANTITY),
 SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)*(1+L_TAX))
FROM CUSTOMER
 JOIN ORDERS ON CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY
 JOIN LINEITEM ON O_ORDERKEY = L_ORDERKEY
WHERE C_NAME LIKE 'Customer#00001%'
GROUP BY O_ORDERPRIORITY;
-- Query D
SELECT N_NAME, SUM(PS_SUPPLYCOST), SUM(L_QUANTITY),
 SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)*(1+L_TAX))
FROM CUSTOMER
 JOIN ORDERS ON CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY
 JOIN LINEITEM ON O_ORDERKEY = L_ORDERKEY JOIN PARTSUPP
 ON L_PARTKEY = PS_PARTKEY AND L_SUPPKEY = PS_SUPPKEY
 JOIN SUPPLIER ON S_SUPPKEY = PS_SUPPKEY
 JOIN NATION ON S_NATIONKEY = N_NATIONKEY
WHERE C_NAME LIKE 'Customer#000001%' GROUP BY N_NAME;
```

9. REFERENCES

[1] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively Parallel Sort-merge Joins in Main Memory Multi-core Database Systems. *PVLDB*, 5(10):1064–1075, 2012.

[2] G. Antoshenkov. Dynamic Query Optimization in Rdb/VMS. In *Proc. Int'l Conf. on Data Engineering*, pages 538–547, 1993.

[3] A. Arvind and K. P. Gostelow. The U-Interpreter. *Computer*, 15(2):42–49, 1982.

[4] C. Ballinger and R. Fryer. Born To Be Parallel Why Parallel Origins Give Teradata an Enduring Performance Edge. *IEEE Data Eng. Bull.*, 20(2):3–12, 1997.

[5] S. Bellamkonda, H. Li, U. Jagtap, Y. Zhu, V. Liang, and T. Cruanes. Adaptive and Big Data Scale Parallel Execution in Oracle. *PVLDB*, 6(11):1102–1113, 2013.

[6] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. Biennial Conf. on Innovative Data Systems Research*, 2005.

[7] P. A. Boncz, S. M., and M. L. Kersten. Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct. *PVLDB*, 2(2):1648–1653, 2009.

[8] S. Borkar. Thousand core chips – a technology perspective. In *Proc. Annual ACM/IEEE Design Automation Conf.*, pages 746–749, 2007.

[9] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing. In *Proc. Int'l Conf. on Data Engineering*, pages 1151–1162, 2011.

[10] L. Bouganim, D. Florescu, and B. Dageville. Skew Handling in the DBS3 Parallel Database System. In *Proc. Int'l ACPC Conf. Parallel Databases and Parallel I/O*, pages 98–109, 1996.

[11] M.-S. Chen, M.-L. Lo, P. S. Yu, and H. C. Young. Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins. In *Proc. Int'l Conf. on Very Large Data Bases*, pages 15–26, 1992.

[12] P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. Hetexchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *PVLDB*, 12(5):544–556, 2019.

[13] J. Cieslewicz, J. Berry, B. Hendrickson, and K. A. Ross. Realizing Parallelism in Database Operations: Insights from a Massively Multithreaded Architecture. In *Proc. Int'l Workshop on Data Management on New Hardware*, 2006.

[14] R. L. Cole and G. Graefe. Optimization of Dynamic Query Evaluation Plans. In *Proc. Int'l Conf. on Management of Data*, pages 150–160, 1994.

[15] A. L. Davis. The Architecture and System Method of DDMI: A Recursively Structured Data Driven Machine. In *Proc. Int'l Symp. on Computer Archiecture*, pages 210–215, 1978.

[16] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. USENIX Symp. on Opearting Systems Design Implementation*, pages 137–150, 2004.

[17] J. B. Dennis, C. K. Leung, and D. P. Misunas. A Highly Parallel Processor Using a Data Flow Machine Language. Technical report, Massachusetts Institute of Technology, 1977.

[18] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA - A High Performance Dataflow Database Machine. In *Proc. Int'l Conf. on Very Large Data Bases*, pages 228–237, 1986.

[19] D. J. DeWitt, J. F. Naughton, and J. Burger. Nested Loops Revisited. In *Proc. Int'l Conf. on Parallel and Distributed Information Systems*, pages 230–242, 1993.

[20] E. Ding, L. A. Dimino, G. Gopal, and T. K. Rengarajan. Parallel Processing Capabilities of Sybase Adaptive Server Enterprise 11.5. *IEEE Data Eng. Bull.*, 20(2):35–43, 1997.

[21] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, 1989.

[22] D. D. Gajski, D. A. Padua, D. J. Kuck, and R. H. Kuhn. A Second Opinion on Data Flow Machines and Languages. *Computer*, 15(2):58–69, 1982.

[23] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query Optimization for Parallel Execution. In *Proc. Int'l Conf. on Management of Data*, pages 9–18, 1992.

[24] K. P. Gostelow and R. E. Thomas. Performance of a Simulated Dataflow Computer. *IEEE Trans. Comput.*, 29(10):905–919, 1980.

[25] G. Graefe. Volcano – An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, 1994.

[26] G. Graefe and K. Ward. Dynamic Query Evaluation Plans. In *Proc. Int'l Conf. on Management of Data*, pages 358–366, 1989.

[27] J. Gurd and I. Watson. A prototype data flow computer with token labelling. In *Proc. National Computer Conf.*, pages 623–628, 1979.

[28] Hitachi Ltd. Hitachi's Database Product Based on Achievement of Collaborative Research by Institute of Industrial Science, the University of Tokyo and Hitachi Obtains the World's First Performance Record for the Largest-Scale Class in the Industry-Standard TPC-H Database Benchmark. `https://www.hitachi.com/New/cnews/131021a.html`, 2013.

[29] W. Hong and M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. In *Proc. Int'l Conf. on Parallel and Distributed Information Systems*, pages 218–225, 1991.

[30] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proc. European Conf. on Computer Systems*, pages 59–72, 2007.

[31] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Comput. Surv.*, 16(2):111–152, 1984.

[32] N. Kabra and D. J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *Proc. Int'l Conf. on Management of Data*, pages 106–117, 1998.

[33] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. D. Blas, and P. Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs. *PVLDB*, 2(2):1378–1389, 2009.

[34] M. Kitsuregawa and K. Goda. Vision and Preliminary Experiments for Out-of-Order Database Engine (OoODE) (Japanese). *DBSJ Journal*, 8(1):131–136, 2009.

[35] M. Kitsuregawa and K. Goda. Database Management System and Method. Japan Patent 4,611,830, U.S. Patent 7,827,167, 2010.

[36] C. E. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid. Server engineering insights for large-scale online services. *IEEE Micro*, 30(4):8–19, 2010.

[37] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. In *Proc. Int'l Conf. on Management of Data*, pages 743–754, 2014.

[38] B. Lewis and D. J. Berg. *Multithreaded Programming With PThreads*. Prentice Hall, 1997.

[39] J. Li, W. Sun, and Y. Li. Parallel Join Algorithms based on Parallel B+-trees. In *Proc. Int'l Symp. on Cooperative Database Systems for Advanced Applications*, pages 178–185, 2001.

[40] B. Liu and E. A. Rundensteiner. Revisiting Pipelined Parallelism in Multi-Join Query Processing. In *Proc. Int'l Conf. Very Large Data Bases*, pages 829–840, 2005.

[41] Z. Majo and T. R. Gross. Memory system performance in a NUMA multicore multiprocessor. In *Proc. Int'l. Conf. on Syst. and Storage*, 2011.

[42] MariaDB Foundation. MariaDB: One of the most popular database servers, 2009.

[43] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology J.*, 6(1):1–12, 2003.

[44] S. Melnik, A. Gubarev, J. J. Long, G. Romer,

S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive Analysis of Web-scale Datasets. *PVLDB*, 3(1):330–339, 2010.

[45] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[46] Oracle Corp. MySQL: The world's most popular open source database, 2020.

[47] K. Ousterhout, C. Canel, S. Ratnasamy, and S. Shenker. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *Proc. ACM Symp. on Operating Systems Principles*, pages 184–200, 2017.

[48] S. Padmanabhan, T. Malkemus, R. C. Agarwal, and A. Jhingran. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *Proc. Int'l Conf. on Data Engineering*, pages 567–574, 2001.

[49] J. Paul, J. He, and B. He. GPL: A GPU-based Pipelined Query Processing Engine. In *Proc. Int'l Conf. on Management of Data*, pages 1935–1950, 2016.

[50] PostgreSQL Global Development Group. PostgreSQL: The world's most advanced open source relational database, 2020.

[51] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki. Scaling Up Concurrent Main-memory Column-store Scans: Towards Adaptive NUMA-aware Data and Task Placement. *PVLDB*, 8(12):1442–1453, 2015.

[52] B. Răducanu, P. Boncz, and M. Zukowski. Micro Adaptivity in Vectorwise. In *Proc. Int'l Conf. on Management of Data*, pages 1231–1242, 2013.

[53] D. A. Schneider and D. J. DeWitt. Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines. In *Proc. Int'l Conf. on Very Large Data Bases*, pages 469–480, 1990.

[54] A. Silberschatz, B. Galvin, and G. Gagne. *Operating system concepts*. Wiley, 2013.

[55] J. M. Smith and P. Y. Chang. Optimizing the performance of a relational algebra database interface. *Commun. ACM*, 18(10):568–579, 1975.

[56] Transaction Processing Performance Council. TPC-H is a Decision Support Benchmark, 2020.

[57] T. Ungerer, R. Robic, and J. Silc. A Survey of Processors wit Explicit Multithreading. *ACM Comput. Surv.*, 35(1):29–63, 2003.

[58] A. H. Veen. Dataflow Machine Architecture. *ACM Comput. Surv.*, 18(4):365–396, 1986.

[59] L. Wang, M. Zhou, Z. Zhang, Y. Yang, A. Zhou, and D. Bitton. Elastic Pipelining in an In-Memory Database Cluster. In *Proc. Int'l Conf. on Management of Data*, pages 1279–1294, 2016.

[60] Y. Wang. DB2 Query Parallelism: Staging and Implementation. In *Proc. Int'l Conf. on Very Large Data Bases*, pages 686–691, 1995.

[61] E. Wong and K. Youssefi. Decomposition - A Strategy for Query Processing. *ACM Trans. Database Syst.*, 1(3):223–241, 1976.

[62] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proc. USENIX Conf. on Networked Systems Design and Implementation*, pages 15–28, 2012.

[63] M. Ziane, M. Zaït, and P. Borla-Salamet. Parallel Query Processing with Zigzag Trees. *The VLDB Journal*, 2(3):277–302, 1993.