




Parallel strategy for multiple scan operations with data replication

Xing Wei¹ · Huiqi Hu¹  · Huichao Duan¹ · Weining Qian¹ · Aoying Zhou¹

Received: 30 November 2017 / Revised: 23 May 2018 / Accepted: 19 July 2018 /
Published online: 13 August 2018
© Springer Science+Business Media, LLC, part of Springer Nature 2018

Abstract

To support the large-scale analytic for Web applications, the backend distributed data management system must provide the service for accessing massive data. Thus, the scan operation becomes a critical step. To improve the performance of scan operation, modern data management systems usually rely on the simple partitioned parallelism. Under the partitioned parallelism, tables are consist of several partitions, and each scan operation can access multiple partitions separately. It is a simple and effective solution for a single scan operation. In this paper, we consider managing multiple scan operations together, where the situation is no longer straightforward. To address the problem, we propose the parallel strategy to schedule batched scan operations together beyond the simple partitioned parallelism. For the sake of performance, first, we utilize replications to increase the parallelism and propose an effective load balancing strategy over replication nodes based on linear programming. Second, we propose an effective chunk-based scheduling algorithm for multi-threading parallelism on each node to guarantee all threads have even workloads under a qualified cost model. Finally, we integrate our parallel scan strategy into an open-sourced distributed data management system. Experimental evaluation shows our parallel scan strategy significantly improves the performance of scan operation.

Keywords Parallel scan · Load balancing · Parallel scheduling · Distributed data management system

This article belongs to the Topical Collection: *Special Issue on Web and Big Data*
Guest Editors: Junjie Yao, Bin Cui, Christian S. Jensen, and Zhe Zhao

✉ Huiqi Hu
hqhu@dase.ecnu.edu.cn

Extended author information available on the last page of the article.

1 Introduction

The backstage of all kinds of Web applications is the big data and the data management system. With the explosion of Web applications, the data become large-scale and the data management system is facing challenges. The distributed data management system can provide the storage service for the massive volumes of data. Catering to diverse Web services for users, modern data management systems are in urgent need of supporting large-scale analytic applications, which request data set by accessing gigabytes or terabytes of data through SQL or SQL-Like interface. Technology supports those large amounts of expensive processing capacity by keeping data within a parallel disk system consisting of tens to hundreds of machines interconnected with high-speed interconnection networks. As many components of the system contribute to the processing performance, the scan operation is considered as the most fundamental and crucial one.

A scan operation accesses the data from their resident nodes and delivers them to the target nodes. It has a wide spectrum of application scenarios in the distributed data management system. For example, to shop online, a customer has to run a scan query which accesses all related commodities among hundreds of millions of records. Enhancing parallelism of the scan operation is a matter of primary importance and many data management systems give the implementations such as *Greenplum Database* [26] and *PostgreSQL 9.x* [22]. The big solution can be called *partitioned parallelism*. For a scan operation, it divides data into several parts and separately scans them with different resources. In the distributed data management systems (e.g. *Greenplum Database* [26]), data are partitioned and processed on different nodes in parallel. While for the non-distributed systems (e.g. *PostgreSQL 9.x* [22]), a configurable number of threads are usually utilized to scan different parts of data (sometimes known as *intra-scan parallelism* [13]). Regardless of the partitioned parallelism on distributed nodes or multi-threading method for one scan operation, its essence is simple for a single scan operation: dividing data and scanning them in separate.

In this study, we devise and implement a parallel strategy for multiple scan operations on the distributed data management systems. Instead of individually scheduling these scan operations, we consider managing them as a batch. Obviously, for a query plan, we have the possibility to execute several scan operations in parallel. For example, when joining over multiple tables, it is required to pull remote data to build (more than one) hash tables simultaneously. As we consider multiple scan operations together, the previous parallel strategy is no longer effective. Under the bad situations, if we allocate a fixed number of threads for each scan operation, it may exhaust resources when processing a large number of scan operations concurrently. Thus the distributed database systems need a justified parallel strategy to execute multiple scan operations. To enhance the performance, we further consider the optimization of our proposed parallel strategy from two aspects:

Firstly, we consider parallelism between storage nodes. On this hand, we investigate how to utilize replicated data to increase the parallelism. As the gold-standard method to provide the high availability of the distributed data management systems, replications are usually used to keep the system be in service when one node crashes [31]. Clearly, the scan operation can benefit from the usage of replications. If multiple scan operations run on a single storage node, they will consume too many resources such as CPU cycles, disk I/O and bandwidth of memory bus. However, the distributed data management systems can avoid overloading one specific node by assigning several scan operations to other replication nodes. Actually, we recognize that the usage of replications can further increase the scan parallelism. For a single scan operation, the performance can possibly be promoted by dividing the scan operations into parts and run them separately in different replications. Following this perspective, it

requires one effective method to divide the scan operations and balance them on different nodes.

Secondly, we consider multi-threading parallelism on each storage node. Scan tasks can be further benefited from multi-threading parallel process. When fetching a number of scan tasks, we have countless ways to schedule and process them on multiple threads. However, the computation resource is so precious that varies scheduling method yield different expenses. Therefore, providing fine-grained multi-threading task scheduling (for the scan operations) is required. Compared with task allocation on parallel nodes, the problem of multi-threading scheduling is more challenging. As discussed in Section 4, we should consider various issues including thread switching cost, load balance and parallel scalability in the scheduling algorithm.

Following the above two thoughts, we propose a strategy which offers two kinds of parallel granularity in the present work. First, we consider using data replications to support parallel scan. We run the scan operations on all replication nodes in parallel. The challenge is how to allocate the appropriate number of scan tasks to different replication nodes with the knowledge of data distribution. Therefore, an effective load balancing method is proposed on the basis of linear programming. Second, we further investigate how to implement multi-threading scheduling for scan tasks on each node. We devise a chunk based approach and propose a cost model to analyze scheduling expenses. Since achieving an optimal scheduling strategy is NP-hard [9], we propose an effective greedy algorithm to make each thread has similarly even cost.

In summary, we have made the following contributions in this paper:

- (1) We consider the parallel strategy to schedule batched scan operations together beyond the simple partitioned parallelism. We further utilize replications to increase the parallelism and propose an effective load balancing method based on linear programming.
- (2) We propose an effective chunk-based scheduling algorithm for multi-threading parallelism on each node which makes each working thread have the even expense under a qualified cost model.
- (3) We implement and integrate the proposed parallel scan strategy into an open-sourced distributed data management system *Oceanbase* [21]. Experimental results show it improves the overall performance of scan operations on TPC-H by up to 78.5% and keeps the load balance of each node when being compared with the original system which only owns the simple partitioned parallelism.

The remainder of this paper is organized as follows: we describe the framework that implements our parallel strategy in Section 2. In Section 3, we present how to parallelize scan operations on multiple replication nodes. Section 4 introduces the multi-threading scheduling strategy on each node. We provide an experimental evaluation in Section 5. The related work is described in Section 6. The Section 7 is the conclusion of this paper.

2 Overview

In this section, we provide a brief overview of the framework that supports our parallel strategy. The architecture is illustrated in Figure 1, and we also list the notations (Table 1) for the ease of understanding.

Data storage The storage is row-oriented and records are organized by the primary-key in ascending order. The data take the horizontal partition for a basic unit. For example, each

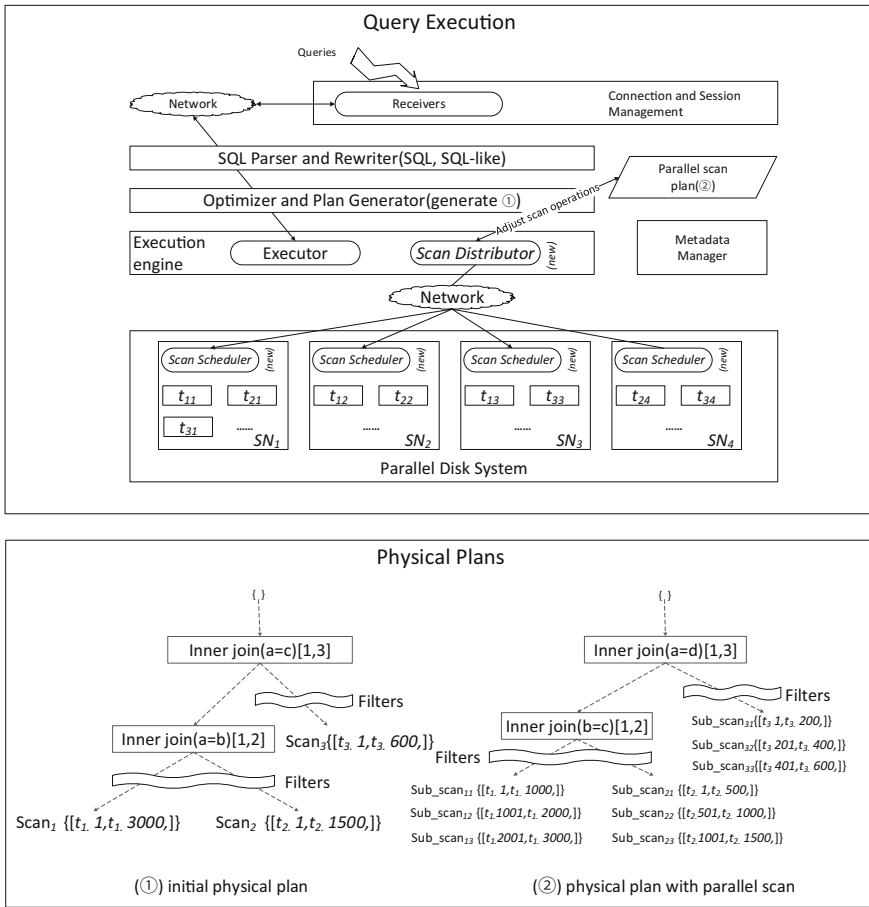


Figure 1 Framework of parallel scan

data table can consist of one or more partitions. We denote a partition of one specific table as a *tablet* t . The table is divided into multiple *tablets* based on the primary-key of the data. Therefore, each tablet records the table’s data in a continues range of primary-key interval. In each *tablet*, indexes (e.g. B+-tree [17], block index [17]) are constructed to support fast data access. To this end, if a request aims to scan a sub-range of the tablet, it is not necessary to probe the all *tablets*, instead, it directly retrieves the target data via the index with fewer costs.

Tablets are residents on a cluster of storage nodes (servers) which are denoted by $\{S_1, S_2, \dots, S_m\}$, where S_j is a storage node and m is the total number of storage nodes. A storage node can contain several tablets. Moreover, a tablet can have multiple replications which resident on different storage nodes. We use $t_{i,j}$ to represent a tablet t_i which is resident on a storage node S_j . All the replications of a tablet contain exactly the same content that there is no distinction for them to provide service. In this paper, we adopt identical deployment of replications that all tablets have the same number k of replications. Using the varying number of replications does not affect our method. For instance, Figure 1 illustrates four storage nodes $\{S_1, S_2, S_3, S_4\}$ and three tablets $\{t_1, t_2, t_3\}$, where each tablet has

three replications. \mathcal{S}_1 has three replications: $t_{1,1}$, $t_{2,1}$ and $t_{3,1}$; \mathcal{S}_2 has two replications: $t_{1,2}$ and $t_{2,2}$; \mathcal{S}_3 has two replications: $t_{1,3}$ and $t_{3,3}$; \mathcal{S}_4 has two replications: $t_{2,4}$ and $t_{3,4}$. Three replications of each tablet t_i are stored on \mathcal{S}_1 , \mathcal{S}_2 , \mathcal{S}_3 and \mathcal{S}_4 respectively.

Scan operation Tablets are the basic units to perform a scan. When a query comes, a query planner (optimizer) will generate its query execution plan. The requests of scan operations which are allowed to probe tablets in parallel are extracted as a set of requesting data contents within several tablets t , which is denoted as $\mathcal{Q}_s = \{[t_1.s, t_1.e], [t_2.s, t_2.e], \dots, [t_n.s, t_n.e]\}$. Where \mathcal{Q}_s is a batch of scan operations offered by the query execution plan, n is the number of requesting tablets, $[t_i.s, t_i.e]$ is the range of primary-key required by a specific scan operation on tablet t_i . For example, as described in Figure 1, When receiving a query, the plan generator will generate an execution plan with a group of scan operations ($\mathcal{Q}_s = \{[t_1.1, t_1.3000], [t_2.1, t_2.1500], [t_3.1, t_3.600]\}$).

Framework Then, we describe our framework for the parallel scan operations. As illustrated in Figure 1, it contains two main modules:

- (1) *Scan Distributor*. When a scan request \mathcal{Q}_s is sent to a *scan distributor* for scheduling, the scan distributor will handle the scan request in two steps: Firstly, it divides each scan operation of \mathcal{Q}_s into multiple sub-scan operations, where its required range of primary key is further divided into smaller disjoint intervals. For instance, in Figure 1, the scan distributor decomposes the three scan operations on t_1 , t_2 and t_3 into further nine parts, which is shown as $\{[t_1.1, t_1.1000], [t_1.1001, t_1.2000], [t_1.2001, t_1.3000], [t_2.1, t_2.500], [t_2.501, t_2.1000], [t_2.1001, t_2.1500], [t_3.1, t_3.200], [t_3.201, t_3.400], [t_3.401, t_3.600]\}$. Then, it assigns all the sub-scan operations of \mathcal{Q}_s and delivers them into target storage nodes to process them. All replications can be used in parallel to process these sub-scan operations. For example, in Figure 1, scan operation $[t_1.1, t_1.3000]$ is divided into $\{[t_1.1, t_1.1000], [t_1.1001, t_1.2000], [t_1.2001, t_1.3000]\}$, each of them becomes a sub-scan operation and is processed on different replications of t_1 (i.e. $t_{1,1}$, $t_{1,2}$ and $t_{1,3}$) respectively. Formally, we call such sub-scan operations generated by the scan distributor *scan tasks*.
- (2) *Scan Scheduler*. There is a *scan scheduler* in each storage node. After receiving the scan tasks (sub-scan operations) from the scan distributor, it is responsible for scan scheduler to schedule and process them with multiple threads. The scan scheduler controls a thread pool, where each thread works continuously to execute these scan tasks. During the process of one scan task, it actually runs several operations in physical, including accessing target data from disk, processing the predict conditions and data serialization. The scan scheduler manages these working threads and generates different scheduling strategies for the scan tasks.

The two keys to our parallel strategy in the framework is how to design the scan distributor and how to schedule the scan tasks with multiple threads on each storage node. Nextly, we introduce our solutions in Sections 3 and 4 respectively.

3 Parallel between replicated nodes

A set of scan operations that belong to a query execution plan is managed by the scan distributor. In this section, we discuss how the scan distributor generates scan tasks and assigns them to storage nodes. Our main goal is to shorten the latency by parallelizing the

Table 1 Notations

t_i	A tablet
$\{\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_m\}$	set of storage nodes, where \mathcal{S}_i is a storage node that can keep multiple tablets.
$t_{i,j}$	a replication of tablet t_i that residents on \mathcal{S}_j .
C_i	cost of scan tasks on \mathcal{S}_i
$[t_i.s, t_i.e]$	a scan operation on t_i from $t_i.s$ to $t_i.e$.
w_i	weight of tablet t_i .
$x_{i,j}$	portion of request that t_i is assigned to node \mathcal{S}_j .
n'	number of assigned scan tasks on a specific node
N	number of working threads on each node

scan tasks. It is easy to achieve the goal by dividing a scan operation into several sub-scan tasks and running them on different replications in parallel. The challenge here is to take the load of each *storage node* into consideration and generate a justified allocation strategy for each *storage node*. We briefly introduce the motivation in Section 3.1, and then we formally describe the method in Sections 3.2 and 3.3.

3.1 Motivation of scan distributor

Considering the storage nodes $\{\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_m\}$ that serve massive data. Our goal is to generate an efficient parallel scan strategy for a query execution plan with the minimum scan time. Since all the storage nodes run scan tasks in parallel, the overall scanning time mainly depends on the slowest storage node that performs the most tasks. Take Figure 1 as an example, there are three replications for each tablet. Just as query \mathcal{Q}_s comes, if the *scan distributor* does not carefully consider load balance and simply divides each scan operation into three equal-range (sub) scan tasks. After dividing scan operations, *scan distributor* will dispatch these scan tasks to the target storage node. These scan tasks will fall into four groups: \mathcal{S}_1 has $group_1 = \{[t_1.1, t_1.1000], [t_2.1, t_2.500], [t_3.1, t_3.200]\}$, \mathcal{S}_2 has $group_2 = \{[t_1.1001, t_1.2000], [t_2.501, t_2.1000]\}$, \mathcal{S}_3 has $group_3 = \{[t_1.2001, t_1.3000], [t_3.201, t_3.400]\}$ and \mathcal{S}_4 has $group_4 = \{[t_2.1001, t_2.1500], [t_3.401, t_3.600]\}$. Obviously, $snode_1$ is assigned to most tasks and spends the most time on executing tasks. The unbalanced distributed strategy can cause the *wooden bucket effect*, where the overall scanning time mainly depends on the storage node with the highest overhead. If one node undertakes overmuch tasks under the parallel environment, the overall scanning performance will be limited to the slowest node.

Therefore, the motivation of designing scan distributor is to make scan tasks well-distributed and each storage node has the similar quantity of workload. In particular, *we seek a strategy to minimize the workload of the node which has the highest load among all the nodes.*

3.2 Formulation of load balance

We first introduce how to split scan operations into scan tasks, and then address the problem of how to minimize the workload of the storage node which performs the most scan tasks. Figure 2 illustrates the process that is exemplified in Figure 1.

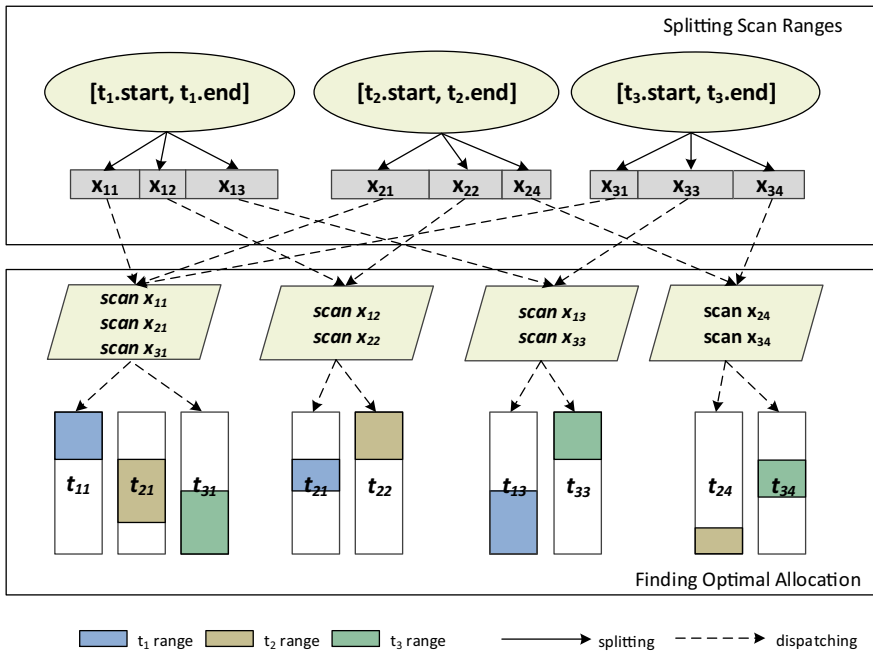


Figure 2 Process of scan requests

Splitting scan ranges First of all, we take *weights* to denote the workloads of these scan operations since they have varies workloads (i.e, the volume data they requested are different). The weight of the scan operation is denoted by w_i , and its value ranges from 0 to 1. In practice, we can estimate the row number (rn_i) of a scan operation based on its primary-key or the statistics (see Section 3.3) and obtain the row size (rs_i) of each tablet through the table schema. Thus, the required size for the scan operation t_i is $rn_i \cdot rs_i$. Besides, w_i can be set by normalizing all these computed sizes.

Next, recall that for each tablet t_i , we have k replications distributed on m storage nodes. Let $\{t_{i,j_1}, \dots, t_{i,j_l} t_{i,j_k}\}$ denote those replications, where t_{i,j_t} ($1 \leq j_t \leq m$) represents a replication of t_i is resident on storage node S_{j_t} . Regarding a scan operation with a range of $[t_i.s, t_i.e]$, we can split it into k continuous sub-ranges, where a sub-range corresponds to a scan task. Therefore, each sub range represents a portion of $[t_i.s, t_i.e]$, which is denoted by $x_{i,j}$. Obviously, if tablet t_i has no replication on node S_j , the storage node cannot contribute to any scan tasks of $[t_i.s, t_i.e]$, that is $x_{i,j} = 0$. Otherwise, if the S_j has a replication of t_i , we have $0 \leq x_{i,j} \leq 1$. Formally, we have:

$$\sum_{j=1}^m x_{i,j} = 1 \quad \text{w.r.t.}$$

$$0 \leq x_{i,j} \leq 1; \quad \text{if } t_i \text{ has a replication on } S_j$$

$$x_{i,j} = 0; \quad \text{if } t_i \text{ has no replication on } S_j.$$

For example, in Figure 2, there are three replications for each *tablet*. To assign a scan task to one replication, a scan operation is split into three *sub-scan* tasks. Therefore, the $[t_1.s, t_1.e]$ served as a scan range, should be transferred into three continuous *sub-scan* tasks with own portions, including $x_{1,1}$, $x_{1,2}$ and $x_{1,3}$. It indicates that scan operation for t_1

is split into three sub-scan tasks (according to x_{11}, x_{12} and x_{13}), whose ranges constitute a complete range of $[t_1.s, t_1.e]$. These sub-scan tasks for tablet t_1 are assigned to node $\mathcal{S}_1, \mathcal{S}_2$ and \mathcal{S}_3 . The other two scan operations can also be transferred into multiple scan tasks with the same method.

Finding optimal allocation The next step is to compute the scanning cost of each storage node. Based on the weight W_i of the scan operation and its assigned portions $X_{i,j}$, we can specify the expected cost C_j of scan tasks that run on the storage node \mathcal{S}_j , which is calculated as:

$$C_j = \sum_{i=1}^n w_i \times x_{i,j}.$$

For instance, the cost on \mathcal{S}_1 is $C_1 = w_1x_{1,1} + w_2x_{2,1} + w_3x_{3,1}$. Since the purpose of our strategy is to make the scanning cost of the highest overhead node as small as possible, we should figure out a group of $x_{i,j}$ for m storage nodes to realize the goal that can be stated as follows:

$$\text{Minimize } \left(\text{Max}_{j (1 \leq j \leq m)} \{C_j = \sum_{i=1}^n w_i x_{i,j}\} \right), \text{ subject to } \begin{cases} 1 = \sum_{j=1}^n x_{i,j} \\ 0 \leq x_{i,j} \leq 1 \end{cases}$$

In fact, from above conditions, it is a variant form of linear programming problem [29] to generate an optimal strategy by giving a set of $x_{i,j}$, which can minimize the cost of the highest overhead node. The problem can be solved through linear programming by assuming that the highest cost of the storage nodes is Z and transforming the object into following equations:

$$\left\{ \begin{array}{l} Z \geq C_1 = \sum_{i=1}^{m_1} w_i x_{i,1} \\ Z \geq C_2 = \sum_{i=1}^{m_2} w_i x_{i,2} \\ \dots \\ Z \geq C_n = \sum_{i=1}^{m_n} w_i x_{i,n} \end{array} \right. \quad \begin{cases} 1 = \sum_{j=1}^n x_{i,j} \\ 0 \leq x_{i,j} \leq 1 \end{cases}$$

By solving the above equation through linear programming, we obtain the optimal assignment, i.e. the portions of the scan tasks that belong to one scan operation should be provided to support the procedure of the division.

Example 1 We take the Figure 3 as an example. Firstly, a set of scan requests ($Q_s = \{[t_1.1, t_1.3000], [t_2.1, t_2.1500], [t_3.1, t_3.600]\}$) are sent to the *scan distributor* to be processed. Then, the *scan distributor* acquires the metadata about the scan tasks to generate the computational scan cost (C_1, C_2, C_3 and C_4) on different *storage nodes*. Next step is to take scan costs into a linear programming problem according to the given conditions. Finally, LPSolver module provides the splitting portions which are used to generate scan tasks ($X_{11} : 0.2489, X_{12} : 0.3475, X_{13} : 0.4036, \dots, X_{34} : 0.4175$).

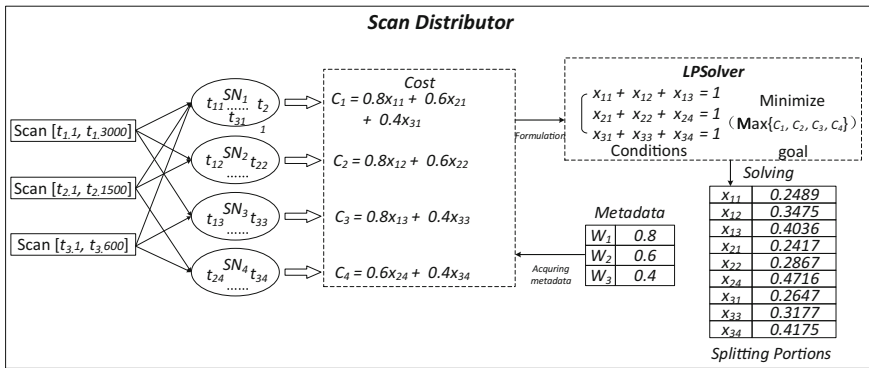


Figure 3 Example of generating optimal distribution

3.3 Generate scan tasks

Despite the optimal portions for the division, to generate scan tasks, another facing question is how to transform the portion $x_{i,j}$ into a specific scan task. In particular, given a scan operation $[t_i.s, t_i.e]$, we divide the range into sub-ranges based on the computed portions. However, the keys between start key and end key may be not always continuous. i.e., some the keys are missing within the interval. For instance, if a scan operation $[t_1.1, t_1.1000]$ is divided into two equal parts according to the portions(0.5 and 0.5), the range $[t_1.1, t_1.500]$ may have only 10 rows of data and range $[t_1.501, t_1.1000]$ may have 500 rows of data under a skew distribution of primary-keys. *The question is how to divide these scan operations according to the optimal portion as even as possible.* We have two strategies:

(1) *Range Based Partition.* In this setting, we do not use any meta information but simply assume the keys which are uniformly distributed. Therefore, sub-ranges are computed directly according to the portions. Formally, for each scan operation $[t_i.s, t_i.e]$, a replication on storage node S_j will receive a scan task with the corresponding sub range.

$$\left[t_i.s + \sum_{t=0}^{j-1} x_{i,t} \cdot (t_i.e - t_i.s), t_i.s + \sum_{t=0}^j x_{i,t} \cdot (t_i.e - t_i.s) \right]$$

(2) *Histogram Based Partition.* To get more concise partitions, we can use the histogram from database statistics. We employ the equal-depth histogram on the primary-key of each tablet. We assume there are m buckets($\{b_1, b_2, \dots, b_m\}$) in scan range($[t_i.s, t_i.e]$) of tablet t_i . Each bucket b_i has a range($[b_i.s, b_i.e]$). Formally, the range that S_j will receive is presented as follows:

$$\left[b_{\lceil \sum_{t=0}^{j-1} x_{i,t} \cdot m \rceil} .s, b_{\lceil \sum_{t=0}^j x_{i,t} \cdot m \rceil} .e \right]$$

Example 2 As described in Figure 4, the actual row number with range $[t_1.1001, t_1.3000]$ is 500. If *scan distributor* takes the range based partition strategy, it will generate three scan tasks which are $[t_1.1001, t_1.1800]$, $[t_1.1801, t_1.2600]$ and $[t_1.2601, t_1.3000]$. Their number of rows are 300, 200 and 100, which are not a good satisfaction of the given ratio

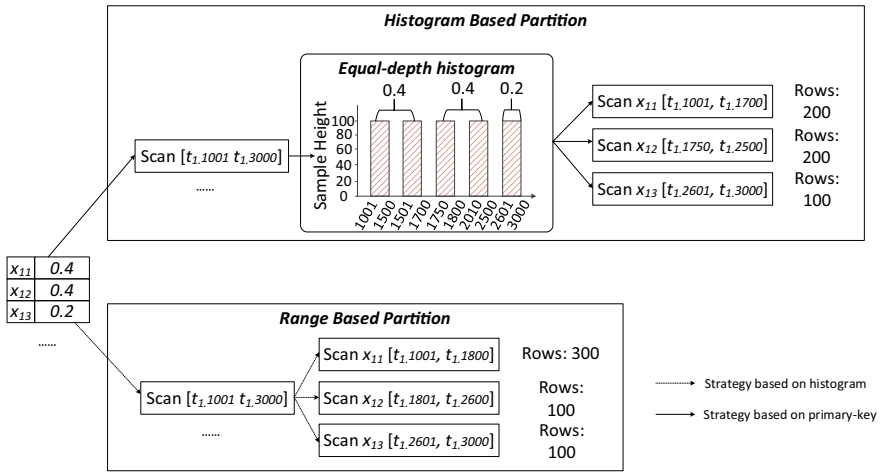


Figure 4 Example of generating sub-scan tasks

2 : 2 : 1. However, the histogram based partition strategy takes a group of continuous buckets to build a sub-scan range. Before building first scan range, we should get needed buckets number by the calculation: $bucket_number \times x_{i,j} (5 \times 0.2 = 1)$. Then, we select first two buckets($[b_1.1001, b_1.1500], [b_2.1501, b_2.1700]$) and choose two boundary values($b_1.1000, b_2.1700$) to form the first sub scan range($[t_1.1001, t_1.1700]$) which has 200 rows data. Based on this method, *scan distributor* can also generate other two scan tasks. The row numbers of these three sub-scan tasks are satisfied with the given ratio 2 : 2 : 1.

4 Parallel task scheduling

Apart from considering the load balance between all storage nodes, the next question is how to schedule these (sub) scan tasks on each storage node by leveraging multi-threading processing. Nextly, we introduce the general architecture of scan scheduler in Section 4.1 and discuss two alternative strategies in Sections 4.2 and 4.3.

4.1 Scheduling framework

We consider that each storage node has a CPU with many cores, which allows more than one task to be tackled by different threads at the same time. Figure 5 explains the architecture for task scheduling on each storage node.

Dynamic job scheduling is adopted in each storage node. Each scheduler owns a thread pool which contains a fixed number of working threads. A working thread is responsible for executing a specific *scan task*. A scan task is to pull data within an arbitrary range from disk into the memory buffer. In the meanwhile, it also needs to process them with predicate filters, make data serialization and then send them to the destination via the network. The scan scheduler utilizes all those working threads in a round robin manner. Firstly, all the scan tasks are registered into a task queue. Then, if a working thread becomes available (i.e. once it accomplishes a scan task), it will claim another undo task from the queue. Based

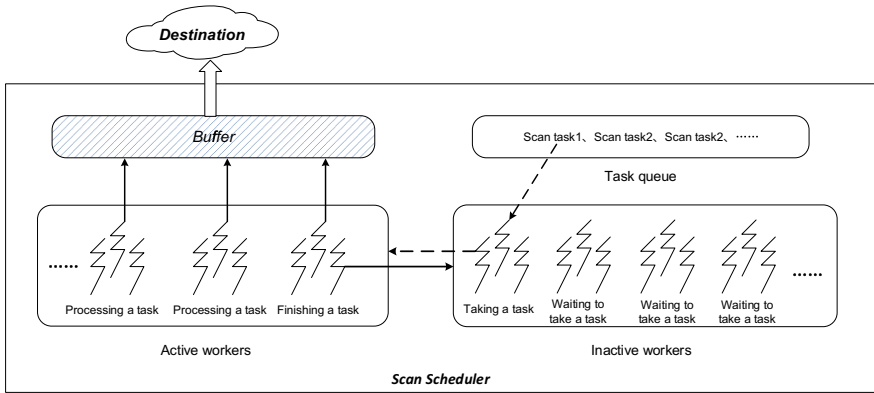


Figure 5 Execution of sub-scan tasks

on this job scheduling framework on one *SN*, we will subsequently discuss two issues that affect the performance.

- (1) *Unbalance Workload.* To process a group of scan tasks on one storage node. A naive scheduling strategy is to assign one available working thread for one scan task. Clearly, one scan task corresponds to one scan job. Apparently, it is an inefficient strategy for the multi-core CPU to execute tasks without taking the workload of each thread into consideration. As illustrated in Figure 6a, the CPU (4 threads) is not fully utilized since the working threads which run the small scan jobs are always idle wait for other jobs to be done.
- (2) *Job Switching Costs.* A method to solve the above problem is to further partition the scan tasks into smaller scan jobs (by further splitting the range of primary key) and keep the workloads balanced. However, switching scan jobs on a working thread will generate additional cost (e.g., thread context switching [32], cache missing [25] or disk addressing [7]). Besides, as the overall number of working thread is limited, over fine-grained partition will involve a lot of switching overhead. Figure 6b shows a quite elaborate scheduling as the workload over all working thread are assigned to be equal. Task t_6 is divided into four scan jobs, t_5 is divided into three parts and t_2 generate two scan jobs. It ignores the expense of switching different tasks for one thread. In practice, over fine-grained scheduling cannot achieve the smooth switch of different tasks in

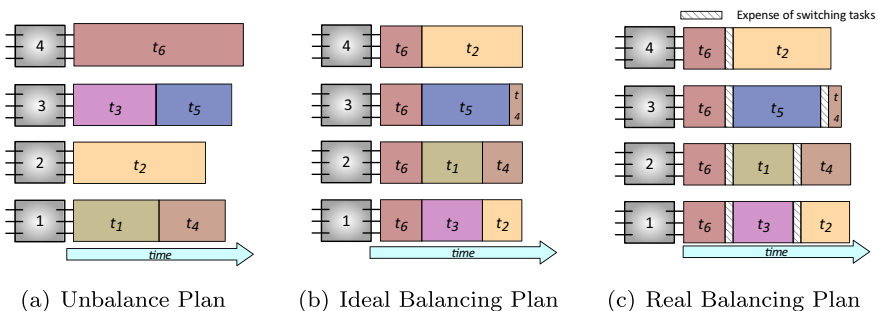


Figure 6 Two specific schedules

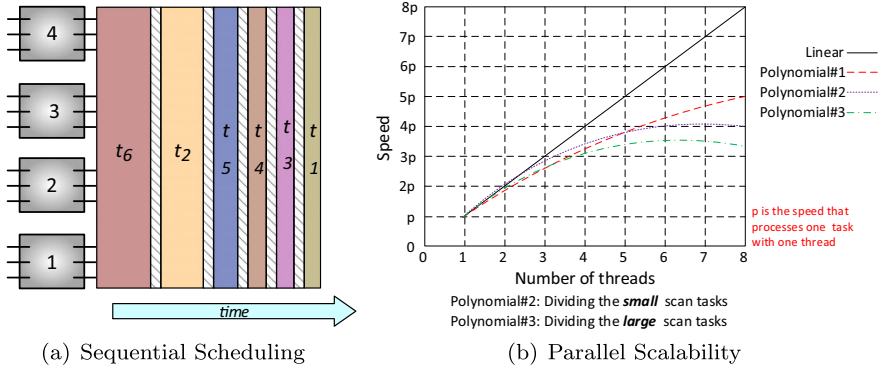


Figure 7 Scalability of parallel scan

Figure 6c. The expense may outweigh the gain of executing tasks with reaching load balance.

4.2 The sequential scheduling algorithm

Both the balance of workloads and the thread switching cost should be considered for scheduling scan tasks. Then, we introduce and analyze the scheduling algorithms based on the two aspects. As the strategy on all storage nodes are the same, for ease of presentation, we denote n' as the number of scan tasks allocated by the scan distributor on a node, and let N be the total number of working thread. At first, We discuss a sequential algorithm.

Algorithmic description The sequential scheduled algorithm assigns each task with all available working threads. It first sorts tasks according to the sizes of the tasks in descending order and then processes them in sequence. Each task is divided into N parts, which consist of N corresponding scan jobs. And each working thread is responsible for executing one scan task. Figure 7a demonstrates an example, all scan tasks are executed sequentially with all available working threads. Ideally, the workloads allocated on each working thread are even. However, it cost numerous job switching expenses. For a working thread, it switches scan task $n' - 1$ times and the total number is $N \cdot (n' - 1)$.

Parallel scalability Another important observation is that parallelism is over considered for the sequential scheduling method. This is because the algorithm uses all threads to process a task. However, the gain in practice is restricted to a limited number threads. As depicted in Figure 7b, we test the scalability of processing scan tasks, and find the performance gain is usually penalized by parallel resource [12] (such as memory bus and the cache) and can not satisfy the linear growth condition.¹ We wish to get N times increase of speed by using N threads to execute the task in “linear” curves. In practice, the scalability may satisfy the polynomial growth (“polynomial#1”). Once it reaches the highest point of performance, it will be stable with regardless of utilizing more working threads. In the worse case, the performance may even begin a little descent after its peak (“polynomial#2” or “polynomial#3”). We also find the size of the scan task is

¹ Similar results are also tested in [15]

also a critical factor to affect the parallel scalability. For a smaller scan task (“polynomial#2”), it is divided into further smaller parts, thus it reaches the maximum degree of parallelism quickly by increasing the number of threads. For a relatively larger scan task (“polynomial#3”), the increase of parallelism is slower than the process of the small task (“polynomial#2”).

The above result suggests that we do not have to divide the task too small or use all working thread to maximize the parallelism. Thus, we propose the chunk based algorithm that caters to our experimental conclusion.

4.3 The chunk based algorithm

As mentioned above, over partitioned scan tasks result in abuse of CPU resources. Therefore, we should also consider load balance and job switching costs for scheduling to save CPU resources. Nextly, we propose the chunk based algorithm.

Design chunks Each scan task is firstly divided into several parts which we call *chunks*, and we utilize the chunk as the basic unit for scheduling. It is worth nothing that the size of the chunk is important and it cannot be too small. We design a lower bound for the size of the chunk, where the partitioned number of chunks cannot affect the linear parallel scalability. This means, for a scan task, we can divide it into M or $M + 1$ chunks. And M is the exact boundary that if M chunks are scanned in parallel, we can get an approximate linear promotion with respect to the non-parallel execution, and if $M + 1$ chunks are scanned in parallel, approximate linear promotion cannot be achieved. In practice, the size of the chunk is obtained from the experiment. We have also tested different size of the chunk in the experiment under the restrictions of the above lower bound. In summary, the chunk is designed that we can maximize the performance without the possibility of wasting computing resources when they are scanned in parallel.

Cost model Then, we give a cost model to analyze a specific scheduling based on chunks. Given a chunk size, each scan task is divided into different numbers of chunks according to their workload. We utilize the chunk as the basic unit for scheduling. Formally, let ϕ_i be the set of chunks which belong to the scan task i . $\phi_1, \phi_2, \dots, \phi_{n'}$ (recall n' is the number of scan tasks) are scheduled on N working threads and each thread will run some chunks picked from them. We denote $T_j = \{c^j_1, c^j_2, \dots, c^j_{|T_j|}\}$ ($c^j_i \in \phi_1 \cup \phi_2 \cup \dots \cup \phi_{n'}$) be the set of chunks scheduled on thread j , where $|T_j|$ is the number of chunks scanned by thread j .

At first, we consider the cost of scanning chunks. Since all the chunks have similar size and the scan tasks are analogous (i.e. scan data from disk and filter data with predicate), we assume they have the same cost $Cost_c$ to scan a chunk. Therefore, the cost of scanning all the chunks for a working thread j will be $|T_j| \cdot Cost_c$.

Next, we consider the costs of job switch. Notice that for $T_j = \{c^j_1, c^j_2, \dots, c^j_{|T_j|}\}$, if several chunks are generated from the same scan task, we can put them together to constitute a scan task without switching costs. Only chunks from different scan tasks cost expense. To this end, the switching cost is $Diff(T_j) \cdot Cost_s$, where $Diff(T_j)$ is the different number of scan tasks executed on thread j (i.e. the number of consisted scan jobs on thread j) and $Cost_s$ is the constant cost unit to switch two scan jobs.

Combine both the above two costs, we can compute the overall cost. Let $Cost[T_j]$ be the cost on thread j , which can be computed as follows:

$$Cost[T_j] = |T_j| \cdot Cost_c + Diff(T_j) \cdot Cost_s \quad \text{w.r.t.}$$

$$T_j = \{c^j_1, \dots, c^j_i, \dots, c^j_{|T_j|}\}$$

$$c^j_i \in \phi_1 \cup \phi_2 \cup \dots \cup \phi_{n'}.$$

Since the goal of scheduling strategy is to make the costs even on all threads, which means to minimize the expense of the thread that has maximum cost, i.e.

$$\text{Minimize } \text{Max}_j (1 \leq j \leq N) \text{ Cost}[T_j].$$

To find the optimal answer for T_1, \dots, T_N , the problem can be reduced to parallel task scheduling, which has been proven to be NP-hard [9]. Subsequently, we propose a greedy solution.

Algorithm 1 The greedy scheduled algorithm

Input: sorted chunks for scan tasks: $\{\Phi_1, \dots, \Phi_{n'}\}$, w.r.t.
 $|\Phi_1| \geq |\Phi_2| \geq \dots \geq |\Phi_{n'}|$

Output: scan jobs for threads $\{T_1, \dots, T_N\}$;
 // Step1. Init $\{T_1, \dots, T_N\}$

- 1 **for** $j = 0$; $j < n'$; $j = j + 1$ **do**
- 2 $i = j \bmod N$;
- 3 // assign all chunks of Φ_j to Thread i
 $T_i = T_i \cup \Phi_j$;
- // Step2. Adjust $\{T_1, \dots, T_N\}$
- 4 **while true do**
- 5 $min = \text{argmin}\{Cost[1], Cost[2], \dots, Cost[N]\}$ ($1 \leq min \leq N$);
- 6 $max = \text{argmin}\{Cost[1], Cost[2], \dots, Cost[N]\}$ ($1 \leq max \leq N$);
- 7 **if** $(Cost[max] - Cost[min]) > Cost_s + Cost_c$ **then**
- 8 try move a chunk from T_{max} to T_{min} ;
- 9 **else**
- 10 break ;
- 11 **for each thread** T_i **do**
- 12 make the chunks from the same scan task into a scan job;
- 13 execute scan jobs;

A greedy algorithm The scheduling algorithm is to make each working thread has the average cost. Algorithm 1 shows the pseudo-code. Each thread has a task queue and the algorithm is to make scan jobs for all the threads. It has two main steps. It firstly sorts all the tasks according to their number of chunks in descending order (Input of Algorithm 1). Next, it iteratively selects the chunks from the same scan task into the task queue of a thread (line 1 to 3). In each iteration, it selects the scan task with the maximum number of chunks which has not been put into a task queue yet. And the queues are utilized in turns, for the next iteration, the chunks of another scan task will be put into the next queue. The first step completes if the chunks of all the scan tasks are pushed into the queue. To further make the costs of all threads even, we make adjustment in a second step (line 5 to 10). The greedy strategy is once again used. In each round of adjustment, we select a pair of task queues with

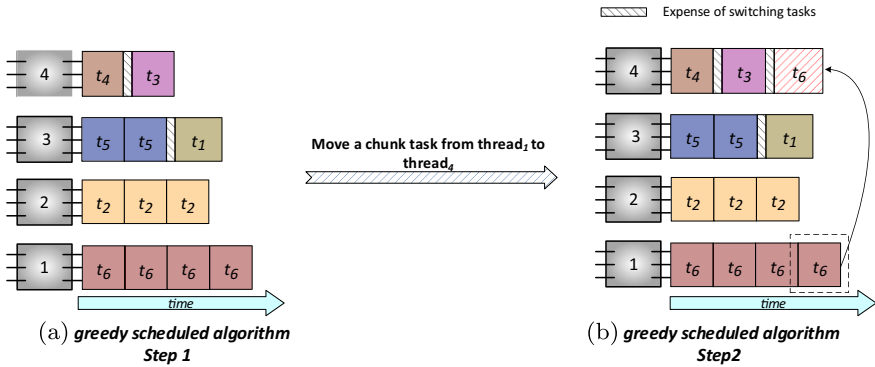


Figure 8 Scheduling algorithm

largest and smallest costs (line 5 to 6), and move a chunk from the most expensive queue to the least expensive (line 8). The adjustment ends up when no such pair can be found that the time costs of the two queues can be further reduced through one chunk shift (line 7). After adjusting the workload of each working thread, each working thread makes the chunks from the same scan task of the queue into a scan job and execute the jobs sequentially (line 11 to 13).

Figure 8 illustrates the example of the greedy algorithm. Firstly, the scan tasks are taken into the task queue of each thread according to the number of chunks in descending order. Then, we observe that T_1 has the most number of chunks (4) and T_4 has the least number of chunks (2). By considering moving a chunk of t_6 from T_1 to T_4 , the maximum costs of T_1, T_2, T_3, T_4 can be reduced, thus the chunk of t_6 is moved to T_4 to make a scan job.

5 Evaluation

In this section, we evaluate the proposed load balance strategy and scheduling algorithms in an open-sourced distributed data management system.

5.1 Experimental setup

System and hardware We have implemented our parallel scan strategy into an open-sourced database *Oceanbase*[21], which is a well-known distributed relational database developed by *Alibaba*. It has two layers: a query processing layer and a storage layer. In the storage layer, tablets (partitioned by primary-key) are kept in storage servers named *chunk-servers*. Queries are processed on the querying processing servers named *query-server*. *query-servers* send scan operations to scan data from *chunk-servers*. The system has implemented a parallel execution strategy which should be classified as *partitioned parallelism* [24]. It allows one scan operation to scan tablets (partitions) of the scanned table in parallel without regard of tablets' locations. For one scan operation, there may be some parts of (sub)scan tasks to be executed in parallel on the same *chunk-server*, and other parts may be executed in parallel on different *chunk-servers* at the same time. For short, we also denote *chunk-servers* by *SN* (storage node) in this section. We have implemented the *scan distributor* and *scan scheduler* with multi-threading parallelism on the *query-servers* and *chunk-servers* respectively.

Table 2 Scan operations of TPC-H

TPC-H [1]	TPC-H [3]	TPC-H [5]	TPC-H [6]	TPC-H [10]	TPC-H [14]
Scanned table	Scanned table	Scanned table	Scanned table	Scanned table	Scanned table
–	orders	orders	–	orders	–
Lineitem	lineitem	lineitem	lineitem	lineitem	lineitem
–	customer	customer	–	customer	–
–	–	supplier	–	–	–
–	–	nation	–	nation	–
–	–	region	–	–	–
–	–	–	–	–	part

All the experiments are conducted on a cluster of four physical machines, where each is equipped with an Intel(R) Xeon(R) CPU (E5-2620 0 @ 2.00GHz , with totally 24 cores, 160GB RAM and 1TB HDD while running an OS of Red Hat with version 4.4.7-4. Each node deploys a *chunk-server* and a *query-server* and servers are connected via 1Gb Ethernet.

Workload We choose six queries from the TPC-H benchmark² as our workload: TPC-H [1], TPC-H [3], TPC-H [5], TPC-H [6], TPC-H [10] and TPC-H [14]. For these six queries, all tables related with TPC-H are included. The related tables for these six queries are shown in Table 2. Predicate conditions for the queries are involved with non-PK columns. Thus, the scan operations require scanning all the tablets of the tables under without utilizing any other indexes on non-primary columns. There are eight tables in TPC-H. Considering the sizes of *lineitem*, *orders* and *customer* are significantly larger than the sizes of other tables (which are very small tables), we choose the $Q - 3$ which involves these three tables to precisely evaluate the effectiveness of our methods.

TPC-H allows setting the size of tables by choosing different parameters of SF and we choose *SF-1* as the default size. Table 3 shows the default data distribution which is formed by the default load balancing mechanism in Oceanbase [21]. Considering the sizes of other tables(except for *lineitem*, *orders* and *customer*) are too small(dozens or hundreds of rows), we do not take any additional parallel strategies to scan them. The system also takes the default storage of three replications for each tablet. The three-replications strategy is adopted by many distributed data management system such as BigTable [5] HBase [1]. Table *lineitem* is large, thus it has three tablets (maximum 256MB for one tablet), and others have only one *tablet*. We utilize the default SF and distribution when no otherwise specification is introduced.

5.2 Experimental results

5.2.1 Parallel between replicated nodes

We first demonstrate the effectiveness of our parallel strategy for multi-replications with $Q-3$, which scans three tables(*lineitem*, *orders* and *customer*). We compare three different strategies: (1) a parallel between tablets (PT). It is the simplest parallel strategy enabled

²<http://www.tpc.org/tpch/default.asp>

Table 3 Default distribution of data

SN_1	SN_2	SN_3	SN_4
Lineitem-1,2,3	–	lineitem-1,2,3	lineitem-1,2,3
Orders	orders	orders	–
Customer	–	customer	customer

by *Oceanbase* (mentioned above) where the naive non-replication parallelism is considered and (2) a simple parallel between replications (PR), which simply divides each tablet’s scan range uniformly and executes these scan tasks on replications between *chunk-servers* (3) our parallel strategy that is proposed for multi-replications in Section 3 (MRP). To divide the scan tasks precisely, we leverage the statistics with the histogram based partition. When run the three strategies, we utilize the same scheduling algorithm on each storage node, which runs a task on one thread until all tasks are completed.

Scan time We compare PT, PR and MRP by reporting the average time cost of scanning the three tables. Results are shown in Figure 9a. It is worth nothing that all three tables are scanned in parallel, thus the performance of scan mainly depends on the largest spent time. We can see MRP achieves the best performance. To scan table *lineitem*, *orders* and *customer*, PT costs 4.62, 0.81, 0.12 seconds respectively. PR overperforms PT by taking 1.15, 0.47 and 0.07 seconds. The time costs of PR on all the three tables are smaller than PT. This is because PR takes advantages of replications on different storage nodes, therefore more nodes (four machines) take part in data scanning while only half number of the machines are utilized for PT. MRP further decreases the time of scanning *lineitem* (0.81 seconds) with a bit of time growth on *orders* (0.52 seconds) and *customer* (0.13 seconds). This results from the effect of load balance between parallel nodes. Clearly, decreasing the longest scan time means the improvement of the entire scan operations. From another perspective, MRP also runs with the minimum overall time costs (0.81 + 0.52 + 0.13), this is because balanced parallel strategy can make the best usage of replications with the consideration of load balance, which can also be revealed by the CPU costs shown in Figure 9b.

CPU cost and system load We show the CPU costs and the exact system load (volume of scanned data) for PT, PR and MRP on the four storage nodes in Figure 9b and c under $SF = 1$. PT makes a majority of CPU’s utilization(49%) on the storage node SN_3 . When processing $Q-3$, it randomly selects SN_3 to run the task of scanning *lineitem*. As the largest table, scanning *lineitem* exhibits more system loads and consumes more system resources.

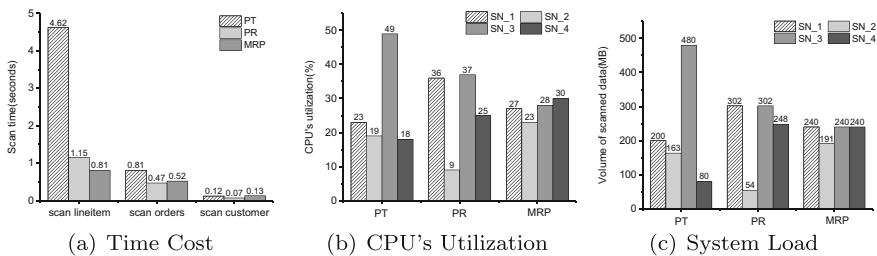


Figure 9 Results of scan time, CPU utilization and system load under $SF = 1$

For PR, we find that the CPU utilization and system loads on different SN s are still unbalanced. It is because that system loads are totally dependent on the distribution of data under PR. Since it dispatches (sub) scan tasks uniformly without load balancing strategy, PR divides the task of scan orders into equal three parts and dispatches them to three replications under the default distribution. To this end, SN_2 may only have only 1/3 of the tasks for scanning orders and take it for the grant that it should have the least CPU's utilization(9%) and volume of scanned data(54MB) on SN_2 . When taking our load balancing strategy into consideration, MRP can make each storage node has the similar system load. Since there is only one task on SN_2 , MRP may reject to divide the tasks of scanning orders and to dispatch them to other SN s under default distribution. From this perspective, MRP runs more similar volumes of scanned data(240MB,191MB,240MB and 240MB) on each *chunk-server* and also has close values of CPUs' utilizations on different SN s(27%,23%,28% and 30%).

Data size Next we vary the data size by doubling it to $SF = 2$ and test the scan time, CPU cost and system load. Results are shown in Figure 10. We can get the same experimental conclusion with $SF = 1$. Both of the time costs and system loads are doubled as the data size has been doubled. This is because that the $SF = 2$ data size does not change the distribution of data and only doubles the volume of scanned data. Under the same distribution, MRP may assign the similar number of scan tasks to each SN . To process the double size of data, scan tasks need to scan double volume of data and have double scan time as multi-threading is not utilized on each node.

Data distribution From the perspective of storage, we change the data(tablets) distribution to verify that our load balancing strategy can work under differenet data distributions. Next we try to adjust the data distribution by two ways: (1) We vary the number of storage nodes from 4 machines to 7 machines without running the storage balancing strategy. (2) We directly generate three new data distributions(shown in Table 4). Compared with default data distribution, each table may have 4 different distributions among the 4 machines and the three tables can have 64 different distributions. Thus, we only change the distribution of one table at each time and generate three new distributions. The first data distribution in Table 4 only modifies the distribution of *lineitem*. The second and third data distributions change the distribution of *orders* and *customer* separately. Results are shown in Figure 11. We observe that the scan time of task does not change obviously with the change of SN s' number in Figure 11a. This is because *Oceanbase* does not run the storage balancing strategy automatically. If we just add new SN s without adjust the number of replications or data distribution, the execution strategy of scan tasks will not be adjusted. Under same data distribution and execution strategy, scan tasks will not be dispatched to the new machines and

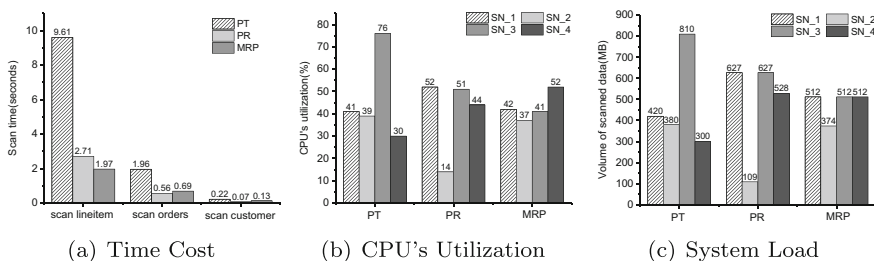


Figure 10 Results with $SF = 2$

Table 4 Compared data distribution

SN_1	SN_2	SN_3	SN_4
Lineitem-1,2,3	lineitem-1,2,3	lineitem-1,2,3	–
Orders	orders	orders	–
Customer	–	customer	customer
Lineitem-1,2,3	–	lineitem-1,2,3	lineitem-1,2,3
–	orders	orders	orders
Customer	–	customer	customer
Lineitem-1,2,3	–	lineitem-1,2,3	lineitem-1,2,3
Orders	orders	orders	–
Customer	customer	customer	–

the scan time will not change. When choosing the first adding data distribution (shown in Table 4), we find the CPU's utilization of each SN is still similar in Figure 11b. The maximum utilization is 49% and minimum utilization is 42%. It means that the load balance in MRP still has worked effectively with the change of related table distribution. Under the second and third data distributions in Table 4, the description of CPU's utilization for each SN has a similar look when compared with the default data distribution and each node has similar CPU's utilization. This means that the changes of data distribution do not have an impact on our load balancing strategy between replication nodes.

5.2.2 Parallel task scheduling

To evaluate the effectiveness of parallel task scheduling, we design the experiments from two aspects: thread-level scalability and processing capacity under different data sizes. In this experimental group, we take two different parallel task scheduling strategies to be compared with the greedy scheduling strategy(mentioned in Section 4): (i) the sequential scheduling strategy that we discussed in Section 4; (ii) a random scheduling strategy, it divides tasks into same chunks and then randomly assigns those chunks to different working threads. Considering the random scheduling and the greedy scheduling algorithm are both based on chunk tasks, we first try to give a preferable chunk size to support the evaluation of different parallel task scheduling strategies.

Chunk size We first vary the chunk size from 2MB to 8MB for the random scheduling strategy and the greedy scheduling strategy. In order to get a more credible conclusion, we use fifty clients to run the $Q-3$ to evaluate the two strategies from the perspective of CPU's utilization and time cost. In the Figure 12a, the CPU's utilization is the average of each SN 's value. We observe that the CPU's utilization drops with the increase of chunk size. It's obvious that the bigger chunk size may cause less switching costs for the two scheduling strategies. The CPU's utilization of random scheduling strategy is significantly higher than the CPU's utilization of greedy scheduling strategy, especially for the smaller chunk size. When the chunk size is 2MB, the CPU's utilization of random scheduling strategy is 63% and the CPU's utilization of greedy scheduling strategy is 46%. However, under the 8MB chunk size, the random scheduling strategy and greedy scheduling strategy are similar(38%

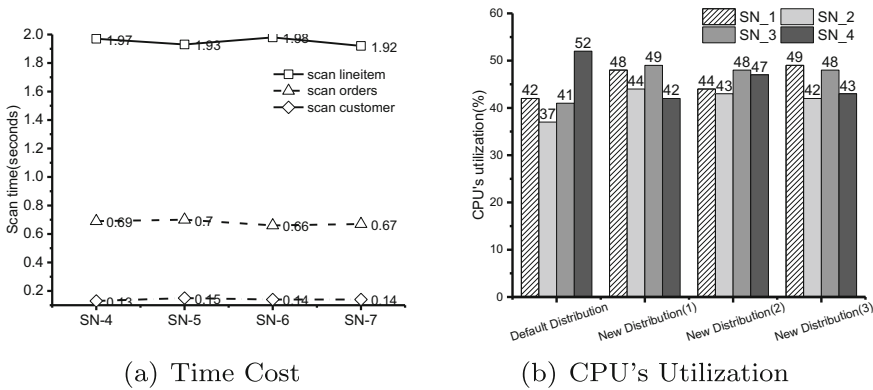


Figure 11 Results for data distributions under $SF - 2$

: 37%). It is because that the greedy scheduling strategy can still make the scan jobs of one working thread as continuous as possible and random scheduling strategy may lead to a group of unordered scan jobs for one working thread. Therefore, the high switching cost of random scheduling strategy consumes more CPU resources. The results in Figure 12b explain that chunk size has an impact on scan time. We find that random scheduling strategy has the least scan time(1.17 seconds) under the 6MB chunk size. It is because that the smaller chunk size may cause higher switching costs(such as 2.01 seconds for 2MB) and larger chunk size may lead to the uneven allocation for each working thread(such as 1.22 seconds for 8MB). The greedy scheduling has the least scan time(0.76) under 4MB chunk size. The reason for why 2MB,6MB and 8MB are not superior to 4MB is similar with the analysis of random scheduling strategy. Bigger size causes the uneven scheduling for each working thread and fine-grained chunk size causes higher switching costs. In order to shorten the scan time, we choose the 6MB chunk size for the random scheduling strategy and the 4MB chunk size for the greedy scheduling strategy.

Thread-level scalability Next, we compare the three scheduling strategies by varying the number of working threads. Results are shown in Figure 13a and b. We have following observations: (1) increasing number of threads can improve the scan performance of random scheduling, sequential scheduling and greedy scheduling. However, these strategies show different thread-level scalabilities. (2) Compared with other strategies, the random scheduling strategy costs the most scan time and consumes the most CPU resources. In Figure 13a, when utilizing six working threads, the random scheduling strategy costs 1.61 seconds. However, the time cost of random scheduling strategy only reduces 0.04 seconds(1.57 records) by allocating more threads(8 working threads). It is obvious that the random scheduling cannot get the further promotion even more working threads are allocated. It is because that the random scheduling strategy generates several unordered chunk tasks and causes too many switching costs. (3) sequential scheduling strategy gets a better scanning performance than random scheduling strategy. This is because the sequential scheduling strategy has fewer costs of switching context than random scheduling strategy. However, the speedup also decreases sharply with the increase of working threads and the CPU's utilization is also higher than the greedy scheduling strategy since the sequential scheduling strategy is also restricted to the CPU's multi-threading scalability(mentioned in Section 4). (4) greedy scheduling definitely has the best multi-threading scalability

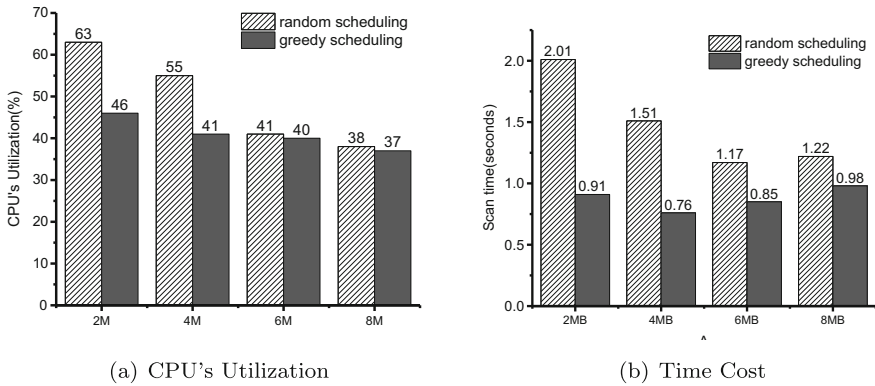


Figure 12 Results for chunk size

and CPU's utilization when being compared with other two strategies. It is because that the greedy scheduling strategy considers reducing switching costs and making job balanced at the same time. These experiments verify that the greedy scheduling strategy can achieve the better thread-level scalability from the perspectives of time cost and CPU's utilization.

Data size In Figure 13c and d, we increase the data size from $SF = 1$ to $SF = 8$. Figure 13c illustrates that the growth of scan time increases with the growth of data size. The Figure 13d shows that CPU's utilization on one SN is also dependent on volumes of data. We can find that the scan time has an approximately linear relation with the value of SF .

5.2.3 Overall performance

Based on above results, we combine the proposed strategies of MRP and greedy scheduling strategy to build our parallel scan mechanism(PSM). Then, we run the six queries of TPC-H to test the overall performance.

Performance improvement To demonstrate the benefit of PSM, we design three comparisons: (1) Method-1: a non-parallel execution which removes the original parallel strategy from the original system, (2) Method-2: an original parallel execution which only keeps *partitioned parallelism* and (3) Method-3: an enhanced parallel execution which takes the greedy tasks scheduling strategy to optimize the execution of multiple scan tasks on the same SN . We run the six queries with fifty clients concurrently and statistics the average time of each query's scan operations from start to end. In Figure 14a, results show PSM is superior to the three comparisons. For instance, $Q - 3$ with method-1 takes 15.4 seconds to finish scan tasks and $Q - 3$ with method-2 only takes 6.85 seconds to have done with all tasks. The other queries($Q - 1, Q - 5, Q - 6, Q - 10$ and $Q - 14$) which take method-2 are also superior to the queries which take method-1. It proves that that tablet parallelism can improve the scanning performance. Further, considering taking the greedy scheduling strategy(method-3) can significantly reduce the total scan time, e.g. the scan time of $Q - 3$ drops from 6.58 seconds to 2.12 seconds when taking the greedy scheduling strategy. Due to obtain the higher parallelism from multi-core CPU, the

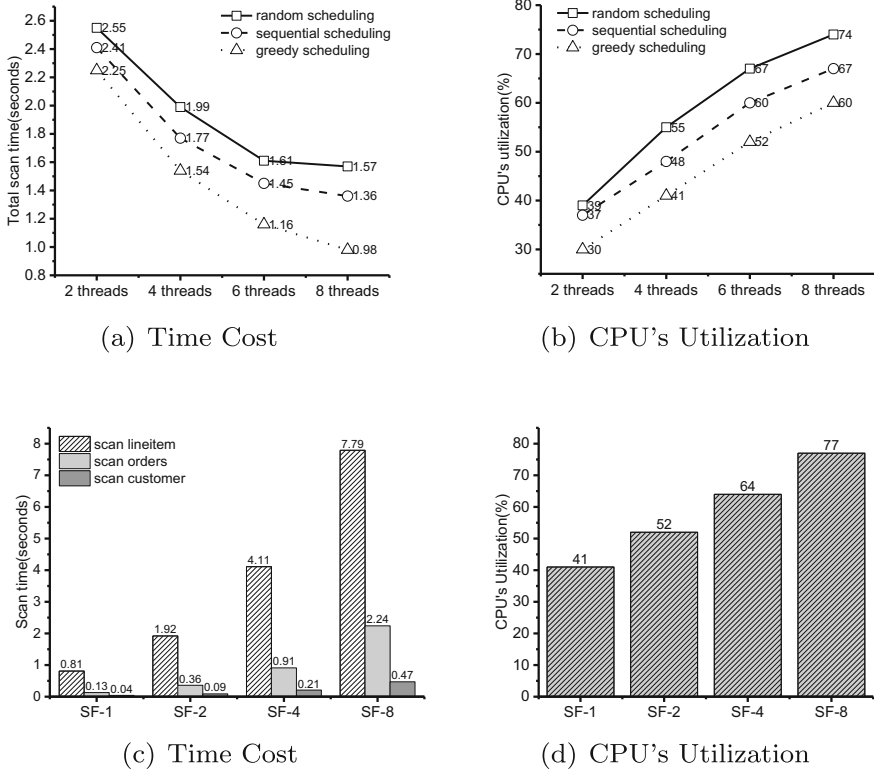
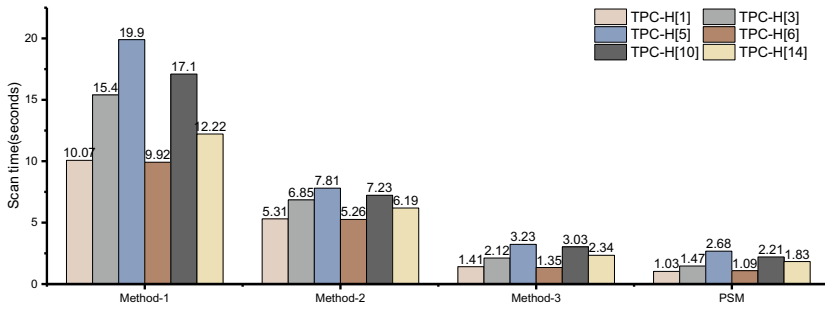


Figure 13 Results for scheduling tasks on storage node

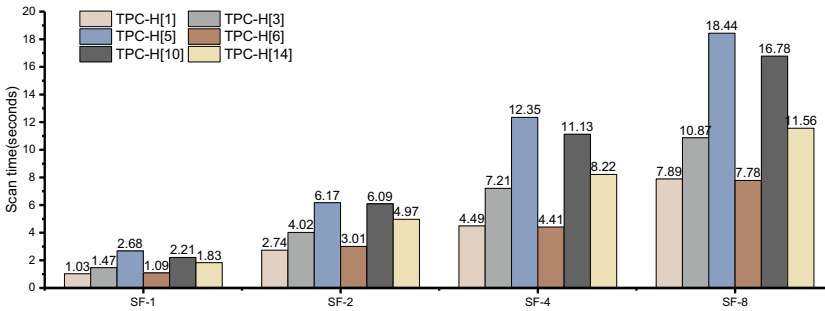
greedy scheduling strategy can also accelerate the scan time of other queries. Combining the load balancing strategy, the PSM can help the scan time of $Q - 3$ ultimately drop to 1.47 seconds. Owing to allocate balancing workload for each SN , the *bucket effect* is eliminated and other queries can also improve the scanning performance. The results demonstrate that the strategies that we proposed in Sections 3 and 4 have worked and the combination can have further improvement of scan operations.

Varying data size Figure 14b runs PSM under different data sizes to seek the correlation between scan time and volumes of scanned data. We observe that the total scan time of $Q - 1$ based on $SF - 1$ is 1.03 seconds. However, $Q - 1$ based on $SF - 2$, $SF - 4$ and $SF - 8$ are 2.74 seconds, 4.49 seconds and 7.89 seconds respectively. Through analyzing $Q - 3$, $Q - 5$ and etc, we find that the total scan time of each query has an approximately linear relation with value of SF . It is easy to understand that more size of data lead to more volume of scanned data for each (sub)scan tasks and need more scan time to complete each queries' scan tasks. It definitely verifies that data size is a critical factor for scan time.

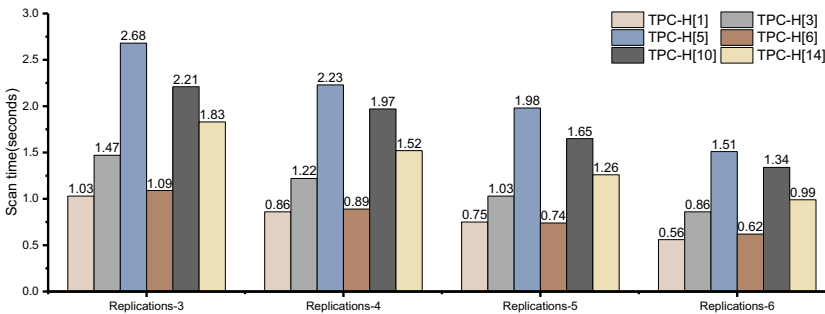
Replication-level scalability In Figure 14c, we verify the replication-level scalability of PSM under the different number of replications for each tablet. We take the scan time from scan beginning to the scan end to represent the scanning performance. Results show



(a) Comparing Different Method



(b) Varying Data Size



(c) Replication-Level Scalability

Figure 14 Results for overall performance

that the total scan time of each query drops gradually with the increase of the number of replications. When taking three-replications strategy, the total scan time of $Q - 6$ is 1.09 seconds. As more replications are taken, the scan time of $Q - 6$ only have 0.89 seconds (4 replications), 0.74 seconds(5 replications) and even 0.62 seconds (6 replications). $Q - 1 - Q - 14$ also can reduce their total scan time when increasing the number of replications. The scan time can achieve a approximately linear decrease by adding the number of replications. The reason is that PSM needs to divide tasks according to

the number of replications (mentioned in Section 3). Clearly, owning more replications means that each scan tasks can be divided into more number of parts (smaller range of primary key) and each part of the scan task can run on more different replications independently to reduce the scan time. It means that PSM can achieve the replication-level scalability.

6 Related work

Scan parallelism Parallel scan has been a primary design point since the very beginning of database systems [8, 28]. The primary problem is the degree of parallelism. Based on a given threshold of parallelism, traditional database systems make it a hard-coded configure in the systems (e.g. IBM DB2 [13], ORACLE [23] and Microsoft SQL Server [20]). Over the past decades, with the development of replica technology, more and more presently-available distributed DBMSs [30] rely on the partition mechanism to achieve *partition parallelism* [24]. However, there is still a strong possibility that some partitions of one scanned table are accessed on same node. The choice that involved in replicating partitions brings us an opportunity to speed up parallel applications [2]. There are several efforts [10, 11] on how to utilize replications to improve query performance, but they only consider how to speed up concurrent queries by retrieving data from different replications at the same time under a uniform data distribution. In this paper, we put forward a new replication parallel strategy to accelerate one or several scan operations from one query under arbitrary distributions. Our parallel strategy enables us to divide a specific scan operation into multiple pieces and run them separately on different replications.

Load balancing Load balancing is a prerequisite for effectively utilizing parallel resources to improve system performance in parallel database systems [3]. A serious problem is to access skewed data distributions which may lead to *bucket effects* of overall performance [33], and a series of solutions have been proposed to solve this problem. However, what we usually need to solve is that the workload of scan tasks are skewed in distributed database system when processing queries. This is because that different scan tasks that are involved in different volumes of scanned data. The balancing goal of task allocation is to guarantee that each node has a similar number of tasks in distributed database systems [16]. In this paper, we find that the workload of full-table scan task is associated with the rows and schema of the scanned table. A complex schema and large number of rows mean that more workload for a scan task. Thus, we introduce the *weight* to describe the workload of scanned table. Through linear programming, we give the optimal tasks division and allocation strategy on the workload-level. Moreover, we employ equal-depth histogram to analyze the workloads of scan tasks. Most of current RDMSs maintain a set of equal-depth histograms for estimating the selectives of given queries [19], but they ignore the benefit of using equal-depth histogram to assign accurate workloads to scan tasks.

Parallel scheduling The authors of [6, 14, 27] study scheduling for parallel task execution. M.S. Chen et al [6] suggest that parallel working threads should be allocated to tasks according the scalability of speed-up curve. This strategy wants to provide enough parallel threads for helping coming tasks obtain their best speed-up ratio to achieve the best overall

performance when processing a dynamic task set. To process a given set of static tasks concurrently, there are also several efforts [14, 27] that has been proposed on how to share parallel working threads with tasks in the multi-threading process framework. The main target in [14] is to try to optimize the execution of query-level tasks in the multi-threading process framework. Meanwhile, the finer-work, that considers how to schedule scan tasks of queries in multi-threading process framework, is proposed in [27] and takes many factors(e.g. cache thrashing [18] and context switching [32]) into consideration. In this paper, we also consider how to schedule scan tasks in multi-threading process framework, but we focus on how to improve the parallel multi-threading scalability and observe the size of tasks is also a critical factor which has relation with cache and context. We try to find a suitable size that can minimize the cost of the system and achieve approximate linear improvement of scan performance by allocating more threads. The dynamic load balancing strategies in [3, 4] for parallel scheduling are also taken into our paper to keep working threads having balanced workloads.

7 Conclusion

In this paper, we have presented an efficient parallel scan strategy in the distributed data management system. We fully exploit the parallelism between replications to improve the performance of scan operations. Taking load balance into consideration, we form a problem about scan tasks allocation and give a linear programming solution. To implement the parallel execution of scan task on one node, we analyze several parallel tasks scheduling strategies and propose a chunk-based greedy strategy to achieve approximately linear improvement of scanning performance with the increase of threads. To evaluate the parallel scan strategy, we integrate it into an open-sourced distributed data management system. Experimental results show the proposed strategy outperforms the original strategy and other compared strategies in the distributed data management system.

Acknowledgments This work is partially supported by National Science Foundation of China under grant numbers 61702189, 61432006 and 61672232, and Youth Science and Technology - “Yang Fan” Program of Shanghai under grant number 17YF1427800. Huiqi Hu is the corresponding author.


Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

References

1. Apache. HBase. <http://hbase.apache.org/>
2. Bal, H.E., Kaashoek, M.F., Tanenbaum, A.S., Jansen, J.: Replication techniques for speeding up parallel applications on distributed systems. *Concurr. Pract. Exper.* **4**, 337–355 (1992)
3. Bouganim, L., Florescu, D., Valduriez, P.: Dynamic load balancing in hierarchical parallel database systems. In: Proc. of the Int. Conf. on Very Large Data Bases (VLDB). Mumbai (1996)
4. Bouganim, L., Florescu, D., Valduriez, P.: Load balancing for parallel query execution on NUMA multiprocessors. *Distrib. Parallel Datab.* **7**(1), 99–121 (1999)
5. Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., Gruber, R.: Bigtable: A distributed storage system for structured data. In: Proceedings of 7th Symposium on Operating System Design and Implementation (OSDI), pp. 205218 (2006)

6. Chen, M.-S., Yu, P.S., Wu, K.-L.: Scheduling and processor allocation for parallel execution of multi-join queries. In: Proceedings of the Eighth International Conference on Data Engineering, pp. 58–67. IEEE Computer Society, Washington, DC (1992)
7. Cockshott, W.P.: Addressing mechanisms and persistent programming chapter 15 in Atkinson others (1988)
8. DeWitt, D., Gray, J.: Parallel database systems: The future of high performance database processing. *Commun. ACM* **36**, 6 (1992)
9. Du, J., Leung, J.Y.T.: Complexity of scheduling parallel task systems. *SIAM J. Discret Math.* SIAM (1989)
10. Ferhatosmanoglu, H., Tosun, A.S., Canahuate, G., Ramachandran, A.: Efficient parallel processing of range queries through replicated declustering. *Distrib. Parallel Datab.* **20**(2), 117–147 (2006)
11. Frikken, K., Atallah, M., Prabhakar, S., Safavi-Naini, R.: Optimal parallel i/o for range queries through replication. In: Proceedings of 13th International Conference of Database and Expert Systems Applications (DEXA), pp. 669–678 (2002)
12. Graefe, G.: Volcano—an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, **6**(1) (1994)
13. IBM: DB2. intra-partition parallelism https://www.ibm.com/support/knowledgecenter/en/SSEPGG_9.7.0/com.ibm.db2.luw.admin.perf.doc/doc/c0005323.html (2009)
14. Johnson, R., Hardavellas, N., Pandis, I., Mancheril, N., Harizopoulos, S., Sabirli, K., Ailamaki, A., Falsafi, B.: To share or not to share? In: VLDB (2007)
15. Krikellas, K., Cintra, M., Vigiias, S.: Scheduling threads for intra-query parallelism on multicore processors. In: EDBT (2010)
16. Krompass, S., Kuno, H., Dayal, U., Kemper, A.: Dynamic workload management for very large data warehouses: Juggling feathers and bowling balls. In: Proc. of the 33rd Intl. Conf. on Very Large Databases (VLDB), pp. 1105–1115 (2007)
17. Kuo, T.-W., Wei, C.-H., Lam, K.-y.: Real-time data access control on B-tree index structures. In: IEEE 15th International Conference on Data Engineering. Sydney (1999)
18. Lee, R., Ding, X., Chen, F., Lu, Q., Zhang, X.: MCC-DB: Minimizing cache conflicts in multi-core processors for databases. *PVLDB* **2**(1), 373–384 (2009)
19. Lim, L., Wang, M., Vitter, J.S.: SASH: A self-adaptive histogram set for dynamically changing workloads. In: Proceedings of 29th VLDB Conference. Berlin (2003)
20. Microsoft: SQL Server parallelism enhancements <http://sqlmag.com/sql-server-2008/parallelism-enhancements-sql-server-2008> (2008)
21. OceanBase. <https://github.com/alibaba/oceanbase/>
22. Open Source DB. <https://www.postgresql.org/>
23. Oracle Database 11g. Parallel execution https://docs.oracle.com/cd/E11882_01/server.112/e25523/parallel002.htm. (2007)
24. Pan, C.S., Zymbler, M.L.: Encapsulation of partitioned parallelism into open-source database management systems. *Program Comput. Softw.* **41**(6), 350–360 (2015)
25. Percival, C.: Cache missing for fun and profit. In: Proc. of BSDCan 2005 (2005)
26. Pivotal. GREENPLUM DB. <http://greenplum.org/>
27. Qiao, L., Raman, V., Reiss, F., Haas, P.J., Lohman, G.M.: Main-memory scan sharing for multi-core CPUs. *Proc. VLDB Endow.* **1**(1), 610–621 (2008)
28. Rahm, E., Stöhr, T.: Analysis of parallel scan processing in parallel shared disk database systems. In: Proc. EURO-PAR Conf., LNCS, p. 966. Springer (1995)
29. Ristau, B., Fettweis, G.: An optimization methodology for memory allocation and task scheduling in SoCs via linear programming SAMOS 89–98 (2006)
30. Sokolinsky, L.B.: Survey of architectures of parallel database system. *Program Comput. Softw.* **30**(6), 337–346 (2004)
31. Son, S.H.: Replicated data management in distributed database systems, ACM SIGMOD, vol. 17 Issue 4, pp. 62–69. ACM, New York (1988)
32. Tsafirir, D.: The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In: Proceeding ecs’07 Experimental computer science on Experimental computer science, pp. 3–3. San Diego (2007)
33. Valduriez, P.: Parallel Database Systems: Open Problems and New Issues, Distributed and Parallel Databases. Springer (1993)

Affiliations

Xing Wei¹ · Huiqi Hu¹  · Huichao Duan¹ · Weining Qian¹ · Aoying Zhou¹

Xing Wei
simba_wei@stu.ecnu.edu.cn

Huichao Duan
stevenduan@stu.ecnu.edu.cn

Weining Qian
wnqian@dase.ecnu.edu.cn

Aoying Zhou
ayzhou@dase.ecnu.edu.cn

¹ School of Data Science and Engineering, East China Normal University, Shanghai, China