The Journal of Supercomputing (2022) 78:2417–2441
https://doi.org/10.1007/s11227-021-03965-4

# Optimizing data query performance of Bi-cluster for large-scale scientific data in supercomputers

**Xia Liao[1] · Yixian Shen[2] · Shengguo Li[1] · Yutong Lu[2] · Yufei Du[2] · Zhiguang Chen[2]**

Accepted: 18 June 2021 / Published online: 29 June 2021


## Abstract

Scientific exploration and discovery heavily rely on increasing datasets and strong supercomputing power. Surging data pose massive data management challenges in existing data query frameworks. Although many data management techniques have been developed to quickly locate the selected data records, the time and space required to build and store these indexes are often too expensive. To deal with the problem of data location in a parallel file system managing large-scale scientific data, we propose an improved high-performance query data framework called "Bi-cluster+." In the aspect of index generation, a hierarchical index data structure is designed, which effectively balances index granularity and index construction overhead. According to the characteristics of the index offset, the write load balancing strategy is designed. The hierarchical index is written independently and in parallel. The in situ index generation is optimized by resource constraint analysis. In the aspect of data retrieval, optimization techniques are proposed to improve the query performance. Such as the strategy of the logical data block merging and reading. With the experiments by using multiple scientific datasets on a supercomputer, our optimizations improve data query performance by up to a factor of 1.9 compared with the original Bi-cluster implementation. The scalability of Bi-cluster+ can keep a good performance by evaluating on 17496 cores.

**Keywords** Scientific data · Data retrieval · Hierarchical index · High-performance computing

✉ Xia Liao
liaoxia@nudt.edu.cn

[1] College of Computer Science, National University of Defense Technology, Changsha 410073, China

[2] National Supercomputer Center in Guangzhou, and the School of Data and Computer Science, Sun Yatsen University, Guangzhou 510006, China

## 1 Introduction

With the rapid development of supercomputers, cross-disciplinary research receives widespread attention. The large-scale simulation, observations, and analysis of scientific applicants generate massive data day and night, such as high-throughput gene sequencing, plasma physics simulation [6], and cosmological observations [26], challenging the traditional data management. Moreover, artificial intelligence like deep learning surges, massive observations, and simulation data have brought scientific research into an unprecedented era of scientific big data [24]. The patterns of scientific discovery will undergo major changes. "Data-intensive science" has become the fourth new model of scientific discovery [13]. The volume of data is usually calculated by TB even PB. Over the past two decades, the computing power of supercomputers continue to increase, from KB to EB. Supercomputers equipped with powerful computing and communication can accelerate to deal with scientific data. Employing the supercomputer for scientific analysis is a pretty good choice.

Tianhe-2A [19] is a group of heterogeneous high-performance computers located in the National Supercomputer Center in Guangzhou with a capability of a peak performance of 100.67 PFlops and sustained performance of 61.4 PFlops after replacing the original Intel Xeon Phi accelerator with the domestic accelerator Matrix 2000. At present, a large number of scientific calculations are carried out on the Tianhe-2A. The fields of application mainly include atmospheric marine environment, astronomical geophysics, industrial design and manufacturing, biohealth medical care, new energy and new materials, and smart cities. PB level data were generated every day. As exa-scale computing era approaches, large-scale data analysis on supercomputers is becoming increasingly important [8]. Traditional supercomputers concentrate on computing-intensive task optimization. With the increasing massive scientific data, supercomputers are required to equip with strong analytical capabilities [14]. The rapid growth of high-performance computing systems has injected new impetus into emerging scientific discovery and engineering innovation.

Index technology [7, 9, 10, 12] is currently a prevalent method for accelerating data positioning so that effectively accelerates scientific analysis. Traditionally, the most widely used application scenarios of index is in database management systems(DBMS). DBMS system is targeted at transaction-oriented workloads, and the data structures used are generally B-trees [3] and bitmap indexes [34]. The user's data are usually treated as a table of rows and columns. Each row of data is a record. When visiting some certain variables, you should visit the entire database owing to several records are stored on the same physical block. Besides, the data strictly follows the ACID (Atomicity, Consistency, Isolation, Durability) atomic characteristics. Loads of reading, writing, querying, and updating are relatively balanced. Queries are returned to users in the form of records. However, scientific data are organized by the array data model. Every variable is stored in its own array. The columns in the table are the records. In this way, when researchers want to visit some certain variables, they just need to visit the continuous data block which can improve the effectiveness of visiting. Moreover,

scientific users often use scientific data for analysis. Massive generated data need only be stored once, but need to be read many times. Loads of writing and reading are imbalanced. Scientific data have its own formats [5] like Hierarchical Data Format version 5 (HDF5) [1] and the Network Common Data Form (NetCDF) [2, 18] and store in the form of file. The scale and volume of scientific data are so huge. Importing the massive data into the database may produce heavy overhead.

In this paper, we propose Bi-cluster+, a novel high-performance data query framework, which extends Bi-cluster [23] framework. In terms of index structure, we introduce another fine-grained inverted index to speed up the process of data retrieval. In the original implementation, we adopted bitmap as a fine-grained index structure. When querying attributes with few distinct values, Bitmap indexes are substantially compact. However, with the increasing cardinality of querying attributes, the size of the bitmap index will linearly grow. Therefore, we introduce another fine-grained index structure when the data is equipped with high cardinality. Integrating the inverted index into our framework, we gain a more compact index size compared with Bi-cluster. The time of index construction was slightly decreased. Moreover, the query time also improved. In terms of hardware resource utilization. For utilization of computing resources, we further improve the computing process of index generation and data retrieval through taking advantage of fine-grained computing resources to effectively utilize the computer resource. Normally, the computing resources are allocated to users in units of computing nodes which are exclusively occupied. To make full use of the computing resources, we adopt the hierarchical schedule of the computing resource and schedule the computing resources in units of cores. For memory bandwidth optimization, in the original implementation, we don't consider the constraints of memory bandwidth. All applications running on supercomputers share the parallel file system and memory bandwidth. Although Tianhe-2A is equipped with a high memory bandwidth, multiple applications accessing the parallel file systems may cause interference with each other. Therefore, we implement a bandwidth-aware data retrieval mechanism to address the memory bandwidth contention issue. Moreover, based on the real-time bandwidth situation, we dynamically adjust the size of the read block. In terms of file format extension, we extend netCDF in the Bi-cluster. Meanwhile, we provide the extensible interface to extend more parallel file format. We evaluate our approach using five scientific datasets on Tianhe-2A. The results show that the proposed optimizations improve query performance by up to a factor of 1.9 comparing to the original Bi-cluster implementation.

Our contributions are shown as follows:

- We optimize the index structure and introduce a second fine-grained index, inverted index to decrease the index size and data query time.
- We improve the adaptability of data-intensive indexing workloads executing on computation-intensive platforms by hierarchical computing resource scheduling and memory bandwidth-aware data query mechanism.
- We further optimize the in situ kernel parallel index generation mechanism by modeling the shared memory mechanism.

- We introduce load balance for parallel data query and flexible merge read strategy to speed up the performance of data retrieval.
- We extend the scientific data formats, supporting NetCDF and other scientific data with an extensive interface.

This work is organized as follows. Section 1 briefly introduces and discusses the challenges and opportunities of the existing data query technology. Section 2 introduces the related work of indexing. Section 3 shows the optimization of Bi-Cluster+. Section 4 describes experimental results, demonstrates and compares with our original systems and other systems to draw conclusions in the real environment. We summarize the paper in Sect. 5.

## 2 Related work

### 2.1 Scientific data query framework

There exist many scientific databases like SciDB [27, 28], Qserv [29], SciQL [37], and ArrayStore [25]. SciDB is a column-oriented database management system (DBMS) designed for multidimensional data management and analytics common to scientific, geospatial, financial, and industrial applications. It is developed by Paradigm and co-created by Turing Award winner Michael Stonebraker. By analyzing the traditional SQL language does not meet the needs of scientific analysis, SciDB explores an array data model to satisfy user needs. SciDB supports nested, multi-dimensional data models that can be applied to the analysis of scientific data. However, during the process of analysis, the original data needs to be imported into the SciDB database. When the amount of data is small, this overhead is acceptable, but as the dataset continues to increase, unacceptable conversion overhead is generated, which seriously affects the efficiency of scientific data analysis.

FastQuery [11] is a parallel file index framework for the massive scientific data designed by the Lawrence Berkeley Laboratory in the United States, which speeds up data queries by directly building index in the original scientific dataset. FastQuery uses Bitmap [33], as the index data structure and maps the array data model to the relational data model. FastBit [31] is applied to compress, bin, and encode bitmap indexes to reduce bitmap size which shows that accelerating query processing by at least an order of magnitude in many different applications [32]. But in some cases, the size of index generation is still large which is even as same as the size of the original data. Moreover, FastQuery generates indexes at a slower speed, and the time overhead is significant when generating indexes for large-scale data.

The in situ data processing method [15, 17] aims to bypass disk access as much as possible. The basic idea is that the data is executed in memory while performing a series of predetermined operations in memory. Currently, many in situ processing methods are only for parallel between nodes and do not consider kernel parallelism.

Reading the continuous data at a time is the motivation for the block index [36]. Therefore, the read time of a single record can be as long as the read time of the data block. In addition, reading one block at a time can greatly reduce the number of

I/O requests compared to random access on a single data record. ADIOS is a middle-ware for high-performance I/O that enables the implementation of the I/O layer away from application scientists [4] and provides users with the flexibility to work between different I/O implementations via XML configuration files. Block index technology combined with ADIOS [20, 35] can accelerate the building index. Good performance can be achieved when retrieving large amounts of data. However, when a tiny fraction of the data needs to be retrieved, the process of data retrieval is very low and relatively time-consuming.

Jeff Dean revamps traditional databases and considers indexes to be models [16]. One of the most significant problems with traditional indexes is that the characteristics of the data are not fully considered, and the worst data distribution is often assumed first, so the expected index has higher versatility. Therefore, such an index often leads to a large amount of waste of storage space, and the performance cannot be maximized. The learned index uses the method of machine learning to learn the distribution characteristics of these data by learning the existing data set, thereby improving the existing index model. Looking at the test results based on the real data set, compared with traditional B-tree, Learned Index has a 60–70% performance improvement.

### 2.2 Indexing techniques

Indexing technology [7, 9, 10, 12] is currently popular acceleration data localization method, effectively accelerate scientific analysis. According to the data structure of the index, it can be divided into tree index, hash index, inverted index, bitmap index and block index.

Tree index is based on tree form index, first proposed in 1962 as AVL tree, then many tree index has been put forward successively, such as B tree, B* tree, B+ tree, R tree, R* tree, R+ tree, T tree, T* tree, T+ tree index and so on. In traditional relational databases, B-tree data structure is used as the index, and leaf nodes are used to store data so as to realize data location. The B-tree is a dynamic, highly balanced indexing scheme that grows and contracts by recursively splitting and merging nodes from the lowest level of the index tree to the root node.

A hash index defines a hash function, which calculates the target address, effectively mapping the keyword and address set to each other. The complexity of query time is $O(1)$. Hash index can only do accurate query, but can not achieve range query. Data skew can occur when the keyword distribution is uneven.

Inverted indexes have two main parts: the attribute value and the physical address of the record with the attribute value. Unlike forward indexes, which have specific records to determine attribute values, inverted indexes map to physical record addresses by attribute values. When the retrieved file data is too large, the forward index cannot meet the requirement of real-time return results. Inverted index is a good choice.

Bitmap index, the index column is mapped into the map array, the number of records on the Bitmap array represents the keyword corresponding to the data row. Bitmap indexing is suitable for low-base applications where space overhead is low.

On the other hand, if the application is high cardinality, you need to check all the data in the index, resulting in too much index data, even more than the original data. Due to the expensive storage requirements and computing overhead, bitmap index is more suitable for scientific applications with few modification requirements and range query requirements. Many different strategies have been proposed to reduce the size of bitmap indexes and increase their overall effectiveness. There are three types of bitmap indexing techniques: compression, encoding, and chunking. The most advanced bitmap indexing technique is FastBit [31], whose parallel implementation FastQuery [11] shows the ability to index and analyze trillions of particle datasets.

Block indexing technology divides the data set into non-overlapping fixed-size data blocks and generates indexes in the form of data blocks. Indexes are established by recording the maximum and minimum values of each block, also known as block-range indexes. Because the block index stores only two values from each block, the total size of the index is very small compared to the size of the original data set. Therefore, the time and space required to build, store, and load these indexes is minimized. Both the query and indexing procedures result in only sequential I/O, and the minimum I/O size can be controlled by the block size specified by the user. The underlying storage system can handle I/O access more efficiently. Because the querying and indexing processes can be performed independently on each block, this approach can be easily implemented in parallel, effectively leveraging the multi-core architecture and computing power of a high performance computing system.

## 3 Bi-cluster+: the extension of Bi-cluster architecture

The original implementation of Bi-Cluster consists of index generation procedure and data retrieval procedure. In terms of index generation, we propose the hierarchical index structure, coarse-grained block index and fine-grained Bitmap index, respectively. First, we evenly partition the data file into a fixed-size data block and establish the coarse-grained for each data block. And then, we analyze the data characteristics (variance, CDF(Cumulative distribution function) as well as sorting feature) and select a portion of the data block as the hot data block to construct the fine-grained index. However, the fine-grained Bitmap index is sensitive to the cardinality of the data file. With the growing cardinality, the size of the index also increases linearly. Consequently, the data retrieval time and overhead of storage also grow correspondingly. Besides, Bi-cluster don't consider the memory bandwidth restricts and the computing resources are utilized ineffectively in units of nodes. The index files are separately stored, it also hampers the performance of the data query performance. Moreover, Bi-cluster can only support HDF5 files.

Based on the aforementioned issues, we extend Bi-cluster in aspects of index structure, resource utilization and supported formats, respectively. The major differences between Bi-cluster and Bi-cluster+ are shown in Table 1. For index structure, we introduce another file-grained inverted index. Compared with Bitmap, the inverted index substantially decrease the size of the index file and improve the data retrieval performance. In terms of resource utilization, we perform fine-grained

**Table 1** Comparison between Bi-cluster and Bi-Cluster+

| | Bi-cluster | Bi-cluster+ |
|---|---|---|
| Coarse-grained index | Block index | Block index |
| Fine-grained index | Bitmap | Bitmap & Inverted index |
| Support file format | HDF5 | HDF5 & netCDF |
| Index built-up | Exclusively occupation an entire core | Fine-grained computing resource allocation |
| In situ index generation | Limited support | Optimized support |
| Index storage | Separately stored | Uniformly stored |
| Memory bandwidth usage | Unconstrained usage | Optimized usage |
| Merge strategy | Contiguous memory block read | Flexible logical memory block |
| Load balance strategy | NULL | Balance the I/O request among nodes |

computing scheduling for index generation and data retrieval. Meanwhile, we propose parallel index writing strategy, load balance strategy to reduce the time of building index and data retrieval while avoiding the memory bandwidth contention. We also further optimize the merge read strategy. With regard to the supported data format, we further extend Bi-cluster+ to support the netCDF data file.

As illustrated in Fig. 1, we marked the improvements explicitly. In respect of index generation, when the data block is loaded into memory, Bi-cluster+ invokes the *MMIndexBuilder* interface to generate the Min–Max coarse-grained index for each data block. By evaluating the data characteristics, we select a portion of hot data to establish the fine-grained index. The hot data block with high cardinality, we invoked *IvIndexBuilder* to build the fine-grained index. While the data block with relatively low cardinality, we invoked the *BMIndexBuilder* interface to build up the Bitmap index with FastBit technique. When the index data has been generated within a node, we perform a parallel write strategy to speed the index storage. Afterward, Bi-cluster+ examine the query conditions which were parsed by *ShemeParse*. First, the data query procedure locates the data range through the coarse-grained index. And then, use the fine-grained index to search for records. During the query process, we perform load balance and dynamically adjust the size of the logical block to meet the I/O throughput and memory bandwidth requirements.

### 3.1 Optimize the index structure

Bitmap index as a fine-grained index for hot data blocks can significantly improve the query time. However, with the growing cardinality of data files, Bitmap index suffers non-trivial storage overhead and longer query time. Although we perform the FastBit technique to compress the fine-grained index file, it still has an obvious impact on the size of the index file. Hence, we adopt the inverted index as a fine-grained index for hot data blocks with relatively high cardinality. We use a flag to indicate which data blocks are equipped with high cardinality. If the cardinality of data blocks exceeds a given threshold, we establish the inverted index. Otherwise, we build up the Bitmap index for other hot data blocks. In the data query procedure,

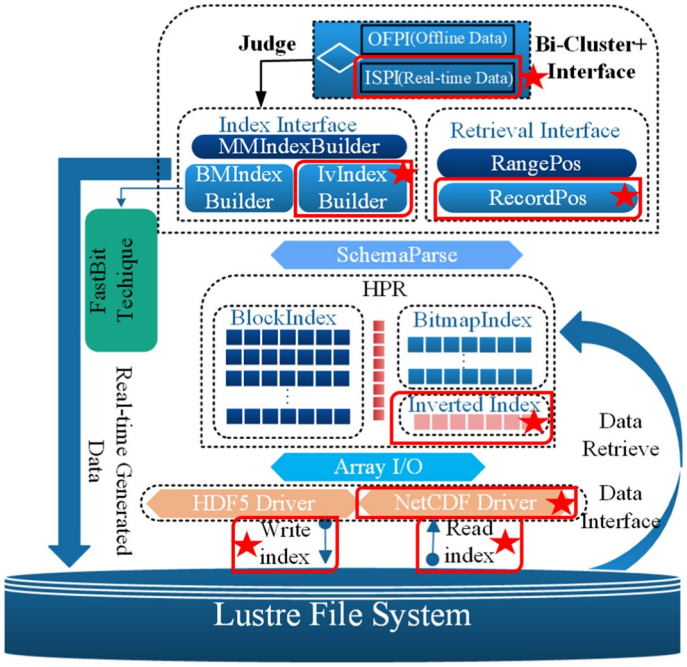


**Fig. 1** Architecture of Bi-cluster+, the red boxes with star explicitly represent the improvements compared with Bi-cluster

we first determine the data range by examining coarse-grained Block index, and then evaluating the flag to perform fine-grained data retrieval.

### 3.2 Improve utilization of computing resources

Establishing the index for large-scale scientific data can be considered as data-intensive workloads. However, traditional supercomputers(referred to as SC) excel at handling the computing-intensive applications. Embedding the data-intensive framework into computing-intensive platforms pose new challenges for the existing computing architecture. Hence, we focus on how to improve the adaptability of Bi-cluster+ on SC. In high-performance infrastructure, despite the abundant computing resources and high memory bandwidth, the existing Master-Slave scheduling diagram may result in inadequate computing resource utilization. Moreover, multiple applications accessing the shared memory bandwidth could also cause resource contention which also degrades the performance of data query framework. Bi-cluster+ exploits the fine-grained computing resource scheduling, scheduled in units of cores, to deal with index generation and data retrieval. Furthermore, Bi-cluster+ introduce the bandwidth-aware mechanism to prevent the bandwidth contention.

As shown in Fig. 2, we perform the hierarchical scheduling to allocate the computing resources. We consider the impact from high-throughput data-intensive index

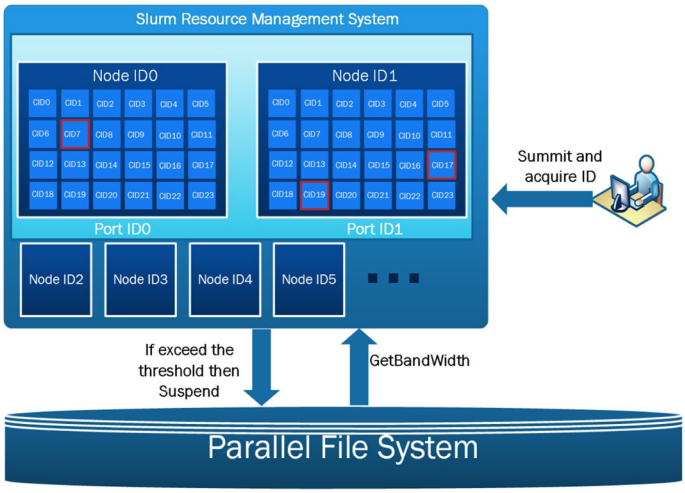


**Fig. 2** Fine-grained computing resource utilization on Supercomputer

generation and data retrieval and schedule the computing resource in a hierarchical way[22]. The cluster in SC is typically composed of a series of homogeneous computation nodes. Supercomputer systems usually separate computing resources from storage resources. Computing resources and storage resources are connected through a high-speed inline network. Each node does not contain or contains only a small number of local disks. Deploying storage resources on supercomputing nodes will increase costs and is not very cost-effective. Because they are also only used for the storage of temporary files. The input and output data of the program still need to be stored in parallel file systems. If computing resources are scheduled in units of cores, it causes the ineffective utilization of computing resources so as to increase the execution time.

Tianhe-2A mounted the Lustre [21] parallel file system which is composed of multiple OST object blocks. The default size of each strip is 1MB, and the theoretical bandwidth can reach 500GB/s. Offline data is often terabytes or even petabytes, and currently each node in Tianhe-2A is equipped with 64GB of memory. For large data sets, the data set needs to be partitioned first. Facing with the constraints of bandwidth resource and memory resource, we investigate how to efficiently handle large-scale scientific data and assume the whole bandwidth $B_w$, the whole memory $M_w$. The request bandwidth vector $B = b_1, b_2, \ldots, b_n$, the required memory vector $M = m_1, m_2, \ldots, m_n$. In order to prevent network congestion and memory overflow, we need to set a bandwidth threshold $B_t$, and memory threshold $M_t$. The utilization of bandwidth and memory must satisfy these constraints.

$$\sum_{i=1}^{n} b_i < B_t, \sum_{i=1}^{n} m_i < M_t \tag{1}$$

**Algorithm 1** Bandwidth-aware Data Query Strategy

**Input:** $N_{build}$, throughput, Bandwidth threshold

```
while throughput < threshold do
    n_task ← Fine_GrainedAllocation(N_build)
    if U_{n_task} < U_threshold then
        BuildIndex(N_data)
    else
        N_build = N_build * 2
    end if
    if CacheQueue ∉ ∅ then
        CQ_i = CacheQueue.pop()
        b_j = GetBandwidth(n_task)
        if throughput + b_j < threshold then
            throughput = throughput + b_j
        else
            ChangeTransRate{CQ_i, threshold − throughput}
            throuput = Threshold
        end if
        ResumeTrans(CQ_i)
    end if
end while
```

As mentioned in the above analysis, multiple tasks may bring contention to each application. Hence, we implemented a bandwidth-aware data query scheduling strategy. Due to the lack of the local disk of each computing load, loading the data chunks into memory occupies high memory bandwidth which may cause I/O competition for other tasks, thereby affecting the normal operation of other tasks. Meanwhile, multiple data blocks in memory may exceed the memory capacity. Hence, we employ hierarchical computing resource scheduling and assign a portion of computing cores to generate index and a portion of cores to write data along with the index file to the parallel file system. We monitor the memory usage and memory bandwidth in real-time. We denote the threshold of memory usage using $U_{threshold}$, threshold represents the threshold of memory bandwidth. When the memory space is inadequate, we assign more cores to write the index and reduce the number of cores to construct the index. If the memory bandwidth is inadequate, on the one hand, we dynamically distribute more cores to deal with data blocks to satisfy the bandwidth constraints. On the other hand, we cache a portion of data blocks to *cacheQueue* and adjust the transmission rate.

### 3.3 Optimize the in situ index generation

We observe that constructing the index for the offline data from the parallel file systems is redundant if we can construct the in situ index while the scientific computing is executing. The index generation for offline data experiences extra two stages. First, we need to read the data from parallel file systems to memory. And then after the index has been constructed, we still write the data from memory to parallel file systems. Transferring large-scale data is time-consuming and occupies lots of computing resources. To tackle this issue, we further optimize the in situ index generation to directly call the index generation method for calculation during scientific computing. Along with the index file, we write the data to a parallel file system.

Task calculation procedure and index construction procedure can be performed in parallel. Compared with the offline data index generation method, in situ index generation can be regarded as an optimized index construction for real-time computing and future scientific computing.

In situ index generation demands a partition of cores as scientific computing and the other to write the data along with index files into parallel file systems. For instance, each node of Tianhe-2A has 24 cores, and some cores can be designated as computing cores, and some cores can complete the process of index writing. We use openMP primitives #pragma omp parallel *num_thread*($K$) for task computation and #pragma omp parallel *num_thread*($24 - K$) for index reading. The computer architecture is shown as Fig. 3. We use *cacheQueue* to temporarily cache the real-time data and accumulate the data to a fixed-size data block. And then, we invoke the index generation procedure to construct the index. During the construction of the index, we evaluate the data feature based on the variances and cardinality. After constructing the index file, we designate the computing resource to write the data along with the index file to the parallel disk.

### 3.4 Parallel index writing process

As for numeric type application, the distribution of some desired data may concentrate on a certain area. Therefore, we can utilize the partial sort by speeding up data retrieval. Accordingly, an inverted index is a good choice to realize this process owing to the low time complexity. Once the inverted index has been built, we can retrieval data at the constant time cost when the index file along with the data loaded into memory. However, establishing the whole data block results in extremely time-consuming. Based on previous analysis, we analyze the data feature and screen out

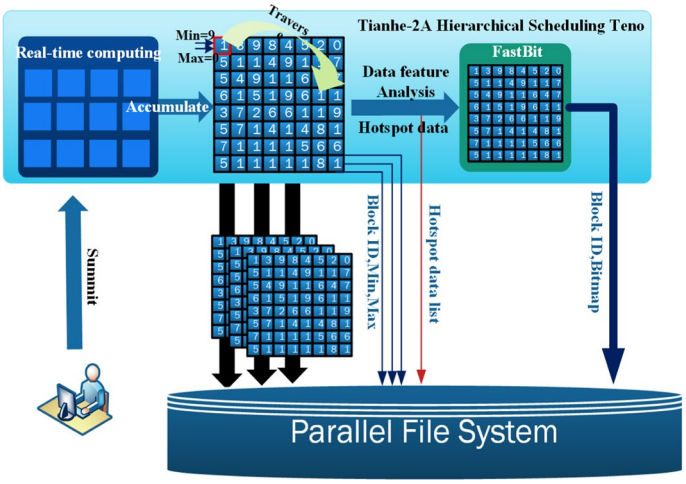


**Fig. 3** Optimization of in situ real-time index generation

the hot data block. We integrate the two fine-grained index and parallel the secondary data retrieval and select the hot data block to build up the inverted index.

Scientific data generate coarse-grained block Min–Max indexes and fine-grained Bitmap indexes after scientific computing. These two indexes are created differently. Min–Max is the maximum and minimum value of the recorded data block. There are only two constant values for each data block. For Bitmap indexes, although the size of each data block is assigned the same size, the generated index will not be the same, because the size of the index generated by Bitmap is related to the data characteristics of the data block. The index block generated for high-cardinality data blocks will be large, while for low-cardinality data, the generated index will be smaller. With regard to inverted index, the size of the index file depends on the data distribution. Hence, we can parallel write procedures for coarse-grained indexes. As for the fine-grained index, we employ a lock mechanism to guarantee the correctness of the data file.

Min–Max block index can perform parallel write indexing when the offset value is known. The index of the data block only needs to record Min and Max data. Each data block needs to perform the same task. The load task is relatively simple and the running time is relatively short, which can be ignored. After the offset value of the Min–Max index has been calculated, the calculated index value can be directly written to the specific location of the index file. So the offset value can be set to directly write the parallel write index. In the construction of the Bitmap index and inverted index, the offset of each data block is different and the base values contained in it are different, and the size of the generated index is correspondingly different. When writing the index load, you need to know the size of the offset value before you can continue to write down the index of a data block. The index is calculated according to the ID of each data block, but due to the uncertainty of the index length, a synchronization lock mechanism needs to be used in the index. Only after the previous index block is written, the next index writing task can be performed. The inverted index is relatively complicated. However, only the hit block can be indexed and the offset is fixed. The Min–Max block index, inverted index and Bitmap index establishment process are shown in Fig. 4.

### 3.5 Load balance and dynamic logical data block merge

In terms of data retrieval, in the original version, we utilize the physical data block as a data partition. The size of the physical data block is fixed. When the query condition is set to retrieve some relevant scattered data, the load imbalance may occur. If we adopt the physical data partition, we just can merge the successive data blocks which can't deal with the low I/O throughput. So in the extended version, we conduct the logical data block which can adjust the data size flexibly. When we retrieve the scattered data point, we can merge the request together and then we can figure out the load imbalance so as to make use of the computing resource and speed up the data retrieval time. When dealing with intensive data requests, the I/O contention may occur. In the same way, we can combine the successive logic data blocks and then reduce the amount of data requested.

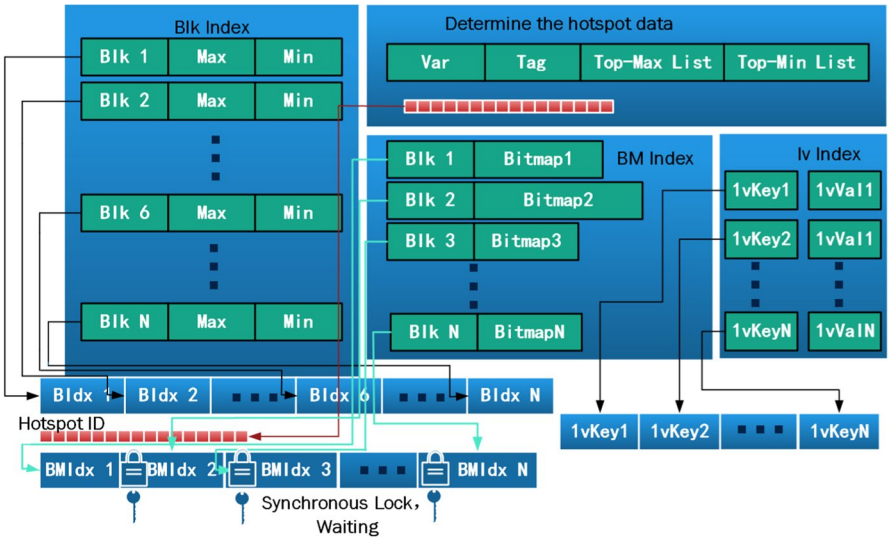


**Fig. 4** Optimization of write load balance

In the query procedure, we first evaluate the block index to access the relative data. Then Bi-cluster+ utilizes two fine-grained indexes to screen the desired data. We adopt the load balance strategy to evenly distribute the data query workloads. When the bandwidth is subject to the bandwidth threshold, if the framework monitors the scattered I/O request, then we merge the logical request to enhance I/O throughput. If the high I/O request results in I/O contention, then we split the merged data block. Therefore the merged read and load balancing techniques mentioned above can still be applied to further optimize their I/O requests. The design is shown in Fig. 5.

## 4 Experiments and analysis

We have evaluated the performance of our Bi-cluster+ framework on Tianhe-2A platform using several real and synthetic data sets. Our framework is compared with the original Bi-cluster, FastQuery, Scan, and block index technologies for the same data sets. In Tianhe-2A platform, each compute node is equipped with two 12-cores Intel Xeon E5-2692 v2 CPUs and 64GB of memory. The details of the test configuration description of compute nodes are shown in Table 2.

The experimental data sets include ClimateGZ, VPDS, COSM, TCP, and PNWNAmet2015. The characteristics of data sets are as follows:

– ClimateGZ is the meteorological data set of Guangdong Province, and the experimental data set size is 3T. The data set collected temperature, humidity, precipitation, radar map, and other weather information of each region in

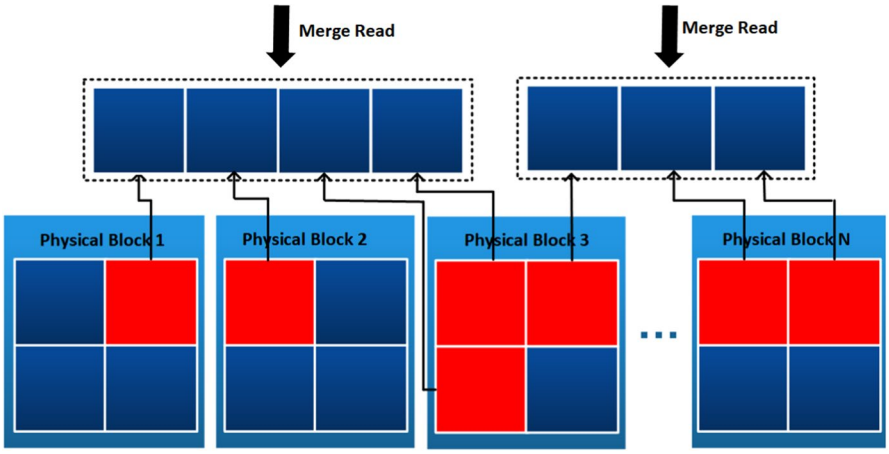


**Fig. 5** Optimization of data retrieval

**Table 2** Compute node configuration description

| Items | Values |
|---|---|
| CPU | Intel Xeon CPU E5-2692 v2@2.2GHz |
| Operating System | Linux 3.10.0 |
| Memory size | 64GB(DDR3) |
| Complier | Intel ifort version 14.0.2 |
| Optimization | -O3 -mavx |

Guangdong since 2000. Forecast the future weather according to historical weather data, and prevent extreme weather such as typhoons.

- VPDS data set is generated by a plasma physics simulation software. The experimental data set is a subset, which has 24 billion particles. The data set size are 900GB. Each particle is associated with seven one-dimensional variables that describe its energy and location.
- COSM data set is generated by simulating the universe observation software. It contains 97.4 billion records of one-dimensional double precision values. The experimental data set size are 974 GB.
- TCP (Three component particle) data set is generated by PiBase simulation software. It can generate real-time data sets of different data sizes for testing. The experimental data set size include 2GB, 8GB, 32GB, 128GB, 512GB, and 2.048TB.
- PNWNAmet2015[30], NetCDF meteorological forcing dataset over a domain covering northwest North America, was created using the trivariate thin plate spline interpolation method and comprised of numerical maximum tempera-

ture, minimum temperature, precipitation and wind data. It records the meteorological data from 1945 to 2012. The entire experimental data are 51.5GB.

In terms of index generation, we compare index generation time on multi-nodes. In terms of data retrieval, we evaluate the throughput and data positioning time, which have been compared with the retrieval reduction ratio of existing frameworks.

### 4.1 Index generation experiment

HDF5 files are the most commonly used in scientific data processing files. NetCDF is usually used to store ocean data. HDF5 uses a hierarchical tree structure to organize data. It is easy to separate variables, and it has rich library support. It's not hard to get simple data structures into NetCDF format, but manipulating them down the road is kind of a pain. Therefore, although our Bi-Cluster+ supports NetCDF, it only performs retrieval tests on it. At present, most of the scientific data in Tianhe-2A was grouped in HDF5 format.

#### 4.1.1 Offline data index generation comparison

We conduct the index generation experiments of FastQuery, block index, Bi-cluster, and Bi-cluster+ on a single node. The process consists of reading data, building indexes, and writing indexes. By analyzing the index time of the four index generation methods at different scales of the data set, the offline data indexing performance is evaluated.

This experimental data set uses ClimateGZ. This data set is divided into four different size datasets: 3GB Small Data Set (Small), 30GB Medium Data Set (Medium), 300GB Large Data Set (Large), and 3TB Mass Data Set (Huge). We conduct the comparative experiments with regard to Bi-cluster, FastQuery and Block Index. The time to generate an index under the dataset of different sizes is shown in Fig. 6.

With regard to offline data located in a parallel file system, we need to load the data files into memory. Each node in Tianhe-2A is equipped with 64GB. However, the size of the scientific dataset exceeds the memory space. So it is necessary to read the data in batches. When the query workload is performed on a single node, the minimum data block size is set to 64MB on the condition that scientific data are Small and Medium. The Small Dataset is divided into 48 data blocks. Each core needs to calculate 2 data blocks and every time it can process 128MB tasks. The medium dataset is divided into 480 data blocks. Each core needs to calculate 20 data blocks and every time it can establish the index for 1.28GB tasks. However, the process of reading data can be completed in one pass, and the entire data set indexing process can be completed within the node, without the overhead of MPI communication between nodes. We adopt openMP parallel shared memory fork-join mechanism and make full use of multi-core parallel to complete the experiment. The indexing time for a small data set is the I/O time generated by one read data block and the hierarchical index processing time for two data blocks. The indexing time

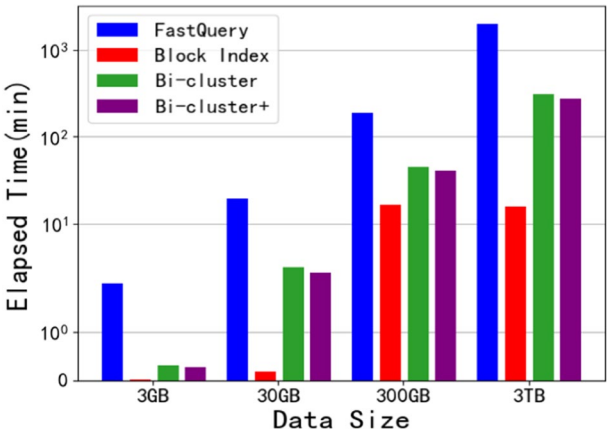


**Fig. 6** Index generation time overhead in single node

for a medium data set was generated by reading a data block, I/O time and 20 data blocks to build a hierarchical index.

Index size is also an important factor to measure the quality of an index method. Table 3 shows the index generation sizes of the Bi-Cluster and Bi-Cluster + frameworks under different data sizes of the ClimateGZ dataset. In meteorological data, it is mainly to analyze whether extreme weather occurs, and variance is a good indicator to represent key data. According to variance, we set 5%, 15%, and 25% as thresholds, respectively. Build fine-grained indexes on the data within the threshold range. The results show that the index size generated by Bi-cluster+ is significantly smaller than that generated by Bi-Cluster. Bi-Cluster only uses bitmap index as fine-grained index. Bi-Cluster+ uses the combination of inverted index and bitmap index. Bitmap index is related to the cardinality of the data when the index is established. The higher the cardinality of the data block, the larger the volume of the bitmap index generated accordingly. While Bi-Cluster+ selects data blocks with low cardinality for bitmap index, and those with high cardinality for inverted index, which effectively reduces the size of index files.

We scale the experiment on 512 nodes by using four 900GB scientific datasets. In the experiment, both Bi-Cluster and Bi-Cluster+ adopted 15% hot spot data

**Table 3** Index size comparison between Bi-cluster+ and Bi-cluster in single node

| | 3 GB | 30 GB | 300 GB | 3 TB |
|---|---|---|---|---|
| Bi-cluster(5%) | 12.41 MB | 105.75 MB | 985.34 MB | 8.49 GB |
| Bi-cluster+(5%) | 11.37 MB | 76.43 MB | 637.25 MB | 6.21 GB |
| Bi-cluster(15%) | 35.70 MB | 317.30 MB | 2.91 GB | 26.17 GB |
| Bi-cluster+(15%) | 26.19 MB | 217.33 MB | 1.77 MB | 20.38 GB |
| Bi-cluster(25%) | 59.72 MB | 526.47 MB | 4.14 GB | 42.50 GB |
| Bi-cluster+(25%) | 37.52 MB | 349.19 MB | 3.02 GB | 26.21 GB |

to establish fine-grained index. We only select a part of the hot spot data for the construction of fine-grained index, which reduces the computational task without greatly reducing the accuracy of the retrieval. Most of the data indexes that reduce the construction are difficult to be retrieved again in the later retrieval process, so it is not necessary to construct the index for these data. The experimental results show that, as shown in Fig. 7, block index is the fastest and FastQuery is the slowest. When reading records, a variable record is used for continuous reading and writing. In the case of a single node, the entire data block is indexed, and a synchronization process is required when writing the index. In Huge data, FastQuery indexing takes more than 12 hours, and the indexing time is too long. As the size of the dataset increases, this overhead is unacceptable. Therefore, it is necessary to adopt parallel to speed up the process of index construction. In all cases, the execution time of Bi-cluster+ is less than or similar to that of Bi-cluster. However, the time overhead of Bi-cluster+ has less improvement than that of Bi-cluster. Scientific data have the characteristic of "one write, more read." We favor the benefits from the query time over the generation time.

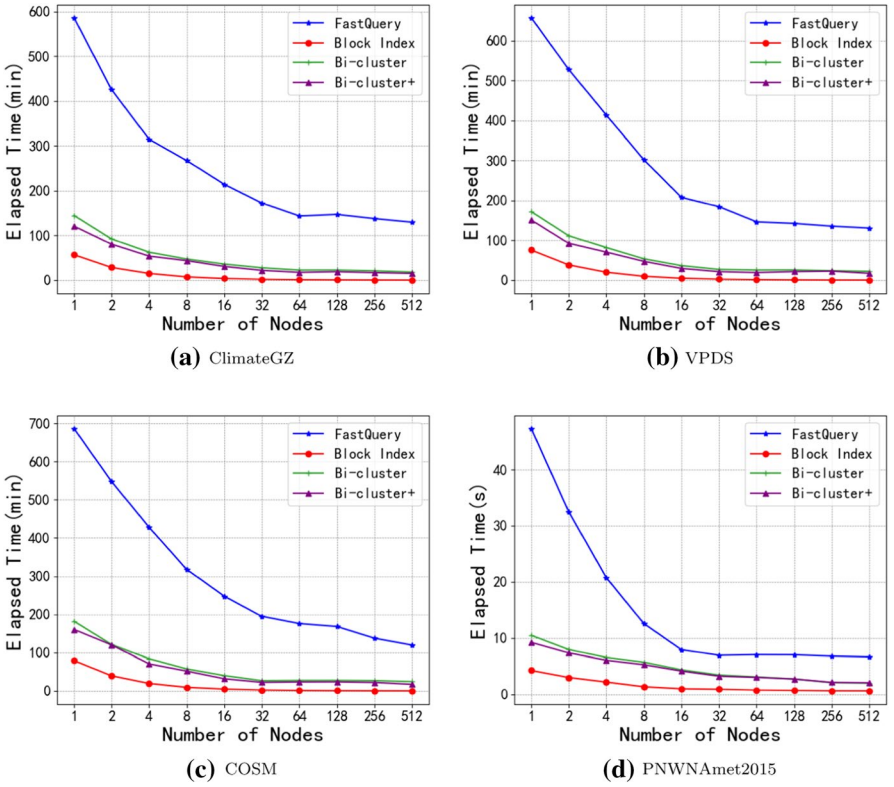


**Fig. 7** Index generation time overhead in multi-nodes

### 4.1.2 In situ real-time data index generation comparison

The off-line data generation index method is mainly for the data on the disk. During the indexing process, the data should be read into the memory before the index can be established. During the reading process, the cost of reading data is generated, and the indexing process and writing The indexing process incurs overhead. The in situ index generation method is to directly index the data when the scientific task is executed, and write the generated data to the parallel file system along with raw data. Compared with reading offline data and then allocating the computing resources to build the index, the in situ index generation method can save computing resources and reduce the I/O overhead of massive data. Therefore, reducing the overhead of in situ indexing is very meaningful. The optimization of the in situ indexing method is a more effective indexing method, providing an interface to the existing scientific task framework, so that the index can be directly established.

From the analysis of Bi-cluster, it can be concluded that when the data set is large, a part of the core is allocated as a calculation index within the node, and another part of the core is used for index writing, which is better than the inter-core index generation method, because it can be in the same node. With shared memory, data can be shared, which reduces data migration overhead. Therefore, we adopt intra-core parallel index generation method for comparison. We compare Bi-cluster+ with Bi-cluster by choosing different three-component particle (TCP) data size, such as 2GB, 8GB, 32GB, 128GB, 512GB, 2.048TB. There are tested in 48 nodes. The results are shown in Fig. 8.

When the size of the data set is 2GB, with 64MB as the minimum unit of logical data block partitioning, the 16 cores of each node need to calculate about 41.7MB of data, and the remaining 8 cores calculate the index in real time and write the index synchronously. In the process of scientific task calculation, each node deal with 41.7MB of data quickly, while the indexing process is relatively slow. When indexing, the synchronization problem is affected by the characteristics of the data set.

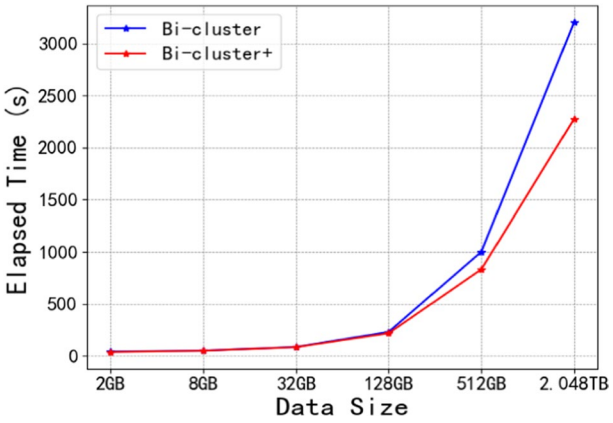


**Fig. 8** Comparison of Bi-cluster+ and Bi-cluster by TCP dataset

When writing an index, record each node number, data block number, and original file and save them to the disk. The offset value of the block index and the inverted index is fixed, and the index offset of the Bitmap is related to the cardinality of the data block, so there will be write synchronization problems.

Bi-cluster+ optimizes the indexing process while making full use of computing resources. As the data set increases, although there exist write synchronization problems in Bi-cluster+, it has improved compared to Bi-Cluster in general. When the data set is terabytes in size, Bi-cluster+ has significant advantages and can achieve 1.4 times acceleration.

### 4.2 Data retrieval experiment

Data retrieval performance is a considerable important indicator for evaluating query performance. This section mainly conducts throughput comparison experiments and retrieval time comparison experiments. We evaluate FastQuery, block index, Bi-cluster and Bi-cluster+. We analyze four retrieval methods I/O throughput and query data time by searching a certain portion of data.

This experiment was conducted on 64 nodes. The size of the dataset is 900GB. The throughput of the four retrieval methods is shown in Table 4. Higher throughput means faster query time and better performance. From the analysis of the throughput of four scientific data sets, the maximum throughput of VPDS is nearly seven times that of ClimateGZ, and the throughput of ClimateGZ is the smallest. This is because each particle is associated with seven one-dimensional variables during retrieval, and in this retrieval, only the temperature in ClimateGZ is queried, and the rest of the scientific data set does not need to be read into memory. In terms of retrieval methods, the throughput of block indexes and high-performance parallel retrieval methods is large, because the value of the block index is continuously read in units of data blocks. FastQuery decreases in order with the size of binning accuracy. Because the binning accuracy is higher, the range of bitmap index construction is relatively small, so the amount of data retrieved is also reduced. Therefore, when retrieving a small amount of data, the advantage of the Bitmap fine-grained index is even greater. The Bi-cluster+ has achieved a good I/O throughput compared with the other solutions. The performance of netCDF is also satisfying. From the perspective of throughput, one-component data, e.g., maximum temperature, minimum temperature as well as longitude and latitude data generate relatively lower throughput.

**Table 4** Retrieve I/O throughput comparison

| Datasets | FastQuery | Block Index | Bi-Cluster | Bi-Cluster+ |
|---|---|---|---|---|
| ClimateGZ | 103.4 MB/s | 14.5 GB/s | 7.6 GB/s | 10.3 GB/s |
| VPDS | 5.9 GB/s | 432.7 GB/s | 216.5 GB/s | 274.7 GB/s |
| COSM | 431.7 MB/s | 84.3 GB/s | 49. 3 GB/s | 69. 5 GB/s |
| PNWNAmet2015 | 91.2 MB/s | 13.7 GB/s | 6.9 GB/s | 9.1 GB/s |

According to the retrieval constraints, we set the retrieval data proportion as $10^{-3}$, $10^{-4}$, $10^{-5}$, $10^{-6}$. The metric measured by the retrieval time is I/O throughput divided by the retrieval time. The more I/O throughput, the shorter the retrieval time, the more efficient the retrieval. We name the metric absolute throughput, expressed by the symbol $R_{out}$, $Q_{I/O}$ is I/O throughput and $T_{query}$ is the retrieval time. The formula is as follows.

$$R_{out} = Q_{I/O}/T_{query} \tag{2}$$

Table 5 compares the query performance of FastQuery, block index, Scan, Bi-cluster, and Bi-cluster+ under different query selectivity settings. Speedup is calculated as the execution time of other method divided by the execution time of Bi-cluster+. Compared to the block index retrieval method, the time spent on coarse-grained queries is the same. But when the data is read into memory, the block index needs to scan all the records to get the data within the retrieval constraint. The speedup of Bi-cluster+ over scan improves from 7.3× to 74× for PNWNAmet2015 dataset as the query selectivity decreases. Compared with Bi-cluster, our performance can achieve 1.1 to 1.9 times improvement, especially when the retrieval is $10^{-6}$, the retrieval efficiency of Bi-cluster+ reaches the highest. From the above experimental results, the optimized Bi-cluster outweigh the original one.

The optimized one combined the hit block from different physical data blocks. The data relativity is increasing. The cache hit ratio is calculated as the amount of valid data obtained divided by the total amount of data based on the query criteria. Based on this, we conduct the cache hit/miss comparative line chart as shown in Fig. 9. The cache hit ratio of both methods improved as the query selectivity decreases. Due to the addition of a variety of fine-grained index techniques, Bi-cluster+ has a significantly better cache hit ratio than Bi-cluster. When the retrieval is $10^{-6}$, the cache hit ratio reaches 63.25%.

**Table 5** Speedup of Bi-cluster+ over FastQuery, Block index technology, Scan, and Bi-cluster

| Datasets | FastQuery | | | | Block Index | | | |
|---|---|---|---|---|---|---|---|---|
| | $10^{-3}$ | $10^{-4}$ | $10^{-5}$ | $10^{-6}$ | $10^{-3}$ | $10^{-4}$ | $10^{-5}$ | $10^{-6}$ |
| *ClimateGZ* | 1.6 | 2.3 | 2.9 | 3.2 | 3.5 | 4.7 | 6.7 | 13 |
| *VPDS* | 1.7 | 3 | 4.3 | 5 | 2.3 | 3.7 | 4.9 | 8.5 |
| *COSM* | 2.7 | 4.1 | 5.4 | 6.1 | 3.7 | 4.9 | 5.1 | 7.6 |
| *PNWNAmet*2015 | 2.1 | 3.3 | 4.6 | 5.1 | 2.9 | 4.7 | 5.0 | 7.2 |
| Datasets | Scan | | | | Bi-cluster | | | |
| | $10^{-3}$ | $10^{-4}$ | $10^{-5}$ | $10^{-6}$ | $10^{-3}$ | $10^{-4}$ | $10^{-5}$ | $10^{-6}$ |
| *ClimateGZ* | 7 | 11 | 23 | 69 | 1.3 | 1.5 | 1.6 | 1.9 |
| *VPDS* | 6.2 | 9.2 | 14.5 | 67 | 1.2 | 1.4 | 1.6 | 1.8 |
| *COSM* | 7.7 | 9.9 | 17.4 | 62 | 1.1 | 1.4 | 1.7 | 1.9 |
| *PNWNAmet*2015 | 7.3 | 9.4 | 26.3 | 74 | 1.2 | 1.3 | 1.5 | 1.7 |

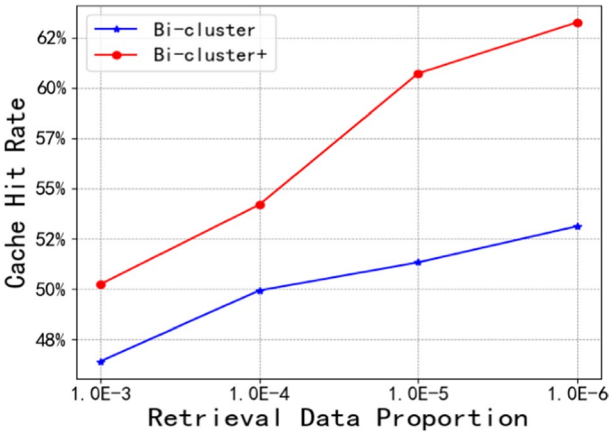


**Fig. 9** Cache hit rate of Bi-cluster and Bi-cluster+

### 4.3 The scalability evaluation

Scalability is an important indicator of the robustness of a framework. So we conduct experiments to analyze the scalability performance. The scalability of Bi-cluster+ framework mainly includes strong scalability experiments and weak scalability experiments. The strong scalability test refers to the same data set, using multiple nodes to determine the performance of the method used, and running on multiple nodes to determine the performance of the method in a parallel environment. The weak scalability test evaluates the stability of our design method as increase the size of the data set and the number of computing nodes in proportion. If the data set size is doubled, the amount of computing resources will be doubled.

The strong scaling of the experiment retrieve the $10^{-6}$ data of ClimateGZ dataset. The size of dataset are 300GB, and the maximum expansion is 512 nodes. The test is performed on 12,288 cores. When there is only one node, such as single-node experimental analysis, the size of each data block is 64MB, and each node requires 720 data blocks, which are processed in 7 times. As the number of nodes increases, the size of the data block will gradually decrease. When the nodes are expanded to 512, each node needs to process 586MB of data, that is, each core only needs to calculate 25MB of data. The test results are shown in Fig. 10.

It can be seen from the above figure that before 64 nodes, the model expansion performance showed a linear expansion trend. When the number of nodes exceeds 64, the indexing speed is affected, and the computing power of Tianhe-2A nodes is not fully utilized. With the increase in the number of nodes, the reduction time of index construction is not obvious. In this regard, the data read time, index build time, and write index time are separated for analysis, as shown in Fig. 11. It was found that at 64 nodes the Bitmap fine-grained index took a long time. Index writing time accounts for an increasing proportion of the entire index generation time. This is because after 64 cores the number of available cores is 1536 and the number of nodes is 64. The index calculation time is very fast. For writing Bitmap indexes, the

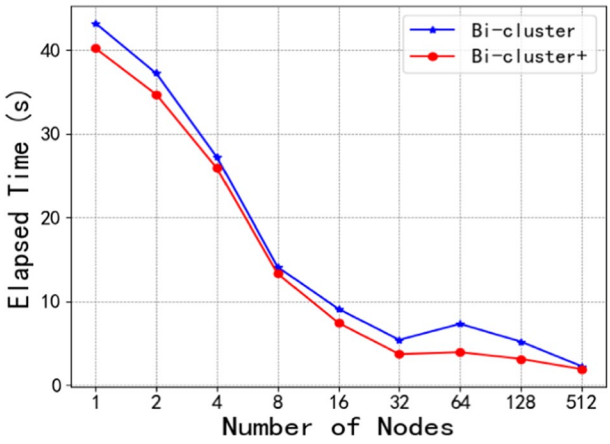


**Fig. 10** Strong scalability of Bi-cluster and Bi-cluster+

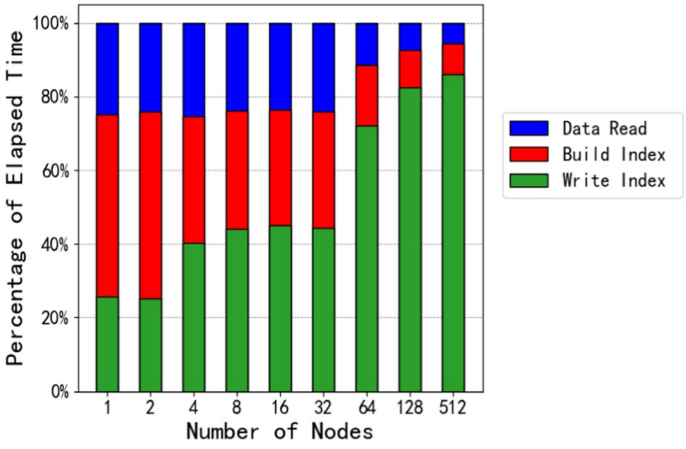


**Fig. 11** Time breakdown of index generation

writing process increases, and the synchronization overhead increases. Multiple processes are queued for index writing. In this way, even if the number of cores continues to increase, the calculation time will be limited by the writing of the index file. Writing the index becomes the bottleneck of the entire indexing process and affects the expansion of performance. But overall, Bi-cluster+ has better strong scalability than original Bi-cluster for the same datasets.

The weak scalability test increase the number of computing nodes and the amount of data in proportion. Ensure that the size of data set processed on each processor is consistent. The experimental data set is TCP, which is generated by the simulation software of PiBase. For computing nodes of different scale, the dataset sizes are 3GB, 9GB, 27GB, 81GB, 243GB, 729TB, and 2.187TB, respectively. The law of the computing nodes and data set is to expand by 3 times. The test results are

shown in Fig. 12. From the experimental results, we can see that the weak scalability of Bi-cluster+ can achieve almost linear expansion. Compared with Bi-cluster, Bi-cluster+ has better weak scalability.

## 5 Conclusion

This paper proposes a high-performance query framework called "Bi-cluster+," which exploits the characteristics of supercomputers to reduces index size, index build time and data retrieval time. In terms of index generation, a hierarchical index structure is proposed, including the coarse-grained block index, the fine-grained bitmap index and the inverted index. By using more accurate index techniques as the secondary index of block index, the limitations of block index are compensated. First, block index is constructed for the original scientific data. Through the analysis of the characteristics of scientific data, select a certain proportion of hot data blocks. After sort these hot blocks, bitmap index and inverted index are established for hot spot data to speed up the query process. The adaptability of data-intensive indexing workloads executed on computation-intensive platforms is improved through hierarchical computing resource scheduling and memory bandwidth-aware data query mechanisms. By modeling the shared memory mechanism, the kernel parallel index generation mechanism is further optimized. The logical block merge read strategy is designed to maximize the query I/O throughput by reading multiple continuous or discontinuous logical data blocks at one time under the condition that the distance from the bandwidth threshold is close enough, so as to overcome the unstable I/O throughput of parallel shared file systems such as Lustre.

Bi-cluster+ framework is evaluated on a supercomputer with parallel file system using multiple real scientific datasets. The results show that it can reduce query time by a factor of 1.1 to 74 comparing to existing approaches, including FastQuery,

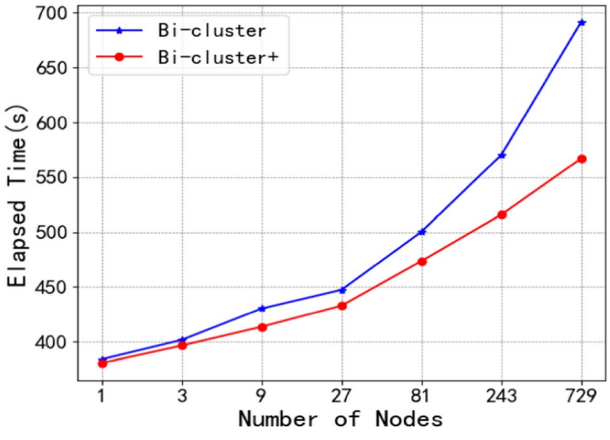


**Fig. 12** Weak scalability of Bi-cluster and Bi-cluster+

block index, scan, and original Bi-cluster. The scalability of Bi-cluster+ can keep a good performance by evaluating on 17496 cores.

**Acknowledgements** Supported by National Key R&D Program of China (2018YFB0204303), National Natural Science Foundation of China (No. 61872392 and U1811461), Guangdong Natural Science Foundation (2018B030312002), the Program for Guangdong Introducing Innovative and Entrepreneurial Teams under Grant (No. 2016ZT06D211), the Major Program of Guangdong Basic and Applied Research (2019B030302002), NSF of Hunan (No. 2019JJ40339), and NSF of NUDT (No. ZK18-03-01).

## References

1. The hdf5 format. http://www.hdfgroup.org/HDF5
2. Netcdf. http://www.unidata.ucar.edu/software/netcdf
3. Aguilera MK, Golab W, Shah MA (2008) A practical scalable distributed b-tree. Proceedings of the Vldb Endowment
4. Behzad B, Luu HVT, Huchette J, Byna S, Prabhat Aydt RA, Koziol Q, Snir M (2013) Taming parallel i/o complexity with auto-tuning. In: International conference on High Performance Computing
5. Blanas S, Wu K, Byna S, Dong B, Shoshani A (2014) Parallel data analysis directly on scientific file formats. ACM
6. Bowers KJ, Albright BJ, Yin L, Bergen B, Kwan TJT (2008) Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. Phys Plasmas 15:199–434
7. Chen C, Huang X, Fu H, Yang G (2012) The chunk-locality index: an efficient query method for climate datasets. In: 26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum, pp 2104–2110
8. Cheng L, Wang Y, Pei Y, Epema D (2017) A coflow-based co-optimization framework for high-performance data analytics. In: Proceedings of 46th International Conference on Parallel Processing, pp 392–401
9. Cheng P, Wang Y, Lu Y, Du Y, Chen Z (2019) Uniindex: an index and query middleware for parallel file systems. Concurr Comput Pract Experience 32(10):1
10. Chou J, Howison M, Austin B, Wu K, Ryne RD (2011) Parallel index and query for large scale data analysis. In: High Performance Computing, Networking, Storage & Analysis
11. Chou JC, Wu K (2011) Prabhat: Fastquery: A parallel indexing system for scientific data. In: IEEE International Conference on Cluster Computing (CLUSTER), Austin, TX, USA, Sept 26–30, 2011, pp 455–464
12. Dong B, Byna S, Wu K (2013) Sds: A framework for scientific data services. In: Parallel Data Storage Workshop
13. Hey T (2012) The fourth paradigm—data-intensive scientific discovery. Springer, Berlin
14. Jha S, Qiu J, Luckow A, Mantha P (2014) Fox GC A tale of two data-intensive paradigms: Applications, abstractions, and architectures. CoRR arXiv:abs/1403.1528
15. Kim J, Abbasi H, Chacon L, Docan C, Wu K (2011) Parallel in situ indexing for data-intensive computing. In: Large data analysis & visualization
16. Kraska T, Beutel A, Chi EH, Dean J, Polyzotis N (2017) The case for learned index structures
17. Lakshminarasimhan S, Shah N, Ethier S, Klasky S, Latham R, Ross RB, Samatova NF (2011) Compressing the incompressible with ISABELA: in-situ reduction of spatio-temporal data. In: Euro-Par 2011 Parallel Processing - 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29–September 2, 2011, Proceedings, Part I, pp 366–379
18. Li J, Liao WK, Choudhary A, Ross R, Zingale M (2003) Parallel netcdf: A high-performance scientific i/o interface. In: Supercomputing, ACM/IEEE Conference
19. Liao X, Xiao L, Yang C, Yutong LU (2014) Milkyway-2 supercomputer: system and application. Front Comput Sci 8(3):1
20. Liu Q, Logan J, Tian Y, Abbasi H, Podhorszki N, Choi JY, Klasky S, Tchoua R, Lofstead J, Oldfield RA (2014) Hello adios: the challenges and lessons of developing leadership class i/o frameworks. Concurr Comput Pract Exp 26(7):1453–1473
21. Schwan P (2003) Lustre: building a file system for 1000-node clusters. In: Proceedings of the the 2003 Linux Symposium

22. Shen Y (2018) Teno: An efficient high-throughput computing job execution framework on tianhe-2. In: 2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)
23. Shen Y, Peng C, Du Y, Lu Y (2019) Bi-cluster: A high-performance data query framework for large-scale scientific data. In: IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)
24. Shoshani, A., Rotem, D.: Scientific data management: challenges, technology, and deployment. CRC Press/Taylor & Francis (2009)
25. Soroush E, Balazinska M, Wang DL (2011) Arraystore: a storage manager for complex parallel array processing. In: Proceedings of the ACM SIGMOD International cCnference on Management of Data, pp 253–264
26. Stephens ZD, Lee SY, Faghri F, Campbell RH, Zhai C, Efron MJ, Iyer R, Schatz MC, Sinha S, Robinson GE (2015) Big data: Astronomical or genomical? PLoS Biol 13(7):e1002195
27. Stonebraker M, Becla J, Dewitt DJ, Lim KT, Zdonik SB (2009) Requirements for science data bases and scidb. In: Conference on Cidr
28. Stonebraker M, Brown P, Zhang D, Becla J (2013) Scidb: a database management system for applications with complex analytics. Comput Sci Eng 15(3):54–62
29. Wang DL, Monkewitz SM, Lim KT, Becla J (2011) Qserv: a distributed shared-nothing database for the lsst catalog. In: High performance computing, networking, storage & analysis
30. Werner A, Schnorbus M, Shrestha R, Cannon A, Zwiers F, Dayon G, Anslow F (2019) A long-term, temporally consistent, gridded daily meteorological dataset for northwestern north america. Sci Data 6(1):1–16
31. Wu K, Ahern S, Bethel EW, Chen J, Childs H, Cormier-Michel E, Geddes C, Gu J, Hagen H, Hamann BA (2009) Fastbit: Interactively searching massive data. J Phys Conf 180:012053
32. Wu K, Otoo EJ, Shoshani A (2006) Optimizing bitmap indices with efficient compression. ACM Trans Database Syst 31(1):1–38
33. Wu K, Shoshani A, Stockinger K (2010) Analyses of multi-level and multi-component compressed bitmap indexes. ACM Trans Database Syst 35(1):1–52
34. Wu K, Stockinger K, Shoshani A (2008) Breaking the curse of cardinality on bitmap indexes. In: International Conference on Scientific and Statistical Database Management
35. Wu T, Chou JC, Podhorszki N, Gu J, Tian Y, Klasky S, Wu K (2017) Apply block index technique to scientific data analysis and I/O systems. In: Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, Madrid, Spain, May 14–17, 2017, pp 865–871
36. Wu T, Shyng H, Chou J, Dong B, Wu K (2016) Indexing blocks to reduce space and time requirements for searching large data files. In: 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)
37. Zhang Y, Kersten ML, Manegold S (2013) Sciql: array data processing inside an RDBMS. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 1049–1052