Scheduling Resources to Multiple Pipelines of One Query in a Main Memory Database Cluster

Zhuhe Fang, *Student Member, IEEE*, Chuliang Weng, *Member, IEEE*, Li Wang, Huiqi Hu, and Aoying Zhou, *Member, IEEE*

Abstract—To fully utilize the resources of a main memory database cluster, we additionally take the independent parallelism into account to parallelize multiple pipelines of one query. However, scheduling resources to multiple pipelines is an intractable problem. Traditional static approaches to this problem may lead to a serious waste of resources and suboptimal execution order of pipelines, because it is hard to predict the actual data distribution and fluctuating workloads at compile time. In response, we propose a dynamic scheduling algorithm, List with Filling and Preemption (LFPS), based on two novel techniques. (1) *Adaptive filling* improves resource utilization by issuing more extra pipelines to adaptively fill idle resource "holes" during execution. (2) *Rank-based preemption* strictly guarantees scheduling the pipelines on the critical path first at run time. Interestingly, the latter facilitates the former filling idle "holes" with best efforts to finish multiple pipelines as soon as possible. We implement LFPS in our prototype database system. Under the workloads of TPC-H, experiments show our work improves the finish time of parallelizable pipelines from one query up to 2.5X than a static approach and 2.1X than a serialized execution.

Index Terms—Main memory database, query processing, resource scheduling, independent parallelism, preemption, filling

1 Introduction

With the evolving of hardware manufacturing technology, the capacity of main memory on a commodity server continues to grow. Under this trend, it is economically and technically feasible to keep the entire dataset in main memory for many data-intensive and performance-critical scenarios. Therefore, in-memory data processing has become a hot topic in both industry and academia in recent decades [1], [2], [3]. On the other hand, data volume in real applications has grown well beyond the processing capability of a single server. Consequently, distributed main-memory databases are gradually becoming a de facto solution to process SQL queries over large dataset with low latency constraints. In order to fully utilize the computational resources in the cluster, pipelined execution is widely adopted in state-of-the art SQL query systems [1], [3], to exploit the *partitioned parallelism* and the *pipelined parallelism*. While previous work [4], [5], [6], [7] focuses on performance modeling and optimization for each individual pipeline, we observed that running a single pipeline at a time could not make full use of the distributed resources in a cluster due to the following reasons.

First, a pipeline typically consumes a large amount of resources in a certain type while underutilizing other types of resources. For instance, when a running pipeline is CPU-bound, the computational resource may be exhausted but the network bandwidth may be almost idle. Second, during the execution of a pipeline, the workloads may not be evenly distributed across the servers [5], [7], resulted from the skewed data distribution and workload fluctuation. Such workload imbalance may lead to overload on some servers but resource waste on others. Third, due to synchronization overhead and thread contention [8], provisioning more CPU cores to a particular pipeline does not always bring proportional performance improvement but may incur inefficient utilization of the CPU cores [9].

To fully use cluster resources from these aspects, this paper additionally focuses on *independent parallelism* [10], in which independent pipelines, i.e., pipelines without data dependency, are executed in parallel. Because this way provides large optimization opportunities in better utilizing resources than just running a single pipeline. However, the independent parallelism is just involved in previous static scheduling strategies [10], [11], [12]. Those strategies may lead to severe resource waste in practice, because they heavily rely on the estimation of runtime workloads at compile time, which is known to be error-prone due to the limited information available at compile time and the unpredictable and fluctuating nature of the runtime workloads [4], [13]. To further illustrate this issue, a straightforward example is given as follows.

A Motivation Example. Fig. 1a shows a physical plan of TPC-H query-3, which involves two equal joins among three tables, `orders` (O), `lineitem` (L) and `customer` (C). Table `orders` and `lineitem` are hash-joined locally, then their results are further duplicated-joined with the data from `customer` that is broadcast via an exchange operator. The whole query plan can be decomposed into three pipelines, P_0, P_1 and P_2, by the two blocking hash build operators. In pipelined execution, a pipeline is further

- *Z. Fang, C. Weng, H. Hu, and A. Zhou are with the School of Data Science and Engineering, East China Normal University, Shanghai 200241, China. E-mail: fzhedu@gmail.com, {clweng, hqhu, ayzhou}@dase.ecnu.edu.cn.*
- *L. Wang is with the Advanced Digital Sciences Center, Illinois at Singapore Pte. Ltd., Singapore 138602. E-mail: wangli1426@gmail.com.*

Manuscript received 8 May 2018; revised 8 Oct. 2018; accepted 19 Nov. 2018. Date of publication 6 Dec. 2018; date of current version 4 Feb. 2020. (Corresponding author: Chuliang Weng.) Recommended for acceptance by G. Li. Digital Object Identifier no. 10.1109/TKDE.2018.2884724

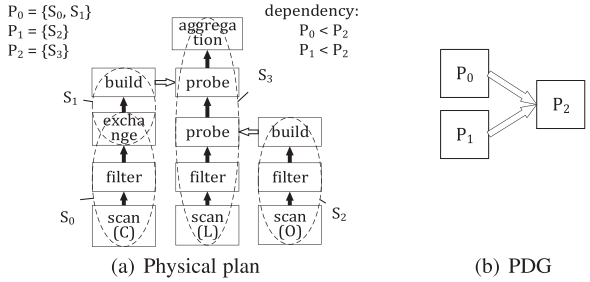


Fig. 1. TPC-H query-3.

divided into stages by its exchange operators, and consequently $P_0 = \{S_0, S_1\}$, while $P_1 = \{S_2\}$ and $P_2 = \{S_3\}$, as shown in Fig. 1a. Particularly within a pipeline, there is producing and consuming relationship between two adjacent stages S_0 and S_1 [4]. Since the execution of P_2 relies on the hash tables built by P_0 and P_1, P_2 cannot be executed until P_0 and P_1 are completed. For simplicity, we assume the query plan in Fig. 1a will be executed on two servers `a` and `b`, where corresponding partitioned data locates.

According to static methods [10], [12], [14], a possible schedule for the physical plan in Fig. 1a may be as follows: on server `a` and `b`, all cores are first divided into two parts for P_0 and P_1, based on their estimated workloads. After P_0 and P_1 are finished, all cores are occupied by P_2. In this scheduling, the core utilization of all pipelines may be shown as Fig. 2a, where the horizontal axes from left to right represent time, and the vertical axes mean the percent of cores on each server. Particularly, the vertical axes on two sides are in opposite directions. The right vertical axis from top to bottom indicates the core utilization of P_1 and P_2. But that of P_0 is expressed by the left vertical axis from bottom to top. By this way, we can clearly see the core utilization of parallel pipelines P_0 and P_1 on each server.

As depicted in Fig. 2a, the core utilization of all pipelines is fluctuating during execution, and a portion of resources are seriously wasted, denoted by some white space. For P_0, the wasted resources are resulted from the unbalance of the producing rate of S_0 and the consuming rate of S_1 in the pipeline. For P_1 and P_2, the wasted resources are caused by synchronization and contention overhead among threads during building and probing hash tables, respectively. In particular, due to data skew, the heavier work of P_1 on server `b` is completed later than that of P_0, delaying the beginning of P_2. Even worse on server `a`, after P_1 is finished, its allocated cores are idle.

Two Challenges. Avoiding resource waste during execution is a primary challenge to scheduling multiple pipelines. Traditional way to this problem attempts to make accurate resource allocation based on static estimation information. However, such static method may lead to suboptimal decisions [13], because of uncertain information at compile time, such as cardinality of intermediate results, skewed data, or correlated attributes. In fact, the allocated resources just mean an upper bound. The actual resource utilization is fluctuating with the immediate data flow. Consequently, the finish time of pipelines is unpredictable in prior. So it is necessary to make scheduling decisions at run time. In this case, once a pipeline is completed, some idle resources occur. To maximize the utilization of newly available resources, we have to dynamically choose some ready pipelines to run first. But different execution order may cause all pipelines accomplished in different

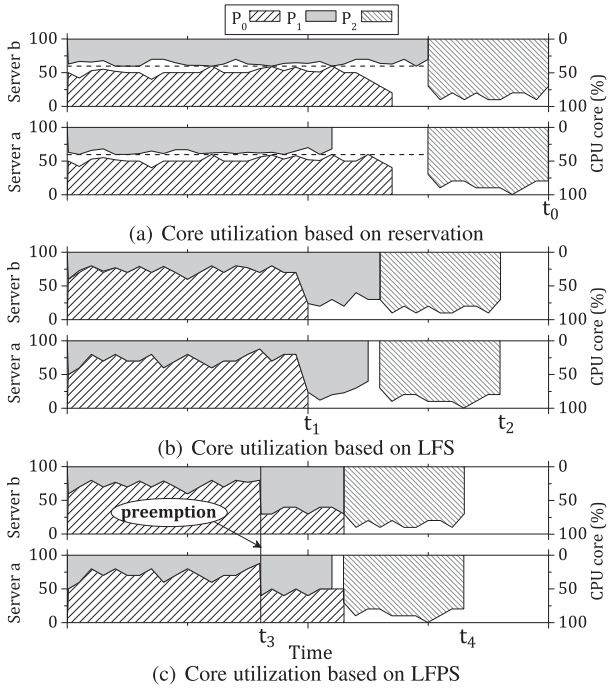


Fig. 2. The execution illustration of TPC-H query-3.

time. Therefore, another challenge is to decide an optimal execution order of pipelines at run time.

Our Proposals. To confront the two challenges, we propose two novel techniques [15]. Specifically, the adaptive filling improves resource usage by issuing more extra pipelines to adaptively fill idle resource "holes" at run time. The rank-based preemption guarantees an optimal scheduling order by allowing the pipelines on the critical path to preempt more resources to finish earlier. Such two techniques can combine with the list scheduling algorithm [16], forming List with Filling Scheduler (LFS) or List with Filling and Preemption Scheduler (LFPS). According to these algorithms, pipelines are dispatched globally. On any local server, resources are coordinated efficiently among different tasks.

Based on LFS, the core utilization of TPC-H query-3 may be like Fig. 2b. P_0 is scheduled first due to its higher rank. Before P_0 is finished at t_1, its core usage is limited by available network bandwidth. The idle cores from P_0 are filled by the extra pipeline P_1. Finally, such filling reduces query response time by t_0 - t_2, compared to the execution based on resource reservation in Fig. 2a. However, after t_1, there are still some idle resource "holes" from P_1, which cannot be filled by other pipelines. But in LFPS, preemption happens at t_3, as the realtime rank of P_0 is lower than that of P_1 to a large extend at that point. Therefore, P_1 preempt cores from P_0, which becomes extra to fill the idle cores from P_1. At last, the query response time is further reduced by t_2 - t_4 in Fig. 2c, compared to the execution based on LFS in Fig. 2b. We see that preemption helps the adaptive filling to make better use of resources to finish the query earlier. As for the left idle resources in Fig. 2c, they can be filled by the pipelines from other queries, forming inter-query parallelism.

Contributions. Our main contributions are as follows:

(1) We propose *adaptive filling* to increase resource utilization by issuing some extra pipelines to adaptively fill idle resource "holes" during execution;

(2) We introduce *rank-based preemption* to guarantee an optimal scheduling order by allowing the pipelines on the critical path to preempt more resources. Moreover, *it facilitates the adaptive filling making better use of resources to finish a query earlier*;
(3) We abstract scheduling resources to multiple pipelines of one query in a cluster to a DAG scheduling problem generally, and propose the List with Filling and Preemption Scheduler algorithm to schedule pipelines at run time;
(4) We design and implement a two-layer scheduling framework in our prototype database to parallelize multiple pipelines, and its performance is evaluated on TPC-H.

The remainder of this paper is organized as follows. Section 2 presents the background about pipelined execution. Section 3 elaborates the problem we concerned in detail and shows the overview of our solution. Section 4 introduces two novel techniques, and scheduling algorithms are designed in Section 5. Section 6 describes the runtime resource coordination on each local server. Section 7 evaluates our proposals with experiments. Next, we discuss briefly in Section 8 and review related work in Section 9. Finally, we conclude our work in Section 10.

2 Preliminaries

2.1 Physical Plan and Iterator Model

A physical plan is converted from a query by a query optimizer. It is defined as a tree structure consisting of a number of data manipulation operators, such as the hash join, filter and scan. A physical plan is often executed in a pipelined manner, enabled by the well-known volcano iterator model [17]. Each operator in a physical plan implements open(), next() and close() functions to initialize its state, generate output tuples and delete its state, respectively. All operators can be classified into two categories, namely *blocking* and *non-blocking* operators, based on their processing logic. A non-blocking operator generates output results on the fly, like the filter or hash probe. On the contrary, a blocking operator, like sort or hash build, collects all children inputs before producing any output.

2.2 Pipeline, Stage, and Task

Pipeline. A physical plan is decomposed into *pipeline*s by its blocking operators. Each pipeline contains a chain of operators, ending by a blocking operator. For instance, the physical plan in Fig. 1a has two blocking operators and thus is decomposed into three pipelines, P_0, P_1 and P_2.

Stage. To achieve partitioned parallelism [10] in the evaluation of a pipeline, exchange operator [17] is introduced to repartition data flow. The exchange operators divide a pipeline into several components, each of which is called a *stage*. For example in Fig. 1a, P_0 has two stages, S_0 and S_1.

Task. Before query evaluation, a stage is duplicated into an appropriate number of copies, each of which is called a *task*. Each task is responsible for processing one partition of the data flow on an assigned server. Note that all the tasks in a given running pipeline are executed simultaneously across servers, to enable both pipelined and partitioned parallelism. Given any two tasks directly connected by the data flow, the task that generates data is called an upstream task, and the task that consumes the data is called a downstream task.

2.3 Pipeline Dependency Graph

In a given physical plan, a pipeline may rely on the output of some other pipelines to initialize the state of its operators, and consequently it cannot be executed before the completion of those pipelines. In Fig. 1a, for example, P_2 cannot start until the completion of P_1 and P_0, because both P_1 and P_0 contain hash build operators, the completion of which is a necessary condition for the execution of the hash probe in P_2. We use a *pipeline dependency graph (PDG)*, similar to the operator dependency graph in [18], to describe the data dependency among pipelines for a given physical plan. Specifically, in a PDG, each node represents a pipeline, and the directed edge between any two adjacent nodes specifies the data dependency between them. Note that a PDG is a tree-like directed acyclic graph (DAG), because the results of a pipeline can only be consumed by at most one pipeline. For example, Fig. 1b is the PDG of the physical plan in Fig. 1a.

2.4 Elastic Pipelining

When a pipeline is running across servers, there is producer-consumer relationship between upstream and downstream tasks. Balancing their processing rate should be concerned when scheduling resources to a pipeline. While traditional static methods could not make optimal decisions due to error-prone estimations at compile time. This problem has been solved by the elastic pipelining [4] in our prior work. It provides an *elastic iterator model*, which effectively adjusts the parallelism of tasks at run time. The adjusting decisions are properly made based on the instantaneous workloads of a pipeline using a *dynamic scheduler*. The scheduler seeks to maximize the throughput of a pipeline under current available resources.

However, just running one pipeline usually cannot fully utilize resources. During the execution of a pipeline, its overall processing rate may be limited by available network bandwidth or CPU cores. The bottleneck of running pipelines may be changing with the fluctuating data flow. Besides, the performance improvement of pipelines does not scale up effectively with the CPU cores investment due to their processing logic [8] in multi-core processors. Therefore, we classify running pipelines into following three types. (1) *Network-bound*: The bottleneck of a running pipeline lays at available network bandwidth. (2) *CPU-bound*: The bottleneck of a running pipeline lays at local CPU cores. (3) *Scalability-bound* [19]: The throughput of a running pipeline would reach a saturation point [8], [11] with incremental CPU cores, beyond which the throughput cannot be improved well. Any type of a single pipeline inevitably wastes some resources. Hence, we focus on parallelizing multiple pipelines to fully utilize cluster resources.

3 Parallelizing Multiple Pipelines

In a main memory database cluster, each physical machine, called a server, is equipped with large memory, multi-core processors and interconnected by high-speed network. In particular, large memory could accommodate all tables and intermediate results to avoid costly disk I/O. As regard to scheduling cluster resources for query evaluation, just taking the pipelined parallelism and partitioned parallelism is not enough to fully utilize resources. From the perspective of the independent parallelism, parallelizing multiple pipelines could provide more chance to make better use of

TABLE 1
Notations about Problem Formulation

Notation	Semantics
O_i	a physical Operator
P_i	a Pipeline
T_i	a Task
$pred(P_i)$	the set of immediate predecessors of P_i
$succ(P_i)$	the set of immediate successors of P_i
P_{entry}	entry pipeline
P_{exit}	exit pipeline
$cost(O_i)$	the cost of O_i
$cost(P_i)$	the average cumulative sum of all operators in P_i
$rank(P_i)$	the cumulative cost of pipelines on the path from P_i to the exit pipeline
CP	Critical Path
PW	Pipeline Width
$serv(T_i)$	the server where task T_i runs
$serv(P_i)$	$\cup serv(T_j)$, where T_j is a task of P_i

resources. In this section, we first present some attributes of a PDG, and their notations are listed in Table 1. Next, we elaborate the problem of scheduling resources to multiple pipelines in detail. Finally, we present an overview of our solution to this problem.

3.1 Attributes of a PDG

Any physical plan can be described as a PDG, $G = (V, E)$, where V is the set of pipelines and E is the set of edges among pipelines. Some attributes about the PDG are defined below, which will be referred to later.

Entry & Exit Pipeline. Each directed edge $\langle P_i, P_j \rangle \in E$ represents the data dependency such that the pipeline P_j can only be executed after the completion of the pipeline P_i. For any given $\langle P_i, P_j \rangle$, P_i is called an immediate *predecessor* of P_j, and P_j is an immediate *successor* of P_i. The set of predecessors of P_i is denoted as $pred(P_i)$. Especially, if $pred(P_i) = \emptyset$, P_i is called an *entry pipeline*, defined as P_{entry}. Accordingly, the set of successor pipelines of P_i is denoted as $succ(P_i)$. If $succ(P_i) = \emptyset$, then P_i is an *exit pipeline*, defined as P_{exit}. Except for P_{exit}, any pipeline only has one immediate successor, i.e., $|succ(P_i)| = 1$, due to the tree structure of the PDG.

Pipeline Width. In a cluster, any task T_i of a pipeline is dispatched to a destination server, called $serv(T_i)$. The set of servers, where any task T_i of P_j runs, is defined as $serv(P_j)$, namely $serv(P_j) = \cup serv(T_i)$. The number of servers in $serv(P_j)$, i.e., $|serv(P_j)|$, is denoted as *pipeline width (PW)*. In particular, if $serv(P_k) \cap serv(P_j) \neq \emptyset$, then there are *resource conflicts* between P_k and P_j. Deciding an optimal $serv(P_j)$ for each pipeline P_j is also an intractable problem, which is the focus of the partitioned (horizontal) parallelism [20], [21]. According to those work, $serv(P_j)$ is determined at compile time with the consideration of data locality in this paper.

Cost of Pipelines. To fairly compare the cost of different pipelines, we express the average cumulative sum of each operator O_j of P_i as the cost of P_i. If P_i has m operators, $cost(P_i)$ is given by

$$cost(P_i) = \sum_{j=1}^{m} cost(O_j)/PW(P_i), O_j \in P_i. \quad (1)$$

This definition is an adaption of the classic cost model [22] to estimate the cost of a plan. The model first estimates the cost of every individual operator of the plan and then sums up these costs. Particularly, the cost of an operator is composed of CPU cost (C_{CPU}) and network cost (C_{NET}), i.e., $cost(O_j) = \lambda C_{CPU} + C_{NET}$, where λ depends on how much slower is the network speed than the CPU speed.

Priority of Pipelines. We define $rank(P_i)$ to quantify the priority of P_i, which is the cumulative cost of pipelines on the path from P_i to P_{exit}, including P_i but excluding P_{exit}, because P_{exit} cannot parallelize with others. Except for P_{exit}, any other pipeline only has one successor pipeline, thus $rank(P_i)$ is defined by

$$rank(P_i) = rank(succ(P_i)) + cost(P_i). \quad (2)$$

Since the rank is computed recursively by traversing a PDG upward, starting from the side of the exit pipeline, it is known as upward rank [23]. The rationality of $rank(P_i)$ refers to the remaining work to do on the path from P_i to P_{exit} in a PDG.

Critical Path. If $rank(P_i)$ is the largest value among all parallelizable pipelines' rank, then the path from P_i to P_{exit} is a *critical path (CP)* of a PDG. It determines the lower bound of the PDG's finish time [24]. To this end, scheduling pipelines according to the order of $rank(P_i)$ can keep the preference of pipelines on a CP so that to minimize the finish time of a PDG.

3.2 Problem Statement

In a main memory database cluster, any SQL query is first converted into a single-server plan. Then this plan is transformed into a multi-server plan, in order to reduce the search space. This approach is commonly adopted in distributed query processing [22]. The multi-server plan includes not only its evaluation cost but also the destination servers for each pipeline. Because the locations of evaluation also decide the shape and cost of plans. For example, a multi-server plan should add an exchange operate if the needed data is on remote servers, additionally increasing the evaluation cost. Optimizing multi-server plans is widely studied in the literature [20], [22], [25], but beyond the scope of this paper. It belongs to the first phase of the *two-phase* optimization technique [14]. Here, we focus on the second phase, resource scheduling.

Given a multi-server plan, its PDG and destination servers of each pipeline refer to the data dependency and resource conflicts among pipelines, respectively. They are two *constrains* to schedule multiple pipelines of a query. Specifically, P_i can be *ready* to run after all its predecessors have been finished. Even though two ready pipelines do not have data dependency, they may compete for the same resources. In this case, allocating resources to which of them first is a tradeoff such that different execution order may cost different time to complete them. Therefore, there are no data dependency and resource conflicts among simultaneously running pipelines. But these running pipelines may seriously waste resources, which is another challenging problem as analyzed in the introduction section.

The *objective* of Scheduling Resources to Multiple Pipelines of one query in a memory database cluster (SRMP) problem is to dynamically decide the execution order of pipelines from a given PDG and improve resource

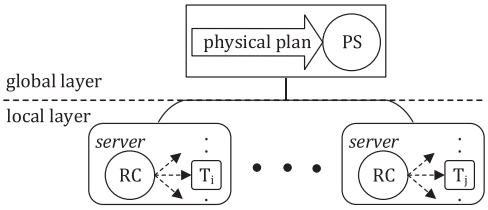


Fig. 3. The two-layer scheduling framework.

utilization of running pipelines such that the finish time of the query is minimized.

3.3 Overview of Our Solution

SRMP can be reduced to the DAG scheduling on multiprocessors problem, which has been proven to be NP-complete [16], [23], [24]. To deal with the two challenges of SRMP, we first present two dynamic techniques in Section 4 to improve resource utilization and determinate an optimal execution order. Based on them, we design a two-layer scheduling framework shown in Fig. 3.

- On the global layer, a *Pipeline Scheduler (PS)* generates a PDG from a physical plan. Based on current available resources, the PS dynamically makes scheduling decisions according to the scheduling algorithms in Section 5.
- On the local layer, resources of each server are coordinated efficiently among tasks by a *Resource Coordinator (RC)* in Section 6. To confirm the efficiency of CPU cores, the RC takes the scalability bound into consideration to avoid allocating excessive cores to a task. To improve resource utilization, the RC schedules extra tasks to adaptively fill resource "holes" from other tasks.

4 Two Dynamic Techniques

To overcome the two challenges of SRMP, we propose two dynamic techniques in this section. First, we analyze idle resources during execution, and present adaptive filling to make full use of resources. Second, we identify out-of-order cases in the critical-path-first strategy, and introduce rank-based preemption to allow higher ranked pipeline to preempt more resources to finish all pipelines earlier. Finally, we analyze two requirements to achieve the two dynamic techniques.

4.1 Adaptive Filling

Idle Resource "Holes". When scheduling pipelines on a cluster of servers, there are idle resource "holes" from three aspects. (1) Multi-dimensional resources of allocated servers cannot fully be used by a running pipeline. Even considering the state-of-the-art dynamic scheduling [4], a running pipeline may also be network-bound, CPU-bound or scalability-bound. In detail, when a pipeline is network-bound, some CPU cores are idle. When a pipeline is CPU-bound, the network is idle. When a pipeline is scalability-bound, the network is idle and some CPU cores are wasted. (2) Due to load imbalance within a pipeline, a part of allocated servers are free after their workloads are completed, while others are busy. Traditional ways to this problem base on work stealing or loading balance [5], [7], [26]. But for a running pipeline, stealing or balancing workloads cross servers is not only complicated to maintain running states but also expensive to move data. (3) Idle servers may occur even after scheduling pipelines as many as possible, because these remaining idle servers cannot satisfy the resource demand of any current ready pipeline.

Filling "Holes". Since it is hard to predicate these idle "holes" in advance, we propose adaptive filling. It allows to adaptively fill above three idle resource "holes" in terms of CPU cores or network bandwidth at run time by issuing some extra pipelines. (1) Extra pipelines help to fully use multi-dimensional resources at run time. If the scheduled pipelines are CPU-bound, extra pipelines try to use free network bandwidth and vice versa. (2) Extra pipelines occupy the idle resources after some tasks of a pipeline are finished. Such way achieves sharing idle resources among different pipelines, rather than within a pipeline. (3) Extra pipelines are dispatched to take up the left available servers after scheduling the most of ready pipelines. In summary, adaptive filling can improve resource utilization with best efforts. It is inspired by the filling policy in static scheduling, which arranges lower prioritized jobs to fill "holes" from already scheduled higher prioritized jobs. This filling policy has been proved effective in the backfilling [27] and the insertion based DAG scheduling [23].

In detail, the pipeline P_i that can be allocated enough resources is called a *pivot pipeline*, and any task T_j of which is pivot on $serv(T_j)$. The pivot task T_j can arbitrarily use the allocated resources on $serv(T_j)$. While other running pipelines that aim at filling the idle resources are called *extra pipeline*s, correspondingly, any task T_k of which is called an extra task. For the example in Fig. 2c, before t_3, P_0 is pivot and P_1 is extra. So all tasks of P_0 are pivot, the left resources from which are filled by extra tasks of P_1. Particularly, when P_0 is bottlenecked by network bandwidth, the idle CPU cores are adaptively used by P_1. In essence, extra pipelines seek to make full use of idle resources but to introduce fewer effects on pivot pipelines. In the two-layer scheduling framework, they are dispatched on the global layer, filling "holes" effectively on each local server. The details about these are described in Sections 5 and 6.

4.2 Rank-Based Preemption

Out-of-Order Cases. To decide the execution order of pipelines, the critical-path-first strategy has drawn much attention in the DAG scheduling [24], [28]. Because the length of a critical path determines the lower bound of the finish time of a query [24]. We also take such strategy to schedule pipelines. In fact, among all ready pipelines, the ones with higher rank could be scheduled first if resources are enough. But higher ranked pipelines would be blocked in two cases. (1) During execution, some pipelines with higher rank become ready after their predecessor pipelines are finished. While their needed resources are occupied by running but lower ranked pipelines, so they have to wait. (2) As time goes, the remaining work of running pipelines becomes less, decreasing their rank. At a certain point, their rank may become lower than the rank of some ready or extra pipelines. But the latters have to wait the completion of the formers. Such two out-of-order cases delay the finish time of the higher ranked pipelines.

Two Types of Preemption. According to the critical-path-first strategy, it is necessary to finish pipelines on a CP earlier, thus we design rank-based preemption. Once any

of the two out-of-order cases happen, the blocked higher ranked pipelines will become pivot to preempt more resources, while the running pipelines with lower rank will become extra and release their resources. Corresponding to the out-of-order cases, there are two types of preemption.

- Event-driven preemption: When a pipeline P_i becomes ready, and its needed resources are occupied by other running pipelines with lower rank, then the allocated resources of which should be preempted by P_i. Therefore, once a pipeline is ready, it is the time to check whether preemption should be issued or not.
- Time-driven preemption: The rank of running pipelines is always decreasing during execution. At some point, the realtime rank of pivot pipelines may decline so much that it is lower than that of others. In this case, other ready or extra pipelines with higher rank should preempt resources from the pivot ones. So we should periodically check the realtime rank of running pipelines to decide whether to issue preemption or not.

Preemption Conditions. The conditions triggering preemption between P_i and P_j are concluded as follows.

(1) $rank'(P_i) - rank'(P_j) > \delta$, where P_i is extra or ready, and P_j is pivot, δ means the positive cost threshold between P_i and P_j;
(2) there are resource conflicts between pipelines P_i and P_j;
(3) there are no resource conflicts between pipeline P_i and any pivot pipeline whose rank is higher than P_i.

We denote $rank'(P_i)$ to stand for the realtime rank of P_i, derived from $rank(P_i)$. It is expressed by

$$rank'(P_i) = rank(succ(P_i)) + cost(P_i) \times rest(P_i), \quad (3)$$

where $rest(P_i)$ represents the *remaining* execution progress of P_i, and $rest(P_i) \in [0, 1]$. In particular, the parallelizable pipelines are derived from multi-join queries. Each parallelizable pipeline can be expressed as {scan-[filter]-[probe]-build}, which includes zero or several filter or probe operators, ignoring exchange operators. Suppose a task T_j of P_i *scans* the data on the *j*th partition of a table that has total n partitions, then the average rest execution progress of all tasks can express the $rest(P_i)$. It is depicted as

$$rest(P_i) = \sum_{j=1}^{n} rest(T_j)/n, \quad (4)$$

where $rest(T_j)$ can be expressed as

$$rest(T_j) = RB_j/TB_j, \quad (5)$$

where TB_j and RB_j denote the number of *total blocks* and *remaining blocks* to process on the *j*th partition data, respectively.

Advantages of Preemption. The preemption can further guarantee the preference of pipelines on a CP to get more resources. As shown in Fig. 2c, since $rank'(P_1)$ is larger than $rank'(P_0)$ to a large extend at t_3, the path from P_1 to P_2 is a current CP. Thus preemption happens, where tasks of P_1 become pivot, then they get more resources. While P_0 becomes extra, and its tasks fill the idle resources from P_1. Finally, they complete almost at the same time, achieving *synchronous execution* [11] such that the predecessors of a pipeline can complete approximately at the same time. However, if there is no preemption shown in Fig. 2b, after P_0 finished, only P_1 uses all CPU cores, the efficiency of cores may be very low. In the meanwhile, the network is idle. This reveals preemption can increase the proportion of parallel execution period of pipelines to improve resource utilization, finally completing a query earlier.

To achieve such two dynamic techniques, there are two requirements. First, it requires efficiently adjusting the parallelism of running pipelines. Fortunately, such requirement can be met by our previous work, the elastic pipelining [4]. Specifically, its elastic iterator model provides an efficient way to change the parallelism of each task at run time. This is similar to the morsel-driven execution in Hyper [6] and work orders in Quickstep [29]. Second, it needs to adapt to the running process of a query. In detail, for the adaptive filling, since the actual resource requirements of pivot tasks are fluctuating on the fly, the extra tasks should adaptively fill the idle resource "holes". On the contrary, when pivot tasks ask more resources, the extra tasks have to release their resources in time. Otherwise, extra pipelines would damage the processing of pivot ones. As regard to the rank-based preemption, the our-of-order cases should be discovered in time and the status of corresponding pipelines should be quickly updated. Such adaptiveness is achieved in our optimal scheduling algorithm in Section 5 and the resource coordinator in Section 6.

5 Scheduling Algorithms

Scheduling algorithms decide the execution order of pipelines on the global layer. In this section, we first describe three baseline algorithms, including a serialized execution, an optimal static approach and a list scheduling method. Then we propose an optimal algorithm, List with Filling and Preemption Scheduler, based on the two techniques in Section 4.

Before describing algorithms, we define the statuses of P_i during its life cycle, which are listed below. The pipelines with the same status are stored in a sorted list ordering by its rank from higher to lower.

- Waiting (*W*): $pred(P_i)$ is not empty, and P_i has to wait until all its predecessors have been finished.
- Ready (*R*): $pred(P_i)$ is empty or any $P_j \in pred(P_i)$ has been finished. All ready pipelines are stored in a Ready List (*RL*).
- piVot (*V*): P_i is running in a pivot role. All pivot pipelines are stored in a piVot List (*VL*).
- Extra (*E*): P_i is running in an extra role. All extra pipelines are stored in an Extra List (*EL*).
- Done (*D*): P_i has been finished.

5.1 Baseline Algorithms

Serialized Scheduler (SS) dispatches pipelines one by one according to the physical plan with the consideration of data dependency. Thus at any time, only a pipeline is running. Each running pipeline can be accelerated by the elastic pipelining [4]. SS is widely adopted in traditional databases and OLAP systems.

Synchronized Phase (SP) [14] splits a PDG into shelves according to the critical path. Note that the path length is

measured by the number of nodes on the path, which is different from this paper. In each shelf, pipelines without dependency are executed concurrently. But resources are allocated statically according to estimated information. This is a common way to achieve the independent parallelism in previous static approaches [10], [12].

Algorithm 1. LFPS Algorithm (Key Part)

```
   Input: RL, EL, VL, δ
 1 foreach P_i ∈ RL ∪ EL in order do ▷ schedule pivot pipelines
 2   preemption ← true;
 3   foreach P_j ∈ VL in order do
 4     if rank'(P_i) − rank'(P_j) > δ then
 5       if serv(P_i) ∩ serv(P_j) ≠ ∅ then
 6         P_j: V→E;
 7         break;
 8     else
 9       if serv(P_i) ∩ serv(P_j) ≠ ∅ then
10         preemption ← false;
11         break;
12   if preemption is true then
13     P_i: R/E→V;
14 foreach P_j ∈ RL in order do        ▷ schedule extra pipelines
15   if could schedule P_j then
16     P_j: R→E;
```

Simple List Scheduler (SLS) is derived from the list scheduling algorithm [30], which is widely used for DAG scheduling problems [10], [12], [16]. The priority of every pipeline refers to its rank in Equation (2), reflecting the importance of the critical path. Besides, each pipeline can be executed in the elastic pipelining manner [4]. The steps of SLS are listed as follows.

(1) Select P_i with the highest rank from *RL*.
(2) Allocate servers to accommodate P_i.
(3) If without enough resources, the pipeline with next higher rank in *RL* becomes P_i, then go to (2).

SLS greedily dispatches the job with higher rank according to available resources. However, it cannot meet the two challenges of SRMP as well. There are not only the three types of idle resource "holes", but also the two out-of-order cases. Fortunately, SLS can combine with the two techniques, adaptive filling and rank-based preemption, to solve these problems, generating *List with Filling Scheduler (LFS)*, *List with Preemption Scheduler (LPS)* or *List with Filling and Preemption Scheduler (LFPS)*.

5.2 List with Filling and Preemption Scheduler

List with Filling and Preemption Scheduler (LFPS) is derived from SLS. It not only combines with the adaptive filling to fully use the idle resource "holes" but also incorporates the rank-based preemption to guarantee an optimal execution order of pipelines. Furthermore, it derives *Event-Driven LFPS (ED-LFPS)* and *Time-Driven LFPS (TD-LFPS)*, according to the two types preemption, namely event-driven preemption and time-driven preemption. They invoke Algorithm 1 to schedule pipelines.

Algorithm 1 describes LFPS in detail. First, pivot pipelines are scheduled according to SLS with preemption (lines 1-13). Then, extra pipelines are issued to fill idle resources (lines 14-16). But we do not dispatch too many extra pipelines (line 15). To issue enough extra pipelines but avoid too much scheduling overhead, we generally arrange at most 4 extra pipelines on each server, which is verified through experiments. As regard to preemption, if conditions are satisfied (lines 3-11), LFPS sends preemption commands to each server and changes the status of tasks (lines 12-13). In particular, resource conflicts between two pipelines are judged by whether their required servers are overlapped or not (lines 5, 9). Because we allocate resources to a pipeline in a coarse manner, taking a server, instead of a core as an allocation unit. The allocated servers can be efficiently used by pipelines in our work. As for δ, it will be analyzed through the experiments in Section 7.5.

The complexity of Algorithm 1 is $O(vm)$, where v is number of pipelines and m is the number of servers in a cluster. m stems from the fact that there is at most one pivot pipeline on a server. Since ED-LFPS invokes Algorithm 1 when a pipeline is ready at run time, the complexity of ED-LFPS is $O(v^2m)$. It is the same as that of some acceptable static list scheduling algorithms [16], [23]. While TD-LFPS invokes Algorithm 1 every τ time, its complexity is $O(vmc)$, where c is the invoking counts. Generally, if the cost time of Algorithm 1 is less than τ to some extent, the scheduling overhead cannot be a bottleneck.

From the perspective of each pipeline, the transition of its status can be expressed as a state machine in Fig. 4.

- W→R: The default status of each P_i is **W**aiting. Once all pipelines in $pred(P_i)$ are finished or $pred(P_i)$ is empty, then the status of P_i is changed to be **R**eady.
- R→V: If there are enough resources for P_i, then the status of a **R**eady pipeline P_i is updated to be pi**V**ot, and its tasks are sent to corresponding servers to run.
- R→E: When a **R**eady pipeline P_i starts to execute in an extra role, then its status is shifted to be **E**xtra, and its tasks are sent to corresponding servers to run.
- E→V: When an **E**xtra pipeline P_i gets preference from a pivot pipeline, its role is changed to be **P**ivot.
- V→E: When a **P**ivot pipeline P_i is preempted by other pipelines, its role is transformed to be **E**xtra.
- E→D: An **E**xtra pipeline is **D**one.
- V→D: A pi**V**ot pipeline is **D**one.

6 Resource Coordinator

Resource Coordinator (RC) dynamically adjusts resources for all tasks on each local server. It aims at achieving high resource utilization and efficiency of each server. In detail, RC schedules extra tasks to adaptively fill the idle cores from pivot tasks. On the other hand, RC checks the scalability bound of tasks to guarantee the efficiency of cores. Additionally, after receiving preemption commands, RC changes the role of specific pivot and extra tasks so that to move CPU cores to new pivot tasks from former pivot tasks. The layout of RC is depicted in Fig. 5 that manages CPU cores for pivot and extra tasks.

6.1 Allocating CPU Cores to Pivot Tasks

The pivot tasks on each local server are from the same pipeline and execute on different data partitions. The cores of each task are controlled by a RC. Like the elastic pipelining, the RC monitors instant processing rate of pivot tasks and improves the throughput of a pivot pipeline by reassigning cores periodically to the bottleneck tasks. Besides, the RC

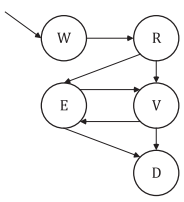


Fig. 4. State machine of a pipeline

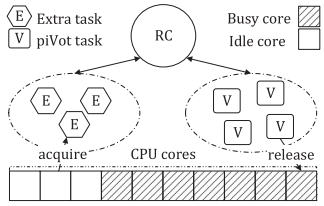


Fig. 5. Resource coordinator

avoids allocating excessive cores to each task according to its scalability. Since the scalability of a task is penalized by synchronization and resource contention. This is related to hardware, processing logic and implementation.

It is difficult to dynamically assess the scalability of core allocation to each task. Analogous to the analytical model [8] to estimate the scalability of tasks at compile time, we statically set an upper bound of cores for each task according to some experiments in prior (see Fig. 8 in [4]). Each task is combined by several operators, and the possible combination is limited. Besides, some critical parameters like input cardinality and tuple size are corrected before the task starts to execute in our dynamic scheduling. So the model works well in practice.

6.2 Allocating CPU Cores to Extra Tasks

On each local server, extra tasks adapt to the changing resource requirements from pivot tasks. As illustrated in Fig. 5, if pivot tasks release some cores, RC will allocate these idle cores to extra tasks. On the contrary, if pivot tasks require more cores, RC will force some extra tasks to release cores. Accordingly, in the former case, RC should decide which extra task can get those idle cores. In the latter case, RC should force some extra tasks to release cores. To achieve such adaptiveness among pivot and extra tasks, we design following rules.

(1) Complementary task first: Tasks are sensitive to different dimensional resources according to their logic. If the pivot tasks are sensitive to available network bandwidth, it is better to run extra tasks that are sensitive to CPU, and vice versa. Such action can not only make full use of multi-dimensional resources, but also bring less impact on pivot tasks. It is similar to overlapping CPU-bound and I/O-bound tasks in [31].
(2) Important task first: Similar to paying much attention to the critical path, the tasks of a higher ranked pipeline are executed first, then the pipeline would be accomplished earlier.
(3) Heavy task first: Tasks with heavy workloads to process may be the stragglers of a running pipeline. If these heavy tasks could get more resources, the throughput of the corresponding running pipeline would be improved.

RC allocates idle cores to extra tasks according to these rules in order on each local server. On the contrary, RC prefers forcing the extra task that is non-complementary, least important and least heavy to release resources, when pivot tasks ask for more cores.

In our implementation, the utilization of CPU is estimated by the number of busy cores, and the utilization of network is measured by whether exchange operators are busy sending or receiving data through network. To schedule complementary tasks, all extra tasks are further classified into a network-bound task set and a CPU-bound task set. Such decisions initially depend on whether a task contains exchange operators or not, then will be updated according to its resource sensitivity. Specifically, if the producing rate of data to shuffle is slow or the total data size to shuffle is small, like the intermediate data after the aggregation operator, the pipeline is not network-bound although it has an exchange operator. In each set, tasks are sorted by their importance and heaviness of workloads. The importance of task T_i is expressed by $rank'(P_j)$ in Equation (3), where T_i is a task of pipeline P_j. The heaviness of a task's workloads is measured by its left blocks of data to process. During execution, expanding and shrinking cores of a task benefit from the elastic iterator model [4] to satisfy the little overhead requirement. In addition, RC also takes the scalability bound into consideration to avoid allocating excessive cores to extra tasks.

7 Experiments

In this section, we first compare the performance of six kinds of scheduling algorithms on TPC-H benchmark. To further analyze the advantages of our optimal algorithm LFPS, we then evaluate the impact of adaptive filling and rank-based preemption in specific cases. At last, we go deep into some details in LFPS.

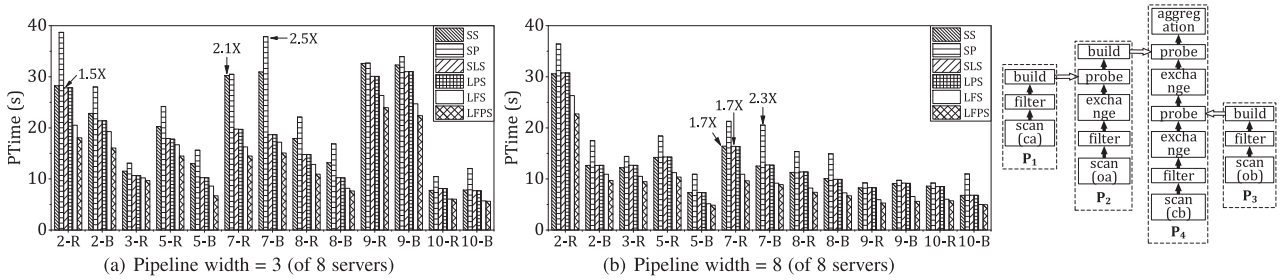
7.1 Environment and Dataset

The scheduling algorithms are implemented in our prototype system GINKGO [32], a main memory parallel database, tailored for efficient query analysis in a cluster. The experimental cluster is equipped with eight homogeneous servers, connected by one Gbps bandwidth network. Each server has 24 threads on 2 sockets, as well as 120 GB memory and runs a Centos 6.5 (64-bit) operating system. All experiments are conducted on TPC-H benchmark with scale factor at 100 in our cluster.

From the tree-like PDG of each query, we can see the exit pipeline cannot parallelize with other pipelines. For the example in Fig. 1b, P_2 should be executed serially after the completion of the two parallelizable pipelines, P_0 and P_1. So the finish time of a multi-join query (FT) equals the sum of the finish time of the exit pipeline (ETime) and the finish time of other parallelizable pipelines (PTime), i.e., FT = ETime + PTime. Although the exit pipeline cannot benefit from the independent parallelism within a query, it can parallelize with pipelines from other queries, forming inter-query parallelism in our future work.

7.2 Performance on TPC-H

We assess the performance of LFPS under TPC-H benchmark with scale factor 100, compared to SS, SP, SLS, LPS and LFS. In particular, all schedulers run pipelines using the elastic pipelining to make better use of resources, except SP allocates resources at compile time. We select seven multi-join queries, because they can provide multiple pipelines from their generated right-deep-tree or bushy-tree plans. Their PDGs are like Figs. 1b and 7. Regardless of the tree type, query 2, 3, 5, 7, 8, 9, 10 have 10, 3, 6, 6, 8, 6, 4 pipelines, respectively. The performance of these queries are examined on two kinds of pipeline width, *three* or *eight* of total eight servers. Because they can test the ability of scheduling algorithms to use the resources from the whole



(a) Pipeline width = 3 (of 8 servers) (b) Pipeline width = 8 (of 8 servers)

Fig. 6. The PTime of TPC-H queries [R: Right deep tree, B: Bushy tree]

Fig. 7. PDG of Test-Query

cluster. The partitioned data of tables are placed on servers in a round-robin fashion. But the small size tables, `nation` and `region`, are duplicated on each server.

From Table 2, we see the finish time of LFPS is up to 2.7 (2.6)X, 1.7 (1.4)X, 1.5 (1.4)X than SP, SS, SLS, respectively, when pipeline width is three (eight). The corresponding cases are in bold. Since FT = ETime + PTime, we analyze the performance of a query from the two aspects, namely ETime and PTime. For ETime, the execution of the exit pipeline in SS, SLS and LFPS is almost the same and costs less time than that in the static approach SP. Because the resources in a pipeline are carefully scheduled based on our previous work, the elastic pipelining. The idle resources "holes" from the exit pipeline can only be filled by the pipelines from other queries. (The values of ETime are not presented directly due to limited space.)

For PTime, LFPS is superior to all competitors as shown in Fig. 6. Its performance is up to 2.5 (2.3)X, 2.1 (1.7)X, 1.5 (1.7)X higher than SP, SS, SLS, respectively, when pipeline width is three (eight). All dynamic scheduling algorithms can get higher performance than the static method SP, because SP could not generate optimal decisions based on estimations at compile time. Sometimes SP is comparable to SS, because the latter cannot dispatch more pipelines to run when pipeline width is three. For dynamic approaches, SLS is superior to SS when pipeline width is three, because SLS parallelizes multiple pipelines to use the whole cluster resources. But it degrades to SS when pipeline width is eight where the PTime in SS and SLS is the same. Since SLS neither makes full use of idle resources at run time nor adapts to the changing of parallel execution. SLS is inferior to LFPS. The performance of LFPS is diverse for different queries due to their various workloads. The advantages of LFPS stem from the combined impact of the adaptive filling and rank-based preemption.

Comparing LFS and LPS with others, we see the individual effect of the two techniques. LFS is always faster than SLS due to filling idle resources by the adaptive filling. In some cases, it gets similar performance to LFPS, because the preemption does not work in those situations, such as for query-10. Importantly, LPS does not speedup the performance, which is similar to that of SLS. It is because executing which pipeline first does not matter if the order does not impact the resource utilization. The preemption works in LFPS because the execution order promotes the adaptive filling. Such internals are further illustrated in following two case studies, but their results extend to other queries.

7.3 Impact of Adaptive Filling

In this section, we test the adaptive filling works for the three types of resource "holes" via List with Filing Scheduler (LFS), which is the combination of SLS and the adaptive filling. Its effect is examined on TPC-H query-3, where tables `lineitem` (L) and `orders` (O) build hash tables, then do hash join with `customer` (C). In detail, P_1 = {scan (L)-filter-[exchange]-build}, P_2 = {scan(O)-filter-[exchange]-build}, P_3 = {scan(C)-[exchange]-probe-[exchange]-probe-aggeration}, where the exchange operators are controlled by changing table partition keys. The data of tables is hash partitioned on four servers in the cluster. We compare PTime, which represents the finish time of two parallel pipelines, P_1 and P_2, based on the execution of SS and LFS. Particularly, the PTime in SS is the sum of $FT(P_1)$ and $FT(P_2)$, which can be expressed separately. The speedup of LFS to SS is measured by $(FT_{SS}(P_1) + FT_{SS}(P_2) - PTime_{LFS})$ $/(FT_{SS}(P_1) + FT_{SS}(P_2))$.

Idle Resources from Different Dimensions. On a local server, tasks are sensitive to different dimensional resources. When tasks of a pipeline are sensitive to CPU, network (NET) maybe idle and vice versa. Extra pipelines fill these idle

TABLE 2
The Finish Time of TPC-H Queries (s) [R: Right Deep Tree, B: Bushy Tree]

PW	Scheduler	2-R	2-B	3-R	5-R	5-B	7-R	7-B	8-R	8-B	9-R	9-B	10-R	10-B
3	SS	31.45	26.46	24.96	50.61	27.2	42.24	**40.29**	48.06	42.47	40.8	41.92	51.47	47.91
	SP	44.23	33.78	43.91	80.67	41	59.8	**65.6**	94.83	77.49	47.1	47.71	67.68	65.14
	SLS	**31.06**	25.02	24.11	48.25	24.45	31.76	28	44.9	39.54	38.3	40.62	51.84	48.01
	LPS	**31.08**	25.05	24.11	48.15	24.45	31.75	28.01	44.91	39.55	38.31	40.64	51.85	48.01
	LFS	23.74	**22.92**	23.66	47.04	22.82	28.3	26.53	42.96	37.47	34.56	34.34	49.78	46.01
	LFPS	21.27	19.69	23.1	44.81	20.89	26.49	24.41	41.05	36.96	32.15	32	49.73	45.93
8	SS	32.61	14.84	19.17	30.71	14.56	24.56	18.61	37.44	37.51	11.33	**11.99**	26.81	28.07
	SP	38.72	20.04	30.45	**70.65**	27.41	45.03	35.83	74.38	70.14	14.23	14.64	39.29	37.5
	SLS	32.76	14.85	19.68	30.79	14.63	24.56	18.8	37.55	37.4	11.39	**12.04**	26.69	28.13
	LPS	32.78	14.88	19.67	30.8	14.64	24.55	18.81	37.57	37.41	11.39	**12.03**	26.7	28.13
	LFS	**28.32**	13.13	17.51	27.75	12.44	19.11	15.24	34.41	34.77	9.04	9.43	24.23	26.31
	LFPS	24.77	11.92	16.5	26.82	12.13	17.8	14.96	33.56	34.14	8.34	8.5	23.87	26.23

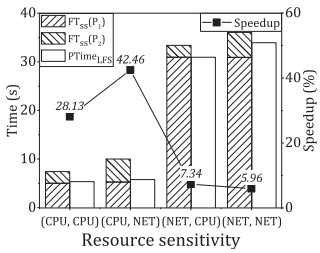


Fig. 8. Speedup for resource sensitivity

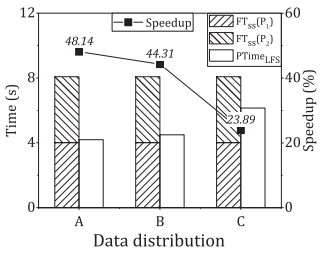


Fig. 9. Speedup for data distributions

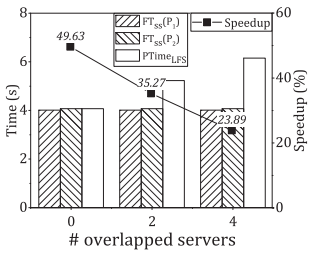


Fig. 10. Speedup for overlapped servers

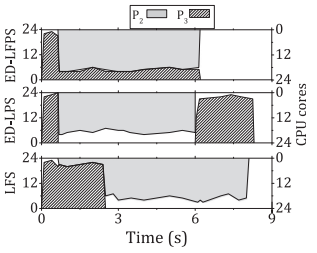


Fig. 11. CPU usage in ED-LFPS

resource "holes" from different dimensions adaptively. We control P_1 and P_2 are sensitive to CPU or NET by changing their data distribution in the cluster. Their resource sensitivity has four cases, (CPU, CPU), (CPU, NET), (NET, CPU) and (NET, NET). We compare the finish time and speedup of pipelines in SS and LFS for all cases.

In Fig. 8, we observe that the impact of filling in (CPU, NET) case is obvious, reaching 42.46 percent speedup. Because pivot P_1 and extra P_2 are sensitive to CPU and network, respectively. But it is not so obvious for (NET, CPU) case, because P_1 costs so much time than P_2 that it decreases the speedup. In other words, P_2 does not have enough work to fill idle CPU cores from P_1, so the speedup is lower. Besides, when both pipelines are sensitive to CPU cores, the speedup of LFS comes from improving CPU utilization and efficiency by extra pipelines. However, the result of (NET, NET) case indicates negligible improvement can be achieved when both pipelines are sensitive to network. This is limited by available network bandwidth. In summary, these experiments show the adaptive filling works well to make better use of different dimensional resources, confirming the importance of scheduling complementary tasks first on RC.

Load Imbalance. In general, workloads are partitioned unevenly across servers, stemming from data skew, unequal selectivity or operations. Consequently, during the execution of a pipeline, some servers are busy while others are free. Fortunately, these free servers can be adaptively filled by extra pipelines. Here, we set P_1 and P_2 build hash tables locally. We control the workloads on each server by changing the skewed data location of table partitions. As a result, the aggregated data size of `lineitem` and `orders` on each server is diverse. The selected data size of each partition from `lineitem` and `orders` is given in Table 3, where A, B and C mean three types of data distribution of two tables. The partitions of two tables with the same id are located on a server.

As illustrated in Fig. 9, the PTime in LFS is less than that in SS for all distributions. But LFS achieves different degrees of speedup for three distributions. In detail, for a pipeline running on skewed partitions, its finish time depends on the heaviest partition. Similarly, the PTime of two pipelines is determined by their heaviest aggregated data size among servers. From Table 3, we see the heaviest aggregated data size of distribution A, B and C is increasing, while the speedup is decreasing. Namely, for data distribution of multiple independent pipelines, the less heaviest aggregated data size among servers, the higher speedup. Particularly, the PTime in LFS speeds up by 48.14 percent for data distribution A.

TABLE 3
Partition Size of Two Tables (64 KB Blocks)

	Partition id	0	1	2	3
A	`lineitem`	293888	29388	2938	293
	`orders`	18	184	1843	18432
B	`lineitem`	293888	29388	2938	293
	`orders`	1843	18432	18	184
C	`lineitem`	293888	29388	2938	293
	`orders`	18432	18	184	1843

Idle Servers. If P_1 and P_2 are conflicted on several servers, then just one of them can be scheduled first, idling some servers in SS. While in LFS, these idle servers can be used by another pipeline as well. This extra pipeline runs not only on the idle servers, but also on some overlapped servers together with the pivot pipeline. We wonder how the number of overlapped servers effects the adaptive filling. We set the pipeline width of P_1 and P_2 to four, and change the overlapped servers to be 0, 2, 4. The two pipelines build hash tables locally, without exchange operators.

In Fig. 10, when servers are not overlapped, the PTime in LFS equals to $max(FT_{SS}(P_1), FT_{SS}(P_2))$, namely the maximal time of building hash tables for the two tables. But with more overlapped servers, the PTime in LFS approaches to $sum(FT_{SS}(P_1), FT_{SS}(P_2))$. Because in such case, there are less idle servers for extra pipeline P_2, then P_2 only fills the left resource "holes" from the pivot pipeline P_1 on overlapped servers. Similar to the case of load imbalance, the PTime in LFS also depends on the aggregated workloads on overlapped servers.

In summary, the adaptive filling improves query performance by filling three types of "holes". Its efficiency is related to the data distribution and resource sensitivity of pipelines. It is also an approach to alleviate the load imbalance problem during the execution of a query.

7.4 Combining Adaptive Filling and Rank-Based Preemption

Although the adaptive filling allows extra pipelines to fill the three types of "holes", is it able to fill all idle resources? We answer this question by comparing LFS with LFPS and List with Preemption Scheduler (LPS), which is a combination of SLS and the rank-based preemption. LPS derives ED-LPS and TD-LPS according to the event-driven preemption and time-driven preemption. Similarly, LFPS derives ED-LFPS and TD-LFPS. The Test-Query is listed below, where tables are used repeatedly to avoid data difference and estimation errors. The data of `customer` is distributed on servers (0, 1, 2, 3), partitioned by the attribute *custkey*, while the data of `orders` is distributed on servers (4, 5, 6, 7), partitioned by the attribute *orderkey*. The PDG of Test-Query is presented in Fig. 7, where P_1 and P_2 can parallelize with P_3, but P_2 and P_3

conflict on servers (4, 5, 6, 7). Additionally, $rank(P_1) > rank(P_2) > rank(P_3) > rank(P_4)$.

Test-Query: SELECT ca.nationkey, count(*) FROM customer ca, orders oa, orders ob, customer cb WHERE oa.custkey = ca.custkey and ob.orderkey = cb.custkey and cb.custkey = ca.custkey and oa.orderdate $\geq$? and ob.orderdate $\leq$? and ca.custkey $\leq$? and cb.custkey $\leq$? GROUP BY ca.nationkey;

ED-LFPS. The execution comparison of LFS, ED-LPS and ED-LFPS is as follows. Initially, P_1 and P_3 are running concurrently. After P_1 is done at 0.65 s, then P_2 is ready but conflicts with P_3. At this point, allocating resources to P_2 or P_3 is different in LFS, ED-LPS and ED-LFPS. Such difference can be seen in Fig. 11, which shows the resource usage during the execution of P_2 and P_3 on a conflicted server. In LFS, P_2 is running as an extra pipeline, using left idle cores from P_3. But this duration is rather short. After P_3 is done, some idle CPU cores are still wasted from P_2. In ED-LPS, P_2 preempts cores from P_3 that is going to sleeping. Such preemption splits the execution of P_2 into two parts, but idle resource "holes" remain. Differently in ED-LFPS, the idle cores from P_2 are filled by extra pipeline P_3 effectively after preemption. Finally, P_2 and P_3 are accomplished almost at the same time. In this progress, the occurrence of preemption lengthens the execution time of P_3, but decreases the finish time of P_2, finally completes the two pipelines earlier. *These experiments show the idle resources cannot be fully filled in the adaptive filling, and just taking the preemption cannot reduce the execution time of queries as expected. In fact, the two techniques promotes each other.*

TD-LFPS. Fig. 11 shows the case where P_3 can be finished by using the idle resources of P_2 after preemption in ED-LFPS, but what if P_3 has so heavy workloads that it is finished later than P_2? To answer this question, we examine ED-LFPS and TD-LFPS on new workloads by changing the conditions in Test-Query to increase the workloads of P_3. Execution details are observed from the resource usage on a conflicted server in Fig. 12. In LFS, the parallel execution duration of P_2 and P_3 is so short that it limits the work of the adaptive filling. After preemption in ED-LFPS, the idle cores from P_2 are occupied by P_3. But after P_2 is done, no pipeline employs the idle cores from P_3. *However, in TD-LFPS, multiple counts of preemption make that P_2 and P_3 are running in an interleaved manner, where P_2 and P_3 exchange the pivot and extra roles periodically according to their realtime rank. In the meanwhile, one extra pipeline fills idle resource "holes" from a pivot pipeline, achieving higher resource utilization. Finally, the PTime in TD-LFPS is reduced by 27 percent in contrast to LFS.*

In summary, the rank-based preemption allows higher ranked pipelines to get more resources. At the same time, the idle resources from the higher ranked pipelines are filled by lower ranked ones in the adaptive filling. Such combination in LFPS fully utilizes cluster resources so that to cut down the execution time of pipelines on the critical path, which determines the lower bound of the finish time of a query. Particularly, TD-LFPS is better than ED-LFPS because the former provides more chance to find the dynamic critical path in time. In practice, we merge such two types preemption in LFPS. In other words, LFPS checks preemption conditions periodically and at the point when a pipeline is ready.

7.5 Details in LFPS

Overhead Analysis. The main overhead of LFPS lays at controlling the thread number of tasks by RC on each server. Specifically, for the adaptive filling, RC adjusts the cores of extra tasks according to the changing requirements of pivot tasks. After preemption happens, RC moves cores from former pivot tasks to new pivot tasks. In the implementation of RC, expanding or shrinking threads for tasks is controlled by the elastic iterator model [4] with little cost. From its experimental study, expanding a thread only takes about 0.2 ms, and shrinking a thread costs at most 2 ms for complex tasks. Such advantages still stand in this paper. Besides, we pay attention to the adaptiveness of allocating cores on RC. It should expand or shrink a thread in time to avoid idle cores. For this purpose, we trace the idle cores in TD-LFPS in Fig. 12. The result is presented in Fig. 13. As expected, idle cores seldom occur during the execution in TD-LFPS. Particularly, the only idle cores mainly happen when preemption is triggered. But their duration is rather short, around tens of milliseconds. The benefits from such adaptiveness sufficiently overweight little overhead of controlling threads.

The Frequency of Preemption. The observed preemption depends on how often to check preemption conditions (This interval is defined as τ) and the rank gap δ of two pipelines in the preemption conditions. First, we examine the effects τ with $\delta = 0$. As reported in Fig. 14, the longer intervals, the fewer counts of preemption. When $\tau = 6$ s, the interval is so large that TD-LFPS becomes ED-LFPS, where the only preemption is triggered by the completion of P_1. But with the interval becoming shorter, it quickly reaches the maximal count of preemption. Such fixed count is related to the interval (about 0.5 ms) of reporting task progress from each worker server. In detail, after P_2 preempts resources from P_3, $rank'(P_3)$ is comparable to $rank'(P_2)$ after 3 s. Then at each checking, preemption happens, resulting in 7 counts of preemption in total. However, multiple counts of preemption increase little overhead, and it is impossible to set τ for every two pipelines to adjust preemption. In fact, τ is a system parameter for all queries. It is generally set to be 1 s in our system. Even so, there are also too many counts of preemption for several pipelines if their rank almost equals. In this case, we can reduce them by adjusting δ for each query. As presented in Fig. 15, the larger δ, the fewer counts of preemption, but the higher PTime. Observed from that curve, it is reasonable to set δ to $30\,\mathrm{percent}$ of the largest rank of all parallelizable pipelines in general. These settings work well to benefit from preemption in our experiments.

8 Discussion

Query Optimization. This paper focuses on resource scheduling that is the second phase of the two-phase optimization technique [14], but it is related to the first phase, namely generating optimal plans. Specifically, the tree type of physical plans decides how many pipelines can run in parallel. But the more paralleliable pipelines do not mean the less evaluation cost. There is a tradeoff between the parallelism and the cost of physical plans in order to reduce query execution time. We will investigate this tradeoff in our system to generate optimal bushy-tree or right-deep-tree plans in our future work.

In essence, generating optimal plans relies on cost estimation, which further impacts the rank computation of pipelines. Accurate estimation guarantees finding correct

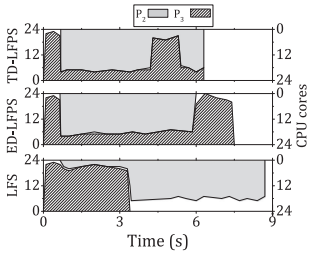


Fig. 12. CPU usage in TD-LFPS

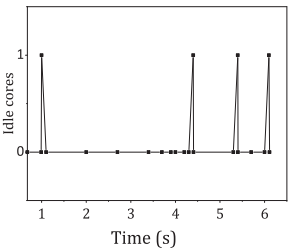


Fig. 13. Idle cores in LFPS

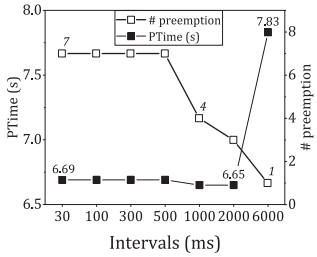


Fig. 14. The effect of intervals

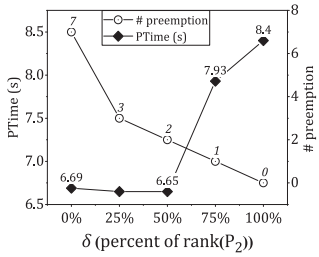


Fig. 15. The effect of δ

critical paths so that the pipelines on critical paths have higher priority to be completed earlier, which decides the lower bound of the execution time of a DAG. However, static estimation is error-prone. It may increase the query execution time if it misleads the critical paths. Such case can be alleviated in our LFPS, because it is resilient to estimation errors from current running and finished pipelines. At run time, LFPS uses these known information to correct the errors of rank during execution. Such benefits stand in the right-deep-tree plans, because all parallelizable pipelines are at the same level, and it is easy to get their accurate information at run time. But in bushy-tree plans, if there is complex dependency among pipelines, it has no idea to the errors resulting from waiting pipelines. In other words, it is too late to correct estimation errors when these waiting pipelines are going to running. So it is still necessary to accurately estimate cost of pipelines. Such issue will be investigated in our future work, probably building an analytic model inspired by [33].

Scheduling Multiple Queries. Scheduling multiple queries is a hierarchical problem that refers to intra-query parallelism and inter-query parallelism [34], [35]. The intra-query parallelism is the focus in this paper and has drawn much attention in previous work [10], [12], [14], [20], [36]. Our work carefully schedules multiple pipelines within a query in order to minimize the finish time of a query. *It can be applied to those scenarios that need to reduce the execution time of a single query*. For example, when taking First In First Out (FIFO) policy, minimizing the finish time of multiple queries requires efficiently executing a single. Besides, some priority scheduling policies need to improve the response time of the most prioritized query.

Our work can be easily extended to the inter-query parallelism, which is essentially to schedule multiple pipelines from different queries, rather than only one query in this paper. This scheduling problem also faces the two critical challenges illustrated in this paper, i.e., resource utilization and execution order of pipelines [31], [34]. These challenges can also be solved by our two techniques, adaptive filling and rank-based preemption, respectively. In detail, as demonstrated in Fig. 2, there are some idle resources when running a query. They can be adaptively filled by other less important queries so that to improve resource utilization. On the other hand, the preemption provides a chance to satisfy the fairness and priority requirements for scheduling multiple queries. Besides, our scheduling is extensible to various policies like [37], because it has the similar elasticity to provide a mechanism that dynamically determines resource allocation among concurrent pipelines. This also follows a well-known system design principle, separating mechanisms and policies. Consequently, our work can handle multiple queries from different users. Additionally, our scheduling is able to elastically share a cluster with other systems, like Hadoop and Spark. For example, the idle CPUs from other applications can also be adaptively filled by the extra pipelines in our system. Nevertheless, details and other challenges like reuse [38] and cache pollution [39] will be investigated in our future work.

9 Related Work

DAG Scheduling. DAG is a common way to represent dependency among jobs. Scheduling jobs with DAG constraint on multi-processor machines is often based on list scheduling algorithms. In detail, the Heterogeneous Earliest Finish Time (HEFT) [23] assigns the highest priority task to the processor that allows for the Earliest Finish Time of the task. Moreover, HEFT uses an *insertion policy* that tries to insert a task between two already scheduled tasks on a processor to use the idle time. To reduce completion time, the Predict Earliest Finish Time (PEFT) [16] introduces a look-ahead feature without increasing the time complexity associated with computation of an optimistic cost table, which is computed before scheduling. For data-parallel tasks, the Critical Path and Allocation (CPA) [28] is a compile-time heuristic, which allocates processors to jobs based on predicting execution time greedily. Then the tasks are scheduled on the available number of processors using a list scheduling algorithm. In summary, these scheduling algorithms schedule jobs statically on a single machine, assuming accurate cost or time estimation to make scheduling decisions.

Backfilling. Backfilling is a scheduling optimization, which makes better use of available resources by running jobs out of order. Specifically, a method based on the backfilling starts jobs one by one according to the priority list until a job could not get enough resources to start, but the scheduler makes resource reservation for these jobs, then schedules smaller priority jobs to fill the idle resources. Such reservation is the essence of backfilling that differs from list scheduling algorithms. Generally, different algorithms based on the backfilling try to make the balance between moving as many short jobs forward as possible in order to improve utilization and responsiveness and avoid starvation for large jobs [27], [40]. Besides, a multiple-queue backfilling approach [41] reduces the likelihood that short jobs get delayed in the queue behind long jobs. Even though the backfilling is often employed in static scheduling, its idea inspires us to propose the adaptive filling. Differently, this technique dynamically fills idle resources with best efforts.

Scheduling in Database. Task scheduling plays an important role in query evaluation, especially on many-core, large memory machines. Numerous work statically schedules tasks at query compile time, which consider various

parallelism strategies, such as partitioned parallelism [20], [21], pipelined parallelism [4], [17] and independent parallelism [10], [12]. As regard to the independent parallelism, the idea of static scheduling is as follows. At first, the data dependency of a plan is decoupled according to the non-overlapping shelves [14] or the synchronous execution [11]. Then static algorithms allocate resources to the tasks without data dependency, sharing multi-dimensional resources [10] as well as considering the scalability of tasks [8]. However, static scheduling may come out suboptimal decisions, due to error estimation, skewed data partition or unrealistic assumptions [13]. Therefore, a lot of dynamic strategies have emerged to finish a query as soon as possible by maximizing resource utilization. For example, dynamic load balance is taken in [5], [42], [43], particularly for NUMA multiprocessors [6], [7], [44]. Besides, overlapping I/O-bound and CPU-bound tasks [31] or active and inactive tasks [45] can also make better use of resources. In addition, [46] aims at mitigating data movement through an elastic core allocation mechanism on NUMA architecture. However, these dynamic methods aim at specific platforms. They cannot meet the challenges of main memory database clusters. In addition, they pay less attention to the critical path of jobs in a query and do not allow preemption in their scheduling. Nevertheless, most of these methods are orthogonal to our work to improve the resource allocation on each local server with NUMA architecture.

Elasticity. The elasticity of running pipelines (tasks) provides a new chance to study query scheduling, because it allows each task to adjust its intra-task parallelism, even to be zero. The elasticity exists in the morsel-driven execution in Hyper [6], work orders in Quickstep [29], [37] and elastic pipelining in our previous work [4]. However, the former two works try to fully use multiple cores on a single server, and [37] specifies the resource sharing among concurrent queries by enforcing policies using a learning-based approach. The elastic pipelining focuses on balancing the producing-consuming relationship among tasks within a pipeline in order to maximize its throughput. In contrast, this paper bases on the elastic pipelining to schedule multiple pipelines within a query to minimize the execution time of a single query. Our these two works aim at scheduling resources on a main memory database cluster and also perform well on a single server.

10 Conclusions and Future Work

We address the problem of scheduling resources to multiple pipelines of one query in a main memory database cluster. Unlike traditional static scheduling methods, we schedule resources to pipelines at run time based on two novel techniques. During the execution of pipelines, the adaptive filling improves resource utilization by issuing more extra pipelines to adaptively fill idle resource "holes". Besides, the rank-based preemption allows the pipelines on the critical path to get more resources. In particular, the occurrence of preemption helps the adaptive filling to further make better use of resources to finish a query earlier. Such two techniques perform well in LFPS. Comprehensive experiments show our work can reduce the finish time of parallelizable pipelines drastically than competitors under the workloads of TPC-H. In the future, as discussed in Section 8, we will study on generating optimal physical plans for distributed environments. Second, we plan to predict the cost of pipelines and combine it with query scheduling. Additionally, we will extend our approaches to schedule multiple queries in main memory database clusters.

Acknowledgments

This research was supported by the National Key Research & Development Program of China (No. 2018YFB1003400), and the National Natural Science Foundation of China (No. 61772204, No. 61732014).

References

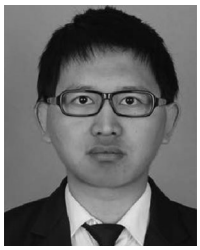
[1] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in *Proc. IEEE Int. Conf. Data Eng.*, 2011, pp. 195–206.
[2] L. Wang, M. Zhou, Z. Zhang, et al., "NUMA-Aware scalable and efficient in-memory aggregation on large domains," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 4, pp. 1071–1084, Apr. 2015.
[3] M. Armbrust, R. S. Xin, C. Lian, et al., "Spark SQL: Relational data processing in spark," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1383–1394.
[4] L. Wang, M. Zhou, Z. Zhang, Y. Yang, A. Zhou, and D. Bitton, "Elastic pipelining in an in-memory database cluster," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2016, pp. 1279–1294.
[5] L. Bouganim, D. Florescu, and P. Valduriez, "Dynamic load balancing in hierarchical parallel database systems," in *Proc. Int. Conf. Very Large Data Bases*, 1996, pp. 436–447.
[6] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann, "Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 743–754.
[7] L. Bouganim, D. Florescu, and P. Valduriez, "Load balancing for parallel query execution on NUMA multiprocessors," *Distrib. Parallel Databases*, vol. 7, no. 1, pp. 99–121, 1999.
[8] K. Krikellas, S. Viglas, et al., "Modeling multithreaded query execution on chip multiprocessors," in *Proc. Int. Workshop Accelerating Analytics Data Manage. Syst. Using Modern Processor Storage Archit.*, 2010, pp. 22–33.
[9] P. Jogalekar and M. Woodside, "Evaluating the scalability of distributed systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 11, no. 6, pp. 589–603, Jun. 2000.
[10] M. N. Garofalakis and Y. E. Ioannidis, "Multi-dimensional resource scheduling for parallel queries," *ACM SIGMOD Rec.*, vol. 25, no. 2, pp. 365–376, 1996.
[11] M. Chen, P. S. Yu, and K. Wu, "Scheduling and processor allocation for parallel execution of multi-join queries," in *Proc. IEEE Int. Conf. Data Eng.*, 1992, pp. 58–67.
[12] M. N. Garofalakis and Y. E. Ioannidis, "Parallel query scheduling and optimization with time- and space-shared resources," in *Proc. Int. Conf. Very Large Data Bases*, 1997, pp. 296–305.
[13] S. Zeuch and J. Freytag, "QTM: Modelling query execution with tasks," in *Proc. Int. Workshop Accelerating Analytics Data Manage. Syst. Using Modern Processor Storage Archit.*, 2014, pp. 34–45.
[14] K.-L. Tan and H. Lu, "On resource scheduling of multi-join queries in parallel database systems," *Inf. Process. Lett.*, vol. 48, no. 4, pp. 189–195, 1993.
[15] Z. Fang, C. Weng, L. Wang, and A. Zhou, "Parallelizing multiple pipelines of one query in a main memory database cluster," in *Proc. IEEE Int. Conf. Data Eng.*, 2018, pp. 1252–1255.
[16] H. Arabnejad and J. G. Barbosa, "List scheduling algorithm for heterogeneous systems by an optimistic cost table," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 3, pp. 682–694, Mar. 2014.
[17] G. Graefe, "Encapsulation of parallelism in the Volcano query processing system," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1990, pp. 102–111.
[18] D. A. Schneider and D. J. DeWitt, "Tradeoffs in processing complex join queries via hashing in multiprocessor database machines," in *Proc. Int. Conf. Very Large Data Bases*, 1990, pp. 469–480.
[19] P. Jogalekar and C. M. Woodside, "Evaluating the scalability of distributed systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 11, no. 6, pp. 589–603, Jun. 2000.

[20] A. Gounaris, R. Sakellariou, N. W. Paton, and A. A. A. Fernandes, "A novel approach to resource scheduling for parallel query processing on computational grids," *Distrib. Parallel Databases*, vol. 19, no. 2/3, pp. 87–106, 2006.
[21] A. N. Wilschut, J. Flokstra, and P. M. G. Apers, "Parallelism in a main-memory DBMS: The performance of PRISMA/DB," in *Proc. Int. Conf. Very Large Data Bases*, 1992, pp. 521–532.
[22] D. Kossmann, "The state of the art in distributed query processing," *ACM Comput. Surveys*, vol. 32, no. 4, pp. 422–469, 2000.
[23] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002.
[24] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 5, pp. 506–521, May 1996.
[25] A. Deshpande and J. M. Hellerstein, "Decoupled query optimization for federated database systems," in *Proc. IEEE Int. Conf. Data Eng.*, 2002, pp. 716–727.
[26] T. Z. J. Fu, J. Ding, and R. T. B. Ma, et al., "DRS: Dynamic resource scheduling for real-time analytics over fast streams," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2015, pp. 411–420.
[27] A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 6, pp. 529–543, Jun. 2001.
[28] A. Radulescu and A. J. C. van Gemund, "A low-cost approach towards mixed task and data parallel scheduling," in *Proc. Int. Conf. Parallel Process.*, 2001, pp. 69–76.
[29] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh, "Quickstep: A data platform based on the scaling-up approach," *Proc. VLDB Endowment*, vol. 11, no. 6, pp. 663–676, 2018.
[30] R. L. Graham, "Bounds for certain multiprocessing anomalies," *Bell Labs Tech. J.*, vol. 45, no. 9, pp. 1563–1581, 1966.
[31] W. Hong, "Exploiting inter-operation parallelism in XPRS," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1992, pp. 19–28.
[32] 2017. [Online]. Available: https://github.com/daseECNU/Ginkgo.git
[33] W. Wu, Y. Chi, H. Hacigümüs, and J. F. Naughton, "Towards predicting query execution time for concurrent and dynamic database workloads," *Proc. VLDB Endowment*, vol. 6, no. 10, pp. 925–936, 2013.
[34] J. L. Wolf, J. Turek, M. Chen, and P. S. Yu, "Scheduling multiple queries on a parallel machine," in *Proc. ACM SIGMETRICS Conf. Meas. Model. Comput. Syst.*, 1994, pp. 45–55.
[35] J. L. Wolf, J. Turek, M. Chen, and P. S. Yu, "A hierarchical approach to parallel multiquery scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, no. 6, pp. 578–590, Jun. 1995.
[36] B. Liu and E. A. Rundensteiner, "Revisiting pipelined parallelism in multi-join query processing," in *Proc. Int. Conf. Very Large Data Bases*, 2005, pp. 829–840.
[37] H. Deshmukh, H. Memisoglu, and J. M. Patel, "Adaptive concurrent query execution framework for an analytical in-memory database system," in *Proc. IEEE Int. Congr. Big Data Congr.*, 2017, pp. 23–30.
[38] A. Bär, L. Golab, S. Ruehrup, M. Schiavone, and P. Casas, "Cache-oblivious scheduling of shared workloads," in *Proc. IEEE Int. Conf. Data Eng.*, 2015, pp. 855–866.
[39] R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang, "MCC-DB: Minimizing cache conflicts in multi-core processors for databases," *Proc. VLDB Endowment*, vol. 2, no. 1, pp. 373–384, 2009.
[40] E. Shmueli and D. G. Feitelson, "Backfilling with lookahead to optimize the performance of parallel job scheduling," in *Proc. Workshop Job Scheduling Strategies Parallel Process.*, 2003, pp. 228–251.
[41] B. G. Lawson, E. Smirni, and D. Puiu, "Self-adapting backfilling scheduling for parallel systems," in *Proc. Int. Conf. Parallel Process.*, 2002, pp. 583–592.
[42] H. Lu and K. Tan, "Dynamic and load-balanced task-oriented database query processing in parallel systems," in *Proc. Int. Conf. Extending Database Technol.*, 1992, pp. 357–372.
[43] L. Wang, R. Cai, T. Z. J. Fu, J. He, Z. Lu, M. Winslett, and Z. Zhang, "Waterwheel: Realtime indexing and temporal range query processing over massive data streams," in *Proc. IEEE Int. Conf. Data Eng.*, 2018, pp. 269–280.
[44] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki, "Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores," *Proc. VLDB Endowment*, vol. 10, no. 2, pp. 37–48, 2016.
[45] I. Psaroudakis, T. Scheuer, N. May, and A. Ailamaki, "Task scheduling for highly concurrent analytical and transactional main-memory workloads," in *Proc. Int. Conf. Very Large Data Bases*, 2013, pp. 36–45.
[46] J. A. M. Simone Dominico, E. C. de Almeida, and M. A. Z. Alves, "An elastic multi-core allocation mechanism for database systems," in *Proc. IEEE Int. Conf. Data Eng.*, 2018, pp. 473–484.

Zhuhe Fang received the bachelor's degree from Zhejiang Normal University, in 2014. He is currently working toward the PhD degree at East China Normal University. His research interests include in-memory database and distributed system. He is a student member of the IEEE.

Chuliang Weng received the PhD degree from Shanghai Jiao Tong University, in 2004. He is currently a professor with East China Normal University (ECNU). Before joining ECNU, he worked with the Huawei Central Research Institute and was an associate professor with Shanghai Jiao Tong University. He was also a visiting research scientist with Columbia University. His research interests include parallel and distributed systems, storage systems, OS, and system security. He is a member of the IEEE.

Li Wang received the bachelor's and master's degrees from Guizhou University, in 2008 and 2011, respectively, and the PhD degree from East China Normal University, in 2015. He is currently a postdoctoral researcher with the Advanced Digital Sciences Center, Illinois, at Singapore Pte Ltd. His research interests include in-memory databases, distributed computing, indexing, and databases on novel hardware.



Huiqi Hu is currently an assistant professor with the School of Data Science and Enginnering, East China Normal University, Shanghai, China. His research interests mainly include database system and query optimization.

Aoying Zhou is a professor with East China Normal University. He is now acting as a vice-director of the ACM SIGMOD China and Database Technology Committee of the China Computer Federation. He is serving as a member of the editorial boards of the *VLDB Journal*, the *World Wide Web Journal*, etc. His research interests include data management for data-intensive computing, and memory cluster computing. He is a member of the IEEE.