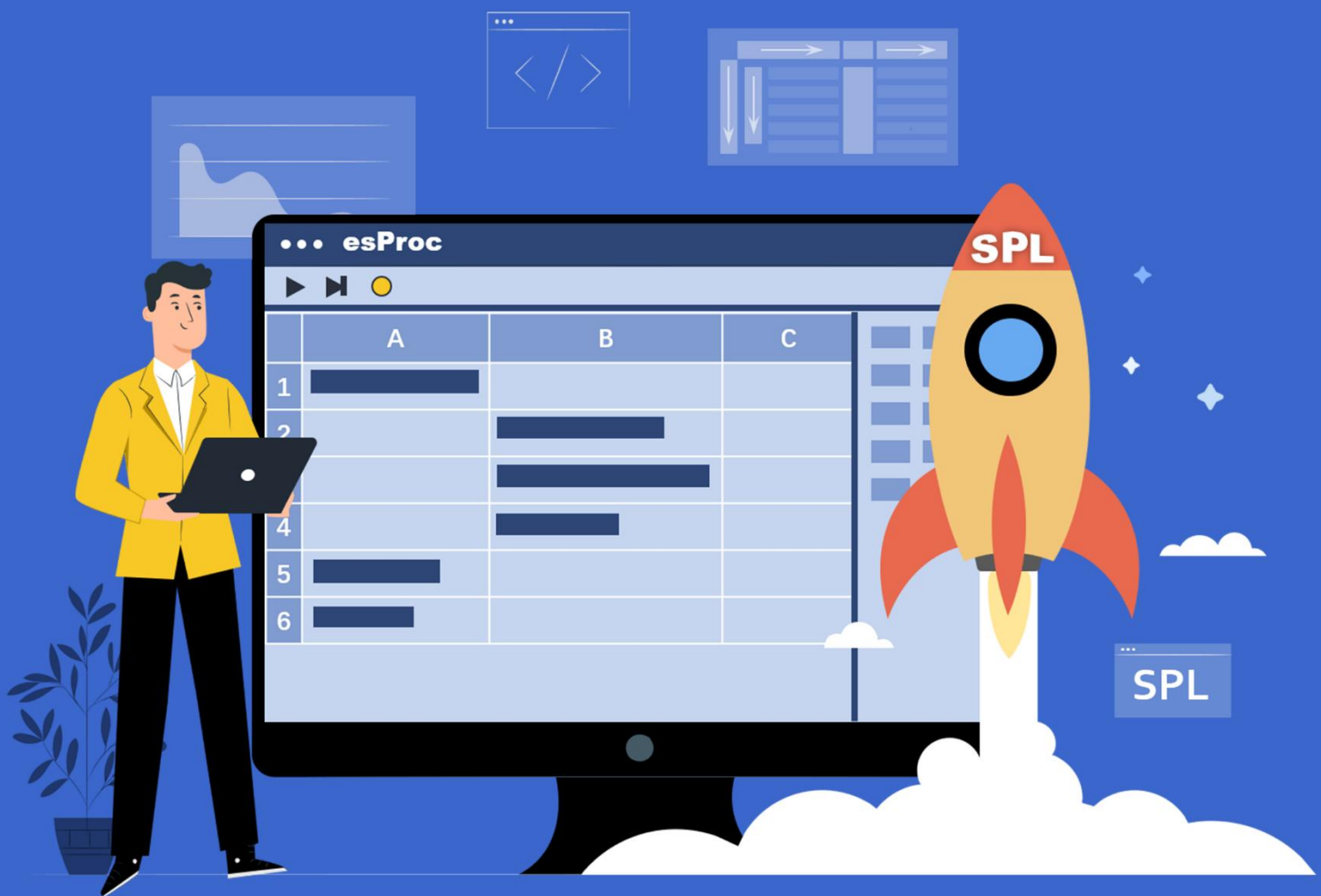
RaqForum

c.raqsoft.com

Performance Optimization



Scudata Inc. | Raqsoft Inc.

Table of contents

Preface

The technical essence of big data is high performance. With sufficient performance, big data analysis can be truly implemented.

Performance optimization should be implemented under limited hardware conditions. Software cannot change the speed of hardware. What we can do is to design algorithms with lower complexity to reduce the actual amount of computation, and naturally we can obtain higher computing performance.

Some big data algorithms have good adaptability and can work in all cases, but they are usually more conservative and difficult to obtain high performance. In order to reduce the amount of calculation, we should carefully study and make use of the characteristics of data and tasks, and design appropriate storage schemes and calculation methods according to actual conditions.

The content of this book is to describe applicable storage schemes and optimization algorithms for different scenarios and objectives. After programmers are familiar with the principles and application prerequisites of these basic algorithms, they can flexibly combine and use them to solve high-performance problems in business. After understanding these algorithms and features, you can also make great progress in the technical selection and understanding of big data products.

The algorithms in this book are mainly oriented to structured data calculation, involving operations such as search, filtering, grouping, sorting and join. These are the basic contents of big data calculation and the most common tasks in data analysis and calculation.

This book does not just simply list and summarize the algorithms in history, many algorithms and optimization technologies are written in the book for the first time in the industry. This book not only discusses high-performance algorithms in theory, but also involves technical means that have no special advantages in complexity but can improve performance in engineering practice.

This book is not for beginners. It has certain professional requirements for readers:

1) Master various operations of relational database and SQL. The meaning of these operations will not be explained in this book.

2) Understand the knowledge equivalent to the data structure course of university computer major, and the relevant concepts will be directly cited.

3) Understand the basic knowledge of algorithm complexity analysis.

4) It is better to be familiar with C/C++ or Java programming language, memory management mechanism of operating system and basic LAN.

The principle and process of some algorithms are cumbersome and difficult. Application programmers do not have to master them. You can also use them as long as you understand the adaptive conditions of the algorithms and are familiar with the application code examples.

This book will use SPL to write application code examples, and directly use SPL data types and syntax to describe calculation objectives, which requires readers to understand in advance. Readers with SPL knowledge can easily convert these terms into the corresponding vocabulary of other programming languages.

SQL is the most commonly used structured data operation language, but it is too rough to apply most of the optimization algorithms in this book. Java, C/C++ and other programming languages still lack the necessary concepts of structured data operation, and to define them from the beginning will take too much length of the book. Moreover, although they can implement and apply these algorithms, the code will be quite long and too much energy will be consumed in details.

SPL may be the only programming language in the industry that can apply these algorithms without too cumbersome. After understanding the mechanism of these algorithms, you can also implement them by yourself in Java, C/C++ and other programming languages, and get better performance.

1 In-memory search

1.1 Binary search

The table sequence T in memory has a field K, given the **search value** k, find the record with the value of field K in T as k. Field K is called the **to-be-searched key**, and the found record is called the **target value** or **target record**.

Conventional sequential search will traverse each member of T in turn for comparison. If there are N records in T, the average comparison number is N/2 and the complexity is O(N). If T is just ordered for K (sequence T.(K) or T.(-K) is an increasing sequence), **binary search** can be used.

Before introducing the algorithm further, let's explain: most of the algorithms discussed in this book are general for table sequence and record sequence, and even valid for sequence. But sometimes it is relatively convenient to use a certain term to describe it. If you understand the principle, you can extend it to other similar scenarios. When applying these algorithms, it is not necessary to strictly abide by the premises and scenarios described in the book, but to understand the principle and then confirm whether the facing scenario is suitable for the algorithm, and sometimes make a few modifications to the algorithm.

Binary search is a common algorithm. Taking T.(K) in ascending order as an example, the basic logic is as follows:

	A	B	C
1	=1	=T.len()	
2	for	=(A1+B1)\2	=T(B2)
3		if k==C2.K	return C2
4		else if k>C2.K	>A1=B2+1
5		else	>B1=B2-1
6		if A1>B1	return null

Compare k with the middle member of T, if they are not equal, exclude half of the members according to the comparison result, and continue to perform this action in the subset composed of the remaining half of the members until the target value is found or the remaining subset is empty. Because half of the members can be excluded each time, you can find the target value (or confirm that it cannot be found) by comparing $\log_2 N$ times at most. The complexity is O(logN). When N is large, it will have a very obvious advantage over sequential search. When N is small, because the process of binary search is relatively complex, it may be slower than sequential search. Usually, when N>=8, binary search will have an advantage.

If T.(K) has duplicate values, there may be multiple target values. After finding one target value using binary search, you need to compare more records continuously forward and backward. When the average number of repetitions is M, each search will make an average of M more comparisons, and the total complexity is O(logN+M). Usually, there are only occasional duplicate values, that is, $M \ll \log_2 N$ is satisfied, and the complexity can still be considered as O(logN). If there are many duplicate values, M cannot be ignored when analyzing complexity. The complexity of sequential

search is always O(N) whether there are duplicate values or not. When there are duplicate values, the complexity of binary search is generally still lower, but in practice, a larger N is needed to show the advantage over sequential search.

SPL provides the binary search option, which can be used directly:

	A	B
1	=10000.new(string(~,"00000"):id, ~\3:K).keys(id)	
2	=A1.find@b("01234")	
3	=A1.select@b(K:1234)	=A1.select@b(K-1234)
4	=A1.pselect@b(K:3210)	=A1.pselect@b(K-3210)

find@b() searches according to the primary key, find one and return. select@b() will handle the situation as if there are duplicate values, pselect@b() only returns one by default, but it will find the first one, and duplicate values are also considered.

It should be noted that logical expressions cannot be used in the parameters of select@b() and pselect@b(), but a colon separator or expressions that return a numeric value should be used, because logical expressions can only calculate out true and false values, and cannot determine the search direction of the next step in case of inequality; A numerical expression of 0 indicates that it is found, and <0 and >0 are used to indicate the search direction of the next step.

You can use SPL to compare the performance difference between binary search and sequential search.

It should also be reminded that the use of binary search requires order in advance. Sorting is a very slow action, and you can't sort first just to temporarily use a binary search, it will only be even slower. Usually, we need to search in a table sequence many times. In this case, we can sort the table sequence first, and then the following multiple searches can get better performance.

1.2 Sequence number positioning

Sometimes the value of the to-be-searched key is just the sequence number (i.e., position) of the target value in the table sequence, or it is easy to calculate the sequence number of the target value through the search value. At this time, the **sequence number positioning** method can be used.

	A
1	=10000.new(~:id, …)
2	=A1(1234)
3	=10000.new(string(10000+~-1):id, …)
4	=A3(int("12345")-10000+1)
5	=10000.new(date(1999,12,31)+~):dt, …)
6	=A5(now()-date(1999,12,31))
7	=A5.groups(month@y(dt):ym; …)
8	=month&y(now())
9	=A7((A8\100-2000)*12+A8%100)

The id field of A1 itself is the sequence number value, and you can directly use the search value as

the position to find the target value. The id field of A3 is also a string converted from continuous numbers, and the sequence number should be reversely calculated when searching. Sequence number positioning is often used for date related search. Dates or months are usually continuous values, which are easy to correspond to sequence number through some operation. The dt field of A5 is an ordered date, and the number of days from the start date is the sequence number of the target value. A7 takes the month and year as the sequence number, and the calculation will be slightly more complicated.

No comparison is required to locate the sequence number. The sequence number is calculated only once, and the complexity of this search is O(1).

In A5 or A7 above, we ensure that each sequence number has a corresponding date when generating data, but there may be a vacant date in the actual business data, if we still use the sequence number positioning, dislocation will happen.

	A
1	=10000.new(date(2000,1,1)+rand(10000)):dt, ...)
2	=A1.groups(dt; ...)
3	=A2(now()-date(1999,12,31))
4	=A2.align@b(10000,dt-date(1999,12,31))
5	=A4(now()-date(1999,12,31))

The dates generated by A1 cannot guarantee to cover every date of the corresponding period, so A2 may have missing dates, A3 may make an error by using the sequence number positioning, and the returned data may not be today's data. After using align() in A4, we can ensure that the record of a date must fall in the position of the corresponding sequence number, and A5 can use sequence number positioning to find it. For the missing date, align() will fill in a null in the corresponding position. In this way, if the target value corresponding to the search value does not exist, null will be returned, and whether the target value can be found can be judged. The @b option of align() means that binary search will be used to find the position during alignment, which will make the alignment faster.

Similar to the sorting mentioned earlier, alignment is also a slower action compared with sequence number positioning. You can't align for a temporary search and then use sequence number positioning. Only if you need to repeatedly search for many times can you align first, so that the subsequent searches can obtain high performance.

After aligning the searched records by sequence number, we can use the efficient sequence number positioning. It seems that it is OK as long as we can find a way to calculate the search value into sequence number.

Of course not. China's citizen ID number, for example, is made up of a series of numbers, which can be understood as a natural number. It is naturally a serial number. But we cannot locate the ID number by aligning it to the natural number and then use the sequence number positioning.

Sequence number positioning requires a sequence of not less than the largest sequence number to store all the searched records and the null values to fill in the vacancy. If the ID number is converted directly into a natural number, the sequence number range will be so large that the current computer

memory cannot bear (10^{17}, the ID number is 18 digits, the last check digit is not counted). The number of sequence numbers to be filled with null is much more than the number of sequence numbers with records. In this case, the sequence number positioning cannot be directly used.

1.3 Position index

Sometimes we want to find the position of the target value in the table sequence, not the target value itself. If the table sequence is out of order for the to-be-searched key (TBS key), binary search cannot be used to improve the performance. After sorting the data according to the TBS key in advance, binary search can be used, but the position information of the target value in the original table sequence will be lost, so we can't complete our search task.

For example, a login system records the login time and user ID of a group of users, sorted by login time. Now we want to find out after the user with the specified ID logged in, for how long there is no other user who logs in.

If we can find the position of the login record of the user with the specified ID in the data table, we can easily complete the task as long as we find the next record to calculate the time difference. However, the original data is not sorted by user ID and we can only use sequential search, which will be very slow. If we sort the data by user ID, the position information will be lost.

Using **position index** can solve this problem:

	A	B
1	=10000.sort(rand())	=20000.sort(rand()).to(10000)
2	=A1.new(elapse@s(now(),-B1(#)):tm,~:id, ...)	
3	=A2.psort(id)	=A2(A3)
4	=B3.pselect@b(id:1234)	=A2.calc(A3(A4),tm[1]-tm)

This code is slightly convoluted and needs to be understood carefully.

When B3 obtains the ordered record sequence by user ID, maintain a position sequence returned by psort() in A3. In this way, after A4 uses binary search to find the position of the target value in B3, use A3 to calculate the position of the target value in the original table sequence A2, and then use positioning calculation to realize the task. The search process (A4 and B4) uses one binary search and two sequence number positioning searches. The overall complexity is equivalent to binary search, which is much faster than the sequential search directly on the original set.

1.4 Hash index

Hash index can be understood as an extension of sequence number positioning.

Use a function to calculate the value of to-be-searched key (TBS key) to a natural number between 1... M, which is called the **hash value** of the record. This function is called the **hash function**. After grouping the records of table sequence T according to the hash value, a two-layer sequence H with length M (i.e. group@n() operation of SPL, where there may be empty members) can be obtained. Now we can use the sequence number positioning for H, after calculating the hash

function of the search value, we can locate the corresponding grouped subset, and then use the sequential or binary search in this grouped subset to further find the target value. The result of this grouping operation to be done in advance is called hash index.

	A
1	=20000.sort(rand()).to(10000)
2	=A1.new(~:id, ...)
3	=A2.group(id%100+1).(~.sort(id))
4	=A3(2345%100+1).select@b1(id-2345)

In A3, the remainder plus 1 is used as the hash function to divide 10000 records into 100 groups. In A4, use sequence number positioning first and then use binary search, the amount of data involved is only 1/100 of the whole data set, and the comparison times will be $\log_2 100$ times less.

The remainder is also a common hash function for searched keys of integer types.

If you are lucky, the number of members of a member in H (a member of H is a set) is relatively average (close to N/M). Since the complexity of hash value calculation and sequence number positioning is O(1), the search complexity after establishing hash index will be reduced by M times, which is equivalent to search a table sequence with length of N/M. If you are unlucky, the distribution of the number of members of a member in H is very uneven, and the hash index search may not improve the performance in extreme cases, which depends on the design of the hash function and the distribution of the TBS key values in the original dataset.

However, luck is usually not too bad. In most cases, hash index will get better performance improvement. It can even have more advantages than using binary search after sorting when the original data is out of order.

The disadvantage of hash index is that it needs to occupy memory space to store the index. If you want to speed up, you need a larger index and it will occupy more space.

SPL provides a hash index for the primary key, which will automatically use the system's built-in hash function.

	A
1	=20000.sort(rand()).to(10000)
2	=A1.new(~:id, ...).keys@i(id)
3	=A2.find(12345)
4	=A1.new(~:id, ...).keys@i(id;100)
5	=A4.find(23456)

When setting the primary key, you can use @i to create an index at the same time. If there is an index in place, the find() function will use it automatically. When creating an index, you can also use parameters to control the length of the hash index (i.e., M value). You can try the impact of different hash index lengths on performance.

1.5 Multi-layer sequence number positioning

As mentioned earlier, we cannot directly use the sequence number positioning to locate the ID number. But in some specific cases, there are flexible means.

We still use the ID number to find people as an example. If the people to be searched have some common characteristics, for example, the birth dates of children who are about to enter school will fall within two or three years, and their regions are also close. These characteristics will lead to the relatively centralized distribution of ID numbers, and not cover the range of all 18-digit natural numbers. Using this characteristic, a multi-layer sequence number positioning can be constructed to reduce the space occupied in the implementation of sequence number positioning.

	A	B	C
1	=T.align@a(999999,int(mid(ID,9,6)))		
2	=A1.run(~=if(~.len()>0,~.align@a(999999,int(left(ID,6))),null))		
3	=A2.run(~.run(~=if(~.len()>0,~.align@a(999,int(mid(ID,15,3))),null)))		
4	'110108200101231234		
5	=int(mid(A4,9,6)）	=int(left(A4,6))	=int(mid(A4,15,3))
6	=A3(A5)(B5)(C5)		

Let T be the table sequence of people to be searched, and ID is the ID number field. A1 divides the members of T into 999999 grouped subsets according to the birth year, month and day digits of their ID. We only use last two digits of the year digits, and there will be no duplication; A2 then divides each non empty grouped subset into 999999 smaller grouped subsets according to the first 6 digits of the ID, that is, the region number; A3 further divides the grouped subset of the grouped subset of A2 into 999 subsets according to the last three digits of the ID (excluding the 18th checking digit). These three lines of code look a little complex, but it is not difficult to understand. The result is a three-layer sequence.

A4 is the search value. In the same way, get the three parts of numbers, that is, A5,B5,C5, and then directly use the three-layer sequence number positioning to locate the target value. The complexity of three times sequence number positioning is still O(1), and it can run very fast.

What is the difference between this multi-layer method and the sequence number positioning mentioned earlier? We take this three-layer sequence as an example to discuss the space it occupies.

In the first layer, if the birth dates of this population are concentrated in two or three years, only about 1000 sequences among the sequences with a length of 999999 are non-empty, while for empty members, there is no next layer sequence, that is, there are only about 1000 sequences in the second layer. The length of each grouped subset in the second layer is also 999999, so the total number of members is about 1000*999999; Similarly, because the regions are close, only about 1000 grouped subsets of each grouped subset in this layer are non-empty, thus, only about 1000*1000 grouped subsets in the third layer will be non-empty. The length of each grouped subset in the third layer is 999 and the total number of members is about 1000*1000*999. The total number of members of the three-layer sequence is less than 2G, and the occupied space is a few G or 10G, which can be loaded in modern computers. As we have discussed before, current computer memory cannot bear single layer sequences.

To sum up: take the multi-layer sequence as a tree structure. If the data meets the above-mentioned characteristics of concentrated distribution, it seems that most branches of the tree will not extend to the deepest level, but break at the shallower level, so it will no longer occupy space. The space occupied by the whole tree may be small enough to be loaded into memory.

Multi-layer sequence number positioning can only be used when there are many empty values in the first few layers after layering. If there are few or no empty values, the space occupied by multi-layer sequence number structure is larger than that of single layer.

In addition to using multi-layer sequence as above, SPL also provides a data type for processing multi-layer sequence numbers, called **serial byte**, which can support up to 16 layers of sequence numbers. The sequence number range of each layer is 0-255, that is, each byte of two long represents one layer.

	A	B
1	func	return k(int(left(A1,2)),int(mid(A1,3,2)),int(mid(A1,5,2)),int(mid(A1,7,4))-1900,int(mid(A1,11,2)),int(mid(A1,13,2)),int(mid(A1,15,2),int(mid(A1,17,1))))
2	=T.run(ID=func(A1,ID)).keys@is(ID)	
3	'110108200101231234	
4	=A2.find(func(A1,A3))	

A1 defines a function that can convert the ID number to a serial byte. You can adjust the order of layers in the serial byte according to the known data distribution characteristics (here, we get the digits according to the ID number from the left to the right directly, not necessarily the best scenario). A2 converts the ID numbers in the table to the serial byte and then creates an index using @s, that is, to construct the above-mentioned tree structure. Then you can use it to find() at high speed.

The use conditions of multi-layer sequence numbers are relatively complex. When there are many layers or the operation of generating a serial byte is complex, sometimes it is not more advantageous than ordinary hash index.

2 Dataset in external storage

2.1 Text file segmentation

Performance problems are often related to a large amount of data, and big data usually cannot be loaded into memory. We should consider the operation schemes of external storage data. Database may be the most common external data storage scheme, but we can't implement optimized storage methods and algorithms in the database, so there's no need to study this scenario.

Considering the engineering feasibility, we will discuss the external storage data stored in the file system.

Text file is a very common file format. Because of its simplicity and universality, it is often used as a medium for data exchange between various data systems. Text files used to store structured data usually have a title row to identify the field names. Each row is a record. The fields in the row are separated by tab or comma, and the rows are separated by carriage return.

The storing scheme of a text file is definite, and the main means to improve its computing performance is only parallel computing. Modern computers usually have multiple CPUs. If a file can be calculated in parallel, it can obtain almost linear multiple of performance improvement.

To implement parallel computing, we need to be able to segment a file and let each thread (CPU) process one of the segmentations separately. For a text file with different length of each row, we can't use the record sequence number (i.e., the row number) to segment like in memory. To get the nth row of text, we need to traverse the previous n-1 rows, which completely loses the meaning of improving performance. Moreover, we even can't know how many rows there are in the file in advance.

The method of bytes location shall be adopted for the segmentation of a text file. The operating system can directly return the number of bytes of the whole file, and also provides a method to quickly locate the specified bytes position in the file. However, a bytes position is not necessarily the beginning of the row (not in high probability). If we read directly from here, we will get a half row record.

The carriage return is used as the separator of the rows (i.e., the records) in the text file, and the carriage return will not appear in the row (record) itself. Using this feature, we can use the method of discarding the head and complementing the tail to realize the random reading of a text file. That is, starting from the specified bytes position, the record is considered to start only after the carriage return is read. When the next carriage return is read, a complete row (record) will be obtained. The characters read before the first carriage return will be discarded, that is, discarding the head; If a segmentation requires that the reading ends at a specified bytes position, it will actually exceed this bytes position until another carriage return appears to ensure the integrity of the row (record), that is, complementing the tail.

SPL has built-in file segmentation reading method, and you only need to specify the total number

of segments and the segment number.

	A	B	C
1	=file("data.txt")		
2	=A1.cursor@t(;4:10)	=A1.cursor@t(;5:10)	=A1.cursor@t(;23:100)
3	=A2.fetch(1)	=B2.fetch(100)	=C2.fetch()

A2, B2 and C2 define three cursors respectively. A2 divides the file into 10 segments and gets the 4th segment; B2 gets the 5th segment; C2 divides the file into 100 segments and gets the 23th segment. Then read out some records in A3, B3 and C3 respectively.

Using this “discarding the head and complementing the tail” method to segment the text file cannot ensure that the number of records in each segment is the same, but can only ensure that the number of bytes in each segment is relatively average.

2.2 Bin file and double increment segmentation

Text files use characters to encode data. Although the universality is good, the performance is very poor. To convert characters into computable values, more computation is needed, and date and time data also need a very complex analysis and judgment process. If the data is directly stored in its form in memory, there is no need to do conversion when reading it out, and much better performance can be obtained.

This is the binary format.

The disadvantage of binary format is that it cannot be viewed directly, and there is no common binary file format standard in the industry.

SPL provides a simple binary format called **bin file**.

	A
1	=file("data.btx").export@b(file("data.txt").cursor@t())
2	=file("data.btx").export@ab(file("data.txt").cursor@t())

A text file can be read out and converted into a bin file. Using the @a option can append data to the existing bin file.

You can try to compare the performance of the same operation using a text file and a bin file, such as simply making a row count:

	A
1	=file("data.txt").cursor@t().skip()
2	=file("data.btx").cursor@b().skip()

A bin file cursor does not need to use the @t option, it must have field names.

Similar to text files, parallel computing after segmentation is also an important means to improve the computational performance of binary files. However, unlike text files, binary files do not have separators between each row, and any value bytes may appear in the stored field values. There is no special value that can be used to separate fields or records. In fact, no separator is also an advantage of binary files, because the number of bytes occupied by many data types is fixed, and there is no need to waste storing a separator (an integer may occupy 4 bytes, and if a separator is added to

occupy 1 byte, the storage capacity will be expanded by as much as 25%).

So how do we do the segmentation at this time?

Early binary files used fixed length records, that is, each record had the same length, so that the bytes position of the nth record could be calculated by multiplication. This method is also used in early databases, so users are required to specify the length of each field when creating a data table in order to reserve the space. Because the actual number of fields occupied by different records may vary greatly, this method requires to reserve space according to the largest one, which is a serious waste. It is rarely used now. At present, modern databases mostly use variable length varchar instead of char.

Another way is to imitate the text file and artificially set a separator to separate two records. However, any byte value may be actual data. Simply using a character as a separator is likely to make an error. Therefore, usually a series of bytes are used, for example, 16 consecutive 255 indicate the end of the record. Generally, this is not the case for normal data, so it will not be misaligned. But this will also cause a lot of waste, and cannot completely eliminate the possibility of mistakes.

At present, the mainstream method is to divide into blocks. Every N records or enough bytes are written as a block. The data in the block is no longer split, and segmentation is performed in all blocks. As long as the number of blocks is enough, the effect of segmentation and parallel computing will not be poor, and there will be many blocks in the case of large amount of data; If the number of blocks is relatively small, it indicates that the amount of data is small, and there is no need for segmentation and parallel computing.

The trouble of using blocks mainly lies in the need for block index information, recording the total number of blocks, where each block starts, and so on. With the addition of data, the number of blocks will continue to increase, and the length of this index will also change. If it is a database system or big data platform, it can have a set of index management mechanism. However, if it is implemented in an ordinary file system, there needs to be a separate file to store the index, which is inconvenient to use.

SPL adopts a method called **double increment segmentation** to realize the appendable blocking scheme using single file in the file system.

1024 (or other number) block index information will be stored in the reserved space in the file header, and then the actual data will be appended. Initially, it is considered that one record occupies a block, and each additional record means an additional block. After addition, the block information should be filled in the index area. When 1024 records are appended, all index blocks are filled.

If there is another action of adding records, it is considered that two records occupy one block, halve the 1024 index block, merge the first and second blocks into a new first block, merge the third and fourth blocks into a new second block,..., merge the 1023th and 1024th blocks into a new 512th block, and clear all the indexes from 513th to 1024th block. Because the actual data is stored continuously, just change the starting position of the data block in each index block.

Now there are 512 empty index blocks, and 512*2 new records can be added. After appending to a total of 2048 records, all index blocks are filled again. If you want to add more records, make another adjustment as just now, change it to every four records per block, and also merge the current

blocks 1 and 2 into a new block 1…. Another 512 blocks index can be vacated to continue appending.

The appending can be carried out on and on, and the block size will continue to double, and the data in the block is always continuous, and there is no redundant separator to waste space. The total number of blocks varies from 512 to 1024 (except when the number of records is less than 512 at the beginning), which also meets the requirement of sufficient number of blocks. The space reserved for the index at the file header is fixed and will not become larger with the addition of data. A single file can realize the mechanism of blocking and segmentation. Moreover, it is easy to count the total number of records (after multiplication, just count the number of records in the last block).

In fact, SPL will not use double increment segmentation immediately. If the total number of records is small (such as less than 1024), it will not do the segmentation. The segmentation scheme will be implemented only when the number of records exceeds a certain number. In this way, if several records are written to the bin file, there will not be an extra file header that looks a little big. When the amount of data is large, the extra file header is not obvious by comparing to data itself.

For the application programmer, these are transparent. After simple appending, a bin file can implement segmented calculation.

	A	B	C
1	=file("data.btx")		
2	=A1.cursor@b(;4:10)	=A1.cursor@b(;5:10)	=A1.cursor@b(;23:100)
3	=A2.fetch(1)	=B2.fetch(100)	=C2.fetch()

The access syntax is basically the same as that of the text file, but if the number of segments exceeds 1024, it is meaningless for the bin file.

2.3 Data types

After using binary files, we can adopt more optimized coding schemes.

An integer may occupy 4 or 8 bytes on the computer. In principle, it is the same size stored in the file. However, in fact, a considerable number of integers are not large. For example, a person's age will not exceed 200. If it is expressed as 4 bytes, 3 bytes are 0. If they are all stored as 4 bytes, it will be a waste.

Don't underestimate this, because the amount of data is often large. If we change the encoding method and store small integers in another encoding method, we can save 2 bytes. If there are 1 billion rows of data, we can save 2G of space and reduce a lot of hard disk access time.

Similarly, there is date time. A complete datetime type will occupy 8 bytes. Often, we only care about the date part, or even the date of recent decades. If the encoding method can be changed, it may be stored in half the space.

However, the coding rules should not be too complex, otherwise decoding will occupy more CPU time. There are many specific codings, which can be designed by yourself after understanding the basic principles.

SPL adopts optimized encoding for small integers, recent dates and short strings when saving bin files. You can try how different is the space occupied by the same number of large integers and small

integers.

The storage space occupied by data in external storage is highly correlated with the data type. The simpler the data type, the smaller the space occupied, and the simplest operation in decoding. For example, an integer is a simple value, the occupied space is relatively determined, and the reading and decoding action is very simple; A string also has a length information, which needs more storage; The date and time itself has no other information, but it will involve relatively complex decoding actions.

If we can change the data type without losing data information, it is possible to save a lot of storage space, reduce parsing actions and achieve good performance improvement.

Among all data types, integer is the simplest and has the best storage and parsing performance. If possible, try to convert other data types to integers.

Date can be converted to the number of days from a certain day, which does not affect the comparison. 50000 days can represent a time period of more than 100 years, which is sufficient in most scenarios. When it is hoped that the components of year, month and day can be obtained more conveniently (some operations need to group by year and month), simply put, it can be converted into an 8-digit decimal integer of yyyymmdd; SPL also provides a method that saves more space, that is, convert the year and month to the number of months from 1970, and represent the day by 5 binary bits (a month has a maximum of 31 days,and a 5-digit binary number can represent any number between 0 and 31), and it is equivalent to ((yyyy-1970)*12+(mm-1))*32+dd. In this way, you can use small integer to represent a date between 1970 and 2140, which basically meets the requirements.

	A	B	C
1	=date(2021,2,12)	=date(2000,1,1)	=year(B1)
2	=interval@d(B1,A1)		
3	=year(elapse@d(B1,A2))	=month(elapse@d(B1,A2))	=day(elapse@d(B1,A2))
4	=month@y(A1)*100+day(A1)		
5	=A4\10000	=A4\100%100	=A4%100
6	=days@o(A1)		
7	=year(A6)	=month(A6)	=day(A6)
8	=((year(A1)-1970)+month(A1)-1)*12+day(A1)		
9	=A8\384+1970	=A8\32%12+1	=A8%32

A1 is the date to be converted and B1 is a base date. A2, A4 and A6 convert A1 into integers in three ways respectively. Lines 3, 5 and 7 inversely calculate the year, month and day components after three coding methods respectively. The integer obtained by A2 method is the smallest, but the amount of inverse calculation is relatively large; The inverse calculation of A4 method and A6 method is less, and the integer obtained by A6 method is smaller. Moreover, SPL provides more convenient basic functions for conversion, and lines 8 and 9 explain the actual execution logic of A6 method.

There are two kinds of **strings**. The text itself generally does not have much room for optimization, such as name, address, etc.; Another kind of string is often a coding method, and its

value range is very small, such as gender, educational background, abbreviation of country or region, etc., in this case, it is only necessary to convert it into the serial number of the coding table:

	A	B	C
1	[...,CN,...,JP,...,UK,...,US,...]		
2	CN	=A1.pos@b(A2)	=A1(B2)

B2 converts A1 into an integer and C2 calculates it back. Search is required during conversion, which is relatively time-consuming (binary search can be used), and the inverse calculation is equivalent to sequence number positioning, which has very good performance. In this way, it takes more CPU time for conversion during storage, but storage and coding have more advantages, and the efficiency of operation is much higher.

It is mentioned earlier that under the appropriate coding mode, small integers will occupy less storage space than large integers. In addition, for SPL, small integers have greater significance. Because SPL is developed in Java, and Java generates objects very slowly, SPL generates all 16-bit integers (0-65535) in advance. If the read integer is found to be within this range, no new object will be generated, which can reduce a lot of computation.

The third method of date conversion just mentioned can convert dates within more than 50 years to 16-bit integers, and the amount of inverse calculation is not large. Although the second method also has a small amount of inverse calculation, the range of converted integers is much larger. It is no longer a small integer, and the storage and operation performance will be poorer.

In fact, the conversion of these data types is also meaningful for in-memory operations.

Integer operation is also less complex than the operation of date and string. It is faster when comparing. Including the calculation of hash function, the calculation of integer is much faster than that of string.

In the previous chapter, we did not pay attention to data types in search operation. There is no difference in complexity analysis, but there is a great difference in practice. When searching, if the to-be-searched key (TBS key) can be converted to integer to store first, for each search, the additional conversion cost of the search value is much less than the benefit of comparison and hash calculation due to the simplification of data type in the search process, and this is also a common engineering means in performance optimization.

The optimization of data types is also reflected in the case of multiple fields.

When we discussed the search algorithms in the previous chapter, all searches were for the TBS key of one field. In fact, these algorithms are also suitable for multi field search. Similarly, there is no difference in algorithm complexity between multi field and single field in theory, but there are great differences in practice. Even two fields need to be represented by a set, and there will be length information. The space occupied and the amount of computation will not be twice that of a single field (assuming that the data types of the fields are the same), but more.

Therefore, if possible, we can consider spelling multiple fields into one field. However, whether this means is effective or not requires specific analysis and measurement. Combining multiple fields into one field can avoid set operation, but it will make the integer larger, or even convert the integer

into a string (to be spelled with another string). Sometimes, the performance loss caused by these cannot offset the benefits of avoiding set operation.

Simply put, if two strings that cannot be converted into integers are put together, or the combination result of two small integers is still not too large (for example, combining two 2-digits will get a 4-digit), merging fields will bring benefits. More complicated situations are not necessarily the case.

2.4 Composite table and columnar storage

Text files and bin files store each record in turn. This method is called **row-based storage**.

Most operations do not use all fields of the data table. Because the hard disk must be read in blocks, almost the whole record must be read out no matter how many fields are read in the row-based storage mode. The optimization that can be done is not to process (such as generating corresponding data objects) after reading, but the reading time of the hard disk is indispensable.

If **columnar storage** is used, redundant reading can be avoided.

The so-called columnar storage is to continuously store the values of the same field of each record in the data table. In this way, if only a few fields are used in the operation, only the data corresponding to these fields will be read, which can reduce the amount of reading and effectively improve the performance.

However, considering that the data will be appended, columnar storage will be much more troublesome than row-based storage. Data is usually appended by records. In case of row-based storage, we only need to append records to the end. In case of columnar storage, we can't do so simply. We need to append the field values behind their respective areas, which requires that there is a reserved space in the field storage area, otherwise it is difficult to ensure the continuity of data of the same field. However, we don't know how many records there are in total, let alone the space occupied by each field, so we can't reserve appropriate space.

The general processing method adopts blocking, and the whole data area is composed of some data blocks with a certain size. The values of the same field will be stored in the same data block. When the data block is full, a new data block will be generated to continue storing. In this way, a field will occupy several data blocks, and it is discontinuous between the blocks, but the field values inside the data block are stored continuously. As long as the data block is large enough (more than 1M of modern hard disk is basically enough), the excess reading of hard disk caused by discontinuity between data blocks can be ignored.

Now that blocks are used, we can conveniently make a **minmax** index on the data block, that is, record the maximum and minimum values of the field values in the block to the block header, which does not take up much space. When performing filtering, if it is found that the field value to be compared is not within the maximum and minimum value range of the current block, the whole block can be directly skipped without reading and parsing, which can further reduce the amount of reading and calculation.

Another trouble with columnar storage is the synchronization of segments.

The number of field values stored in each data block is different. Except for the first data block of each field, any other data block of two different fields cannot guarantee that they store the fields of the same batch of records. The columnar storage realized by simply using data blocks cannot realize segmentation.

The commonly used method in the industry is to batch the data by records. For example, every 1M records are used as a batch, and the columnar storage is used in the above block mode. The segmentation can only be based on the whole batch, and it cannot be carried out within the batch.

This method is relatively effective when the amount of data is very large. Because a batch must be large enough so that the data blocks in columnar storage can continuously store enough data. The number of batches should also be large enough, otherwise the segmentation will be limited, because the segmentation can only be based on each batch. This method is suitable only when the number of records reaches 100 million or even greater.

This problem can be solved by using the double increment segmentation method.

The data is no longer batched, but the a fore mentioned index area is established for the data area (composed of multiple data blocks) of each field, and the starting data block of each segment and the position in the block are recorded in the index area. When adding records, all index areas will be filled and double-increased synchronously to ensure consistency. The number of records (actually the number of field values) corresponding to all the index blocks in the index area is the same. In this way, it can also ensure that the segment with the same serial number of different fields will always correspond to the field values of the same records. Good segmentation effect can be obtained without a large amount of data.

The columnar storage file of SPL is called the **composite table**, which implements the above data storing, minmax index and double increment segmentation mechanism.

	A	B	C
1	=file("data.ctx")	=file("date.btx")	
2	=A1.create(…)	=A2.append(B1.cursor@b())	
3	=A1.open()		
4	=A3.cursor(…;;4:10)	=A3.cursor(…;;5:10)	=A3.cursor(…;23:100)
5	=A4.fetch(1)	=B4.fetch(100)	=C4.fetch()

Unlike bin files, a composite table needs to be created before appending data (A2 creates, B2 appends data). When creating a composite table, we need to first specify the data structure (part... in A2), which is somewhat similar to the need to CREATE TABLE for tables in the database. A4, B4 and C4 generate the cursors of different segments. Note that one more semicolon should be written. When creating a composite table, columnar storage will be used by default. When reading, specify the field names to be used in the parameters (... in A4, B4, C4), so as to take advantage of columnar storage to reduce the amount of reading.

The encoding method of structured data is generally not very compact (even if the optimization method mentioned in the previous section is used), so there is often some room for compression. After the data is compressed, the space occupation of the hard disk can be reduced, so as to reduce

the reading time, but decompression will increase the amount of CPU computing and consume more computing time. The compression effect is also related to the algorithm used. Algorithms with high compression rate usually occupy more CPU time during decompression. Therefore, whether to compress or not is not a definite choice. It can only be determined according to the actual situation or even after certain tests.

Columnar storage is more conducive to data compression than row-based storage. In structured data, the data type of the same field is generally the same, and even the values are very close in some cases. The data blocks composed of such a batch of data usually have a good compression rate.

Columnar storage and compression are used by default when creating a composite table. SPL provides options for the user to choose row-based storage or no compression.

When column storage is adopted, if a field in the data table is orderly, it means that the field values of adjacent records are more likely to be the same. In this way, only the number of duplicates and one value can be stored instead of storing the same value many times, and the saved space is considerable.

In order to use this scheme, we can sort the data in advance and then store it in the file. However, a data table can only have one sort scheme. When it is ordered by field A, it cannot be ordered by field B at the same time. Which field should be preferred in sorting?

If we do not need to consider other factors, we can generally choose the field with more repetition to be listed first. To be precise, if T.id(A).len() is less than T.id(B).len(), then sort by A first and then by B, the space occupied is usually less.

When the ordered data is appended to the columnar storage composite table, SPL will automatically execute the above scheme, only record the value once and number of repetitions, without user intervention.

2.5 Order and data appending

Even if the amount of storage is not reduced, ordered storage is of great significance for searching and traversal. We will gradually talk about how to use order to improve operation performance later.

	A
1	=file("data.ctx").create(#ID,...)
2	=file("data1.ctx").create(#K1,#K2,...)

When creating a composite table, if the field names in the parameter are prefixed with #, it indicates that the composite table will be ordered by these fields (in the order of fields, and must be the first few fields). However, SPL does not check when appending. Programmers need to ensure that the written data is indeed orderly.

The trouble with ordered storage mainly comes from data appending, as the new data may not always be concatenated to the end of original data in an orderly manner. For example, the existing data of composite table are sorted by ID field, and the ID field of new data usually contains the same batch of values. In this case, to ensure the data of whole composite table are in order by ID, we need

to re-sort all data by ID in principle, rather than simply appending new data to the end of existing data. However, sorting big data is a very time-consuming action.

Fortunately, since the large amount of original data are already in order, we only need to sort new data and then use the low-cost ordered merge algorithm to merge new data with original data, which is equivalent to reading and writing all data only once, and avoids the phenomenon of generating many temporary files in conventional big sorting algorithm, hereby obtaining fully ordered data more quickly.

	A
1	=file("data.ctx").open()
2	=file("data_new.ctx")
3	>A1.create(A2)
4	=A1.cursor()
5	=A2.open()
6	=file("data_append.btx").cursor@b()
7	>A5.append([A4,A6].merge())

This action can be simplified as:

	A
1	=file("data.ctx")
2	=file("data_new.ctx")
3	=file("data_append.btx").cursor@b()
4	>A1.reset(A2;A3)

Or merge new data file directly onto the original file:

	A
1	=file("data.ctx")
2	=file("data_append.btx").cursor@b()
3	>A1.open().append@m(A2)

For big data, it is very time-consuming even just to read and write all data. To speed up, a dual-file approach can be adopted, that is, store data in two files. One file stores the large amount of historical data and the other stores the recent data. When appending data, the new data are only merged into the latter file in general, and merge all data after an appropriate period of time.

	A	B
1	=file("data_history.ctx")	
2	=file("data_recent.ctx")	
3	=file("data_new.btx").cursor@b()	
4	if (day(now())==1	>A1.append@m(A2.open().cursor())
5		>A1.create@y(A2)
6	>A2.append@m(A3)	

This code shows that the historical data and recent data are merged on the first day of the month, and on the other days of the month, the new data are merged into the recent data file. If we append data once a day, the recent data file will store data for up to one month, and the historical data file will store all data from a month ago. In other words, the historical data file may be very large and the

recent data file is relatively small, which makes the amount of data to merge each day not large, and allows us to append data quickly, and the time-consuming full data merging is done only once a month.

After the adoption of the dual-file approach, it needs to read data from the historical data file and recent data file respectively when fetching data from the composite table, and then merge the two parts of data and return the merged result so as to ensure the returned result set is still in order.

	A
1	=file(“data_history.ctx”).open().cursor()
2	=file(“data_recent.ctx”).open().cursor()
3	=[A1,A2].merge(…)

Due to the need to merge data when accessing compsite table, the performance of reading two files will be slightly lower than that of reading one file.

This coding method is a bit complicated. With the advent of multi-zone composite table, SPL can simplify this code.

2.6 Data update and multi-zone composite table

We previously only discussed how to append data in the external storage data table, not how to modify it.

We have been trying to store data as compact and continuous as possible in order to reduce the storage capacity of the hard disk and reduce the reading time. After compact storage, the data cannot be updated directly, and the modified data may not be able to fill the empty space of the original data (because there are various compression codes, the same type of data does not necessarily occupy the same number of bytes). If it can't be filled in, it can't be stored in the original location. If it can't fill in enough, it will also cause empty space. Either case will cause the data to be no longer continuous and compact, especially for insertion and deletion. Columnar storage will exacerbate this problem, which is faced by data blocks of all fields.

The OLTP database, in order to adapt to the frequent modification requirements, usually adopts the uncompressed row storage method, which is not suitable for the OLAP services dominated by reading and computing. Moreover, even so, it is still possible to generate many discontinuous data blocks after multiple writings, resulting in a serious decline in accessing performance.

Data modifiability and high-performance computing are a pair of contradictions. It is impossible to realize high-performance operation with a huge amount of data and allowing frequent and large modifications at the same time.

This book focuses on high-performance computing, so we sacrifice the need for frequent and large modifications.

Both text files and bin files can be considered not to support modification (modification means rewriting all the file), and composite tables can support a small amount of modifications.

When the amount of modification is small, SPL writes the modified (also including inserted and

deleted) data to a separate data area, which is called the **supplementary area**. When reading, the supplementary area data is merged with the normal data, so that the existence of the supplementary area cannot be felt during access. Moreover, to ensure high-performance merge processing, the amount of data in the supplementary area should not be too large.

To modify, there must be a primary key. SPL requires that the primary key of a composite table must be in order (that is why there is an insert action. Relational databases do not distinguish between insert and append).

	A	B
1	=file("data.ctx").open()	
2	=10.new(rand(1000):ID,...)	>A1.update(A2)
3	=10.new(rand(10000):ID,...)	>A1.delete(A3)

Because too many modifications are not allowed, SPL only provides the use of in-memory record sequence as a parameter to perform modification and deletion actions, rather than the use of a cursor during appending. When modifying, if the primary key value does not exist in the composite table, it will be inserted.

After many times of modification, the supplementary area will become larger and larger, and the impact on performance will increase. Therefore, the supplementary area needs to be cleaned on a regular basis: merge it into the original data area to ensure compact and continuous storage, and then remove it .

	A	B
1	=file("data.ctx").open()	
2	if (day(now())==1	>A1.reset()

reset() will mix the supplementary area into the normal storage area, and rewrite part of the data (starting from the smallest primary key value in the supplementary area, and the previous data will not be changed) to maintain the compact and continuous storage of whole composite table.

The historical data used by OLAP service generally does not have a large number of frequent updates, but sometimes batch deletion of historical data is necessary. The data from too many years ago have lost their significance for query analysis. If they are still stored in the data table, it will occupy a lot of space and affect the query performance. However, we require data to be stored continuously. Even if the data itself is in chronological order, deleting the data in the initial period will lead to all data rewriting, which is very time-consuming.

SPL provides a **multi-zone composite table**, which allows multiple files to form a composite table. These files are called the **zones** of the multi-zone composite table.

	A	B
1	=file("data.ctx":to(12)).create(#dt,...;month(dt))	
2	=file("data.ctx":[5,6,7,8,9,10,11,12]).open()	

When creating a composite table, we can set that the composite table is composed of multiple physical files. Each file will have a number called **zone number**, that is, the to(12) part of the parameter in A1, indicating that the composite table will be divided into 12 files with zone numbers

of 1, 2,..., 12. At the same time, an expression for calculating the zone number from the data is given, here is month(dt). When using append@x() to append data, SPL will calculate month(dt) for each appended record and append it to the file of the corresponding zone number according to the result.

When using, we can select only part of the zones to form a logical composite table. These zone numbers can be discontinuous, but must be orderly. When early data needs to be deleted, just delete the file of the corresponding zone, and then open the composite table without adding this zone to the list. There is no need to rewrite the data in other zones.

Similarly, the historical data file and recent data file mentioned in the previous section can also be respectively stored as two zones of multi-zone composite table (if the amount of historical data is large, it can be stored as multiple zones). In this way, multiple files can be regarded as a composite table logically to access, making the code more concise.

3 Search in external storage

3.1 Binary search

If the records of the data table stored in the file of external storage are in order to the to-be-searched key (TBS key), as long as the file format can support relatively arbitrary segmentation, we can implement external storage binary search to avoid traversing the whole file sequentially.

First divide the file into two segments, get the first record of the second segment, compare it with the search key, and then decide whether to continue searching in the first half or the second half according to the comparison result. Then divide the file into four segments, get the first record of the 2nd or 4th segment, continue the comparison, and then divide the file into eight segments, ... , finally find the target value.

If there are duplicate values, it will be more complex. To search forward, we can directly continue to traverse; To search backward, we need to continue to do binary search in the previous part until we find the first searched value, which is much more troublesome than in-memory search.

Binary search is also applicable to interval search, that is, to find the records of the TBS key in a certain range. Just find the first starting searched value, and then traverse forward until all the ending searched values are found. There is no need to search on either side.

This process is not difficult to understand, but it is cumbersome to implement. SPL has done the encapsulation and it can be used directly:

	A	B
1	=file("data.txt")	
2	for 1000	=10000.sort(rand()).to(9000).sort()
3		>A1.export@at(B2.new(A2*10000+~-1:ID,...))
4	=A1.iselect@t(123456,ID)	
5	=A1.iselect@t(123456:234567,ID)	

A1-B3 generates 9 million lines of text file with orderly ID. Using iselect() function, we can find the target value in an ordered file using binary search, and it also supports the searched value in a certain interval. You can compare the speed difference with traversing using a cursor.

Bin files can also use this function.

	A
1	=file("data.btx"))
2	=A1.export@b(file("data.txt").cursort@t())
3	=A1.iselect@b(123456,ID)
4	=A1.iselect@b(123456:234567,ID)

Composite tables are OK, of course, but will use another function.

	A	B
1	=file("data.ctx")	=file("data.btx").cursor@b()
2	=A1.create(#ID,…)	>A2.append(B1)
3	=A1.open()	
4	=A3.find(123456)	
5	=A3.cursor(;ID>=123456 && ID<=234567)	

When using find(), the TBS key is required to be the primary key of the composite table, that is, it is unique in the composite table. If you perform an interval search (multiple target values will be returned), you can directly generate a cursor and add conditions. SPL will automatically jump to the appropriate data block to read the data according to the order of the key, instead of actually traversing.

3.2 Hash index

When using binary search, we still need to read the original file many times to locate the target value, and many readings are redundant in the process. If we can get the physical position of the target value efficiently, we can read the target value directly.

Establish an association table between the to-be-searched key (TBS key) value and the physical position of the target value, quickly obtain the physical position from the association table with the search value, and then read the target value. This association table is the external storage **index**.

There are many TBS key values in the index, so it is necessary to perform the same search action for the index as for the data table. The difference is that we can specially design the index structure to facilitate fast search.

The **hash index** will calculate the hash value of the TBS key value in the original data table, store the TBS key value and its corresponding record physical position according to the hash value, and store the records with the same hash value together.

Usually, the original data table is very large, and the index is also too large to be stored in memory. At this time, a small index should be established for the index itself. The range of hash value is known in advance and can be divided into several segments for storage. For example, the hash value of 1-100 is stored as a segment, and 101-200 is stored as another segment to ensure that each segment is small enough to be loaded into memory. A small index with a fixed length is saved in the index header to record the hash value range stored in each segment and the physical position and length of this segment. When the search value is determined, calculate the hash value, read out the small index first, and find out which segment to search in the next step. At this time, the in-memory search technology can be used. Generally, the hash value of each segment is stored in order, and binary search can be used. If the number of hash values stored in each segment is fixed, sequence number positioning can also be used. Then read out this segment, find the corresponding physical position of the target value (this is also an in-memory search technology), and then read it out from the original data table.

To sum up, there are three steps: firstly, read out the small index in the header, secondly, read out the index segment where the search value is located, and thirdly, read out the target value using the

physical position. In this way, the target value can be found after reading a total of three times, and the use of index has obvious advantages over binary search. Moreover, the small index in the header can be always loaded in memory after being read once, and it is not necessary to be read again for the following search, and the amount of reading can be further reduced.

When the amount of data is very huge, the small index in the header may be too large to be loaded into memory. At this time, it may be necessary to create another smaller index for the small index. That is, the index may be divided into multiple levels. We use the loading order to represent the index level. The first loaded minimum index is called the first-level index. Find the appropriate second-level index through the first-level index, and then find the third-level and fourth-level (four levels are large enough). The last level can locate the physical position of the target value. The maximum number of reads per search is the total number of index levels, plus the one that reads the target value. The first-level index can also reside in memory without repeated reading.

Another problem with hash index is that the hash function may have bad luck, resulting in too many records corresponding to a slice of hash values to be stored in one index segment. At this time, we need to do the second round of hashing, and change a hash function to redistribute these records. With bad luck, there may be a third round, a fourth round, …. As a result, the level of hash index is not fixed, and there may be more levels for some search values.

The whole process is not simple. The external storage task is indeed much more troublesome than the in-memory task.

SPL provides indexing function for composite tables:

	A	B
1	=file("data.ctx").create(ID,…)	
2	=1000.sort(rand())	
3	for A2	=10000.sort(rand())
4		>A1.append(B3.new(A3*10000+~-1:ID,…).cursor()))
5	=file("data.ctx").open()	
6	=file("data.idx")	
7	=A5.index(A6:1000000;ID)	
8	=A5.icursor(;ID==123456;A6).fetch()	

First, generate a disordered composite table. Do not add # before the ID field in A1.

A7: create an index for the composite table of A5; the TBS key is ID; the index file is data.idx. When creating a hash index, it needs to give a hash value range (1000000 in this code). In this way, we can use corresponding index file (A6 in this code) to search in A7.

Different from the previous binary search, the index search of the composite table always assumes that the amount of returned data may be large, so a cursor is used.

Theoretically, we can also establish index according to the record sequence number and physical position, so that we can logically realize the effect of sequence number positioning. However, through the above analysis, we can see that the time cost of external storage search is mainly in the processing of index segmentation and related multi-level indexes. Even if we use sequence number

positioning, we also need to go through the same process, and the time saved by sequence number positioning in in-memory operation cannot be felt any more, so the method of sequence number positioning is no longer specifically proposed for external storage search.

3.3 Sorting index

Sorting index is more common in external storage.

The hash index can only do equivalence search for the search value, that is, the judgment condition is equal, not interval search, that is, the judgment condition is that the to-be-searched key (TBS key) is in a specified interval. Moreover, the hash index may have two rounds of hash when it is unlucky, and its performance is not stable enough. Sorting index can solve these problems.

The principle of sorting index is to store the TBS key values in order in the index, and then use binary search to find the physical position of the target value corresponding to the searched value. This scheme can do equivalence search or interval search.

In practice, we cannot simply do binary search to the index directly. It is still similar to hash index, we need to divide the index into several segments and establish a small index for these segments. First read out the small index, find the index segment corresponding to the searched value using binary search, and then read out the index segment to find the physical position of the target value. The small index can also reside in memory to reduce the amount of reading.

Similarly, when the amount of data is huge, the index will be divided into levels. However, the sorting index can determine the number of records stored in each index segment in advance, and the second round of hash caused by bad luck with hash index will not happen here. If 1K values are stored in each level, four levels can correspond to the huge scale of 1K*1K*1K*1K=1T records, which is more than enough. Generally, three levels are enough.

Sorting index is usually a little slower than hash index for equivalence search, but the difference is not great. Because it has a wider adaptability, most external storage indexes are sorting indexes. Hash index will be used only in individual scenarios that have extreme performance requirements and only need to realize equivalence search. The external storage index mentioned later in this book refers to the sorting index if not stated otherwise.

The SPL default index is the sorting index.

	A
1	=file("data.ctx").open()
2	=file("data.idx")
3	=A1.index(A2;ID)
4	=A1.icursor(;ID==123456;A2).fetch()
5	=A1.icursor(;ID>=123456 && ID<=654321;A2)

If the hash range is not specified in the index() function, it will be considered to establish the sorting index. Interval search can also be performed based on the sorting index.

Why do we make a sorting index instead of sorting the original data table directly by the TBS key?

This is mainly a size problem. There are only the TBS key and physical position in the index, which is equivalent to a data table with two fields. The original data table often has dozens or even hundreds of fields, which is much larger than the index. If the whole table is sorted, it will occupy at least twice as much storage space.

Moreover, even if the data is ordered, the performance of performing binary search to the original data table is still not as good as that of using index. As mentioned earlier, direct binary search has no multi-level index, cannot locate precisely, and has many futile reading actions. Simple binary search is usually used in scenarios where performance requirements are not very high and you do not want to maintain indexes. Although the index is smaller than the original data, it still occupies considerable storage space, and it needs to be updated at the same time when the data is appended.

The essence of index is sorting, which is also the principle in database. Therefore, it is very slow to build an index for a big data table in the database, and the addition, deletion and modification of a table with index will be much slower, because an orderly index should be maintained at the same time. After understanding this principle, indexes should be consciously established and avoided in database project.

Sometimes we need to use two fields as the TBS key. Theoretically, this is no different from a single field, only the comparison action is a little complicated. However, after understanding the essence of index is sorting, it can also guide the establishment of this index.

For example, there are two fields A and B as the TBS key, so how do we build the index? Do we need to build an index for A and B respectively, or do we build a joint index of (A,B)? The cost of building an index is not low, so we should build as few indexes as possible.

If it is ordered for A, it is generally not ordered for B. While if it is ordered for (A,B), it must be ordered for A, not necessarily for B. We can draw the following conclusions:

1) Building separate indexes for A and B will be helpful for (A,B) joint search, but only one index can be used. For example, using the index of A will traverse the conditions of B in the records that meet the relevant conditions of A, but cannot use the index of B after meeting the relevant conditions of A.

2) The joint index of (A,B) can also be used for the search of A, but it cannot be used for the search of B, let alone for the search of (B,A). Indexing is built in a clear order.

If we are allowed to build two indexes, we should build indexes for (A,B) and (B,A). In this way, the search conditions for A, B, (A,B) and (B,A) can all use the indexes. The effect of building indexes only for A and B is much worse.

SPL implements these strategies to automatically find the best index. These principles are also valid for databases. Good commercial databases can all intelligently find the most appropriate index.

3.4 Row-based storage and index with values

As mentioned in the previous chapter, columnar storage is a common means to improve performance. However, for most search tasks, columnar storage will lead to worse performance.

Even if it has been stored in order, the usual columnar storage can be regarded as unable to perform binary search that does not depend on the index. The reason is the same as the difficulty of

columnar storage segmentation mentioned before, that is, the synchronization of each column cannot be guaranteed. The columnar storage using double increment segmentation mechanism can perform binary search (that is what the composite table does), but it also needs to read the data from the data blocks of each field separately, and the performance will not be very good.

Index can also be established on the columnar storage data table to avoid traversal, but it is very troublesome. Theoretically, if we want to record the physical positions of each field in the index, the index size will be much larger than the index of row-based storage, and may even be as large as the original data table (because each field has a physical position, the amount of data in the index is the same as the original data, only the data type is simple). Moreover, when reading, it is also necessary to read in the data areas of each field, and the hard disk has a minimum reading unit, which will cause the total reading amount of each column to far exceed that of row-based storage, and the result is that the search performance is very poor.

Columnar storage is more suitable for data traversal. For scenarios that require high-performance search, try not to use columnar storage.

SPL adopts the double increment segmentation mechanism, and can quickly find each field value in columnar storage according to the record sequence number, so what SPL actually stores in the index is not the physical position of each field value, but the sequence number of the whole record. In this way, the size of the index can be much smaller and has little difference from the row-based storage. However, the data blocks of each field still need to be read separately, and the performance still cannot catch up with the index of row-based storage.

By default, the SPL composite table uses columnar storage, but also provides row-based storage mode. You can use option @r to specify when creating:

	A	B
1	=file("data.ctx").create@r(ID,…)	
…	…	

For scenarios with high requirements for both traversal and search, we can only trade space for time. The data is stored twice redundantly. The columnar storage is used for traversal and the row-based storage is used for search.

SPL also provides a kind of **index with values**. When establishing the index, other field values can be copied at the same time. The original composite table can continue to use columnar storage for traversal, while the index itself has stored field values and used row-based storage. Generally, the original composite table is no longer accessed during search, which can obtain better performance, but it will also occupy more storage space.

	A
1	=file("data.ctx").open()
2	=file("data.idx")
3	=A1.index(A2;ID;…)
4	=A1.icursor(…;ID==123456;A2).fetch()
5	=A1.icursor(…;ID>=123456 && ID<=654321;A2)

When creating an index in A2, copy the fields to be referenced in the... part of the parameter, and

it can copy these field values in the index. When getting the target value later, as long as the involved fields are in this part, it is not necessary to read the original composite table. You can compare the performance and spatial differences between valued indexes and ordinary indexes.

3.5 Index preloading

We know that the index of big data is often very large, and multi-level indexes need to be established. For each search, we need to read in them level by level before we can finally locate the target value. Because of the high complexity of external storage access, even if the operating system cache can avoid the actual hard disk action, there will still be more in-memory moves and object generations. The time to read and sort out these index segments becomes the main cost of search by index.

Of course, if it's just one search, it takes very little time. On modern computers, it can be completed at the millisecond level, which is generally not felt. However, if you have to search hundreds or thousands of times, you will feel an obvious delay.

When processing index search, the database will generally automatically cache the index segments that have been read. If they are used again in a short time, they will not be read again, so as to effectively reduce the search delay. If the new search values always involve index segments that have not been accessed before, it will appear a little slow. This phenomenon is easy to occur when the system just starts.

SPL also provides this mechanism of automatically caching index segments, and also provides a method of actively preloading some index segments. When the system starts, they will be loaded actively, and the subsequent search will be faster without waiting until these index segments have been accessed.

	A
1	=file("data.ctx").open()
2	=file("data.idx")
3	=A1.index@3(A2)
4	=A1.icursor(…;ID==123456).fetch()

@3 means that the first three levels of index segments are preloaded into memory.

The index of SPL composite table has four levels, each level is 1K key values, and the four levels can support up to 1T records. The first three-level indexes may occupy about several gigabytes of space. After loading the first three-level indexes, the index segments do not need to be read again for composite tables with no more than 1G records, and the index segment will only be read once more for composite tables with larger data volume.

If there is not enough memory, you can also use @2 to load only two levels.

The preloaded index segments will exist throughout the process and will be automatically used by other threads of the same process.

3.6 Batch search

The search discussed above is mainly for a single search value. If there are multiple search values at the same time, should we simply repeat it many times?

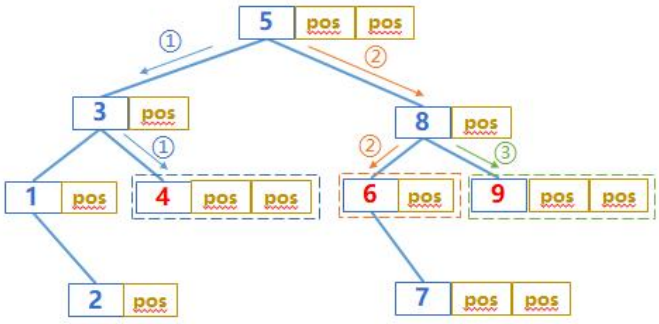
For in-memory search, it's OK to do this. Although some intermediate results may be reused in the two searches to reduce the amount of computation, more complex code is also required to determine which information can be reused (thus increasing the amount of computation). On the whole, better performance may not be obtained.

For external storage search, because the time to read the file is much slower compared to processing judgment, if the later search can reuse the results already read in the previous search, it is often worthwhile even if the code is complex, resulting in more judgment time. More detailed optimization methods are needed for multiple search values.

Let's examine the sorting index search of external storage. The multi-level index will form a tree structure. We assume that it is a K-ary tree (the K in SPL is 1024). Given the search value k_1,..., k_n, if the search starts from the root node every time, the number of index segments to be read is $n*\log_K N$ times. If we cache all the index segments we have read, we can reuse these cached data and reduce the number of reads when looking for the subsequent search values.

However, when n is large, we will find that we may need to cache a lot of index segments, because we can't predict which cached index segments will be used in the future. Too much cached data will occupy a lot of memory. Sometimes, we may be forced to discard part of the cache. As a result, we may repeatedly read an index segment during subsequent searches.

This phenomenon can be avoided if k_1,…,k_n are sorted and then searched in turn. If the index segments read to find k_i are not used when searching for k_{i+1}, they can be discarded. These index segments will not be used again when searching for k_{i+2},….



When searching for 6, if the index segments read out for searching for 4 are not used they will not be used when searching for 9

In this way, few index segments need to be cached when searching for these batch of values (there are only $\log_K N$ segments at most at the same time, that is, the number from the root node to the leaf node, and there will not be too many even when N is large). It is easy to put them in memory, so as to ensure that repeated reads caused by discarding will not occur. The more search values per batch, the greater the overall advantage.

The case of binary search is similar. When there is no index, more data needs to be read, and the reuse effect of read data will be more obvious.

After finding the position of each target value using the index, we can't read the data in the original data table immediately. We have to wait to find the positions of all target values, and then sort the positions before reading the data in the original data table.

The reason is similar. The original data table is usually stored in blocks and one data block is read at a time. If two target values are in the same data block, it can be read only once. After sorting by position, it is easy to reuse the data blocks of the target values. Otherwise, the data blocks in the original data table may be read repeatedly, resulting in a waste of time.

Many performance optimizations are focused on details, and many savings together may achieve considerable performance improvement.

Such an algorithm has been built in SPL. You can directly use contain as a condition in the function. If the search value is a sequence, the icursor() function will sort it first and then execute the above method.

	A
1	=file("data.ctx").open()
2	=file("data.idx")
3	=10000.id(rand(1000*10000)+1)
4	=A1.icursor(…;A3.contain(ID);A2).fetch()

You can compare the time difference between performing multiple single value searches and one batch search.

In addition, even if the preloaded index of SPL is used, it is still meaningful to sort the search values for the scenario with a huge amount of data (the fourth level index segment is required). The first three levels are preloaded and will not be read repeatedly. The fourth level index segments can only be read temporarily (there are too many segments in this level). After they are used, they will be discarded to reduce the memory occupation (the first three levels already occupy a lot of memory). Repeated reads may still occur when the search values are out of order. Since the first three levels have been cached, the proportion of this time in the whole search process will be more obvious.

Multiple search values may also occur in multiple concurrent scenarios. Each task has only one search value, but hundreds or thousands of tasks may be concurrent at the same time. These tasks are independent of each other and need to be searched separately, which will also lead to a long total time. If these tasks can be aggregated and multiple values can be searched at one time, the total time will be much shorter than the sum of each search time.

It can be processed at the application level. The concurrent search values in a time interval (such as 0.5 second) can be collected, sorted, searched together, and then returned to their respective requesters. In this way, each task will wait up to 0.5 second plus batch search time, and the front-end users will not have an obvious sense of waiting. If we search separately, when the number of tasks is large, the tasks behind may have an obvious sense of waiting.

However, this mechanism should be implemented in the interface stage of the application, and the relevant examples cannot be given here.

3.7 Search that returns a set

Sometimes it is necessary to find multiple target values for one search value, such as the user's transaction records through the user ID.

Establishing index for user ID can avoid full traversal and improve search performance, but it is still not enough. If the storage positions of multiple target values in the external storage data table are discontinuous, even if the positions of the target values can be quickly found by using the index, the actual reading will still jump through different positions in the hard disk. Moreover, because there is a minimum unit for hard disk reading, usually this unit is much larger than the space occupied by one record, resulting in most of the read contents being wasteful (even more serious in case of columnar storage). Moreover, too much jumping on the mechanical hard disk will also consume a lot of time. In particular, this search is often accompanied by concurrency, which will further aggravate the jumping of the hard disk.

The solution is to sort and store the data according to the to-be-searched key (TBS key) in advance to ensure that the target values corresponding to the same search value are physically continuously stored. In this way, almost all contents of the data block read out are the required target values, and it will not cause the hard disk to jump, and the performance improvement is quite obvious.

	A	B
1	=file("data.ctx").create@r(#ID,…)	
2	for 10000	>A1.append((rand(1000)+1).new(A2:ID,…).cursor()))
3	=file("data.ctx").open()	
4	=file("data.idx")	
5	=A5.index(A4;ID)	
6	=A5.icursor(;ID==3456;A4).fetch()	

Regenerate a composite table with ordered IDs. There may be several records under each ID. After creating an index, the search method is no different from that before, but it will obtain much better performance, especially in concurrency. Using row-based storage or valued index will also have better performance than using columnar storage directly. You can try to generate a disordered composite table to compare the performance difference.

Review again: even for a data table that is ordered by the TBS key, it still makes sense to establish an index. The index can directly locate the target position, unlike binary search of external storage, which will read out adjacent data to try.

The generation order of such transaction data is usually time, while the TBS key is usually not time. Therefore, only by appending the new data generated continuously (in chronological order) to the existing data table and keeping them in order by the TBS key can an efficient query effect be obtained. To achieve the objective, we can use the method discussed in the previous chapter to implement appending new data to form an ordered composite table.

3.8 Merging multi indexes

In the previous sections, we explained that multiple indexes cannot be used at the same time. After establishing indexes for two fields respectively, only one index can be used in the joint condition query for the two fields, and the conditions of the other field still need to be traversed.

In fact, this statement is not always true. Some optimization of index query algorithm can still make multiple indexes work together to a certain extent.

For the convenience of narration, we record the record set satisfying x as S(x). Assuming that there are indexes for fields A and B respectively in the table, we need to find S(A==a && B==b). Using the index of A can quickly locate S(A==a), but if we want to find the records belonging to S(B==b) from them, we can't use the index of B any more. And vice versa, we can use the index of B to quickly find S(B==b), but we can't continue to use the index of A for them.

Generally, the physical positions of records stored in the index are also ordered, that is, in the index of A, the physical positions of records stored in S(A==a) are sorted from small to large; B's index is similar. This is easy to meet when building an index, or even naturally, because the original order of members with the same value is generally not changed during sorting.

Using this, we can use the indexes of A and B at the same time when searching S(A==a && B==b).

The process is not complicated. Use the index of A to obtain the cursor based on S(A==a) and use the index of B to obtain the cursor based on S(B==b). Both cursors are orderly for the physical position of records. Then, we can use the merge algorithm to calculate their intersection to get S(A==a && B==b). This is equivalent to traversing both S(A==a) and S(B==b) once, and both indexes can be used at the same time.

However, this algorithm does not necessarily have more advantages in complexity than traversing and calculating whether B==b is true in S(A==a), because the latter only needs to traverse one set of S(A==a), while the previous algorithm needs to traverse two sets of S(A==a) and S(B==b).

The advantage of this method is in practice. To get the intersection using merging, there is no need to read out the records of the original data table to calculate whether B==b is true, just compare the index. The index blocks may already be stored in memory, or even in external storage, they are stored continuously, only have two columns (TBS key and physical position) and in row-based storage format. The reading performance is much better than the original dataset with multiple fields, discontinuous storage and columnar storage. In this way, the performance of merging and comparison will be better than that of reading records and then doing judgment.

If we choose the smaller one of S(A==a) and S(B==b) as the criterion to merge the larger one, we can stop merging as long as the smaller one has been traversed, and the members in the larger one that have not been traversed can be abandoned. In this way, the total number of traversing will not be very large. When building an index, we can usually know the number of records of S(A==a).

Of course, this method is also applicable to the case where there are conditions on more fields.

This mechanism has been implemented in SPL. For multi field conditions written on the cursor, such as A==a && B==b &&, SPL will automatically find the available index in the given index file list and merge without the intervention of programmer.

3.9 Full-text searching

In structured data query, we often query records where a string field contains a certain substring. If the condition is like(X,"abc*"), that is, the substring is located in the front of the searched field, we can use the index of the field to locate quickly. Because records that meet the condition like(X,"abc*") must be stored continuously in the sorting index of X, we can quickly locate the first record using the substring abc (or we can quickly judge that none of them exist), and then continue to traverse until the original condition is no longer satisfied. This mechanism is implemented in SPL, when this condition is found, it will check whether X has a sorting index and take advantage of it.

If the substring is located in the middle of the field to be searched, that is, the condition in the form of like(X,"*abc*"), we can no longer use the various indexes mentioned above, and we need to establish an index for text.

Search oriented full-text retrieval can certainly solve this problem, but its index is too large (because it also faces a larger amount of data), and it may also involve very professional natural language understanding and word segmentation technology, and is not suitable to be introduced into structured data computing tasks.

The scale of structured data is also much smaller than the web pages faced by search, and the search scope will generally be limited to a certain field, so the amount of data involved will be smaller. Moreover, the string as the field content is usually a few characters to a sentence, and it is rarely a whole article like a web page. In this case, a simpler technology can be used.

Take apart the characters of the field (string type) and get all combinations of the two (or three) characters that have appeared. For example, from the string abcd, we can get the combination of ab, bc and cd. Then, establish a sorting index for these extracted combinations. The to-be-searched key value is these character combinations, corresponding to the record of the field where the character combination has appeared.

The index built in this way will not be very large. There are about thousands of commonly used characters (Chinese characters and English characters). If only two characters are taken, the maximum number of such combinations is thousands by thousands, about one million to ten million, which is not very large. If we get three characters, it will be several billion to ten billion, and modern better servers can support it.

Let's look at how to do the search. Take the combination of two characters as an example, and continue to use the symbols in the previous section.

It is easy to prove that $S(like(X,"*abc*"))\subseteq S(like(X,"*ab*"))\cap S(like(X,"*bc*"))$. Using this index, we can easily find $S(like(X,"*ab*"))$ and $S(like(X,"*bc*"))$. Then we can find $S(like(X,"*ab*"))\cap S(like(X,"*bc*"))$ by using the index merging technology in the previous section, and then further check whether the condition like(X,"*abc*") is true in this set to find the final target values (the target values are included in this set, and they not necessarily equal). But in any case, this can make the computation much less than hard traversal, and does not need high-cost full-text retrieval technology.

This simplified full-text indexing is implemented in SPL:

	A
1	=file("data.ctx").open()
2	=file("data.idx")
3	=A1.index@w(A2;X)
4	=A1.icursor(;like(X,"*abc*");A2)

Using index@w() will establish this index. At present, the two-character scheme is used, which is also sufficient in many scenarios. When searching, you can directly use like as a condition.

4 Traversal technology

4.1 Cursor filtering

Records can be searched at a high speed by using index or order, however, the cost of establishing and maintaining indexes, as well as keeping data in order is not low. Since it is impossible for us to pre-build indexes for all query conditions, we will use, if necessary, the sequential search, i.e., the traversal.

While searching the external storage data table in traversal manner, the easy way to think of is to read out the data and generate records one by one, and then calculate whether the search conditions are satisfied according to the records, so as to decide to keep or discard the records. Regrettably this simple method will produce some unnecessary actions. For example, when we want to find the information of a person over the age of 18 including his/her name, gender, age and address, we will first read out these fields and generate a record, then judge whether the person is over the age of 18, as a result, the actions of reading fields and generating records for those people under the age of 18 are not essential because these records will be discarded. In contrast, we can omit a lot of actions (especially in cases of many records that do not meet the conditions) if we are able to immediately judge whether the condition is true following the read of the age field. The reason is, we will read other fields and generate records only when the condition is true, and skip directly and no longer read other fields and generate record in case of false condition.

SPL implements this mechanism for the composite table. While creating a cursor on the composite table, a filter condition can be attached, in this way, SPL will read out the field values used to calculate the condition only. If the condition is not satisfied, the next step will be canceled. Only when the condition is met will other required fields be continuously read out, and the record be created.

Using select() function on a created cursor cannot achieve this effect. Since SPL does not know the data source of this cursor, and cannot apply the filter condition before the cursor creates records, the only way is to judge after the records are fetched out.

	A
1	=file("persons.ctx").open()
2	=A1.cursor(sex,age;age>=18).groups(sex;avg(age))
3	=A1.cursor(sex,age).select(age>=18).groups(sex;avg(age))

The calculation results of A2 and A3 are same, while A2 uses the above-mentioned **pre-cursor filtering** technology, its performance is much better.

SPL has not yet implemented this optimization to text files and bin files.

Sometimes there may be multiple filter conditions. For example, the filter condition for searching female persons over 18 years old is:

```
age>=18 && sex=="Female"
```

As the commutative law is a law of && operation, the writing order of these two conditions does

not affect the operation result but the operation performance. We should consider which condition is more likely to be false and put it in the first place. The reason for doing so is that, if the former condition is calculated as false, the subsequent condition does not need to be calculated, and the entire logical expression is surely determined as false, therefore, this record can be skipped directly, and the field used to calculate the subsequent condition is not necessarily read out any more.

For example, if we know roughly that less than half of people in this data is female, and most of people are over 18 years old, then the condition of women is easier to be calculated as false, in this case, we should write it as:

```
sex=="Female" && age>=18
```

This writing method will have better calculation performance than the previous one.

The writing method is exactly opposite for ||-related conditions. In this situation, we should put the condition that is easily calculated as true in the first place. If true is determined after calculation, the subsequent condition needs not to be calculated.

Normally the amount of data during traversal is very large, and the filter conditions may be calculated many times, in this case, the calculation speed is quite important, thus we should try to choose a syntax that makes it calculation faster while writing the code.

For the example given above, and according to the knowledge we discussed earlier, we can get to know that sex=="female" is a string operation, and its operation speed is not as fast as that of age>=18. In case that the possibility of false result calculated by these two conditions is almost the same or it cannot be judged, then we should put the calculation speed section in the first place enabling the conditions to calculate a definite result as soon as possible.

Of course, a better way is to use the data type conversion method mentioned earlier to convert the sex field into a small integer, and judge it with an integer operation like sex==1.

In addition, try not to put some unnecessary calculation expressions in the filter conditions, but try to calculate them in advance.

	A
1	=file("persons.ctx").open()
2	=A1.cursor(;year(now())-year(birthday)>=18)

The filter condition in A2 is not satisfactory. Since now() returns an instant information, and a prior calculation can not be preformed, year(now()) will be calculated provisionally during every operation, resulting in a relatively poor performance. If written as:

	A
1	=file("persons.ctx").open()
2	=year(now())
3	=A1.cursor(;A2-year(birthday)>=18)

It will be much better. In this case, year(now()) only needs to be calculated only once. A better writing method is:

	A
1	=file("persons.ctx").open()
2	=year(now())-18
3	=A1.cursor(;year(birthday)<=A2)

In this way, subtraction can also be omitted. Furthermore, using the conversion data type discussed in the previous chapter to convert birthday into a small integer, the performance will be further improved.

Sometimes it is necessary to determine whether a field value is in a certain set, for example, specify a name list and find out the people with these names.

	A
1	=file("persons.ctx").open()
2	[John,Alice,Mary,Steven,…]
3	=A1.cursor(;A2.contain(name))

Usually the sequential search method is used by contain(), but the performance is not very well. If the list is a little long (10 or more), the binary search can be used to improve the speed.

	A
1	=file("persons.ctx").open()
2	[John,Alice,Mary,Steven,…]
3	=A2.sort()
4	=A1.cursor(;A3.contain@b(name))

In this way, the traversal speed will be much faster.

In Chapter 8, we will introduce one more efficient method used for performing this set membership judgment in traversal.

4.2 Multipurpose traversal

We know that the data reading time accounts for a large proportion in the traversal operation of the external storage data table. Since reading is unavoidable, we hope to get as many things as possible during one time reading, which means, the data read out in the traversal process can be used to the maximum extent.

For example, when we intend to group and aggregate the orders, hoping to count sales by product, then finding out the maximum order amount in each region, such two tasks with different grouping keys cannot be written in one grouping operation. A simple method to write will be:

	A
1	=file("orders.ctx").open()
2	=A1.cursor(product,amount).groups(product;sum(amount))
3	=A1.cursor(area,amount).groups(area;max(amount))

This method will traverse the data table twice, and the amount field will be repeatedly read (assuming it is columnar storage, and more contents will be repeatedly read in case of row-based storage).

Actually, we can calculate both grouped results in one traversal with some techniques.

	A
1	=file("orders.ctx").open()
2	=A1.cursor(area,product,amount).groups(area,product;sum(amount):samount,max(amount):mamount)
3	=A2.groups(product;sum(samount))
4	=A2.groups(area;max(mamount))

In this way, more amount of computation will be performed by CPU, and more memory will be occupied as it needs to calculate and keep a more detailed grouped result set. One advantage of this method is that a better performance can usually be obtained due to much less traversal. However, if we need to do different grouping and aggregating for more fields, the code will be much more cumbersome. If we want to do some further non-simple-grouping operations to the cursor, such as filtering followed by grouping, it is almost impossible to write with this technique.

Using SQL in the database will face this problem.

SPL provides **multipurpose traversal** technology to solve this kind of problem. The above operation can be written as follows:

	A
1	=file("orders.ctx").open()
2	=A1.cursor(product,area,amount)
3	=channel(A2).groups(area;max(amount))
4	=A2.groups(product;sum(amount))
5	=A3.result()

A3 uses the channel() function to define a **channel** synchronized with cursor A2, on which an operation (also a group) is attached. When traversing the cursor in A4 to calculate the group, the read data will be concurrently sent to the just-mentioned operation (a group on the channel attached by A3). After traversal, the corresponding calculation results will be kept in the channel and can be taken out in A5.

In comparison with the previous code traversed twice, the computation amount of CPU in this code is the same, and only two small grouping are performed for both codes. However, the reading amount of hard disk is much less in this code, and the amount field is read only once.

A cursor can define multiple synchronous channels and attach multiple sets of operations at the same time. Moreover, such operations can be written at will. Besides the grouping, it can also be written in multiple steps. For example, A3 can be written as:

```
=channel(A2).select(amount>=50).groups(area;max(amount))
```

It can count the orders with an amount of over 50.

This mechanism works for all cursors, not limiting to composite table.

SPL also provides a statement-pattern channel syntax, such code looks more neat:

	A	B
1	=file("orders.ctx").open()	
2	=A1.cursor(product,area,amount)	
3	cursor A2	=A3.groups(area;max(amount))
4	cursor	=A4.groups(product;sum(amount))
5	cursor	=A5.select(amount>=50).total(count(1))
6		

After the cursor is created, use the cursor statement to create a channel for it, and attach operations on the channel. Multiple channels can be created. If the cursor parameters are not written in the subsequent statements, it indicates the same cursor will be used. After all cursor statements (code blocks) are written, SPL will consider that all channels have been defined completely, and will start traversing the cursor, calculate the operation results of each channel and store them in the cell where the cursor statement is located.

Here, B3 and B4 respectively define corresponding operation targets, and B5 adds the number of orders with a calculated amount of more than 50. These calculation results will be placed in A3, A4, and A5 respectively (note that it is not B3, B4, B5).

The idea of multi-purpose traversal can also be applied to data split. For example, there is a large text file from which we hope to pick out the compliant data (meeting the given conditions) to perform further analysis, in this case, using the select() function of the cursor is OK. Furthermore, we may also hope to know what non-compliant data (failing to meet the conditions) are, in order to prevent this from happening again. Since one-time filtering principle cannot separate the records that meet and do not meet the conditions at the same time, the multi-purpose traversal technique can now be used.

	A
1	=file("data.txt").cursor@t()
2	…
3	=channel(A1).select(!(${A2})).fetch()
4	=A1.select(${A2})
5	>file("result.btx").export@b(A4）
6	=A3.result()

When the filter condition is filled in A2, the channel in A3 will filter out and fetch the records that do not meet the conditions, and the cursor in A4 will filter out the records that meet the conditions, then in A5, the cursor will be traversed to write the records that meet the conditions to a new file, at last, A6 can take out the channel results. In this case, we assume there are few records that do not meet the conditions, and the memory is capable of storing such records.

However, this method still needs to calculate this condition twice (records that meet and do not meet the condition to be calculated separately). SPL provides a method directly in its select() function, which can take out the records that do not meet the condition concurrently, but such records can only be written to another file in the format of bin file.

	A
1	=file("data.txt").cursor@t()
2	…
3	=A1.select(${A2};file("error.btx"))
4	>file("result.btx").export@b(A3)

Similarly, it is also possible to split the big data table into multiple groups. For example, the order records will be spitted into multiple files by region for distribution purpose, in this case, the channel can also be used. However, the number of channels needs be predetermined in the code. We should know how many channels are needed in advance, and it cannot temporarily generate a new channel in the process of grouping.

To solve this problem, SPL attaches a function that can split and write-to-files when performing serial number group on the cursor. For example, the following code divides the orders into 12 months (take months as example due to its characteristics of easy serialization, and in other cases, you can do serialization yourself).

	A
1	=file("orders.txt").cursor@t()
2	=12.(file("order"/~/".btx"))
3	=A1.groupn(month(dt);A2)
4	=A1.skip()

The groupn() function in A3 is a delayed cursor, which merely records the action, and will be actually calculated during cursor traversal. We should prepare a corresponding number of file objects (A2) in advance. Similar to select(), groupn() can only write data to bin files.

4.3 Parallel traversal

We discussed a method for segmentation of external storage data set in Chapter 2. Such method can not only be used for binary search, but more importantly, for the parallel computing in segment. By letting multiple CPUs share the amount of computation, it is often possible to implement a performance improvement close to linear growth.

SPL provides a convenient parallel computing syntax:

	A	B
1	=file("orders.txt")	
2	fork to(4)	=A1.cursor@t(area,amount;A2:4)
3		return B2.groups(area;sum(amount):amount)
4	=A2.conj().groups(area;sum(amount))	

The fork statement will start multiple threads to execute their own code blocks in parallel. The number of threads is determined by the length of the parameter sequence after fork. At the same time, fork will assign these parameters to each thread in turn. Accordingly, the value in code block, which is quoted from the cell where fork is located, can be obtained. When the execution of all threads ends, the calculation result of each thread (the value of return in the code block) will be collected into the cell where fork is located, and assembled into a sequence in the order of threads. Then the code

continues to execute.

In this example, A2 will generate four threads, each thread will get 1,2,3 and 4 as parameter respectively, and a corresponding segment cursor will be generated in B2 to perform the grouping and aggregating calculations. After all the four threads are executed, the return value of each thread (actually, a table sequence) will be collected in the cell of fork (A2), then these table sequences will be concatenated to perform an aggregation again, after that, the grouping and aggregating result for original data table is obtained.

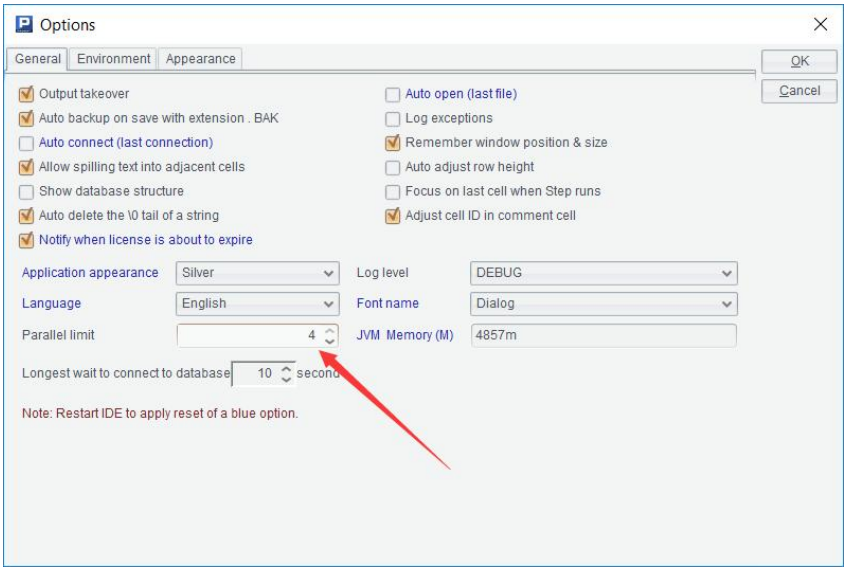
Besides the data table segmentation, it can also be used for the parallel computing of multiple files. For example, orders are placed in 12 files (one file each month), let's calculate the number of orders with an amount of more than 50 in each region:

	A	B
1	fork to(12)	=file("orders"\A2\".txt")
2		=B1.cursor@t(area,amount;A2:4)
3		=B2.select(amount>=50)
4		return B2.groups(area;count(1):quantity)
5	=A1.conj().groups(area;sum(quantity))	

It will start 12 thread calculations. Note that sum() should used outside the thread instead of count() when count() is used inside the thread.

SPL simplifies the mechanism of parallel computing, assuming that all threads are started at the same time, and the whole task is completed only when all threads are executed. Although it can not be used to deal with system-level complex parallel tasks, the syntax and code are much simpler, which is enough for most structured data computing.

When executing in parallel, we should first set the number of parallels in esProc options:



After setting, SPL will physically generate the threads with a number up to the set value. If the parameter sequence after fork is longer, serial execution will be forced. For example, in the previous example, the sequence length of fork parameter is 12, while the number of parallels set here is 4, in this case, SPL will actually only start 4 threads, and put the first four tasks of fork into the threads to

execute in turn. When a thread becomes idle after execution, fill in a task that has not been executed, and repeat according to this rule until all tasks are executed. In this way, there seems to be 12 threads logically, but only 4 threads in fact.

Under this mechanism, splitting the task will be beneficial. Since the actual execution time of each task is not always same, if each thread physically executes only one task, the fast-executing thread, after execution, has to wait for other slow-executing threads. If the task is split, the fast thread can execute more tasks, the total waiting time will be less, the load will be more balanced, and the overall performance will be better. However, the disadvantage of task splitting is that it will occupy more memory to store the results returned by threads, and there are also some costs generated from thread switching and scheduling, therefore it is not the case “more task splitting, the better performance”, and it depends on the actual situation to determine an appropriate value.

Typically, each thread is executed by a corresponding CPU. If the number of threads generated exceeds the number of CPUs, the operating system will use time-sharing technology to switch different threads on the CPUs. These mechanisms will consume CPU and memory resources, resulting in a decline in the overall performance, hence the maximum number of parallels should be set to a value that does not exceed the number of CPUs, usually slightly less, as some CPUs also need to perform other tasks such as operating system processes.

Some servers have a lot of CPUs, perhaps as many as dozens, but sometimes we find that setting more parallel numbers cannot achieve better performance. One reason is that the scheduling cost rises with the number of threads, and the more likely reason is that the hard disk does not have a same parallel computing capability. Since multiple threads will share same set of hard disks, and the data also need to be read from hard disk while performing external storage calculations, and the total throughput capacity of hard disk is definite, there is no way to allow hard disk to run faster once its capacity limit is reached, resulting in the occurrence of a phenomenon that CPU is idle waiting for the hard disk. Especially for mechanical hard disks, parallel reading will also cause more severe head runout, leading in a significant decrease in performance of hard disk, and it is very likely to result in "the more parallel reading, the worse performance".

Server's CPU and hard disk configuration should be balanced to adapt to the computing tasks. Some servers have many high-class CPUs, but such servers are equipped with low-speed mechanical hard disks for large capacity purpose, as a result, they often fail to give full play to the efficiency of the CPU and cause waste. If the money spent on CPUs is used to buy memory or high-speed SSDs, it is likely to spend less and get better performance.

4.4 Load from database in parallel

Sometimes we need to read data from the database to execute complex operations, however, we often find the time to read data is very long when database is not heavily loaded. This is mainly due to poorer performance of database’s access driver; this problem can be effectively alleviated by using the parallel technology.

	A	B
1	fork to(4)	=connect(…)
2		=range(MinID,MaxID+1,A1:4)
3		=B1.query("select * from T where id>=? and id<?,B2(1),B2(2))
4		>B1.close()
5	=A1.conj()	

This code will fetch the records in **T** table whose **id** value is between **MinID** and **MaxID** in parallel. The conditional interval is divided into multiple segments, and each thread takes one of segments. Note that the database needs to be connected separately in each thread, otherwise, database will force the requests of same connection to be executed serially, as a result, the parallel logic fails.

Database does not have an interface to achieve segmentation split, you can only use where to split, and you can use a more appropriate split scheme based on your understanding on the data characteristics. Since where computing will consume database resources, the parallel data-fetching can be used only when database resources are sufficient (slow data-fetching is due to slow driver rather than the slow database computing). Too complex where computing will also make the effect of parallel data fetching worse.

If the database cursor is needed due to the large amount of fetched data, only when the read data is processed in each thread can the parallel effect be achieved, and you cannot return the delayed cursor in the thread without substantive calculation.

	A	B
1	fork to(4)	=connect(…)
2		=range(MinID,MaxID+1,A1:4)
3		=B1.cursor("select * from T where id>=? and id<?,B2(1),B2(2))
4		=B3.groups(…)
5	=A1.conj().groups(…)	

Parallel schemes can also be used while data-fetching with multiple different SQLs:

	A	B
1	=["select …","select…",…]	
2	fork A1	=connect(…)
3		=B2.query(A2)
4		>B2.close()

For multiple SQLs, some bigger parallel thread numbers can be used. Since different SQL has different execution speeds, the load can also be balanced even if there is no more number of threads physically. When the threads are assigned to SQLs having faster execution speed, more tasks can be executed to avoid thread waiting.

4.5 Multi-cursor

Using fork can flexibly implement parallel calculations, but its code is still a bit cumbersome. Particularly, when we count a very common single data table, attention needs also be given that the function (from count to sum) may be changed while aggregating once again the results returned by the thread. Fortunately, SPL provides a simpler multi-cursor syntax, which can directly generate parallel cursors.

	A
1	=file("orders.txt")
2	=A1.cursor@tm(area,amount;4)
3	=A2.groups(area;sum(amount):amount)
4	=A1.cursor@tm(area,amount;4)
5	=A4.select(amount>=50).groups(area;count(1):quantity)

Using @m option can create parallel multi-cursors. Same as the usage for single cursor, SPL will automatically process the actions of paralleling and results re-aggregating, and will automatically and correctly process the functions that should be used in the second round.

On the multi-cursor, it can also use channels to implement multipurpose traversal:

	A	B
1	=file("orders.txt").cursor@tm(area,amount;4)	
2	cursor A1	=A2.groups(area;sum(amount):amount)
3	cursor	=A3.select(amount>=50).groups(area;count(1):quantity)

Multi-file cursors can also be concatenated into a multi-cursor to perform parallel calculation:

	A
1	=12.(file("orders"\~\".txt").cursor@t(area,amount))
2	=A1.mcursor()
3	=A2.select(amount>=50).groups(area;count(1):quantity)

In addition, it can create the in-memory multi-cursor on in-memory table sequence, and improve the operation performance by using parallel technology.

	A
1	=file("orders.txt").import@t()
2	=A1.cursor@m(4)
3	=A2.groups(area;sum(amount):amount)

For scenarios with high-performance CPUs and low-speed hard disk, we can convert single cursor into multi-cursor. This conversion enables single thread to be used when cursor fetches data to avoid the parallel tasks of hard disk, and enables multiple threads to be used during calculation to improve performance by means of multiple CPUs, which is suitable for situations that need more CPUs to parse such as text files.

	A
1	=file("orders.txt").cursor@t(area,amount)
2	=A1.mcursor(4)
3	=A2.groups(area;sum(amount):amount)

The aggregating operation on multi-cursor needs a second round (the calculation results of each thread need to be aggregated again). The calculation logic of the secondary round may be different. From the above example, it can be seen that SPL has implemented common operations, so it is enough to directly consider multi-cursor as single cursor operation, however, if you encounter uncommon operations, it is necessary to do the second round yourself.

	A	B
1	=file("orders.txt").cursor@tm(area,amount;4)	
2	fork A1	return A2.groups(area;count(1):C)
3	=A2.conj().groups(area;sum(C):C)	

Let's take count as an example (in fact, count has been processed by SPL). SPL also provides a simplified fork syntax for multi-cursor, allowing each thread of multi-cursor to be paralleled.

4.6 Grouping and aggregating

Grouping is a common traversal type operation, which needs to read and calculate all records participating in grouping. For this type of operation that needs the participation of the whole set, the index makes little sense (useful only in very few scenarios, we will discuss in the next chapter). Since some programmers do not understand the principle of grouping, they will add indexes to the database tables in case of slow grouping, doing so will only increase the load of database.

Grouping process is typically divided into the following steps: firstly, generate an empty grouped result set; secondly, traverse each original data; thirdly, calculate the grouping key value of each original data; and lastly, search the grouped subset corresponding to the key value in grouped result set, and add this record to grouped subset. If the grouped subset cannot be found, add a new grouped subset composed of this record.

During this operation process, the occurrence times of some actions like reading the records from original data, calculating the grouping key values of records, and adding the records to grouped subset, are definite (equal to the record number of original data). Since there is no way to reduce the occurrence times, the only method to reduce operations is to find the grouped subset in the grouped result set. This method is a standard search operation, which generally adopts the hash method in case no special conditions are available. The hash method works in a way that the grouped subsets are arranged according to the hash value of their corresponding grouping key values (equivalent to sequence number positioning), and the grouping key values and hash values are calculated when there is a new record, which can quickly find its own group in a small number of grouped subsets with same hash value. This grouping algorithm is also called **hash grouping**.

The sum of grouped subsets is as large as original data set. If the memory cannot store the data due to too-large data amount, then, nor the grouped subset does, therefore, this method only applies to in-memory data sets. However, in most cases, grouping always comes with aggregation, we do not

need to keep grouped subsets, but only need to calculate the aggregation value of grouped subsets, and these aggregation values can often be achieved by cumulative methods such as summing, counting, computing maximum / minimum value. In this way, the grouped subset can be discarded, and just keeping the grouping key value and corresponding aggregated value will work (it is equivalent to a table sequence rather than a set of sets), as a result, the result set will be much smaller, and it is also possible to get a small grouped result set that can be stored in memory even if the amount of original data is large. In this process, it still needs to find the target record to do accumulation, and it also needs to use hash scheme, this kind of grouping and aggregating is still called hash grouping.

Sometimes, even if only the aggregation value is needed, the grouped result sets may still be too large to be stored in memory, that means the number of grouping key values is very large. This situation is called **big grouping**, while another situation where the result set is small enough to be stored in memory, is called **small grouping**.

When dealing with big grouping, it is necessary to extend hash grouping algorithm to external storage. To do this, we need to expand the range of hash values first to let the grouping key values be dispersedly under different hash values to the utmost extend, resulting in a situation that not many grouping key values correspond to the same hash value. Since the range of hash function is known in advance, divide this range into several intervals that can be stored in memory (simple equal division is OK). In the process of data traversing, every time a batch of records is read, calculate the hash value of its grouping key value, and write to different buffer files based on the interval in which it is located, and then release the memory space to read the next batch of records until the end of traversal. After that, read the data from each buffer file separately, and do hash grouping again. Since the hash values of data in each buffer file fall within one of intervals, and they can definitely be stored in memory, separate data read and hash grouping can be performed without causing out of memory.

SPL designs two functions for the aggregation of big grouping and small grouping respectively. Small grouping will directly return the result set, while big group will return a cursor. This cursor is based on the above-mentioned buffer file, and the second round of grouping and aggregating will be performed during data fetching.

	A
1	=file("orders.btx").cursor@b(area,amount)
2	=A1.groups(area;sum(amount):amount)
3	=A1.groupx@u(area;sum(amount):amount).fetch()

Both functions have same parameter rules. The groups() function of small grouping will directly use hash algorithm, while the groupx() function of big grouping will use hash algorithm only after adding @u option.

Small grouping does not need to generate buffer files, while big grouping certainly does. When the function of big grouping is used to achieve small grouping, a lot of time will be wasted for writing buffer file, even if the final grouping result is small. It is very important to predict the size of the grouped result set in advance and select an appropriate function. Therefore, SPL provides two

grouping functions, you can compare the calculation time of A2 and A3 respectively.

We also find that the order of grouped result sets returned by groupx() is disordered, it seems that it has nothing to do with the grouping key value as well as the order of original data set. In fact, this is exactly the characteristic of hash grouping, and the reason why it looks disordered is that the result set of hash grouping is sorted according to the hash value of grouping key value. On the contrary, SPL will perform a sort according to grouping key value before the groups() function of small grouping is returned, therefore, it looks that the key values are ordered. If you use groups@u, same result will occur.

There is another method to achieve big grouping, and this method can achieve big sorting as well.

This method works in a way that traverse each record in turn, and perform the hash grouping method described at the beginning of this section (hash range cannot be too large), but what is different is that this method needs to constantly monitor the number of effective grouping key values in the grouped result set in the memory. Once the number reaches a threshold, the following steps should be performed, first, sort the current grouped result set according to grouping key value; second, write it to a buffer file; and then clear and release the memory space occupied by the grouped result set; and lastly, continue to traverse the remaining records. Repeat these steps until the traversal is completed, and you will finally obtain a batch of buffer files. Since the data in these files are arranged in an orderly manner according to grouping key value, you only need to perform two operations, namely, ordered merge algorithm as well as ordered grouping and aggregating. Both these two operations can be achieved with only a small amount of memory. This kind of grouping algorithm is called **sort grouping**.

Similarly, the result from this kind of big grouping is also a cursor based on buffer files, and the second round of merging as well as grouping and aggregating are performed in the process of cursor data-fetching.

	A
1	=file("orders.btx").cursor@b(area,amount)
2	=A1.groupx(area;sum(amount):amount).fetch()

The groupx() without options works according to this algorithm.

Compared with hash grouping, sort grouping has some advantages. The grouped result set of sort grouping is directly ordered according to grouping key values, which may be used in the next round of calculation, more conveniently, big grouping algorithms (and functions) can be used to achieve small grouping. After carefully studying the above algorithm process, you will find that if the actual result set is small, it will not really trigger the action of writing buffer files, because the grouped result set in memory will never be large enough to the threshold at which the buffer files should be written.

Another advantage of sort grouping is that it is more stable. Since there are some cases where you are unlucky with the hash function, these cases just lead to a waste of time for in-memory search, while for big grouping, it may happen that there are too many grouping key values under a hash value, and cannot be stored in the memory, in this case, a second round of hashing is needed, which

is very troublesome and low in performance.

Therefore, the big grouping provided by SPL by default is sort grouping (no options).

However, when you are sure that it is a large grouped result set (buffer files will definitely be written), and you are lucky with the hash function, then the hash grouping is more efficient than sort grouping. The reason is that the sort itself is relatively slow, and multiple buffer files need to be read at the same time when merging in second round, resulting in the concurrent read of more hard disks. The second round of hash grouping, however, only needs to read one buffer file at a time, which will not cause concurrent read of hard disk. Therefore, SPL also provides the method of hashing big grouping.

Database usually uses an optimized hash grouping method. This method will first try a small range of hash functions. If too many grouping key values are found, it will do the second hashing and perform buffering. In this way, the phenomenon that buffer data is always written can be avoided. This method has better performance in case of small grouping, but its algorithm process is much more troublesome, and its performance will decline seriously in case of big grouping.

However, even if the sort grouping can be adaptively to both small grouping and big grouping, groupx() is still more complex and a little worse in performance in comparison with groups() in practice. More importantly, the parallel effect of big grouping is not good. Specifically, multiple threads will accumulate to the same intermediate result set at the same time, and will often deal with the waiting state because of the preemption of write rights. On the other hand, if each thread has its own intermediate result set, it will result in the split of memory (each thread can only use a fraction of memory), furthermore, when there may be no need to write buffer files (the whole memory is enough to store grouped result set), the phenomenon of writing buffer files will also occur (as a fraction of memory is not enough), in this case, the hard disk will read and write very slowly, and it's easy to offset the benefits of CPU's parallel operation. Even if in the case that buffer files are definitely needed, and multiple threads write buffer files at the same time, it will cause concurrent write to the hard disk, and often seriously affects the performance. Therefore, groupx() function does not necessarily achieve better performance for multi-cursor operation.

As a result, if you clearly know that the result set is small, you still need to use groups() to get the best performance. In case you can predict the size of result set, you can also choose an appropriate number of parallels. When the size of result set is not clear, using groupx() will be more secure, and the performance loss in case of single thread is not large.

Following the understanding sort grouping, big data sorting will also be relatively simple, which can be described as the following simple steps: order the data after a batch of data is read, and then write them to buffer files, and finally perform the merge algorithm to sort these buffer files. The size of the result set of sort operation is same as that of original data set, and it will not become smaller like grouping, so big sorting will definitely generate buffer files.

Similarly, it is not easy to obtain a linear performance improvement for parallel computing of big sorting. Although sorting in the memory can be faster, the concurrent writing of multiple threads to hard disk may offset the advantage.

SPL does not directly provide a hash grouping style big sorting algorithm; you can work out the algorithm yourself after understanding the program cursor technology as well as serial number

segmentation mechanism in the next chapter. Usually, big sorting is only used in the data preparation stage, and the merge algorithm can be performed in most cases, and there are not many cases where sorting is performed to original big data.

4.7 Understandings about aggregation

Let's consider this question: how to find the top 10 out of 100 million order amounts.

The simple idea is to sort the 100 million records from large to small by amount, and take the amount field of the first 10 records, and then discard the rest.

Writing SQL in the database to solve this problem uses exactly this idea.

However, sorting itself is a very slow action, moreover, and big sorting also involves data-buffering, it will not only lead to significant decrease in performance, but it is difficult to perform the parallel computing.

Actually, it is easy for us to think of a simpler algorithm.

As long as we keep a result set with 10 members, first, fill with 0 (any small number will do), and then, traverse the order table. If the current order amount is bigger than the smallest one in the result set, replace the smallest one with this amount. The result set, after traversal, is the number we want.

This algorithm only needs to traverse the original data table once, and there is no need to sort (actually, the traversal times of sort is $\log_2 N$), let alone buffer the files. Moreover, the parallel computing in segment is also very easy, it is nothing more than calculating the top 10 of each segment, and then calculating the top 10 of the union of the said top 10. This algorithm still does not involve a concurrent write to the hard disk.

If you want to take the top M out of N members, the complexity of sort is N*logN, while the complexity of the above algorithm is N*logM. For this example, even for an all-in-memory operation, the CPU computation can decrease by around 8 times.

This idea is not uncommon. If the question is changed to calculate the maximum value, then almost everyone will think of using this method. But when the question is changed to take the top M, we will be more used to thinking of sort method first.

This requires us to expand our understanding on aggregation operation.

Usually, the aggregation operation we understand is to calculate a set to a single value, such as summing, counting, computing maximum / minimum value. However, if we expand our idea to regard the case where the return value is a small set as aggregation operation, then we can use aggregation operation to solve relevant issues.

Let's look at this example again, we can regard "taking the top N" as an aggregation operation which is the same as summation and counting operations, except that it returns a set rather than a single value.

	A
1	=file("orders.btx").cursor@b()
2	=A1.total(sum(amount), top(-10,amount), top(-10;amount))

The top() function in SPL, the same as sum(), is considered as an aggregation function, and only has a direction from small to large. The -10 in the parameter means to take the last 10 which are the 10 with maximum amounts. The top(-10,amount) will return 10 maximum amount values, while top(-10;amount) will return 10 records that maximize the amount.

Another advantage of regarding top() as an aggregation function is that it can be used in grouping and aggregating where it is still the same as functions such as sum(), count(), max() and min():

	A
1	=file("orders.btx").cursor@bm(4)
2	=A1.groups(area;top(-10,amount),top(-10;amount))

In this way, the top 10 order amounts and corresponding orders in each region can be calculated, moreover, the parallel computing of multi-cursor can be used. Attention should be given that the values of the last two fields in A2 calculation results are sequence (set).

If top() is not regarded as an aggregate function, it will be very difficult to do this operation in grouping and aggregating. In SQL, you need to use window functions to barely describe this kind of operation, and the operation performance is very poor.

Aggregation operation is, in essence, an operation for a set, however, when we actually perform this operation, it is often unnecessary to get all members of set ready. Many aggregation operations can be performed by using the cumulative method to gradually traverse the members in a set, in this way, the operation for big data can be performed.

Operations like summing, counting, computing maximum / minimum value as well as "taking top M" just mentioned, all meet this characteristic.

We call this type of aggregate functions **iteration function**, and the following characteristics can be abstracted from its operation process:

1) An initial value is given as the calculation result;

2) Each time a new set member is encountered, perform a computing on this member and last calculation result to get a new calculation result;

3) After traversal, the calculation result can be returned.

To calculate iteration function, you only need to keep a current result value, and the traversed set members can be discarded. Even for the aggregation of grouped subsets in grouping operation, only one current result value needs to be kept for each group, and occupied memory is small. To perform the calculation of iteration function, it only needs to traverse the original data table once, and there is no need to buffer the files, and the parallel computing can also be performed.

SPL designs a general form for iteration functions:

iterate(x,a)

Where, a is an initial value; x is an expression, in which ~~ represents last calculation result, and ~ represents the current set member. The calculated x is used as the new calculation result, i.e. the ~~ of the next calculation. After traversal, this function will return current ~~.

We can use this form to define common aggregation operations such as sum() and count():

```
sum         iterate(~~+~,0)
max         iterate(if(~~<~,~,~~),-inf())
min         iterate(if(~~>~,~,~~),inf())
countiterate(if(~,~~+1,~~),0)
```

For top(), although it is more troublesome, it can also be defined. You can take it as an exercise.

We can now expand aggregation operation to more general cases. As long as these cases can be described by iterate(), and the calculation results occupy just a little memory, the cumulative method can be used to achieve higher computing performance. This method can be used in grouping where original data table only needs to be traversed once and buffering the files is not needed (buffering is still needed in big grouping, but it comes from big grouping itself, and is not caused by aggregation calculation). However, there are some differences in parallel computing of iteration functions. Iteration function itself can perform parallel computing for segmented data tables to obtain multiple calculation results (one for each segment), but the manual coding is needed to perform the second round of aggregating (re-aggregate the calculation results of each segment to one result), and the parallel computing cannot be performed directly based on multi-cursor (fork syntax of multi-cursor can be used, but it still needs to do the second round of aggregating operation manually).

We can also use iterate() to perform some aggregation operations that are not defined in advance, such as continuous multiplying, getting the union, etc

	A
1	=file("orders.btx").cursor@b()
2	=A1.total(iterate(~~&area,[]))
3	=A1.groups(product;iterate(~~&area,[]))

Here, getting the union is exactly an undefined aggregation operation.

SPL stipulates that, in the iteration function for calculating the records, the field name can be directly used to represent the field of current record, and there is no need to write it as ~.area. A2 will calculate the region to which all orders are sold, and A3 will calculate the region to which each product is sold.

In structured data operations, the common simple aggregation operations are the above-mentioned several operations that have been defined, or operations that can be derived from such operations. For example, the continuous multiplying can be achieved by taking logarithm, summing and then using exponent. For "getting the union" in the above example, it can also be replaced by DISTINCT operation (id() function in SPL). Custom aggregation operations that need to be written with iteration functions are not uncommon, but such operations may involve more complex business backgrounds, which are not easily illustrated with simple examples.

Assuming that the order table is ordered by time, and if you want to calculate the total tax of each product with an initial tax rate of 5% and a tax rate reduced to 3% for subsequent orders after the cumulative tax amount exceeds 10,000, then this aggregation can hardly be described by conventional functions but iteration function, in this way, it can also be calculated at a higher performance.

	A
1	=file("orders.btx").cursor@b()
2	=A1.groups(product;iterate(~~+amount*if(~~>=10000,0.03,0.05),0):tax)

The following iteration function will calculate the maximum number of consecutive orders with an amount exceeding 50 in each region.

	A
1	=file("orders.btx").cursor@b()
2	=A1.groups(area;iterate([max(~~),if(amount>=50,~~(2)+1,0)],[0,0]):C)
3	=A2.run(C=max(C))

In the next chapter, we will discuss the form of iteration function applied to ordered detail data , and will give more meaningful examples.

4.8 Redundant grouping key

Sometimes, there are some redundant fields in data table for the purpose of convenient processing. For example, there may be both customer number and customer name in the order table, however, the customer name can be determined by customer number, in this case, the customer name belongs to redundant information. This kind of data structure is not uncommon in the wide table on the server of multi-dimension analysis.

If we want to do grouping and aggregating by customers, the customer number is usually used as grouping key because there may be same customer name. Furthermore, if we hope to list the customer name in result set, we need to write customer name in grouping keys. Indeed, this writing way is often used in SQL.

However, this writing way will increase the amount of computation. In the process of grouping, the calculation and comparison of grouping key values account for a significant ratio in amount of computation. With one more grouping key, the calculation of hash value and comparison will be much more complex, resulting in a decrease in computing performance.

SPL allows such redundant grouping keys to be written in the aggregation parameters as aggregation value, as a result, these unnecessary calculations can be avoided.

	A
1	=file("orders.btx").cursor@b()
2	=A1.groups(cust_id;cust_name,sum(amount))

If the aggregation parameter does not have an aggregation function, it will be considered as a redundant grouping key to take any one from the current grouped subset as the aggregation value. Here, cust_name will not participate in the calculation of grouping keys, but it can still be calculated correctly in the result set.

The main purpose of designing redundant fields in the data warehouse is to avoid the difficult-to-optimize join operation, so space is sacrificed for time. In the following chapters, we will discuss how to efficiently achieve this type of join in SPL without having to generate a wide table with redundant fields (fewer fields in the traversed table will increase computing performance). The

processing of redundant grouping keys is rarely involved in SPL practices, generally, it is only used in the operation for migrating and replicating traditional data warehouse.

5 Ordered traversal

5.1 Ordered grouping and aggregating

If the data table is ordered by grouping keys, we can implement the **ordered grouping** algorithm.

The process of ordered grouping is very simple, you only need to compare the key value of the current record and the last grouped subset when traversing. If two values are the same, continue to put this record into this grouped subset. On the contrary, if two values are different, it indicates that this grouped subset has been calculated completely, in this case, re-create a new grouped subset, and then put this record into the new subset. Repeat these steps until the end of the traversal, and you can get all grouped subsets.

Similarly, for the big data, the more common way is to obtain the aggregation value of each grouped subset. In this case, only one grouped result set needs to be kept. When traversing at a record, judge the grouping key value so as to decide whether to accumulate the current record value to the last grouped record or generate a new grouped record. This algorithm applies to all accumulable aggregation operations (describable by iteration function).

The ordered grouping only needs to compare adjacent records, so its complexity is very low, and the luck problem in hash grouping does not exist in this algorithm. The records in grouped subsets or grouped result set are calculated one by one, thus it will not change the calculated grouped subsets and grouped records when a new data is traversed. Such computing process does not need a large memory space to store calculation results, which is well suited for returning the results in the form of a cursor, therefore, it is naturally suitable for big grouping.

In fact, we introduced the sort grouping of big grouping earlier, its second round uses exactly this algorithm.

For instance, for the orders ordered by time, the total amount of orders per day can be calculated by ordered grouping algorithm:

	A
1	=file("orders.btx").cursor@b()
2	=A1.groups@o(dt;sum(amount))

In SPL, the groups@o() represents the ordered grouping, yet here still returns a table sequence, using groups() is considered as small grouping.

The groups@o() for small grouping can work on multi-cursor. The data table ordered by grouping key values can be deemed that it is composed of successive grouped subsets. Segmentation does not necessarily happen between two grouped subsets (possibly within a certain grouped subset), but groups@o() will finally adjust the calculation result.

SPL does not provide a corresponding groupx@o(), the ordered grouping algorithm to return a big result uses another function:

	A
1	=file("orders.btx").cursor@b()
2	=A1.group(dt).new(dt,~.sum(amount))
3	=A1.group(dt;sum(amount))

By default, the group() function on the cursor will consider that the cursor is ordered by grouping key values, it will return a cursor, and each grouped subset can be taken out in turn, after that, further operations can be performed. If the aggregation expression is written in the parameters, the cumulative method will be used directly to perform the aggregation, in this case, a cursor will still be returned, and however, the cursor members will be the grouped records consisting of grouped aggregation values.

Here, A2 and A3 perform the operations of the same result in different ways. In the way of A2, each grouped subset needs to be taken out, thus it requires that the grouped subset cannot be too large; while in the way of A3, the calculation process of iteration function is used to perform the aggregation calculation, which can calculate normally when it encounters a big grouped subset. If you only need to get the grouped aggregation value, it is more advantageous to use A3 code.

Both A2 and A3 are cursors, and fetch() is still needed to take out calculation data. Moreover, the group() function is a delayed cursor, actual calculation will be performed only while data-fetching.

For big grouping, group() cannot directly support the multi-cursor. If the segmentation position falls within a certain grouped subset, an incorrect calculation result will occur. Theoretically, we can adopt the method of discarding the head and complementing the tail discussed in the text file segmentation section, which is described as follows: starting from the segment position (except the first segment), find the next record with a different grouping key value, then start traversing this segment. After this segment is traversed, continue to the next segment and fetch a record and continue to traverse until this grouped subset is completely traversed. In this way, a correct calculation result can be ensured in case the multi-cursor is used for segmentation.

Unlike a text file, however, a certain grouped subset may be very large, which may cause a very large "head" to be discarded in the above-mentioned method, resulting in a decrease in performance due to repeated read. Moreover, since it is not clear how the cursor is calculated out (cursor may come from various sources) during the execution of group() function, the action of the said method should be well handled when the data table is segmented, it requires you to know which field the data table is sorted by when segmenting.

SPL's composite table provides another segmentation mechanism. Because SPL needs to specify the data structure in advance when creating a composite table, therefore, some information like ordered fields is known in advance, it can ensure that the segmentation point will definitely fall between the grouped subsets when segmenting. The text files and bin files do not need to specify the data table structure in advance, this mechanism is not provided.

	A
1	=file("orders.ctx").create@p(#dt,…)
2	=file("orders.ctx").open().cursor@m(dt,amount;;4)
3	=A2.group(dt;sum(amount))

While creating a composite table, @p can be used to inform SPL not to put records with the same

value in the first field (usually an ordered field) into two segments while segmenting. Currently, the segmentation processing is only provided for the first field, as the past practice has shown that it is enough.

Then, you can use multi-cursor to calculate. Note that the syntax of multi-cursor for composite table are a little different from those for text and bin files, which need to be written at the parameters after the second semicolon.

5.2 Ordered grouped subsets

When the data table is ordered by grouping key, the grouped subsets can be read out in turn in the form of cursor, which allows us to do some complex operations.

Let's take the one-year account transaction table as an example. We want to count the number of accounts with the consumption times of more than m within n consecutive days, where n and m are the parameters entered by users at program interface, and hope to find the result immediately.

It is a relatively complex operation which is unlikely to be written in a simple aggregation function (nor with an iteration function). Generally, the calculation will be easy when these transaction records are read into the memory, that is, you need to take out the transaction records under one account at a time to calculate. Since the transaction data under one account is generally very small, the memory is sufficient to hold them.

The subset grouped by account is exactly the transaction records under one account, yet the number of accounts in this situation may be very large, if this is the case, it is a typical big grouping, and it is impossible to store all grouped subsets in the memory. When the transaction records of an account are fetched every time, if there is no index, the whole table needs to be traversed, which is completely unacceptable; Even if there is an index, too-many fetching times may cause a slow computing speed because the original data table is usually sorted by transaction time (refer to the explanation in section 3.7: Search that returns a set).

If we sort the data table by account in advance (sort the transactions in an account by date), and then use the ordered grouping technology, we can easily take out these grouped subsets to perform the calculation:

	A	B	C
1	=file("trades.ctx").open().cursor(id,dt)		>m-=1,n-=1
2	for A1;id	if (k=A2.(dt).len()-m)<=0	next
3		=@+if(k.pselect(A2(#+m)-A2(#)<=n),1,0)	
4	return B3		

For the ordered cursor, the for statement will fetch a grouped subset at a time, then judge whether there are m transactions within n days.

During actual operation, the code will be further optimized to read more accounts each time.

These lines of code can also be concatenated into the group function:

	A	B
1	=file("trades.ctx").open().cursor(id,dt)	>m-=1,n-=1
2	=A1.group(id).((k=~.(dt).len()-m)>0 && k.pselect(~(#+m)-~(#)<=n)	
3	return A2.total(count())	

For each grouped subset taken out by group(), a logical expression will be calculated. If the result is true, it indicates that there are m transactions in n days. Furthermore, A2 will also return a cursor, we only need to traverse the cursor and count the number of true.

If the data table uses the composite table segmentation method described in the previous section, this operation can also work based on multi-cursor, in this case, we only need to add options at the function that generates the cursor:

	A	B
1	=file("trades.ctx").open().cursor@m(id,dt;;4)	>m-=1,n-=1
2	=A1.group(id).((k=~.(dt).len()-m)>0 && k.pselect(~(#+m)-~(#)<=n)	
3	return A2.total(count())	

The method to maintain and append the ordered data has been discussed in the previous chapters. As long as we pre-sort the data, and implement the technique of converting the date to an integer as mentioned earlier, it is also possible to obtain a high performance with immediate response even if the amount of data is very large. This calculation method can be one or two orders of magnitude faster than the cursor calculation method on traditional database (you cannot write this kind of logic in a single SQL statement).

The ordered grouped subset technology is very useful for improving the performance of complex analysis on massive accounts.

With grouped subsets, it is also very easy to achieve DISTINCT, and we only need to take one record in each group. For example, let's calculate in which months each account in the transaction table was traded.

	A
1	=file("trades.ctx").open().cursor(id,dt)
2	=A1.group(id,month(dt)).(~(1))

From the cursor calculated out in A2, we can know the account and months in which the transaction occurred, actually, only the first record of each grouped subset is taken. Note that this is just a cursor, we also need to use the fetch() to actually calculate and fetch the data. There will be many examples like this below, the only thing we need to do is to get the cursor because the result set may be too large to be taken out completely. After getting the cursor, we can do further calculation or save the result set as a file.

The operation for taking the grouped subset is relatively common, SPL provides the options:

	A
1	=file("trades.ctx").open().cursor(id,dt)
2	=A1.group@1(id,month(dt))

The group@1() will do the same thing as above, but will not firstly generate the grouped subset. In this way, the memory consumption will be less, and it can also suit to the situation where the

grouped subset is sometimes large (but it is usually a small grouping under this situation).

In fact, group@1() can be understood as DISTINCT whose effect is basically the same with that of id() function. When the data is in order, DISTINCT can be performed efficiently, while when the data is out of order, DISTINCT is as complex as grouping.

Let's take the above account transaction table again as an example, we now want to add a monthly cumulative amount information for each record, i.e., the cumulative transaction amount of the account in a month after this transaction is completed, and then filter out the transaction (including date) when the cumulative transaction amount of each account exceeds 100 for the first time every month.

This information can be calculated after reading the grouped subset:

	A
1	=file("trades.ctx").open().cursor(id,dt,amonut)
2	=A1.group(id,month(dt)).(~.derive(itertate(~~+amount,0):mca))
3	=A2.(~.select@1(mca>=100))

This kind of cumulative calculation can be performed using the iteration function for detailed data. The iteration function here still has the aforementioned characteristics: there is an initial calculation result, and the traversed members are used to calculate new result each time. Unlike the aggregation function that only returns the final result, the iteration function will return the current calculation result every time when detailed data is involved. Consequently, the monthly cumulative amount as of each transaction can be calculated out in the field added for the derive function in A2.

It should be noted that each piece of data, fetched by the cursor that is calculated out in A2, is a table sequence, and this table sequence needs to be filtered in A3 (take out the first record whose cumulative amount reaches the requirement), instead of filtering the cursor directly.

However, this algorithm needs to take out the grouped subset. If we change to another situation where the order table is sorted by product and date, and the requirement is changed to calculate out the date when the monthly cumulative order amount of each product exceeds 100, then the method of taking out the grouped subset in advance followed by calculating will not work because there may be many transaction records of one product and it may be a large grouped subset.

For the cumulative calculation on the ordered cursor, we can also use the grouping parameter of iteration function to achieve:

	A
1	=file("orders2.ctx").open().cursor(product,dt,amount)
2	=A1.derive(iterate(~~+amount,0; prudoct,month(dt)): mca)
3	=A2.select(mca>=100).group@1(product,moth(dt))

The parameter after the semicolon in the iterate() parameter is used to represent the grouped fields. When these fields (or expressions) change, SPL will restart the calculation of a new round of iteration function (re-set the calculation result as initial value and continue to iterate). During the iteration calculation, we only need to compare the previous record, without the need to firstly take out the whole grouped subset, in this way, it can either make the memory footprint less or support the large grouped subset. Moreover, since the cumulated amount field will be added to original

record, the select() can be directly performed to cursor in A3.

Finally, we need to use group@1() to perform DISTINCT. What is taken out at this time is the first record of grouped subset. Although the grouping key is product and month, the taken-out record is the record before grouping, i.e., the record containing the dt field.

5.3 Program cursor

Let's continue to use above example. Now we want to find out those records where transactions have occurred for n consecutive days in each month, and then count the transaction amount by the day of the week of the occurrence date.

The latter task is very simple, which is a common grouping and aggregating operation. However, the former task is a little troublesome. Even if the data table has been sorted by account and date, to perform this complex operation, we still need to take out the grouped subsets first, and then write a few lines of code to filter out the result. After that, these records will be stored in memory, then how can we proceed to the next step to perform the grouping and aggregating operation?

An easy way to think of is to gradually write the calculated data into a buffer file, and then group and aggregate this file:

	A	B
1	=file("trades.ctx").open().cursor(id,dt,amount)	
2	for A1;id	=A2.align@a(31,day(dt)).group@o(~==[])
3		=B2.select(~.len()>=n).conj().conj()
4		=file("temp.btx").export@ab(B3,dt,amount)
5	=file("temp.btx").cursor@b().groups(day@w(dt);sum(amount))	

A2 takes out each grouped subset; B2 aligns the subset to 31 days according to the transaction date, and then splits them into continuously empty and non-empty subsets using the ordered grouping operation; In B3, find out the subsets whose time span exceeds n, at this point, we can get the records that transactions have occurred or have not occurred for n consecutive days, and then union these records, by this time, we can finally obtain the record that transactions have occurred for n consecutive days (transactions not occurred in n consecutive days are empty sets, which will not change the union result). Note that we need to conj twice here, because the result of align@a is a sequence of the sequences.

After the calculation, write the result into a temporary file. We only need to write two fields, and finally perform the grouping and aggregating operation.

Obviously, this calculation process will be very slow as the result of an action of writing and reading, because we need to write the intermediate data into a buffer file. In fact, these data can be directly used for grouping and aggregating operation, and there is no need to write them into external storage. However, since the grouping functions can only be based on table sequence or cursor, if we hard-code each batch of data to achieve grouping and aggregating, it will be too troublesome.

SPL provides the program cursor that allows us to implement this mechanism, that is, simulate the data generated in the loop as a cursor.

	A	B	C
1	Func	=file("trades.ctx").open().cursor(id,dt,amount)	
2		for A1;id	=A2.align@a(31,day(dt)).group@o(~==[])
3			return B2.select(~.len()>=n).conj().conj()
4	=cursor@c(A1).groups(day@w(dt);sum(amount))		

By defining a subprogram, the required records can be calculated and returned in the loop of this subprogram. The cursor@c() will collect the returned values and concatenate them into a cursor. When we fetch data from the cursor (such as groups() here), the cursor() function will execute the subprogram and collect the returned values. Once the collected values are sufficient to meet the number requested in this fetch, the execution of the subprogram will be suspended, and return result to this fetch, but the subprogram will not be closed. When we need to fetch data next time, the subprogram will continue running until the whole loop is over, at this point, the cursor() function will also return the signal indicating the cursor ends.

This process can concatenate the data continuously calculated in the loop as a cursor, and the intermediate data do not have to be written into a file, as a result, the operation of this complex process can also obtain higher performance. This kind of cursor is called **program cursor**.

As we discussed earlier, SPL provides a hash method for big grouping, but it does not provide a similar sort algorithm. We can use the mechanism of program cursor to implement a rough one, such as sorting the order table by order amount:

	A	B	C
1	func	=file("orders.btx").cursor@b()	=100.(file(~）)
2		=B1.groupn(int(amount/100)+1;C1)	>B1.skip()
3		for C1	return B3.import@b().sort(amount)
4	return cursor@c(A1)		

In B2, the orders are split into 100 parts by amount (here, we assume that the order amounts are basically evenly distributed in the range of 0-10000, and you can adjust the split method according to actual situation. It is necessary to ensure that the split expression and the field values to be sorted are monotonically nondecreasing or monotonically nonincreasing, and the number of records corresponding to each split value is small so that it can be loaded into memory). Then, we just need to return the sorting results of each part in order, the cursor@c() function will collect these returned values and concatenate them into a cursor.

5.4 First-half ordered grouping

For the transaction table example taken above, we now want to change its order in a way that the transaction records of account are changed to be sorted by the day of the week of the occurrence date, that is, put together all the transactions occurred on Sunday, put together all the transactions occurred on Monday, Tuesday… Essentially, it changes the sorting method of original table, yet it belongs to big sorting operation. The big sorting algorithm will generate the buffer file, and the performance will not be good.

However, we found that the original data table has been ordered by account, it can be considered

as a “half ordered” table, moreover, the amount of unordered data in each (ordered) account is not large. In this case, we can use the program cursor mentioned above to perform the big sorting.

	A	B	C
1	func	=file("trades.btx").cursor@b()	
2		for B1;id	return B2.sort(day@w(dt))
3	return cursor@c(A1)		

Fetching the data of each account followed by sorting the “second-half”, the program cursor can collect the results and return the desired big sorting result.

Of course, this method can also be used for grouping. For example, if we want to calculate the total transaction amount of each account on the day of each week, we just need to change the above A3 to:

```
return cursor@c(A1).group(id,day@w(dt);sum(amount))
```

It can also be handled in the subprogram, in this way, the amount of returned data will be less:

	A	B	C
1	func	=file("trades.btx").cursor@b()	
2		for B1;id	return B2.groups(id,day@w(dt);sum(amount))
3	return cursor@c(A1)		

This situation is not uncommon in reality. SPL directly provides an option at the group() function:

	A
1	=file("trades.btx").cursor@b()
2	=A1.group@q(id;day@w(dt);sum(amount))

Note that group@q() has three groups of parameters. The first represents the ordered grouping keys, the second represents the unordered grouping keys, and the third is aggregating expressions. By means of the three group of parameters together with the @q option, SPL will know that it should firstly use the ordered cursor to fetch the data with same id, and then group by day@w(dt).

This code can also be used directly to sort.

	A
1	=file("trades.btx").cursor@b()
2	=A1.group@qs(id;day@w(dt))

If the @s option is added, it means that only sorting is performed without grouping, that is, sort the data with the same id by day@w(dt).

5.5 Second-half ordered grouping

We have handled the "first-half ordered" situation, then is there something we can take advantage of in the "second-half ordered" situation?

Still, we take the account transaction table as an example, where the data under each account is sorted by date, now we want to count the total transaction amount under all dates. Since the whole table is not ordered by date, the ordered grouping algorithm cannot be used. Yet, because the number of dates in a few years is not large, it still belongs to a small grouping, which can be easily written with groups():

	A
1	=file("trades.btx").cursor@b(dt,amount)
2	=A1.groups(dt;sum(amount))

However, this code does not take advantage of the characteristic that the data in the account is ordered by date, and remains a conventional hash grouping mechanism.

Let’s take this example to explain how to take advantage of "second-half ordered" situation.

Assuming that we use the ordered grouping mechanism to implement the "second-half ordered" situation. In this way, when traversing the records, we only need to compare the grouping key with the current last grouped record to determine whether to generate a new group or accumulate this record. As long as the dt is in order in the same account, errors will not occur, and the calculation and comparison of hash values can also decrease. When traversing to the next account, the dt will start resorting, and the dt of new record may be smaller than that of the current last grouped record. Once this phenomenon is found, we need to compare this dt with the first record of the current grouped records to find an appropriate position where we can either accumulate this record or insert a record, and then traverse the next transaction record. There is no need to compare once again the record that has been compared in grouped record table because it is also in order, and it will be compared with the grouped record table from the head again when the next account is traversed.

As a result, when the accounts are traversed one by one, this grouped record table will be compared again and again from the head to the tail. Every time a record is traversed, the ordered grouping algorithm only compares it once, but many times of comparison may be needed before this algorithm can find an appropriate position. However, as long as the grouped record is compared, it will not be compared again until the next account is traversed.

As a whole, the comparison times will certainly be more than that of ordered grouping, but if most of the data are in order, the occurrence proportion of comparison from the head (when traversing to the next account) will be relatively less (within the same account), and the increased comparison times will not be too many, which may be more advantageous than hash grouping.

Here comes a key problem that it may involve the insertion to grouped record table. The conventional sequential insertion is a very complex action. To maintain order, it needs to move all the following members backward to make room, and the complexity of this operation will offset all the advantages of taking advantages of order. Thus, we need to use the linked list to maintain the grouped record table, in this way, only the property of member before and after the insertion point is changed when inserting, and a large number of members will not be moved. We can't do binary search after using the linked list, yet the grouped record table is originally compared in order, therefore it has little impact here.

SPL also makes this mechanism an option, which can be used directly:

	A
1	=file("trades.btx").cursor@b(dt,amount)
2	=A1.groups@h(dt;sum(amount))

Executing the groups@h() will not use the conventional hash method but the linked list mechanism mentioned above.

Big grouping can also perform this kind of second-half ordered grouping with a similar algorithm. The difference is that when the grouped record table in memory is large enough to a certain extent, it will be written out to the buffer file, and then the process will start again, finally, these buffer files will be merged. Generally, the process is similar to that of sorting big grouping, and it is also self-adaptively to big grouping and small grouping.

	A
1	=file("trades0.btx").cursor@b()
2	=A1.groupx@h(id;sum(amount))

Suppose that the trade0 is sorted by date, and the transactions under the same date are sorted by account, this code can calculate the total transaction amount of each account at higher performance.

However, the main cost of big grouping depends on the writing and reading of buffer files, this algorithm will still have advantages, but not as significant as small grouping.

Theoretically, this algorithm works for any data, and does not necessarily require "second-half ordered" situation because any data can actually be regarded as second-half ordered data. However, if the ordered degree is too low, leading to frequent comparison with grouped record table from the head, the performance will not be comparable to hash grouping, therefore, attention should be given to the ordered degree of data when using this algorithm.

The higher ordered degree in the second-half, the more obvious the advantages of this algorithm will be. When the ordered degree reaches the extreme, it becomes an ordered grouping.

Unlike the method for first-half ordered situation, which can be used for sorting, this algorithm has little significance on sorting. The quick sort algorithm commonly used by small sorting has already taken advantage of the ordered degree of original data, while the main cost of big sorting depends on the writing and reading of buffer files, this algorithm does not reduce the number of buffer files, so it has no obvious advantages.

Since the grouping operation may involve aggregation, the actual buffer file is usually smaller than original data, sometimes much smaller, as a result, the time cost proportion from comparison in memory will become higher, and the advantages of this algorithm will appear. However, if each grouping key is almost unique in the whole table, and actual aggregation rarely occurs, and there is little difference between buffer file and original table, this algorithm will not have much advantage over hash grouping or sort grouping.

5.6 Serial number grouping and controllable segmenting

SPL provides an in-memory serial number grouping and aggregating function groups@n, and a same function for external storage cursor.

If the grouping key value can be computed to the serial number in a simply way, then there is no need to calculate and compare the hash values when grouping. Instead, we can directly use serial number to find the group (to do aggregating and accumulating), and the performance will be better.

Note that the computation must be a simple process. If it is too complex (for example, the serial number can only be obtained after searching a table), it cannot catch up with computing the hash value.

	A
1	=file("trades.ctx").open().cursor(dt,amonut)
2	=A1.groups@n(month(dt);sum(amount))

In this way, the transaction amount can be grouped and aggregated efficiently by month. This code is suitable for month or year that can be easily computed to serial number. We can use the data type conversion scheme discussed in Chapter 2 to quickly compute the serial number of month and year from the integer date.

The groups@n() also supports multi-cursor.

The groups() will return small groups, while for big grouping, can we also use serial number to improve performance?

It certainly can in theory. Actually, this is a simplified hash method for big grouping, and the grouping key itself is the hash value, thus the time of computation and comparison is omitted, and the returned cursor will naturally be ordered by grouping key.

	A
1	=file("trades2.btx").cursor(id,amount)
2	=A1.groupx@n(id;sum(amount))

Suppose that the account id is continuous natural number, in this way, the transaction amount of each account can be calculated.

However, unlike the serial number grouping for small grouping that has obvious performance advantages over hash grouping, the serial number grouping for big grouping is not significantly faster than the hash method for big grouping. Since the time of big grouping is mainly consumed at writing and reading the buffer files, the time difference between using serial number and hash value to locate the grouped records in memory can be neglected relative to the reading and writing time of buffer file.

Compared with conventional hash grouping, the significance of groupx@n() is that it does not need the second round of hashing caused by "unlucky hash function" that may occur in hash grouping.

In the process of hash grouping, knowing the range of hash values (which can be determined by hash function) can control the number of segments in a relatively reasonably way (let's review the hash method for big grouping, that is, segment the hash values first, and then write the corresponding groups into different buffer files). For the serial number grouping, however, it is not known what the maximum serial number is, and the system may be conservative when it estimates the serial number by itself, resulting in too many segments, and affecting performance as well. Therefore, group@n() adds a parameter that allows manual control of the segmentation rules:

	A
1	=file("trades2.btx").cursor(id,amount)
2	=A1.groupx@n(id;sum(amount);1000000)

The third parameter of groupx@n() represents how many serial numbers (i.e., grouped records) contained in each segment, which allows the programmers to control the number of segments according to data characteristics and memory capacity.

For groupings other than serial number grouping, it also makes sense to manually control the segmentation. SPL provides another option:

	A
1	=file("trades2.btx").cursor(id,amount)
2	=A1.groupx@g(id;sum(amount);id\1000000)

When using the groupx@g(), the hash method for big grouping that can manually control the segmentation scheme will be adopted. The third parameter is an expression based on the current record, and the computation result is an integer to determine which segment the current record should be divided into. The data in each segment is not large in amount, which can be read into memory once again to perform hash grouping operation. In this example, the id does not have to be a continuous natural serial number, but it is still an integer, and can be used to compute the segment serial number.

The method for serial number can also be used to sort. If the field (or expression) for sorting is originally a natural number, just regard it as serial number and list it out without the need to compare again, and the complexity will be much lower than that of quick sorting.

For in-memory sorting, SPL provides the sort@n() function to fulfil the serial number sorting. Correspondingly, sortx@n() can also be used for big sorting, at this time, a method similar to hash grouping will be used for sorting, which works in a way that each time a batch of data is read, the data will be written into multiple buffer files by segment, then sequentially read data from each buffer file and perform serial number sorting.

	A
1	=file("trades2.btx").cursor(id,amount)
2	=A1.sortx@n(id)
2	=A1.sortx@n(id;1000000)

sortx@n() also has a parameter similar to groupx@n() to control the segment size.

Similarly, there is also sortx@g(), which can be used to manually set the data segmentation scheme. The big sorting implemented by program cursor discussed in the previous section uses exactly this strategy, equivalent to the following code:

	A
1	=file("orders.btx").cursor@b()
2	=A1.sortx@g(amount;int(amount\100))

Since the hash value and sorting field may be in different order, sorting cannot use hash value to segment as grouping does. Thus, a monotonical segmentation serial number should be calculated based on the sort field. Essentially, there is no hash sorting, but there will be a similar method like sortx@g().

5.7 Index sorting

We know that the index is essentially the sorting. If we want to sort the data table by the to-be-searched key (TBS key), is it possible to take advantage of the index that has already been

built?

Unfortunately, this method has no effect in most cases.

If the original data table is not sorted by the TBS key, even if the index has been built, and the physical location of each record can be read out in order from the index, and no sorting operation is required, the index sorting algorithm will not have much advantage over directly performing the big sorting algorithm. The reason is that although the big sorting algorithm needs to perform sort operation, its access to external storage data tables is basically continuous, and a large number of data can be read and written each time; while if the index is used to read data in order, although the sort operation does not need to be performed, the physical position of the data in external storage is often discontinuous, which may lead to a large amount of unnecessary reading actions. On a whole, the index sorting is very likely to perform worse than big sorting.

In fact, the index has no effect on most traversal-pattern operations, and it is mainly used for situations where there are very few returned result sets.

However, although the overall sorting time with a ready-made index is not necessarily faster than that of big sorting algorithm, it has one advantage that can quickly start data output. On the contrary, the big sorting algorithm needs to traverse all the data and generate the buffer files before it starts outputting data, resulting in a longer waiting time.

Which application scenarios will use index sorting to quickly start data output?

For example, when the big data needs to be transmitted in order, the remote transmission itself is very slow and often not much faster than big sorting. If the data can be transmitted earlier, a lot of time will be saved. To achieve it, using the index sorting algorithms will be more advantageous, which can immediately start returning data for transmission without waiting.

	A
1	=file("data.ctx").open()
2	=A1.index(file("data.idx");ID;…)
3	=A1.icursor@s(…;;file("data.idx"))

The icursor@s() will ensure that the data sorted by specified index is returned (the order will not be ensured when there is no option, but it is generally sorted by physical location, which is faster). The data will also be returned as a cursor, and taken out by fetch() (for transmission).

6 Foreign key association

6.1 Foreign key addressization

Let's briefly review the concept of foreign key association: when the field F in the data table T is associated with the primary key K of the data table D, D is called the foreign key table of T, also known as the dimension table, T is called the fact table, and field F is called the foreign key field. If the primary key of D has multiple fields, the foreign key association can also be defined in a similar way. The goal of foreign key association is to be able to reference the fields of the record in D corresponding to field F when accessing the records of T.

The characteristic of foreign key association is that the foreign key of fact table must be associated with the primary key of dimension table. Since the primary key is unique, the records of fact table will only be associated with the unique record of dimension table, which is a typical many-to-one association. This characteristic should be taken advantage of when optimizing the performance.

If the amount of data is not very large, both the fact table and the dimension table can be loaded into memory. We can build the association relationship first, and then we can refer to the fields of dimension table quickly.

Suppose that the foreign key p_id in the orders table is associated with the primary key pid of the product table (the foreign key field and the corresponding primary key field are often named the same in the actual design for data table structure, and the small difference is made intentionally here to distinguish), we want to take the product unit price from product table and multiply it by the sales quantity in the order table to obtain the order amount, and then count the total order amount of different manufacturers (also a field in the product table).

	A
1	=file("product.btx").import@b()
2	=file("orders.btx").import@b()
3	>A1.keys@i(pid)
4	>A2.switch(p_id,A1)
5	=A2.groups(p_id.vendor;sum(p_id.price*quantity))

In this code, A1 reads the product table, A2 reads the order table, and A3 sets the primary key and establishes the index for the product table. The action of A4 is to convert the foreign key field p_id in A2 to the corresponding record in A1, that is, after the execution of A4, the field p_id in A2 will become a record in A1, which is equivalent to executing:

A2.run(p_id=A1.find(p_id))

A3 builds an index to make the switch() run faster. Since the fact table is usually much larger than the dimension table, this index can be reused many times.

When A5 traverses the fact table, since the value of the p_id field is already a record, we can directly use the operator "." to reference this field, in this case, both p_id.vendor and p_id.price can be executed normally.

After the switch() operation is performed, the content stored in the p_id field of the fact table A2 in the memory is already the address of a record in the dimension table A1. This action is called **foreign key addressization.** At this time, when the operator "." is used to reference the dimension table fields, we can take out them directly without the need to use the foreign key value to search in A2, which is equivalent to be able to take out the dimension table fields in constant time. Otherwise, even if the fast hash method is used to search the index, it still takes some time, and it's also related to the size of the dimension table.

This is equivalent to reusing the results of the third step switch(). For the first three steps, that is, reading the table, building the index and using switch() to make association, all are one-off in advance. After that, the operation for the fact table can quickly reference the dimension table fields.

The reason for being able to do so exactly takes advantage of the uniqueness of foreign key association towards the dimension table mentioned above, which means, one foreign key field value will uniquely correspond to one dimension table record, and each p_id can be converted to its uniquely corresponding record in A2.

In fact, the code writing becomes simpler and easier to understand after the foreign key addressization.

This operation is actually the join in the database. The relational algebra does not stipulate that one party of the association must be the primary key, and the uniqueness of either party in the association cannot be guaranteed, nor can the foreign key field be converted to dimension table records in advance. The database usually uses the hash algorithm to perform in-memory joining, that is, calculate the hash value of the associated fields of both tables, put the records with the same hash value into a group, and then traverse the correspondence. The complexity of performing a join operation is equivalent to building an index on the association key of the first table (there is no clear definition for dimension table and fact table when no primary key is assumed), and then traversing the second table so as to use the association key of its each record to search in the first table, the computation amount of this process equals to that performed in A3 and A4 in this example.

The preparations for foreign key addressization will be the same thing (like A3 and A4 do). Once the preparations finished, the result can be reused without the need to build an index and search based on this index each time the join operation is performed. In this way, the performance of subsequent calculations will be much better. If only one operation needs to be performed, plus the time to do the preparations, it will not be faster than the hash algorithm for database.

We can do the foreign key addressization following the read of data table when the system starts, this action is called **pre-association**. After that, we can deal with the association query and aggregation at a high speed. This method is very meaningful for building a high-performance in-memory database, while it cannot be achieved based on relational algebra theory system.

If the fact table has multiple foreign keys associated with different dimension tables, and dimension table also have dimension tables, the pre-association can be performed in advance for both cases:

	A
1	=file("area.btx").import@b().keys(aid)
2	=file("product.btx").import@b().keys@i(pid)
3	=file("orders.btx").import@b()
4	>A2.switch(a_id,A1)
5	>A3.switch(p_id,A2;a_id,A1)
6	=A3.select(p_id.a_id.state==a_id.state)
7	=A6.groups(p_id.vendor;sum(p_id.price*quantity))

In this code, A1, A2 and A3 read data, A4 and A5 perform the pre-correlation, and A6 and A7 are the operation code.

This code will select the orders with the same transaction place and production place, and then count the total order amount by manufacturer. Three table associations are involved in this code, among which the order table has two foreign keys and dimension table, and the region table, as the dimension table, is associated with other tables twice.

In principle, the hash join algorithm can only parse one association relationship at a time. If there are multiple association relationships, it needs to parse many times and traverse related tables many times as well. After distinguishing the fact table from dimension table, we can perform foreign key addressization for multiple foreign keys of the same fact table in one traversal. From the perspective of theoretical analysis, although the comparison times of the hash algorithm that needs multiple traversals and separate parsing are not greater than one traversal that needs to perform multiple foreign key addressizations, each traversal in hash algorithm is accompanied by the generation and copying of a batch of data records, as a result, the actual computation amount of hash algorithm is much greater than that of one traversal.

Moreover, when there are many association layers and data tables involved, the optimizer of the database often cannot find the most appropriate parsing order, and different order will result in a different data generation and amount of copy operation. If each association involves large table, these unnecessary calculations will be very large. We will observe that when the amount of data does not increase much, and only the number of associated tables and layers increases, the computing performance of database may decrease sharply.

After the fact table and dimension table are clearly distinguished, the multi-layer association of multiple tables will also be clear. The addressization for the association between small tables can be performed separately without involving other large tables. In this case, the degree of performance decrease is only linearly related to the data amount and the number of associated layers.

In practice, the association relationship is often much more complex than that in the example here, and the advantage of foreign key addressization will be more obvious.

Using switch() to achieve foreign key addressization can obtain a better computing performance, but it still has a few problems:

1) After the addressization, the value of foreign key field itself is lost, which can be obtained only by taking the primary key from the dimension table, thus, the speed will slow down.
   In the example shown above, if we want to get the pid after the addressization, we need to use p_id.pid.

2) If no associated records are found in dimension table, the foreign key field will be converted to null, and the original value will be completely lost.
3) The foreign key may be composed of multiple fields, in this case, we cannot use the switch() function to convert.

SPL also provides the join() function to adapt to these scenarios:

	A
1	=file("area.btx").import@b().keys(aid)
2	=file("product.btx").import@b()
3	=A2.join(a_id,A1,~:area).keys@i(pid)
4	=file("orders.btx").import@b()
5	=A4.join(p_id,A3,~:product;a_id,A1,~:area)
6	=A5.select(product.area.state==area.state)
7	=A6.groups(product.vendor;sum(product.price*quantity))

The join() will add a field in the original table sequence to save the address of dimension table record, and the original field value will not change, in this way, all the above problems can be solved. The difference is that join() will return a new table sequence, thus attention should be given to the order when performing the addressization. For the data table that is both a fact table and a dimension table, it needs to associate its own dimension table first to obtain a new data table, and then be associated with the fact table.

6.2 Instant addressization

Address is an in-memory concept. The foreign key addressization can only be implemented in all-in-memory operation, yet the big data often needs an external storage to perform computing.

Let's first consider the situation where the fact table is large and the dimension table is small, which is also a common situation in reality. The fact table is used to store ever-growing events and will easily become very large, while the dimension table is used to store the code information, and will change little in information growth, be relatively stable in scale, and won't become too large.

If only the dimension table is in memory, the foreign key addressization for the fact table cannot be performed in advance, which means the pre-association cannot be achieved. In this case, the association can only be made temporarily.

	A
1	=file("product.btx").import@b().keys@i(id)
2	=file("orders.btx").cursor@b(p_id,quantity)
3	=A2.switch(p_id,A1)
4	=A3.groups(p_id.vendor;sum(p_id.price*quantity))

The foreign key addressization can be implemented by using the switch() function on the cursor. It will return a delayed cursor, and the substantive association calculation will occur only when fetching the data. Unlike the all-in-memory operation where the operation is performed based on the original data table because the switch() will change the foreign key field of original data table, the operation in A4 can only be performed based on the result returned by A3, and the current association is made during the cursor reading.

This method is called **instant addressization**. In this method, only the index established on the dimension table can be reused, and the action to search for the record of dimension table has to be done every time the operation is performed, resulting in a significant decrease in performance compared to all-in-memory operation.

Theoretically, the instant addressization algorithm will still have more advantages than the hash join algorithm. For the large table that cannot be stored in memory, if the hash join algorithm is used, the records in the table need to be split, according to the hash value, into several parts that can be stored in memory, this process is commonly known as **partitioning**. After that, the in-memory hash joining between the parts corresponding to the hash value will be performed. Partitioning will cause the external storage to generate additional writing and reading actions resulting in a decrease in performance, while the instant addressization will not result in the generation of the buffer data in external storage, and there are no unnecessary write and read actions.

However, when the database encounters a join operation of two tables, one large and one small, it will not strictly implement the hashing & partitioning operation. Usually, it will read the small table into memory first, and then read the large table data in batches to perform in-memory joining. As a result, the actual complexity and performance are not much different from the above code.

In theory, the relational algebra system cannot distinguish between dimension table and fact table but the size of the tables. In case of two tables, the database optimizer can easily design a correct execution path according to the size of table, but when there are many tables and the association is complex, the optimizer may be "confused" and cannot design a reasonable execution path, instead, it will execute the original hashing & partitioning algorithm. Therefore, we will observe that even if there is only one large table, there may be a sharp decrease in performance when there are many associated tables.

After conceptually distinguishing the dimension table from the fact table, it is clearly that we should first read the dimension table into memory, build the index and make pre-association between the dimension tables, and then traverse the fact table to perform the calculation. Compared with all-in-memory operation, this algorithm has no difference in overall structure, except that the fact table needs to be read with a cursor and the association can only be made with dimension table temporarily, and buffer files will not be generated. Moreover, the pre-association between the index on the dimension table and the dimension table can still be reused.

	A
1	=file("area.btx").import@b().keys(aid)
2	=file("product.btx").import@b()
3	=A2.join(a_id,A1,~:area).keys@i(pid)
4	=file("orders.btx").cursor@b(p_id,a_id,quantity)
5	=A4.join(p_id,A3,~:product;a_id,A1,~:area)
6	=A5.select(product.area.state==area.state)
7	=A6.groups(product.vendor;sum(product.price*quantity))

The pre-processing actions performed in A1, A2, and A3 can be executed when the system is started. The indexes in A1 and A3 can also be reused when associating the dimension table in A5.

Both join() and switch() functions can perform the parallel computing based on multi-cursor. In this

example, A4 can be written as a multi-cursor with the cursor@m() option, and the subsequent code does not need to be changed. The in-memory join operation in the previous section can also be converted to the in-memory multi-cursor to perform parallel computing.

6.3 Foreign key sequence-numberization

Associating the dimension table with the foreign key of fact table is essentially a search action. However, the pre-association cannot be done when the fact table is larger. If there is a way to improve the search speed, a good association performance can also be obtained.

Usually, the hash index is already established on the dimension table. After reviewing the in-memory search technology discussed in Chapter 1, we can find that the sequence number positioning algorithm is the only method faster than hash search.

If the dimension table's primary key value itself is a sequence number, it can be used directly, but this is not the case in most situations. Therefore, we need to convert the foreign key value of fact table to a sequence number, which is what we called **foreign key sequence-numberization**.

	A
1	=file("product.btx").import@b().keys@i(pid)
2	=file("orders.btx").cursor@b()
3	=A2.run(p_id=A1.pfind(p_id))
4	>file("orders_new.btx").export@b(A3)

Using pfind() can find out the sequence number of search value in the table sequence, and replace the foreign key value in the original table with this sequence number, after that, we can use faster sequence number positioning algorithm to achieve the association involving external storage.

	A
1	=file("product.btx").import@b()
2	=file("orders_new.btx").cursor@b(p_id,quantity)
3	=A2.switch(p_id,A1:#)
4	=A3.groups(p_id.vendor;sum(p_id.price*quantity))

The parameter # allows the switch() function to use the sequence number positioning, in this case, it is unnecessary to generate the index of dimension table (which can save some memory), and the subsequent calculation code could remain unchanged.

The association performance of foreign key sequence-numberization is almost the same as that of addressization. In this way, the same high performance as in-memory pre-association can be obtained for data table in external storage. The only difference is that the fact table is too large and needs to be read from the external storage.

However, the sequence-numberization needs to prepared in advance, that is, change the foreign key value of fact table. Moreover, since the conversion result depends on the order of records in dimension table, the fact table needs to be regenerated if the dimension table changes such as insertion or deletion, as a result, the maintenance and management cost is relatively high.

The sequence-numberization can also be used in the join() function to adapt to different scenarios where there is the multi-field foreign key, or the foreign key value needs to be retained. Examples

for such scenarios will not be given here.

The primary key of some dimension tables is not a sequence number, but it can be associated with a certain sequence number by simple processing. Similar to the sequence number positioning search described in Chapter 1, sometimes we need to use align() to sort the dimension table by sequence number.

For example, in the above example, pid is not a continuous sequence number, since we know that it is always within 1-1000, we can first process the dimension table records to a record sequence sorted by sequence number:

	A
1	=file("product.btx").import@b().align(1000,pid)
2	=file("orders.btx").cursor@b(p_id,quantity)
3	=A2.switch(p_id,A1:#)
4	=A3.groups(p_id.vendor;sum(p_id.price*quantity))

In this case, there is no need to regenerate the fact table, and the original table is available directly.

In another case, the dimension table's primary key itself is not a sequence number, but the sequence number can be obtained through simple operation.

For example, pid is a string, the first character is a capital letter, and the last two are numbers. In this case, we can use

(asc(left(pid,1))-asc("A"))*100+int(right(pid,2))+1

to convert pid to a natural number between 1-2700, which can associate with the sequence number. So, is it possible to avoid regenerating the fact table at this time?

It depends on the complexity of conversion calculation. If it is a simple addition or subtraction calculation, it is not a big problem. However, if the conversion can be achieved only by an expression like the one above, it still needs to regenerate the fact table, otherwise, we have to do such complex operation on the foreign key fields of fact table every time we search for the records in dimension table, it will be a waste of time. As a result, it is probably not as good as hash index. Therefore, in order to improve performance, we need to do such calculation in advance.

	A
1	=file("orders.btx").cursor@b()
2	=A1.run(p_id=(asc(left(p_id,1))-asc("A"))*100+int(right(p_id,2))+1)
3	>file("orders_new.btx").export@b(A2)

This conversion has a slight advantage over the previous method, which can be achieved based only on the data of fact table and does not depend on the order of records in dimension table. As long as the dimension table maintains this rule, there is no need to regenerate a new fact table after the insertion or deletion occurs.

	A
1	=file("product.btx").import@b()
2	=A1.align(2700,(asc(left(pid,1))-asc("A"))*100+int(right(pid,2))+1)
3	=file("orders_new.btx").cursor@b(p_id,quantity)
4	=A3.switch(p_id,A2:#)
5	=A4.groups(p_id.vendor;sum(p_id.price*quantity))

The primary key of dimension table does not need to be converted, which can be done in the data preparation stage after each load.

We also discussed the multi-layer sequence number positioning technology in Chapter 1. The primary key of some dimension tables cannot be used when they are directly associated with sequence number, in this case, using the multi-layer sequence number makes it possible to use such primary keys (take the aforementioned ID card number as an example, if the ID number is directly converted to a sequence number, it will occupy too much memory space, but if the data distribution is appropriate, the ID number can be converted to multi-layer sequence number). In this way, we can establish the multi-layer sequence number index for the dimension tables and then use the foreign key to make association (let's review again: the foreign key association is essentially a search action, and any index works). Likewise, however, since the operation of converting the primary key to multi-layer sequence number is usually not too simple, a good association performance can be obtained only after the foreign key is converted first, and it still needs to regenerate the fact table. The advantage of this conversion method over full sequence-numberization is only limited to what we just mentioned, that is, this conversion does not depend on the order of records in dimension table, and there is no need to re-convert when the insertion or deletion occurs to the dimension table.

The multi-layer sequence number is a sequence, which is inconvenient to store and read. In SPL, we can use the serial byte to process multi-layer sequence number as a single value.

6.4 Inner join syntax

We know that the foreign key of the fact table does not always have a corresponding dimension table record, and there may be invalid value. In this situation, we will take a very common action: delete this fact table record if no corresponding dimension table record is found for the foreign key; on the contrary, do the addressization association. This action is called **inner join**.

SPL provides the inner join syntax for cursors:

	A
1	=file("product.btx").import@b().keys@i(id)
2	=file("orders.btx").cursor@b(p_id,quantity)
3	=A2.switch@i(p_id,A1)
4	=A3.groups(p_id.vendor;sum(p_id.price*quantity))

The switch@i() will delete the fact table record that cannot be associated.

The join() also has @i option for the same purpose.

The switch@i() will first fetch the record from the cursor and then judge the association. Even if it

fails to associate, the record has already been generated. After reviewing the pre-cursor filtering discussed in Chapter 4, we will find that to get better performance, we need to finish the association judgement before generating the record.

SPL provides this mechanism for the cursor of composite table:

	A
1	=file("product.btx").import@b().keys@i(id)
2	=file("orders.ctx").open().cursor(p_id,quantity;p_id:A1)
3	=A2.groups(p_id.vendor;sum(p_id.price*quantity))

The association relationship can be written in the filter condition parameters of composite table cursor, in this way, SPL will judge whether the association can be done before generating the records. If it can, the addressization conversion will be performed at the same time.

If we only need to judge whether there is association, without performing the addressization for the foreign key, the conventional conditional syntax will do:

	A
1	=file("product.btx").import@b().keys@i(id)
2	=file("orders.ctx").open().cursob(quantity;A1.find(p_id))
3	=A2.total(sum(quantity))

SPL does not provide the corresponding syntax for the join() function. We can only read the records before processing in case of multi-field foreign key.

6.5 Index reuse

Sometimes the dimension table will be filtered first and then associated. For example, we just want to know the products from a certain place of origin:

	A
1	=file("product.btx").import@b().keys@i(id)
2	=file("orders.btx").cursor@b(p_id,quantity)
3	=A2.switch@i(p_id,A1)
4	=A3.select(p_id.state=="CA")
5	=A4.groups(p_id.vendor;sum(p_id.price*quantity))

In this way, the amount of calculation will be relatively large, and in A4, it needs to repeatedly take out the p_id.state to do judgement. If the dimension table A1 is filtered first, and only the required records are left, then the inappropriate records in the fact table will be filtered by switch@i() in A3, and the performance will be better.

	A
1	=file("product.btx").import@b().select(state=="CA")
2	=file("orders.btx").cursor@b(p_id,quantity)
3	=A2.switch@i(p_id,A1)
4	=A3.groups(p_id.vendor;sum(p_id.price*quantity))

The calculation amount of this code will be much smaller than the previous one.

However, here comes a problem: after the dimension table is filtered and becomes a record

sequence, its original index cannot be used. At this time, an index will be rebuilt when executing the switch(), and building an index also takes time such as calculating the hash value of primary key, and the time to build an index is not short when there are a large number of records. Moreover, because the filtering condition is unpredictable, the index cannot be prepared in advance.

In fact, the index on the original table can still be used. At least the hash value does not need to be recalculated, the only thing we need to do is to delete the filtered records from the index table. SPL provides this mechanism:

	A
1	=file("product.btx").import@b().keys@i(id)
2	=file("orders.btx").cursor@b(p_id,quantity)
3	=A1.select@i(state=="CA")
4	=A2.switch@i(p_id,A3)
5	=A4.groups(p_id.vendor;sum(p_id.price*quantity))

select@i() will reuse the index of original table on the filtered record sequence.

The reuse of index is not always faster, because the filtered records need to be deleted from the index table. If the number of filtered records is large (the remaining is small), this action will not be fast. In this case, the method of rebuilding the index may be faster. Which method to use depends on the actual situation.

6.6 Aligned sequence

For the foreign key that has been sequence-numberized, we can also use the **aligned sequence** to process the filtering on the dimension table.

	A
1	=file("product.btx").import@b()
2	=file("orders_new.btx").cursor@b(p_id,quantity)
3	=A1.(state=="CA")
4	=A2.select(A3(p_id))
5	=A4.groups(A1(p_id).vendor;sum(p_id.price*quantity))

In this code, A3 will produce a sequence having the same length as dimension table A1. The members of this sequence are all boolean value, and the dimension table record that meets the condition corresponds to true, otherwise it corresponds to false. Then, in A4, as long as the member of the aligned sequence is taken out with the sequence-numberized primary key, we can judge whether the dimension table record has been filtered, so as to quickly decide whether to filter the fact table record.

Because there is no need to do a substantial search, the performance of aligned sequence is quite good, and is very effective in processing the filter of dimension table.

For composite table, the pre-cursor filtering will be more advantageous:

	A
1	=file("product.btx").import@b()
2	=A1.(state=="CA")
3	=file("orders_new.ctx").cursor(p_id,quantity;A2(p_id))
4	=A4.groups(A1(p_id).vendor;sum(p_id.price*quantity))

For foreign keys not sequence-numberized, we can also use the aligned sequence to filter in a disguised form:

	A
1	=file("product.btx").import@b().keys@i(id)
2	=file("orders.btx").cursor@b(p_id,quantity)
3	=A1.(state=="CA")
4	=A2.run(p_id=A1.pfind(p_id))).select(A3(p_id))
5	=A4.groups(A1(p_id).vendor;sum(p_id.price*quantity))

This operation has only one more action of taking members with sequence number than that without filtering, thus the performance is not much different.

6.7 Big dimension table search

When traversing the fact table and using the foreign key to search for the record of dimension table records, only one record will be taken at a time; However, the fact table is usually not ordered by the foreign key fields (the fact table may have multiple foreign keys. Being ordered by one foreign key will not be ordered by another. In most cases, the fact table is not ordered by any foreign key), and dimension table records are also randomly found; If the number of records in the fact table is slightly large (not too large to be loaded into memory), reading the dimension table is a typical frequent, random and small-amount-data access, which will lead to an extremely low performance to the hard disk.

Fortunately, the stability of dimension table is relatively high, it is usually not very large and can be loaded into memory. This is the case discussed in previous sections.

Occasionally, however, we will encounter a situation where the dimension table is too large to be loaded into memory. For the dimension table in the external storage, we can no longer use the above algorithms, otherwise the hard disk will face the frequent, random and small-amount-data access.

Let's first consider the case where the fact table is relatively small and can be loaded into memory.

Associating the dimension table with the foreign key is essentially a search action. For the big dimension table, it is the search in external storage. Reviewing the algorithms introduced earlier, we will find that in order to obtain high performance, it needs to store the data table in order by the to-be-searched key (the primary key of dimension table), or build an index.

If there are multiple records in the fact table, there will be multiple foreign keys, the problem becomes the batch searches in external storage. In this case, the foreign keys of the fact table need to be gathered for searching to avoid frequent access caused by separate search.

	A
1	=file("customer.btx")
2	=file("orders.btx").import@b(c_id,amount)
3	=A1.iselect(A2.id(c_id),cid;area,discount).fetch()
4	=A2.switch(p_id,A3:pid).groups(p_id.area;sum(amount*discount))

The dimension table stored in the bin file is ordered by the primary key, and the dimension table needs to be searched randomly, so it cannot appear as a cursor. After the foreign key values of the fact table are concatenated into a sequence, using the iselect() function (binary search) can relatively quickly fetch the required dimension table records, avoiding the traversal of whole big dimension table. The foreign key of fact table may have duplicate values, so we need to use the id() to do DISTINCT operation and sort at the same time, and then search for them in the dimension table file, and finally associate them with the fact table.

SPL encapsulates this process into a joinx@q() function:

	A
1	=file("customer.btx")
2	=file("orders.btx").import@b(c_id,amount)
3	=A2.joinx@q(c_id,A1:cid,area,discount)
4	=A3.groups(area;amount*discount)

Unlike the addressization method used earlier, in this code, only part of records and fields are taken out from the dimension table, and such records and fields are for temporary use only in most situations, in this case, it doesn't make much sense to do addressization after constructing a table sequence, hence the joinx@q() function directly joins the dimension table fields to the fact table records.

If the dimension table is stored in the composite table, we can also use the index to search, which can obtain better performance.

	A
1	=file("customer.ctx").open()
2	=file("orders.btx").import@b(c_id,amount)
3	=A2.joinx@q(c_id,A1:cid,area,discount)
4	=A3.groups(area;amount*discount)

Generally, the composite table is sorted by the primary key, and even if there is no index, it can also be found more quickly by the binary search.

Similar to in-memory dimension table, this method can also resolve the association of multiple big dimension tables at the same time. After resolving the dimension tables in external storage, we can continue to resolve the in-memory dimension table on the obtained table sequence. Clarifying the relationship between dimension table and fact table is the key foundation for solving the foreign key relationship and obtaining high performance.

The relational algebra theory does not distinguish between dimension table and fact table. For the join of two tables, many databases still read the small table into memory first and traverse the large table. For the current situation, it will first read the fact table into memory and then traverse the

dimension tables. For these two situations, i.e., large fact table and small dimension table, big dimension table and small fact table, the database processing method is the same.

The method for searching dimension table we are discussing here also needs to read the fact table, but it does not need to traverse the big dimension table, and the processing method for the above two situations is not the same. In this method, every record of the fact table needs to participate in the calculation, and traversal of the fact table is unavoidable. For the dimension table, however, not every record needs to participate in the calculation (the maximum number of records used for calculation does not exceed that of fact table). It does not take much time to read the small dimension table, and it will not have much impact even if there is waste due to read the whole dimension table (moreover, as discussed in the previous sections, these dimension tables are usually reused, hence there is no waste). However, it is not worth traversing the big dimension table. When the fact table is relatively small, only a small part of dimension table records will be used, in this case, it is not necessary to traverse the whole big dimension table.

Database, or relational algebra system, has a relatively simple understanding of data association, and does not deeply reflect the more essential characteristics of data association, hence it is impossible to design a higher performance algorithm theoretically, and we have to hope for the engineering optimization of the database. Specifically, some well-designed database optimizers can find that one of the association keys is the primary key of the big table, and will "intelligently" choose the search technology instead of traversal, but they will still get "confused" when many tables involved.

6.8 One side partitioning

Finally, let's deal with the situation where both the dimension table and the fact table are very large, usually the fact table will be larger. For this situation, it is very difficult to achieve a high-speed calculation in any way, but we still try to find a way to calculate as quickly as possible.

Is it possible to read the fact table with a cursor, and then execute in batches the dimension table search algorithm as introduced in the previous section?

No, because the fact table is very large, too many batches of searches will always find all dimension table records, or even more than once. This algorithm will lead to serious frequent and small-amount-data reads. As mentioned earlier, the total time-consuming of index sorting is not necessarily shorter than that of big sorting algorithm, hence there is a high probability that the performance of this algorithm will not be as good as the algorithm that sorts both the fact table and the dimension table followed by merging and joining.

For the situation where both tables are large, the database generally adopts the hashing & partitioning method which works in a way that first calculate the hash value of the association key in two tables respectively, and then divide the data whose hash value is within a certain range into a part so as to form the buffer data in external storage, and ensure that each part is small enough to be loaded into memory, and finally execute the in-memory join algorithm for each pair of part (two tables) one by one. This method will split and buffer both large tables, which can also be called **two-side partitioning** method. When we are unlucky with the hash function, a second round of

hashing may be required since a certain part may be too large.

If the dimension table is stored orderly by the primary key and can be read in segments after adopting an appropriate storage scheme, thereby we can think that the dimension table has been logically partitioned (we can simply regard each segment as a part, and can also easily calculate an appropriate number of segments after knowing the total size of the data table). At this time, we only need to split and buffer the fact table by the segment of dimension table's primary key into which fact table's foreign key falls, i.e., partitioning the fact table, and then gradually associate each part (of the fact table) with the corresponding segment of the dimension table, because the partitioning action has ensured that the fact table records in this part will only be related to the dimension table records in the corresponding segment.

The logical process of the algorithm is as follows:

	A	B	C	D
1	=file("customer.btx")			
2	=10.(A1.cursor@b(id;~:10).fetch(1).cid)			
3	=file("orders.btx").cursor@b(c_id,amount)			
4	=10.(file("temp"/~))			
5	=A3.groupn(pseg(A2,c_id);A4)			
6	func	for 10	=A1.curor@b(cid,area,discount;B6:10).fetch()	
7			=A4(B6).cursor@b().join(c_id,C6:cid,area,discount)	
8			for C7,1000	return C8
9	=cursor@c(A6).groups(area;amount*discount)			

The overall process is similar to the two-side partitioning algorithm. However, this algorithm does not need to partition the dimension table, only needs to partition the fact table, hence it can be called **one-side partitioning**. Moreover, this algorithm can divide the dimension table into equal segments, and there won't be a situation where we need to do a second round of hashing and partitioning caused by encountering an unlucky hash function. Probably, there may be a too large fact table part, in this case, the algorithm for the large fact table and the small dimension table can be employed, and no secondary partitioning is required. The amount of buffer data from one-side partitioning algorithm is much less than that of two-side partitioning algorithm, and the performance will be better.

The process is still more complex, and SPL also encapsulates the algorithm:

	A
1	=file("customer.btx")
2	=file("orders.btx").cursor@b(c_id,amount)
3	=A2.joinx@u(c_id,A1:cid,area,discount)
4	=A3.groups(area;amount*discount)

This code uses the joinx() function without @q option, SPL will think that the fact table is also large and needs to be partitioned. Similar to the algorithm in the previous section, the dimension table needs to be accessed in segments, and appear as a data file rather than a cursor.

The order of data fetched from the cursor returned by joinx@u() looks unordered. It is known from the above algorithm that these data are joined and returned after being fetched one by one according

to the segments of the dimension table, and the overall order should be the same as the primary key of dimension table. However, the data in each segment are fetched by the order of the fact table parts, and each segment is not necessarily ordered by the primary key of dimension table, hence it looks like unordered on the whole.

If we want to return the data by the original order of fact table, just remove the @u option.

At this time, SPL will generate a sequence number recorded in the original fact table, and write the data after they are sorted by this sequence number during partitioning and buffering, and then write it once again into the buffer file after the joining at each segment is done, and finally merge and sort all the buffered data generated in the latter round by the original sequence number. In this way, it is equivalent to doing one more big sorting than joinx@u(), and the performance will be worse. But sometimes the fact table is originally ordered, and this order needs to be used for the next round of calculation, so it must continue to maintain this order.

7 Merge and join

7.1 Ordered merge

We have mentioned the ordered merge algorithm many times, for example, this algorithm was used when we introduced the appending of ordered composite table in Chapter 2. It can be used to achieve the intersection, union and difference operations for large sets. Taking union as an example, the logic of this algorithm is roughly as follows:

	A	B	C	D
1	func		=file("A.btx").cursor@b()	=file("B.btx").cursor@b()
2			=C1.fetch(1))	=D1.fetch(1)
3		for	if C2==null && D2==null	break
4			else if C2==null	>r=D2,D2=D1.fetch(1)
5			else if D2==null	>r=C2,C2=C1.fetch(1)
6			else if C2.id<D2.id	>r=C2,C2=C1.fetch(1)
7			else if C2.id>D2.id	>r=D2,D2=D1.fetch(1)
8			else	>r=C2,C2=C1.fetch(1),D2=D1.fetch(1)
9			return r	
10	return cursor@c(A1)			

In this code, both the data tables A and B are ordered by the id field. It will take a record from the cursors C1 and D1 respectively and compare which record has a smaller id, and then save this record having a smaller id for returning, and continue to read the cursor to which this record belongs. This action will be repeatedly performed until all the data of both cursors are all taken out. In this way, the returned cursor is the union of two data sets based on the id field.

To get the intersection, it can be changed to return the data only when both records are equal, and the loop ends when reading one cursor ends. Getting the difference can be implemented similarly. This algorithm can also be extended to the situation with n tables, in that case, the code will be more complex.

The ordered merge algorithm can also be used to perform the join operation. Take the inner join as an example, the logic of this algorithm is as follows:

	A	B	C	D
1	func		=file("A.btx").cursor@b()	=file("B.btx").cursor@b()
2			=C1.fetch(1))	=D1.fetch(1)
3		for	if C2==null || D2==null	break
4			else if C2.id<D2.id	>C2=C1.fetch(1)
5			else if C2.id>D2.id	>D2=D1.fetch(1)
6			Else	=new(C2,D2)
7				>r,C2=C1.fetch(1),D2=D1.fetch(1)
8				return D6
9	return cursor@c(A1)			

This logic is very similar to the intersection operation. The full join or left join can be achieved in a similar way, in this case, we only need to adjust the intermediate judgment code. Likewise, this logic can also be extended to the situation with n tables.

The ordered merge algorithm can be finished by only traversing both tables once. Even for large data tables, there is no need to buffer the data. If the size of two tables is N and M respectively, the complexity will be O(N+M). To perform set or join operation on unordered data, we need to compare every piece of data each other if no optimization is taken. The complexity will be O(N*M) which is much larger than O(N+M). Usually, the database uses the hashing & partitioning algorithm described in the previous chapter. Although the complexity of this algorithm can decrease by K times (K is the average number of repetitions of the hash value), it is generally still much greater than the ordered merge algorithm. Moreover, for the operation of big data in external storage, the hashing & partitioning method will also lead to read and write buffer files, and may encounter the unlucky situation we have mentioned many times. Therefore, there will be huge performance advantages by using the ordered merge algorithm.

SPL implements the ordered merge algorithm involving the set and join operations:

	A
1	=file("A.btx").cursor@b()
2	=file("B.btx").cursor@b()
3	=[A1,A2].merge@u(id)
3	=[A1,A2].merge@i(id)
3	=[A1,A2].merge@d(id)

The options @u, @i and @d of the merge() function represent the union, intersection and difference respectively.

	A
1	=file("A.btx").cursor@b()
2	=file("B.btx").cursor@b()
3	=joinx(A1,id;A2,id)
3	=joinx@1(A1,id;A2,id)
3	=joinx@f(A1,id;A2,id)

The joinx() function without option, with options @1 and @f represent the inner join, left join and

full join respectively.

Both merge() and joinx() assume that the cursor, as the parameter, is already ordered. Executing these two functions for an unordered cursor will result in incorrect calculation results.

Both functions are delayed cursors, and further calculations need to be defined.

In addition to the foreign key association, there are also two types of associations in common join operation, i.e., homo-dimension tables association, and primary-sub table association. The former is the one-to-one association, and the two tables are associated by the primary key; while the latter is the one-to-many association, and the association occurs between the primary key of primary table with the previous part of primary keys of sub-table. Both the two tables in these two associations are related to the primary key. If the data is ordered by the primary key, the low-complexity ordered merge algorithm can be adopted. When the large dimension table is ordered by the primary key, the foreign key association algorithm introduced in the previous chapter can also achieve high performance.

For the database, being ordered by the primary key is the key to achieve high-performance join operation. It is enough for the database to be ordered by the primary key only, and the cost of meeting this premise is not very high. In SPL, we will assume that all the data tables in the external storage are by default ordered by the primary key, and only few data tables used for special search targets will be sorted differently.

The definition of join operation in relational algebra is too simple (Cartesian product and then filtering) and does not involve the primary key. If the definition is strictly implemented, the ordering characteristics of association keys cannot be assumed in theory, hence these optimization algorithms cannot be used. Moreover, according to the theory of relational algebra, it is difficult to ensure that the data table is ordered by any field, and even if we want to improve the performance through the engineering optimization, it is also difficult to reach a relatively high level.

7.2 Merge in segments

There is an inconvenience in the ordered merge algorithm for big data, that is, the data need to be read one by one from two (or more) cursors before comparison. Such logic cannot directly achieve the parallel computing in segments. Since this logic cannot ensure that the associated key values of two tables are distributed synchronously in two corresponding segments, the key value in the second segment of table A may correspond to the third segment of table B.

We can once again use the ordering characteristic of the primary key to solve this problem.

When the tables A and B are associated by the primary key, take A as the reference table to divide it into segments. After that, we can quickly take out the start and end values of the primary key of each segment, and then use the primary key interval of each segment as a condition to search in table B. Since the table B is also ordered by primary key, and the records in the interval into which the primary key value falls are stored continuously, these records can also be quickly located to generate the cursor. In this way, we can achieve the synchronous segmentation and the parallel computing in segments:

	A	B
1	=file("A.ctx").open()	=file("B.ctx").open()
2	=[-inf()] | 9.(A1.cursor(;;~+1:10).fetch(1).id)| [inf()]	
3	=10.("id>="/A2(#)/" && "id<"A2(#+1))	
4	fork to(10)	=A1.cursor(;;A4:10)
5		=B1.cursor(;${A3(A4)})
6		=joinx(B4,id;B5,id)
7		=…
8	=A4.conj()….	

In A2, the id value of the first record of each segment in table A will be firstly read, and A3 will use these segmentation points to form the interval condition. After that, in each thread, table A is still segmented normally, and table B is segmented with corresponding condition. In this way, it can ensure that the key values in two tables are synchronously corresponding, and the calculation will continue after the associated cursor of segment cursor is obtained.

SPL provides corresponding function with which associable multi-cursors can be generated:

	A	B
1	=file("A.ctx").open()	=file("B.ctx").open()
2	=A1.cursor@m(;;4)	=B1.cursor(;;A2)
3	=joinx(A2,id;B2,id)	…

In A2, the multi-cursor of table A is generated normally, and in B2, the multi-cursor of table B can be generated based on A2. Since the primary key (field name with # while executing the create()) can be defined when creating the composite table, there is no need to explicate the field name here. Instead, search will be directly performed according to corresponding primary key of the two tables (different names are allowed for the primary key field of two tables). The joinx() in A3 will also return a multi-cursor.

There is another problem.

Since the primary key is unique, it is impossible to assign two records with the same primary key to two segments while segmenting. However, in case of the primary-sub table association, the association keys of the sub-table are not all primary keys, and there may be duplicate values, hence the records with the same association key may appear in two natural segments. As a result, when the parallel segmentation is to be performed in case of the primary-sub table association, the primary table should be used as the reference table, the sub-table should follow the primary table to do segmentation instead of reversing the order at will.

7.3 Association location

What we discussed in the previous two sections is the full-table traversal. In real tasks, the association tables are often filtered with conditions. Of course, we can filter after the association, but it will traverse all the association tables, and will take a lot of time in case of large table. Sometimes the filter can be performed quickly, and the remaining result set after filtering may be very small.

The two tables are associated by the primary key, hence using the primary key of the filtered table to search for the records of association table can avoid full-table traversal and obtain better performance.

The logic of this algorithm is somewhat like the foreign key association between a small fact table and a large dimension table. However, we can take advantage of the characteristic that both the primary keys of two tables are ordered to deal with the larger filtered result set.

	A	B
1	=file("A.ctx").open()	=file("B.ctx").open()
2	=A1.cursor(;…)	
3	for A2,1000	=B1.find(A3.(id))
4		=join(A3,id;B3,id)
5		…

In B3, the primary key from A3 is used to take out data. Since the primary key of the records taken out from A3 is ordered, there will be no duplicate record taken out from B1. Moreover, table B is always scanned from front to back, which will be at worst traversed only once. As a result, the repeated traversals that may be caused by adopting this method that we assumed while discussing the large fact table and large dimension table will not occur.

However, this code is only an example to describe the algorithm process. In fact, the find() function cannot be used in such a simply way because it will search from the beginning every time, and will not take advantage of the results of the last search (it only needs to search forward from the last location point). This algorithm that uses the association to locate cannot be implemented easily, and its code is more complicated. Moreover, the cost of this algorithm itself is not low, which may offset the advantage of smaller reading amount. For the specific application scenario, testing is necessary.

SPL encapsulates these operations. Let's take the primary-sub table as an example.

	A
1	=file("orders.ctx").open().cursor(dt;area=="CA")
2	=file("details.ctx").open().news(A1,dt,price,quantity)
3	=A2.groups(dt;sum(price*quantity))

After the primary table is filtered, use the filtered primary table to take out the records of associated sub-table. Because the primary table has a one-to-many relationship with sub-table, the news() should be used here to indicate that each record taken from the primary table may correspond to multiple sub-table records. At this time, the field of primary table will be duplicated according to the number of the records of associated sub-table, this action is equivalent to executing joinx().

Alternately, the sub-table can be filtered first, and then take out the records of the associated primary table.

	A
1	=file("details.ctx").open().cursor(price,quantity;quanity>10)
2	=file("orders.ctx").open().new(A1,dt,sum(price*quantity):amount)
3	=A2.groups(dt;sum(amount))

By default, the returned cursor is based on the latter association table (if there is no any filter

condition, the number of records in the returned cursor is the same as that in the associated table), in this code, the latter table is the primary table, and the records of the sub-table need to be aggregated first.

These functions can be applied to multi-cursor.

7.4 Attached table

The foreign key association may be established between a certain fact table and multiple dimension tables arbitrarily, even multiple foreign key associations can be established between a same pair of fact table and dimension table. However, the homo-dimension tables association and the primary-sub table association established based on the primary keys of two tables can't be so arbitrary.

The homo-dimension tables association is an equivalent relationship (If A is homo-dimension with B, then B is homo-dimension with A; If A is homo-dimension with B, and B is homo-dimension with C, then A is homo-dimension with C), and hence we can take advantage of the homo-dimension relationship to divide all the data tables into several groups. In this way, the data tables in the same group are associated with each other in the homo-dimension relationship, rather than with the tables outside the group. In other words, if the homo-dimension association is to be performed, it definitely occurs between the tables within the same group. Among the homo-dimension tables in the same group, there is usually a largest table whose primary key values are complete, and the primary key values of other tables belong to a subset of this complete values. For example, the customer table has the primary key values of all customers, while the VIP customer table has only the primary key values of part customers but has more attributes of the VIP customers.

The primary-sub table association is slightly more complicated, but there will be some fixed characteristic. In a reasonable data structure design, the sub-table will only establish the association with the unique primary table. For example, the primary table of the order detail table will only be the order table, and will not be another, hence the primary table is unique from the perspective of the sub-table.

Based on this understanding, we can bind together the homo-dimension table and primary-sub table of the same group for storage.

For the homo-dimension tables in the same group, first find out the table with complete primary key values, which is called the **base table**, and the remaining homo-dimension tables are called the **attached tables**. After the base table is determined, the fields of attached tables will be stored as the additional fields of base table records; or as the fields of base table, but these fields have no value for many records.

For the primary-sub table relationship, take the primary table as the base table, and sub-table as the attached table. The fields of the sub-table are also considered as the additional field of primary table records. The difference is that the value of these additional fields is a set, and the length of the value set of the additional fields from the same sub-table is the same (as the additional field of the same primary table record). Similarly, these additional fields may also have no value.

Due to the relatively fixed characteristic of the homo-dimension tables association and the

primary-sub table association, binding storage will not affect the association relationship, nor will it affect the foreign key association with other tables.

In doing so, we can obtain the following benefits in terms of performance:

1) The base table and the attached table have common primary keys. When the fields of attached table are stored as the additional fields of base table records, only one set of primary keys need to be stored, and there is no need to store primary keys (associated with the base table) of attached table again. As a result, the storage amount will be smaller. When the columnar storage method is adopted, the amount of data to be read during association will also become less.
2) The fields of attached table, as the additional field of base table records, can be directly referenced (the fields from the sub-table is a set, using different reference method), and there is no need to do association and comparison, thereby reducing the amount of calculation. In particular, if the base table is filtered, the attached table will be filtered automatically, therefore, there is no need to use the association location method introduced in the previous section, and vice versa.
3) As the field of base table records, the attached table fields are bound together with the base table records, hence it is naturally synchronized during segmentation, and no special segmentation following is required.

However, this storage method also has disadvantages. Because the storage scheme becomes more complicated, there will be a lot of unnecessary judgments when referencing additional fields.

Generally, when the primary key or the association is relatively complex, for example, there are multiple primary key fields, or the N is bigger in the 1:N ratio of primary-sub table association (it means that there will be more comparisons in conventional association), using the attached table scheme will have greater advantages. For the homo-dimension tables association, if the primary key is single and simple, the advantages of the attached table scheme are not obvious, and may even have disadvantages.

Theoretically, in the primary-sub table association relationship, the attached table can have its own attached table, but it is not very common.

Let's take the primary-sub table as an example.

SPL implements the attached table function on the composite table, and it needs to specify the additional field when the composite table is created.

	A
1	=file("orders.ctx").open().cursor()
2	=file("details.ctx").open().cursor()
3	=file("order_detail.ctx").create(#ID,…)
4	=A3.attach(detail,#seq,…)
5	=A3.append@i(A1)
6	=A4.append@i(A2)

In A3, a conventional composite table is created, and an attached table is added based on A3 in A4 where there will be a name and fields of the attached table. For the sub-table, you need to design its

own primary key (if it has a common primary key with the primary table, there is no need to specify), and then append the data like a normal data table. In this way, SPL will attach the records to correct primary table records based on the primary key of sub-table. It should be noted that except for the primary key, the base table and attached table cannot have fields with the same name, otherwise confusion will occur.

The reason why it is called the composite table is because it is a combination of the base table and the attached table. The base table and the attached table in composite table is called the **real table**.

After creating the composite table with attached table, the field of the attached table can be referenced in calculations.

	A
1	=file("order_detail.ctx").open()
2	=A1.cursor(dt,detail.sum(quantity):quantity)
3	=A2.groups(dt;sum(quantity))

The quantity is the field of attached table detail. Because it is the sub-table, the fetched data is a set. When referencing from the primary table, the aggregation operation is required.

The records of the sub-table can also be restored:

	A
1	=file("order_detail.ctx").open()
2	=A1.cursor(dt,detail{price,quantity}:amount)
3	=A2.run(amount=amount.sum(price*quantity))
4	=A3.groups(dt;sum(amount))

Since there are multiple sub-table records, which will be used as one field of the cursor of primary table after restoring, so the fetched data is a table sequence. Actions such as generating a table sequence are more complicated, and will lose performance and may offset the advantages of reducing associations.

The base table field can also be referenced from the attached table:

	A
1	=file("order_detail.ctx").open().attch(detail)
2	=A1.cursor(dt,price,quantity)
3	=A2.groups(dt;sum(price*quantity))

Just write directly when referencing the base table fields, and the performance will be better than the above method of generating table sequence field.

All these operations can support multi-cursor.

It should be noted that when we introduce these association algorithms, we often say that they may not always achieve better performance. Association is a complex operation, and the implementation code of its optimization algorithm is also very complex. Although most of these algorithms can reduce the computational complexity from the theoretical analysis point of view, when the actual code is very complex, the impact on engineering practice cannot be ignored.

All these algorithms have been verified by actual testing and can indeed optimize the

performance in some scenarios, but not in all scenarios. As for which algorithm to use, it should be selected according to the actual situation after being familiar with these algorithms.

8 Multi-dimensional analysis

8.1 Partial pre-aggregation

Essentially, the backend operation of the multi-dimensional analysis is the grouping and aggregating calculation, and the grouping methods mentioned earlier can all be used. However, when the amount of data is very large, it is not easy to achieve instant response.

To solve this problem, an easy way to think of is the pre-aggregation, that is, calculate the aggregation results in advance so as to return them directly when requested by the frontend. In other words, this method is to trade space for time, which is equivalent to converting the traversal problem to a search problem, and can achieve the instant response theoretically.

However, the full pre-aggregation is basically infeasible, and we can know the reason through simple calculations.

To do the full pre-aggregation for 50 dimensions, 2^{50} intermediate result sets will be required, and a conservative estimate on the required capacity is over one million terabytes, hence it is not operable. Even if only 10 dimensions are pre-aggregated (10 dimensions are not large in number since we need to pre-aggregate both the observation and slice dimensions), it still requires at least hundreds of terabytes of storage space, therefore, the practicality is very poor.

What we can do is only the partial pre-aggregation, that is, we can only pre-aggregate the combination of some dimensions. When the front-end requests, searching for the pre-aggregation data under a certain condition and then aggregating on them can improve the performance by dozens of times on average, which can often meet the requirements.

Which dimension combinations to pre-aggregate generally depends on practical experience. In addition, the engineering means can also be used to determine which combinations to pre-aggregate. For example, you can either record the historical query requests and do the statistical analysis, or dynamically generate new ones and delete the infrequently used ones. When it comes to the algorithm, there is not much to discuss.

If there are multiple pre-aggregated data, which one to choose to respond to frontend request?

Suppose the frontend requests the statistical values on the dimensions A,B, searching for the pre-aggregated data that includes dimensions A,B would be OK. If there are multiple pre-aggregated data that meet the criterion, just select the one with the smallest amount of data and then calculate.

These logics are relatively simple.

SPL provides partial pre-aggregation functions for the composite table:

	A
1	=file("T.ctx").open()
2	=A1.cuboid(file("1.cube"),D1,...;sum(M1),...)
3	=A1.cuboid(file("2.cube"),D1,...;sum(M1),...)
	...

Using the cuboid() function can create pre-aggregated data. What we need to specify a file name

for the pre-aggregated data, and the remaining parameters are the same as those of grouping and aggregating algorithm.

It is also very simple to use:

	A
1	=file("T.ctx").open()
2	=A1.cgroups(D1,…;sum(M1),…;file("1.cube"), file("2.cube"))

The cgroups() function will automatically search for the most appropriate pre-aggregation data according to the above logic before calculation.

The pre-aggregation scheme is quite simple, but it is limited by the capacity and has many application limitations. Thus, it can only deal with the most common situations.

It is difficult to fully apply the pre-aggregation scheme in the following situations:

1) Unconventional aggregation: in addition to the common summation and count operations, some unconventional aggregations such as count unique, median and variance, are likely to be omitted and cannot be calculated from other aggregation values. In theory, there are countless kinds of aggregation operations, and it is impossible to pre-aggregate them all.
2) Aggregation combinations: the aggregation operations may be combined. For example, we may want to know the average monthly sales, which is calculated by adding up the daily sales of a month and then calculating the average. This operation is not a simply counting and averaging operation, but a combination of two aggregation operations at different dimension levels. Such operations are also unlikely to be pre-aggregated in advance.
3) Conditions on metrics: the metrics may also have conditions during the statistics. For instance, we want to know the total sales of orders with transaction amount greater than 100 yuan. This information cannot be processed during pre-aggregation, because 100 will be a temporarily entered parameter.
4) Time period statistics: time is a special dimension, which can be either enumerated or sliced in a continuous interval. The starting and ending points of query interval may be fine-grained (for example, to a certain day), in this case, the fine-grained data must be used for re-counting, rather than using the higher-level pre-aggregation data directly.

8.2 Time period pre-aggregation

For the statistics on the time period, the pre-aggregation will work after taking some techniques.

If the data in the original data table are stored day by day, then we can pre-aggregate the data by month. When the statistics on a time period is needed, we can read the data of the whole month spanned by the time period from the pre-aggregated data and do the re-aggregation, and then read the data of the dates at both ends of the time period that do not constitute a whole month from the original data table, and do aggregation once again, at this time, we can obtain the query target. In this way, the amount of calculation of statistics on long time period can be reduced by ten times or even more.

For instance, we want to query a certain statistical value in the interval from January 22 to September 8, and the pre-aggregated data has been prepared by month in advance. We can first

calculate the aggregate value from February to August based on the pre-aggregated data, and then use the original data table to calculate the aggregate values from January 22 to January 31 and September 1 to September 8. In this process, the amount of calculation involved is 7 (Feb. – Aug.) + 10 (Jan. 22 - 31) + 8 (Sep. 1 - 8) = 25. If the aggregation is performed completely based on the original data table, the amount of calculation will be 223 (the number of days from Jan. 22 to Sep. 8). As a result, the calculation amount is almost reduced by 10 times.

The original data table mentioned here can also be a certain fine-grained pre-aggregated data.

SPL has already implemented this method by adding the conditional parameter at the cgroups() function:

	A
1	=file("orders.ctx").open()
2	=A1.cuboid(file("day.cube"),dt,area;sum(amount))
2	=A1.cuboid(file("month.cube"),month@y(dt),area;sum(amount))
3	=A1.cgroups(area;sum(amount);dt>=date(2020,1,22)&&dt<=date(2020,9,8); file("day.cube"),file("month.cube"))

If it is found that there are time period condition and higher-level pre-aggregated data, SPL will use this method to reduce the amount of calculation. In this example, SPL will read the corresponding data from the pre-aggregated files month.cube and day.cube respectively before aggregation.

The time period pre-aggregation technology is essentially to solve the slicing (dicing) problem.

8.3 Redundant sorting

The aggregation operation without the slicing condition always involves all data. If the data is not pre-aggregated, there is no way to reduce the amount of calculation. However, when there are slicing conditions, if the data are organized reasonably, it may not be necessary to traverse all the data.

By simply creating an index on the dimension will have some effect, but just a little. Although using the index can quickly locate the records that meet the condition, if the physical storage location of these data is not continuous, there will still be a lot of waste while reading. When the target data are too scattered, using index may not be much better than full traversal. The reason is that in multidimensional analysis operation, even if slicing is performed, the amount of data to be read is still very large. The main application scenario of index is often to select a small amount of data.

If the data are stored orderly by a certain dimension, the slicing condition of this dimension can be used to limit the target data to a continuous storage area, in this way, there is no need to traverse all the data, and the amount of reading can be effectively reduced. However, each dimension may have a slicing condition in theory, if the data are sorted by each dimension, it is equivalent to being copied several times, as a result, the cost of such storage is somewhat high.

A compromise is to store two copies of data sets, that is, one copy is sorted by dimensions D1,...,Dn and stored, and the other copy is sorted by Dn,...,D1 and stored. In this way, although the amount of data will be doubled, it is still acceptable. For any dimension D, there will always be a

data set that makes D in the first half of its sorted dimension list. If it is not the first dimension, the data after slicing will generally not be concatenated into a single area, but the data are also composed of some relatively large continuous areas. The closer to the top the position of dimension in the sorted dimension list, the higher the degree of physical order of data after slicing will be.

While calculating, it is enough to use one dimension's slicing condition to filter, and the conditions of other dimensions are still used for traversal calculation. In multidimensional analysis, the slice on a certain dimension can often reduce the amount of data involved by several times or tens of times. It will be of little significance to reuse the slicing condition on other dimensions, and it is also very difficult to implement. When multiple dimensions have slicing conditions, we usually choose the dimension whose range is smaller than the total value range after slicing, which often means that the filtered amount of data is smaller.

The cgroups() function implements this selection. If there are multiple pre-aggregated data sorted by different dimensions and there are multiple slicing conditions, the most appropriate pre-aggregated data will be selected. When cuboid() creates the pre-aggregated data, the order of grouped dimensions is meaningful as different pre-aggregated data will be created for different dimension orders.

It is also possible to manually select a properly sorted data set with code, and store more sorted data sets.

The redundant sorting method is not only appropriate for multidimensional analysis, but also for the conventional traversal with filter condition. The reason for taking multidimensional analysis as an example is that the relevant features of this method will be more obvious while slicing the dimensions in multi-dimensional analysis, and it is more suitable to explain.

8.4 Dimension of boolean sequence

In the previous chapter, we used the aligned sequence to improve the association after the dimension table filtering. This technique can also be used to improve the slicing performance of the enumerated type dimension.

The so-called enumerated type dimension means that the value of the dimension is a limited number of values that have been determined in advance such as gender, place of origin; In multidimensional analysis, almost all dimensions are the enumerated type dimension except for the time dimension.

Generally, the condition for slicing (more precisely, dicing) of this type of dimension is to give a dimension value set, and then filter out all the records whose dimension value is in this set. Write it in code as follows:

```
T.select( V.contain( D ) )
```

where, T is the data table, D is the slice dimension, i.e., a field, and V is the dimension value list used as condition.

The calculation of contain() will not be fast. When V is not large, the sequential search will be used; when V is larger, the binary search will be used, but it still needs to compare several times to determine whether the data is in the slice. If the data type is complex, the performance will be worse.

We first convert the enumerated type dimension to integers (generally, the number of dimension value will not be too many, and most of them can be converted to small integers less than 65535), and then convert the slice condition to the aligned sequence composed of boolean values during query. In this way, the slice judgment result can be taken directly from the specified position of the sequence during comparison, avoiding complex contain() operation.

Convert dimension to integers:

	A
1	=file("T.ctx").open()
2	=file("T_new.ctx).create(…)
3	=DV=A1.cursor(D).id(D)
4	=A1.cursor().run(D=DV.pos@b(D))
5	=A2.append@i(A4)

In A3, all possible values DV of dimension D will be calculated. Usually, such values are known in advance (such as gender, city, etc.), and there is no need to traverse the table. Why it is written this way here is to indicate its logical meaning. In A4, the DV will be employed to convert the dimension D to integers. The DV will be saved separately for use during query.

Code for slice aggregation:

	A
1	=file("T_new.ctx").open()
2	=DV.(V.pos(~))
3	=A1.cursor(…;A2(D))
4	=A3.groups(…)

A2 converts the parameter V to a boolean sequence with the same length as DV. When a member of DV is in V, the member at the corresponding position in A2 will be non-null (playing the role of true when judging), otherwise it will be filled in as null (i.e., false). Then, when traversing to perform slicing, just use the dimension D that has been converted to integer as the sequence number to take the member of this boolean sequence. If it is non-null, it indicates that the original dimension D belongs to the slice condition list V. The operation complexity of taking the value by sequence number is far less than that of contain(), which greatly improves the slicing performance.

You can do the same for all enumerated type dimensions.

8.5 Flag bit dimension

The **flag dimension**, also known as **binary dimension**, refers to the enumerated dimension with only two values yes or no (or true/false) such as whether a person is married, has attended a college, has a credit card, etc. The flag dimension is very common, and flagging the customers or things is an important means in current data analysis. This dimension is rarely used for grouping and aggregating, but usually for slicing condition.

The data set in modern multidimensional analysis often has hundreds of flag dimensions. If such large numbers of dimensions are handled as the ordinary field, it will cause a lot of waste whether in storage or operation, and it will be difficult to obtain high performance.

There are only two values for the flag dimension, only one bit is required to store them. Since a 16-bit small integer can store 16 flags, one field is enough for storing these flags that originally require 16 fields. This kind of storage method is called the **flag bit dimension**, which will greatly reduce the storage amount, i.e., the reading amount of hard disk. Moreover, the small integer will not affect the reading speed.

	A
1	=file("T.ctx").open().cursor()
2	=file("T_new.ctx).create(…)
3	=(n\16).(bits(~.(if("tag"/(~*16-15),1,0))):"bits"/~)
4	=A1.new(…,${A3.concat@c()})
5	=A2.append@i(A4)

Let n be the number of flag dimensions, and the flag dimensions are named in the form of tag1, tag2,... A3 will generate partial parameters of the new() function to convert these conventional boolean flag dimensions to the data represented in bits. Every 16 tags form a small integer, which are named bits1, bits2...respectively. As a result, bits1 will correspond to the original tag1 through tag16, and bits2 corresponds to tag17 through tag32, and so on. In this code, letting n be a multiple of 16 is for simplicity's sake, and the ... part of new() in A4 refer to other fields, just copy them.

In practice, the field naming and arranging methods may be different, and the number of flag dimensions may not be a multiple of 16, but you can use this code example to rewrite your own code.

Usually, there are multiple slicing conditions for the flag dimension, in this case, the records whose flag dimension values are all true will be taken out. Alternatively, describing from the perspective of flag bit dimension, it is to select the records that make these flag bits 1 simultaneously. This judgment can also be performed by bit operation.

	A
1	=file("T_new.ctx").open()
2	=tags.splits@c().(int(mid(~,4))-1).group(~\16)
3	=A2.(~.sum(shift(1,-(~%16))))
4	=A3.pselet@a(~!=0)
5	=A4.("and(bits"/~/","/A3(~)/"=="/A3(~)))
6	=A1.cursor(…;${A5.concat("&&")})
7	=A6.groups(…)

The condition tags is a comma-separated string composed of the flag field names, such as "tag3, tag8, tag23", which means to select the data whose values of these three flag dimensions are all true. A2 parses out the sequence numbers and groups them, and each group corresponds to a new flag bit dimension; A3 calculates the value of flag bit dimension corresponding to each group. For example, tag3 and tag8 are grouped into a same group, corresponding to bits1, the value calculated from A3 will be the binary number 0000000010000100 (the lower bit is on the right, the third and eighth bits counted from right to left are 1); A4 takes out the sequence number that is not 0, i.e., the bit dimension sequence number that needs to be judged (no judgement needed for 0); Then, in A5,

convert these values to the condition for the flag bit dimension, for example, the condition for this group corresponding to tag3 and tag8 is

and(bits1,132)==132

(132 is the decimal value of the binary number just mentioned)

The and() function is the bitwise AND operation, which can judge whether the 3rd and 8th bits of bits1 (counting from the right) are both 1. Correspondingly, for the original flag, it can judge whether tag3 and tag8 are both true. Therefore, this operation can achieve the judgment for two flag conditions by only once.

The flag bit dimension effectively decreases the amount of storage, and in most cases, the judgments for multiple tags may be combined into one bit dimension, which significantly improves the performance. For in-memory calculations, this method that can greatly decrease the occupation of memory and reduce the amount of judgment to a certain extent is also very meaningful.

8.6 In-memory flag change

The flag data may change over time, for example, the customers may be re-flagged once a month. In this case, if we want to query the flag situation of a certain month in the past, we need to save the flag information of each time point. However, this action may cause very large amount of data in multidimensional analysis, and will occupy a lot of storage space. Although it has little impact on external storage, it is a problem when we want to perform high speed in-memory queries as the insufficient memory space may not be able to hold the result data of all time points.

If the flag changes a lot each time, there is nothing we can do. However, the flag changes usually little, much smaller than the total amount of data at a certain time point. In this case, we can save the initial state and the information of change each time, then quickly calculate the data at a certain time point.

Since there are only two values for flag data, the new flag value can be calculated correctly as long as the changed flag name is saved (whether it changes from true to false, or from false to true). By using the bit dimension, we can calculate the changed bit, and then just perform a simple XOR operation with the original value.

	A	B
1	=file("T_new.ctx").open().import()	
2	=file("T_change.btx").import()	
3	for join@1m(A1,id;A2,id)	=A3.#2.tags.(~-1).group(~\16)
4		=B3.(~.sum(shift(1,-(~%16))))
5		=B3.pselect@a(~!=0)
6		=B5.(("bits"/~/"=xor(bits"/~","/B4(~)/")")
7		>A3.#1.run(${B6.concat@c()})

Both the original data table and the changed information take id field as the primary key and are stored orderly. In A3, join@m() means that the ordered merge algorithm is used to perform in-memory association. After the changed information is aligned with original data, the XOR calculation is performed in the loop body. The operation of B3:B6 is similar to that in the previous section, which is

to calculate the flag bit dimension that needs to be changed and its calculation expression. Due to little information change, converting the information to bits for storage may occupy more space, so it is better to directly store the changed flag sequence number set.

9 Cluster

9.1 Computation and data distribution

When the amount of data is large, multiple machines can be used to share computing tasks, that is, the **cluster**. In a cluster, the machines participating in the computing are called **node machine**, and there is usually a control program for managing and assigning the computing tasks to each node, and aggregating the computing results, this program is called **master computer**.

The node is generally a physical machine, because the logical machine can not share the computing tasks. The master computer is mainly responsible for control, and does not actually undertake much computation, so it is probably a logical machine. A thread of a certain node or an application outside the cluster can serve as the master computer.

The multi-machine parallel computing is similar to that of multi-thread on a single machine, except that the tasks assigned to the thread are assigned to nodes.

SPL's fork statement can also support multi-machine parallel computing, but it needs to start the esProc server on each node and configure the access address in advance.

	A	B
1	["192.168.0.101:8281","192.168.0.102:8281",..., "192.168.0.104:8281"]	
2	fork to(4);A1	=file("orders.ctx").open().cursor(area,amount)
3		return B2.groups(area;sum(amount):amount)
4	=A2.conj().groups(area;sum(amount))	

Since the order table of one year is very large, we divide it into four parts by quarter and store them on four nodes respectively, and each part is named as the same file name (different contents). A1 is the access address of four nodes (the last section of the address is used to represent the node below). The fork statement in A2 sends its code block to the esProc on the node for execution. After being executed by each node, the return value will be transmitted to the master computer. Following the collection of the return value of all nodes, the master computer will continue to execute forwards until the final aggregation is finished.

Using multi-cursor to perform parallel computing is also available on nodes:

	A	B
1	["192.168.0.101:8281","192.168.0.102:8281",..., "192.168.0.104:8281"]	
2	fork to(4);A1	=file("orders.ctx").open().cursor@m(area,amount;;4)
3		return B2.groups(area;sum(amount):amount)
4	=A2.conj().groups(area;sum(amount))	

In this way, multiple machines can be used to share computing tasks, and improve computing performance.

Generally, the physical configuration of each node in a cluster is the same, and the computing power is basically the same. In the example above, one year’s data is divided into 4 quarters and placed on 4 nodes, which can be considered as relatively average. In this way, the amount of data

traversed by each node is almost the same, and hence it will not happen that some nodes have to wait due to too heavy task burdened on a certain node.

If the requirement is changed to filter the order table by the specified time period first followed by grouping and aggregating, the situation will be different.

Assume that the time period, as the filter condition, is within the first quarter, and that the data is ordered by date (it is often the case), then the records that meet the conditions can be quickly located by using the orderly characteristics of data. As a result, 102, 103 and 104 will quickly find that they have nothing to do, and almost the whole task is on 101.

If the **snake distribution** method is adopted, this problem can be avoided. Assuming that there is little difference in the amount of data each day, we use the remainder of dividing the date of the data by 4 to determine which node the data is distributed to, that is, the data of day 1, 5, 9,... are placed on 101; the data of day 2, 6,... are placed on 102;...and the data of day 4, 8,... are placed on 104. In this way, if the requirement mentioned above is performed, no serious task imbalance will occur.

Theoretically, this will cause new imbalance. For example, when we want to count the data of day 1, 5…, the whole task will be burdened on 101. But in actual business, there is almost no such computing requirements. For most application scenarios, the snake distribution is better than the sequential distribution.

A reasonable data distribution scheme should be designed according to the calculation objective, and there is no scheme that can adapt to all operations.

9.2 Multi-zone composite table of cluster

For the conventional operations on the data table, coding with fork is a bit troublesome. SPL also provides the cluster table and cluster cursor to simplify the code, but the situation is a little more complicated than that of a single machine.

Let's review the concept of multi-zone composite table in Chapter 2. For the ease of deleting the old data, the multi-zone composite table can be composed of multiple physical files, that is, the zones; each physical file will have a number, i.e., zone number.

The zones of multi-zone composite table can also be distributed on the nodes of the cluster. Let's start with simple situation: the number of zones of multi-zone composite table is the same as that of nodes in a cluster, and each node has one zone.

	A
1	["192.168.0.101:8281","192.168.0.102:8281",..., "192.168.0.104:8281"]
2	=file@w("orders.ctx":to(4),A1)
3	=A2.create(…;(day(dt)-1)%4+1)
4	=A3.append(…)

Using the file@w() to create a writable cluster file, making the four zones correspond to the nodes one by one, in this way, the i^{th} zone file will be generated on the i^{th} node. Then, create the cluster composite table, and finally, append the data. Note that one zone expression is needed for the multi-zone composite table (the same as for a single machine).

Except for the different file created during calculation, other syntax is basically the same as that

for a single machine:

	A
1	["192.168.0.101:8281","192.168.0.102:8281",..., "192.168.0.104:8281"]
2	=file("orders.ctx":[1,2,3,4],A1)
3	=A2.open()
4	=A3.cursor@m(area,amount;dt>=arg1 && dt<arg2;4)
5	=A4.groups(area;sum(amount))

When the **cluster file** is created in A2, the zone number also needs to be written because it is allowed to use part zones of multi-zone composite table. SPL will search for corresponding zone on each node according to certain rules. The composite table and the cursor created based on the cluster file are called **cluster table** and **cluster cursor** respectively. To this point, related operations can be performed.

Let's review the multi-machine parallel computing framework described in the previous section, that is, the node will calculate separately, and then transmit the calculation result to the master computer for aggregating; during the calculation of node, there is no data exchange between the nodes. When this framework is performed based on the zone mechanism of cluster table, each node only needs to process the data of its own zone, and does not depend on other nodes, and hence it is almost the same as single machine operation. As a result, many operations can be simply transplanted from a single machine to a cluster. Moreover, by means of the syntax of cluster table, the code writing is also very similar to that of a single machine operation.

We briefly explain the working principles of these common operations, and will not give detailed examples here.

The search without index can directly use this framework and syntax.

There are two ways to deal with the index. The simple way is to create a separate index for each node, so this framework can still be used. In this way, each node is independent when searching, and the results will be searched respectively and returned to the master computer for aggregating. However, all nodes will be used for every search, it will result in consuming more resources. The complex way is to create a multi-zone-based index and then sort by zone. By this way, according to the search value, it can immediately locate the zone of the index and find the location of target value. For the accurate search (there is only one returned result set), only two nodes are involved (the zone where the index is located and the zone where the target value is located), as a result, it will consume less resources. In short, for a single task, there is almost no performance difference between the two ways, but for the scenarios where more concurrency exists and the extreme performance is required, the latter will have more advantages.

For the filtering and conventional small grouping, this framework and syntax can also be used directly.

When the ordered grouping (and other various ordered traversals) is performed, attention should be given that the records with the same grouping key value cannot be assigned to different zones, this requirement is usually easy to achieve.

Under this framework, using the sorting algorithm will be easier for the big grouping (big sorting). After the nodes are grouped (sorted) respectively, being ordered by the grouping keys (sorting expression) should be kept while aggregating to the master computer, and then do the final merging. The hash method for big grouping can also be used on the node, but the results still need to be transmitted to the master computer for merging, and a certain order is still required. Although the final merging can also be implemented by using the order of hash values and then sorting by the grouping key in case of same hash values, it is relatively troublesome. The disadvantage of this framework is that it will put the burden of final merging calculation on the master computer, causing master computer's computing power and network capacity to become a bottleneck.

For the single table operations described above, it is relatively easy to implement the distributed computing. Except for the final operation result of the node to be transmitted to the master computer at the end, the nodes are independent during most of the operation time and there is no data transmission between them. Expanding the cluster size will not increase the network burden, but can effectively share the amount of calculation.

Usually, for the distributed database, the clusterization of single-table operation also uses this method, it will not cause excessive network transmission between nodes. However, some databases adopt the hash method when the big grouping is performed, and the data will be distributed between nodes (distribute the records with the same hash value to the same node and then perform the single machine grouping). The advantage is that the nodes will share the amount of aggregating calculation (equivalent to the final merging action in the above framework). However, its disadvantage is that a large amount of network transmission will limit the cluster size. When a certain limit is reached, more nodes will not increase the computing performance.

As for the join operation, it should be discussed based on specific association. The homo-dimension tables association and the primary-sub table association are relatively simple. As long as the data is properly distributed to make the data with the same primary key in the association table be assigned to the same zone number, it can ensure that the associated data is in the same zone (i.e., in the same node), and there is no need to transmit the data between nodes. The data distribution required here is only related to the primary key, which is similar to the requirement for ordering the primary key, and can be easily processed and achieved during data sorting and appending. Once the data is properly distributed, the operation code is still the same as that of a single machine.

Since the dimension table in the foreign key association needs to be accessed randomly, the situation will be more complicated. We will discuss it in the following two sections.

The database does not distinguish various situations of join operation. A common method is to extend the hash method from single machine to the cluster, that is, each node distributes its own data to all nodes (forming the buffer file) according to the hash value to ensure that the associated data are in the same node, and then perform the single machine join operation on each node. The previous distribution process of this algorithm will generate a large amount of network transmission, and also cause the phenomenon of limited cluster size. As a result, the performance improved by multi-machine sharing of calculation will be completely offset by the performance decrease caused by the transmission, in this case, the performance will not be improved even if more nodes are added.

The join operation has been a difficulty for the distributed database.

9.3 Duplicate dimension table

Let’s discuss the foreign key association, starting with the small dimension table, that is, the situation where the dimension table can be loaded into node’s memory.

So, which node's memory should be used to hold the dimension table?

All nodes’ memory should hold one copy.

Since the fact table under a cluster is very large and need all nodes to store, while the dimension table record will be accessed randomly, and hence the fact table on any node may associate with all dimension tables’ records. If the dimension table was only stored on a certain node, it would cause a large amount of network transmission. When the dimension table is small enough, it can be loaded on every node, in this way, the association operation becomes a local operation.

This kind of dimension table that has been duplicated in multiple copies is called the **duplicate dimension table**.

Since the small dimension table is not large, there is no need to store it in multiple zones. Instead, it can be duplicated and stored in the external storage of every node, and loaded on every node.

	A	B
1	["192.168.0.101:8281","192.168.0.102:8281",..., "192.168.0.104:8281"]	
2	fork to(4);A1	=file("product.ctx").open().import()
3		>env(PRODUCT,B2)
4	=memory(A1,PRODUCT)	
5	=file("orders.ctx":to(4),A1).open().cursor(p_id,quantity)	
6	=A5.switch(p_id,A4)	
7	=A7.groups(p_id.vendor;sum(p_id.price*quantity))	

First, use the fork framework to load the dimension table on every node and name it as a certain global variable. Then, on the master computer, use the memory() function to create a duplicate dimension table based on the global variables of all nodes. The subsequent operations on the cluster table are basically the same as those on a single machine. For a certain node, the duplicate dimension table is exactly the table sequence in local memory, which can be used, like the dimension table on a single machine, in the functions such as switch() and join() to achieve the addressization. When SPL processes the operation on each node, it will search for this dimension table locally for association.

The dimension tables are often reused in multiple computing tasks, and the fork code block can be executed when the node is started, and the program of master computer could just start from the establishment of the duplicate dimension tables. For small multi-layer dimension table, the pre-association can be performed on the node in advance.

By distinguishing the dimension table and the fact table, and by means of the smaller feature of dimension table, we can load the dimension table on every node in advance. Such foreign key association will not generate network transmission.

Database does not distinguish between dimension table and fact table; it generally determines whether to copy the table to the node based on the size of tables. For the situation of associating two tables, there is little difference from the above-mentioned scheme if the database is well-optimized. But when there are more tables or more complex association layer, the database may be "perplexed". Some distributed databases that are not optimized well will store every table in partitions, resulting in a poorer association performance.

9.4 Segmented dimension table

The dimension table needs to be accessed randomly, but the external storage does not have such ability, therefore, we should load the dimension table into memory whenever possible. If the dimension table is so large that one node cannot hold, we should try to load it into multiple nodes' memory.

For the relatively large dimension table, storing it in the external storage will also use the multi-zone composite table, and when it is loaded on the node, multiple zones will be used as well. Each node holds a zone, and the zones in the memory of multiple nodes collectively constitute a complete dimension table, called **segmented dimension table**.

	A
1	["192.168.0.101:8281","192.168.0.102:8281",..., "192.168.0.104:8281"]
2	=file("product.ctx":to(4),A1).open().memory()
3	=file("orders.ctx":to(4),A1).open().cursor(p_id,quantity)
4	=A3.join(p_id,A2,vendor,price)
5	=A4.groups(vendor;sum(price*quantity))

In A2, after the cluster table is created, it will be loaded as a segmented dimension table that can be used to be associated with the cluster table in A4.

Unlike the duplicate dimension table, the data of the segmented dimension table is stored on all nodes, and hence no any node has a dimension table with all data. In this case, the addressization mechanism which converts the foreign key of fact table to the dimension table record fails, because the dimension table record associated with a certain fact table record on a certain node may be on another node, and a cross-node address does not work. Therefore, we can only use the join() function to take out the fields of dimension table record to be referenced and then perform subsequent calculations.

In order to quickly locate the node where the associated dimension table record is on, the zone expression of multi-zone composite table will also be used. When associating the dimension table, the master computer will calculate the zone expression with the foreign key of fact table so as to obtain the node where the dimension table record associated with this foreign key will locate. The multi-zone composite table, as a dimension table, its zone expression must be calculated based on the primary key (the primary key of dimension table corresponds to foreign key of fact table).

Network transmission and hard disk reading have some similarities, both have a more complex preparation action, and both are not suitable for frequent and small-amount-data access. When

executing the join() function for cursors, SPL will fetch a large number of foreign keys and transfer them to appropriate nodes for query at a time, instead of processing only one foreign key each time. SPL does not provide the method for obtaining the record of segmented dimension table for a single foreign key.

We emphasize again that unlike the homo-dimension tables association and the primary-sub table association, although the segmented dimension table and associated fact table may use the same zone number, it just represents a split method, and does not mean that data with the same zone number will be associated.

While using the segmented dimension table to associate, network transmission will occur during the operation. The transmission amount, however, is not very large, only involving the fields of the associated record between the fact table foreign key and the dimension table, and there is no need to transmit other fields of fact table. As a result, the calculation can be achieved directly, and no buffer data is generated in this process. On the contrary, when the distributed database performs the join operation, it has the following disadvantages: it needs to transmit all fields of two tables involved in the associated result set; it also needs to buffer these transmitted data, only in this way can subsequent single-machine join operations be performed on each node; there may be "unlucky hash function" that cannot be avoided in hash algorithm, resulting in a very unbalanced amount of computation on different nodes, which seriously affects the overall computing efficiency. On the whole, the segmented dimension table has more advantages than the hash join algorithm of database.

For larger dimension table that cannot be held by multiple nodes, the external storage scheme should be used. That is, distribute the fact table in a cluster, and still duplicate the dimension table on every node, and execute the one-side partitioning algorithm to the zone of each fact table on the node. In this way, it returns to the situation where there is no dependency between nodes, and hence it can be solved by using the previous framework.

9.5 Redundancy-pattern fault tolerance

While performing the cluster operation, the fault tolerance must be considered. For the single machine operation, if the machine fails, the operation fails. For the cluster operation, however, the failure of only a few nodes may not affect the cluster to continue working.

To be fault tolerant, redundancy must be used. If, as the situations discussed in the previous sections, each node only holds the data of one zone, then the failure of any node will cause the data to be incomplete, thereby making it impossible to perform proper calculation.

The esProc server can store the data of multiple zones at the same time. When a node fails, if all zones can be found on the remaining nodes, then the calculation can still proceed. However, it will increase the amount of tasks executed by some nodes, and the overall computing performance may also decrease.

Still, we take four zones as an example, and let each node hold two zones as follows:

101: zone 1, 2; 102: zone 2, 3; 103: zone 3, 4; 104: zone 4, 1

Let’s examine the previous code:

	A
1	["192.168.0.101:8281","192.168.0.102:8281",..., "192.168.0.104:8281"]
2	=file("orders.ctx":[1,2,3,4],A1)
…	…

If all nodes run normally, the file() function in A2 will take zone 1 on 101, take zone 2 on 102,..., and take zone 4 on 104 to constitute a cluster file. If 103 fails, resulting in a failure to find zone 3 on 103, it will continue to search 104. If zone 3 is still not found, it will turn back to 101 to search, and will eventually find and take zone 3 on 102, and then take zone 4 on 104. As a result, the four zones of this cluster file will be taken from 101, 102, 102, and 104 respectively, and 102 will execute the operation of two zones.

Such scheme using the data by multiple times to implement fault tolerance is called the **redundancy-pattern fault tolerance**.

There is a problem here. According to our previous assumption that when each node holds only one zone, each zone of the multi-zone composite table of cluster can only correspond to unique node. But if there are redundant zones on the node, this correspondence is not unique. For example, we can take out four zones from 101, 102, 103 and 104 respectively, and we can also take the same zones from 104, 101, 102 and 103. For the single-table operation, how zones are distributed will not affect the calculation results, but for the multi-table association operation (like the homo-dimension tables association), a same zone distribution method is required to continue the operation, because zone 1 on 101 and zone 1 on 104 cannot be associated.

	A
1	["192.168.0.101:8281","192.168.0.102:8281",..., "192.168.0.104:8281"]
2	=file("A.ctx":[1,2,3,4],A1)
3	=file("B.ctx",A2)
…	…

The file() function can generate a new cluster file according to the known zone distribution scheme of a certain cluster file, and A3 will adopt the same zone distribution as A2. In this way, the association calculation can be performed based on the constituted cluster table.

Let’s continue to examine the distribution scheme just mentioned:

101: zone 1, 2; 102: zone 2, 3; 103: zone 3, 4; 104: zone 4, 1

If both 103 and 104 fail, not all zones can be found on 101 and 102, which makes it impossible to continue to calculate. In other words, such redundant distribution scheme only tolerates the failure of one node. It is found after a closer analysis that this scheme can continue to work when any node fails, but can't work once any two nodes fail.

We call the **fault tolerance** of this distribution scheme as 1, while for the scheme with only one zone for one node in the previous example, its fault tolerance is 0. For the cluster with n nodes, if each node has all data, its fault tolerance is n-1. When the fault tolerance is k (the operation can still be performed after any k nodes fail; the cluster fails when k+1 nodes fail), the data is required to be duplicated by k times (that is, in addition to one original data, k more copies should be stored). The fault tolerance is sometimes called **redundancy**.

For the cluster with n nodes, we can simply use the **loop distribution** scheme to achieve k fault tolerance:

Node 1: zone 1, zone 2,…, zone k+1

Node 2: zone 2, zone 3,…, zone k+2

…

Node i: zone i, zone i+1,…, zone i+k

…

Node n: zone n, zone 1,…, zone k

where, zone m is equivalent to zone m-n when m>n.

9.6 Spare-wheel-pattern fault tolerance

Loading the data into memory in advance can obtain much better performance than external storage. When the amount of data is so large that the memory of one machine cannot hold, we can use the nodes of a cluster to load in segments so as to share the calculation at the same time. The multi-machine parallel computing framework and cluster table mentioned above can also support in-memory operations.

However, in a cluster environment, we will also face the problems of fault tolerance and x fault tolerance. Moreover, for the data in memory, the redundancy-pattern fault tolerance scheme mentioned in the previous section cannot be used.

In the redundancy-pattern fault tolerance scheme, we duplicate k more copies of the data, that is, the k fault tolerance is obtained by k+1 copies of data. In other words, in order to obtain k fault tolerance, the utilization rate of storage is only 1/k. This is acceptable for external storage, because the hard disk is cheap enough and its capacity can be expanded almost infinitely. However, since memory is much more expensive and there is an upper limit on capacity expansion, a utilization rate of only 1/k is intolerable.

In order to solve this problem, we can use the **spare-wheel-pattern fault tolerance**, which works in a way that still divides the data into n zones, loads these zones on n nodes respectively, and then prepares k idle nodes as spare node. In this way, when a running node fails, a certain spare node will be started immediately to load the data of the failed node, and reconstitute a cluster with complete data together with other nodes to continue to provide services. After troubleshooting, the failed node will return to normal state, and can be used as a spare node. The whole process is very similar to the mode of replacing the spare wheel of a car.

If more than k nodes fail at the same time, the cluster with complete data cannot be constituted even if all k spare nodes are used up. At this time, we have to declare that the cluster fails.

The memory utilization rate of this scheme can reach up to n/(n+k), which is much higher than 1/k of redundancy-pattern fault tolerance. In this scheme, the amount of data to be loaded into memory is usually not very large, and the instant loading time is not much when the node fails, and hence the cluster service can be restored quickly. Conversely, if the spare-wheel-pattern fault tolerance is used for external storage, the amount of data that needs to be instantly prepared in case of node failure may be very large, which will cause the cluster to be unable to provide services for a long time.

The node that loads the data into memory needs a loading action:

	A	B
1	if z>0	>hosts(z)
2		=file("orders.ctx":z).open().memory()
3		>env(ORDERS,B2)

Parameter z represents the zone number. Use the hosts() function to register the zone number of in-memory data of the node on esProc server, and then load the data and name it as a global variable of the node.

At this point, the operation code can use the loaded in-memory data:

	A
1	["192.168.0.101:8281","192.168.0.102:8281",..., "192.168.0.105:8281"]
2	=hosts(4,A1)
3	=memory(A2,ORDERS)
4	=A3.cursor().groups(area;sum(amount))

Note that there are 5 nodes in A1, and the hosts() function needs to find 4 normally-run nodes (corresponding to 4 zones) from these nodes. If not found, it will return a signal indicating that it has failed. Once found, it will check for the completeness of the data zone numbers loaded on 4 nodes. If there is a missing zone number, it will take the zone number as a parameter to execute the loading code on the node so as to form a cluster with complete data, and then generate a cluster in-memory table for calculation.

9.7 Multi-job load balancing

Similar to the multi-thread parallel computing on a single machine, the multi-machine parallel computing framework introduced in the first section will also wait for the slowest node to return the result before proceeding. We can try to make the amount of data to be calculated by each node more balanced, but we can't guarantee the execution speed of each node is the same, and hence the phenomenon of waiting for the slow node by the fast node is still unavoidable. In most cases, such wait will have little impact, but it is intolerable in pursuit of ultimate performance.

Theoretically, just like the way for handling multi-thread, we can split the task into smaller pieces and dynamically allocate them to balance the load on each node, so as to reduce the wait. However, the multi-machine situation is more complicated since the dynamic load balancing can be achieved only with data redundancy. If a certain data is saved only on one node, then only this node can calculate it. In this case, other nodes have to wait even if they are faster, otherwise, the data needs to be transmitted through the network, which will not only cause network latency, but also consume the computing resources of the node where the data is located. As a result, the loss outweighs the gain. Since multiple threads on the same machine share the stored data, this problem does not exist.

	A
1	["192.168.0.101:8281","192.168.0.102:8281",..., "192.168.0.104:8281"]
2	=callx("sub.dfx",to(1000);A1)

The callx() function can implement this mechanism. The sub.dfx in the parameter is the script for calculating the job on the node, taking the job number as the parameter. A2 will generate 1000 jobs, only part of which will be allocated to nodes at first to make the job number of each node reach its limit, and then the node that finishes a job will be allocated another job, hereby achieving the dynamic load balancing. When a calculation error occurs because the data for a certain job is not on the allocated node, the callx() will reallocate this job to another node. If no node can execute the job, the calculation fails.

At the end of calculation, A2 will obtain a sequence composed of the return value of every sub.dfx, which is in order of job number.

There is another problem when the job amount is large. After each job is finished, the result will be returned to the master computer, resulting in too much number of returns, this situation will also bring a heavier burden to the network. Fortunately, many operations allow the node to return the results after performing a round of aggregation, in this way, the number of returns will be reduced to that of nodes.

The callx() function has another parameter that indicates the operation script for aggregating first.

	A
1	["192.168.0.101:8281","192.168.0.102:8281",..., "192.168.0.104:8281"]
2	=callx("sub.dfx",to(1000);A1;"reduce.dfx")
3	=A2.sum()

Assuming that the calculation task is to sum the return values of each sub.dfx, the code of reduce.dfx is:

	A
1	return p1+p2

where, p1 and p2 are the two parameters of reduce.dfx; p2 is the return value of each job, and p1 is the current aggregation value of the node. Each time the node finishes a job, it will call the reduce.dfx so as to obtain a new aggregation value to replace the current one. After the node finishes all jobs, it will transmit the last aggregation value, as its return value, to the master computer. At this point, A2 will get a sequence composed of the return values of all nodes, in the order of the node in its parameters (i.e., A1). The p1 and p2 here are a bit like ~~ and ~ in the iteration function.

In distributed computing terminology, such aggregation action performed by node is called **reduce**.

Postscript

This book contains 60 sections, describing dozens of basic high-performance algorithms or storage schemes for big structured data. The flexible use of such methods has been made in many actual scenarios. Compared with SQL on conventional relational database, the algorithms implemented in SPL can often improve the performance several times or even a hundred times.

As a programming language, SPL theoretically would not be faster than SQL. The reason why SPL actually runs faster is that it can implement the high-performance algorithms that SQL cannot implement. If a computing task has already adopted the best algorithm when it is implemented in SQL, rewriting it in SPL will not run faster. However, there are too many high-performance algorithms that are hard to be implemented in SQL, as long as the code is somewhat complex, it is almost certain that we will be able to find the points that can change the storage scheme and algorithm so as to optimize the performance. For more than ten scenarios we have performed, we could always find certain points to improve every time, in this case, rewriting in SPL can effectively improve the overall performance. Of course, coding in C/C++ or Java may also get a higher performance, but the development efficiency is too low.

It needs to be noted once again that each algorithm in this book has its own scenario to which it adapts; and even some algorithms cannot be used at the same time due to certain contradictions between them, therefore, the algorithm needs to be selected according to practical situation. We have been emphasizing that we should first fully understand task's objectives and data characteristics, and then design an optimization scheme according to actual conditions. Moreover, the big data computing task in reality does not simply correspond to the algorithm presented in this book one by one but a comprehensive task, and hence we should use these basic algorithms in an appropriately combined manner to cope with real task. Consequently, it is more important to learn the analysis methods.

Due to space limitations, this book does not give all examples and tests. If you are interested in learning more test examples and codes, go to Raqforum to find them.