Hindawi
Wireless Communications and Mobile Computing
Volume 2018, Article ID 4263831, 11 pages
https://doi.org/10.1155/2018/4263831

Research Article

Dynamic Pricing for Resource Consumption in Cloud Service

Bin Cao,[1] **Kai Wang**,[1] **Jinting Xu**,[1] **Chenyu Hou**,[1] **Jing Fan**,[1] **and Hangning Que**[2]

[1] *College of Computer Science & Technology, Zhejiang University of Technology, Hangzhou, China*
[2] *NetEase, Inc., Hangzhou, China*

Correspondence should be addressed to Jing Fan; fanjing@zjut.edu.cn

Received 24 February 2018; Accepted 19 April 2018; Published 24 May 2018

Academic Editor: Shangguang Wang

This paper studies dynamic pricing for cloud service where different resources are consumed by different users. The traditional cloud resource pricing models can be divided into two categories: on-demand service and reserved service. The former only takes the using time into account and is unfair for the users with long using time and little concurrency. The latter charges the same price to all the users and does not consider the resource consumption of users. Therefore, in this paper, we propose a flexible dynamic pricing model for cloud resources, which not only takes into account the occupying time and resource consumption of different users but also considers the maximal concurrency of resource consumption. As a result, on the one hand, this dynamic pricing model can help users save the cost of cloud resources. On the other hand, the profits of service providers are guaranteed. The key of the pricing model is how to efficiently calculate the maximal concurrency of resource consumption since the cost of providers is dynamically varied based on the maximal concurrency. To support this function in real time, we propose a data structure based on the classical B+ tree and the implementation for its corresponding basic operations like insertion, deletion, split, and query. Finally, the experiment results show that we can complete the dynamic pricing query on 10 million cloud resource usage records within 0.2 seconds on average.

1. Introduction

In recent years, the development of cloud computing is extremely rapid [1]. With the cloud computing, we can deploy different kinds of computing resources in the cloud, rather than a specified server [2]. In other words, there is no need to use hardware equipment, which is very convenient for the companies or organizations so they can focus on their core businesses rather than expending resources on computer infrastructure and maintenance [3–7]. Therefore, more and more users are looking for the cloud computing, which makes the cloud resource providers emerge. Due to the reason that the service deployed in the cloud may migrate between different resources, and in the meantime mobile users such as smart devices cause the dynamic change in resource consumption, hence, a dynamic pricing scheme is needed for both service providers and consumers.

The traditional cloud resource pricing models can generally be divided into two categories. The first one is on-demand service; i.e., the users should pay for the fee, which is based on the using time and the actual resource consumption. This pricing model is suitable for the short-term users. The second one is reserved service; i.e., each user pays a fixed fee for a month or year and could use the cloud resource during this period of time without limitation.

However, both of the two pricing models have disadvantages. For the first one, users are required to pay a huge fee if they use the cloud resource for a long time. But in this case, the concurrency for these users maybe little; e.g., only one computer is online, and compared with users who have multiple computers online simultaneously, it is unfair to charge users with little concurrency. As for the second one, it is unfair for the users who use the cloud service for little time monthly. They must pay for the same fee with the users who use the cloud resource for a long time. Therefore, it is meaningful to design a more reasonable pricing model for the cloud resources service.

In this paper, for cloud resource, we propose a flexible dynamic pricing model which takes into account occupying time, resource consumption, and maximal concurrency. In our pricing model, the fee of cloud service for the user is mainly composed of three parts: the monthly rental, the fee of his maximal concurrency, and the fee of his using time and resource consumption. As we all know, it is inevitable that many users use the cloud resources at the same time, which may cause a great concurrency of resource consumption on the cloud servers. The greater number of concurrency is, the higher load is on the cloud servers, which results in the higher cost for the cloud resource providers. In other words, the cost of the cloud resource providers is mainly determined by the maximal concurrency of all the users. In fact, many service providers also price their service based on the maximal concurrency of resource consumption that the user needs, which also motivates our work.

However, the dynamic of service migration and user mobility causes a large number of records for the resource consumption, which makes it challenging to find the maximal concurrency of resource consumption in real time when we calculate the fee of the cloud servers for users in our dynamic pricing model.

Suppose that there are 100,000 users using the cloud resources with their usage records. In order to calculate the cost, the cloud resource provider needs to query their usage records and work out the maximal concurrency. However, it is difficult to find the maximal concurrency of resource consumption since it may occur at an arbitrary instant time. It would be extremely inefficient if we calculate the concurrency for each timestamp when data amount is large. Therefore, we should design a new efficient algorithm to find the maximal concurrency. As we know, this problem is similar to the aggregation query [8] since both of them calculate the overall information at a given time. However, different from the aggregation which has a time interval as a given condition, our problem needs to calculate the maximal concurrency without a time interval. Therefore, there is no existing algorithm that can solve this problem directly as far as we know. In order to solve the problem, we propose a data structure called B++-tree and corresponding operations including insertion, deletion, and split. Additionally, query processing algorithm based on B++-tree is also presented. Basically, we solve the efficiency issue by storing all users records into B++-tree and calculating the maximal concurrency of resource consumption by traversing all leaf nodes of B++-tree.

The contribution in this paper can be summarized as follows:

(1) We propose a more reasonable pricing model of cloud resource, which takes into account occupying time, resource consumption, and maximal concurrency.

(2) We propose a data structure called B++-tree and the operational algorithm based on B++-tree to calculate the maximal concurrency which is the fundamental of the cost for the cloud resource providers.

(3) We performed extensive experiments to test our algorithm. The experiment results show that we can complete the maximal concurrency query on 10 million data within 0.2 seconds.

The rest of this paper is organized as follows. Section 2 discusses the related work. Section 3 introduces the pricing models we proposed. Section 4 introduces the B++-tree and the implementation details for its corresponding operations. Section 5 presents the experiment. Section 6 is our conclusion.

2. Related Work

Our problem can be divided into two major subproblems: (1) how to design a reasonable pricing model and (2) how to find the maximal concurrency of resource consumption. Therefore, we investigate related work from these two parts, respectively.

For the first part, cloud computing is different from the classic distributed system. The pricing model of the cloud service should take the pricing fairness, evolving system dynamics, and cost of failures into account [9]. The existing pricing schemes in the cloud market can be summarized into three types: trading on-demand service, reserved service, and spot service, respectively [10]. Trading on-demand service means that the cost of a user is based on the time he used for cloud resources. But users pay a fixed fare for cloud resources in the reserved service. These two static pricing schemes are the main pricing models in the current cloud market [11, 12]. Different from them, the spot service is a dynamic pricing scheme where users' payments depend on the relation of their demand and the available cloud resource. Based on the main idea of the spot model, many kinds of dynamic pricing model have been proposed in recent literature, like auction mechanisms [13–16]. Zheng et al. [17] developed a predator-prey model which can simulate the interactions between demand and resource and compute the fare of cloud service. Zhang et al. [18] proposed a joint pricing and scheduling strategy and proved the worst-case competitive ratios of the pricing functions. However, none of above works consider the dynamic pricing for cloud service that we do since the algorithm proposed in this paper can efficiently report the total cost for the providers no matter how the service migrates or user behaves.

For the second part, as far as we know, there are few algorithms that can find the maximal concurrency of resource consumption to solve our problem directly. But we find a closed problem called temporal aggregations, in which there are some temporal aggregations operators such as count, sum, and average. These operators are similar to the method of calculating the maximal concurrency. There are also many algorithms to solve temporal aggregations problem. Kline and Snodgrass [19] proposed aggregation tree, which is a data structure, to support incremental computation for temporal aggregation. However, the aggregation tree is unbalanced whose time complexity is $O(n^2)$ for constructing a tree, where n is the number of intervals. Moon et al. [20] presented a balanced tree algorithm whose time complexity is $O(n \log n)$, where n is the number of intervals. Besides, Moon et al. [21] also proposed a bucket algorithm and parallelized it on a shared-nothing architecture. Yang and Widom [22] presented a data structure called SB-Tree which combines B-tree [23, 24] and segment tree [25]. SB-Tree can feedback a query in $O(n \log n)$ and an update in $O(n \log n)$, where n is the number of intervals. However, there is a difference between our problem and temporal aggregation. The time interval is

Table 1: User records.

User	Time interval	Concurrency
A	[5–10]	2
B	[10–20]	4
C	[0, 15]	6
D	[5, 15]	1

Table 2: Intervals after splitting from Figure 1.

Users	Time interval	Concurrency
C	[0–5]	6
A, C, D	[5–10]	9
B, C, D	[10, 15]	11
B	[15, 20]	4

a condition given in the temporal aggregation. But in our problem, we do not know when the concurrency of resource consumption is maximal. In other words, we do not have a certain time interval in our problem which needs to be calculated by our algorithm. Therefore, the algorithms for temporal aggregation cannot solve our problem directly.

3. Pricing Model

As mentioned in Section 1, the maximal concurrency of resource consumption plays an important role in pricing the cloud service. In other words, the price is mainly decided based on the maximal concurrency on cloud servers at the same time.

To further illustrate the meaning of the maximal concurrency, we give an example as follows. Table 1 and Figure 1 show the using time and usage records of four users, A, B, C, and D. Each user record consists of a time interval and its concurrency. The time interval represents the time range of users using the cloud resources. We can easily see that user A uses the cloud resources in time interval [5, 10] and his concurrency is 2.

According to Table 1 and Figure 1, we can split the original interval again, which is shown in Table 2. In Table 2, obviously we can see that 11 is all the users' maximal concurrency of resource consumption. So we find out the cost of the cloud resource provider. After that, we can calculate the user's price based on his usage data and the profit for the cloud resource provider.

Next, based on the maximal concurrency of resource consumption, we can design the following price model.

3.1. Pricing Model

Modeling the Concurrency. Suppose that there is a rate q, which means the cost of each concurrency for the cloud service provider. In addition, in a certain month, the total number of users is n and the maximum concurrency of resource consumption is m. Therefore, the cost of the cloud service provider is $C = q \times m$. As for users, we propose a new pricing model. Firstly, each user needs to pay for B as the monthly rental. In addition, the provider will charge a fee

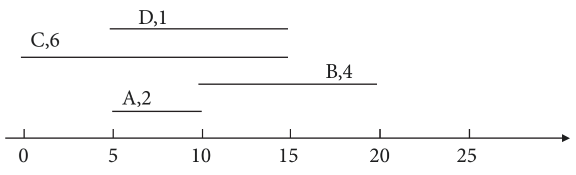


Figure 1: Graphical representation of user records.

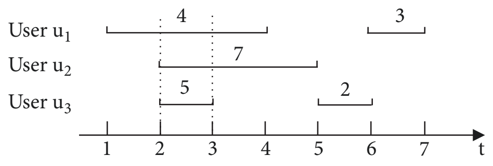


Figure 2: The consumption of the cloud server.

for each user based on his maximal concurrency. Therefore, a user u_i should pay $\text{price}(u_i) = \max_i \times q + B$ for his resource consumption. Finally, the profit of the cloud service provider is

$$\text{profit} = \sum_{i=1}^{n} \left(\max_i \times q + B\right) - m \times q. \quad (1)$$

Example. As shown in Figure 2, assume that the user's monthly fare $b = 1$, the rate $q = 2$. From time 1 to 7 is a month. The maximum concurrency of each user u_1, u_2, and u_3 is 4, 7, and 5, respectively. The cloud service fare of users u_1, u_2, and u_3 is $\text{price}(u_1) = 4 \times 2 + 1 = 9$, $\text{price}(u_2) = 7 \times 2 + 1 = 15$, and $\text{price}(u_3) = 5 \times 2 + 1 = 11$. The maximum concurrency of resource consumption is the dotted line part in the figure, which is $m = 4 + 7 + 5 = 16$. Therefore, the profit of the cloud service provider is $\text{profit} = 9 + 15 + 11 - 16 \times 2 = 3$.

However, this pricing model has some drawbacks. For example, the user u_3 uses the cloud server for a little time. But he needs to pay for more fare than user u_1 because of his higher concurrency. Therefore, the merely using concurrency model is not reasonable in some special cases, and we further improve this model by taking resource consumption into account.

Combining Resource Consumption. Assume that a user u_i has x records of cloud resource consumption on the current month and each record starts at t_i and ends at t'_i. The concurrency of the record is m_{t_i}. Therefore, the cloud consumption of u_i is $U_i = \sum_{i=1}^{x} (t'_i - t_i) m_{t_i}$. In addition, the maximal concurrency of user u_i is $\max_i$. The final cloud service fare of u_i can be calculated by following equation:

$$\text{price}(i) = \alpha U_i p + (1 - \alpha) \max_i q + B, \quad (2)$$

where $0 < \alpha < 1$ is a factor that adjusts the using time and the maximum concurrency of users. p is the resource consumption rate which means the cost of each resource consumption. q is the maximum concurrency rate. b is the monthly rental of cloud service. The values of α, p, q, and b can be adjusted according to actual needs.

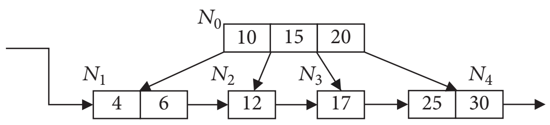


FIGURE 3: An example of B+tree.

Example. Still using Figure 2 as an example, assume that the user's monthly rental $b = 1$, the maximum concurrency rate $q = 2$, the usage rate $p = 1$, and α is 0.5. Then the consumption of users u_1, u_2, and u_3 is $U_1 = 4 \times 3 + 3 \times 1 = 15$, $U_2 = 7 \times 3 = 21$, $U_3 = 5 \times 1 + 2 \times 1 = 7$. The maximum concurrency of each user u_1, u_2, and u_3 is 4, 7, and 5, respectively. Therefore, the cloud service pare of users u_1, u_2, and u_3 is $\text{price}(u_1) = 0.5 \times 15 \times 1 + 0.5 \times 4 \times 2 + 1 = 12.5$, $\text{price}(u2) = 0.5 \times 21 \times 1 + 0.5 \times 7 \times 2 + 1 = 18.5$, and $\text{price}(u3) = 0.5 \times 7 \times 1 + 0.5 \times 5 \times 2 + 1 = 9.5$. The profit of cloud service providers is $\text{profit} = 12.5 + 18.5 + 9.5 - 2 \times 16 = 8.5$.

This pricing model compensates for the defect of the former price model which only takes the maximum concurrency of resource consumption into account. This pricing model adds the factor of users' resource consumption, which makes the model more reasonable. Besides, we can adjust the coefficient α to adapt to different cloud resources and make the model more flexible.

4. Algorithm Description

In Section 3, we propose a flexible dynamic pricing model for cloud resources. As the maximal concurrency of resource consumption plays an important role in the pricing model, how to efficiently calculate the maximal concurrency of resource consumption is a difficulty in our problem. Because when we calculate the price for resource consumption in cloud service, we should calculate the maximal concurrency of resource consumption first. Therefore, we propose a new data structure and the operational algorithms based on it to solve this problem.

In this part, we will introduce the new data structure called B++-tree which extends from B+ tree. We first introduce the structure of the B+ tree and then introduce the structure of the B++-tree. Finally, we introduce the insertion, deletion, and split operations of B++-tree, which can calculate the maximal concurrency of resource consumption.

The nodes of B+ tree can store a lot of index entries, which helps reduce the height of the tree. Besides, the leaf nodes of B+ tree are connected by pointers. It is very suitable for the query. The following is a brief introduction for the B+ tree.

Each B+ tree has a parameter called capacity c, which determines the maximal capacity for each node. For each interior node, it contains k ($k < c$) elements and points to $k + 1$ child nodes. For each leaf node, it contains l ($l < c$) elements and have a pointer to its right sibling node as shown in Figure 3. In this way, it is efficient to traverse all leaf nodes by utilizing their pointers rather than traversing from the root.

But a simple B+ tree cannot solve our problem, because we need to store the concurrency of resource consumption into the tree. So we propose a new data structure called B++-tree, which extends from B+ tree.

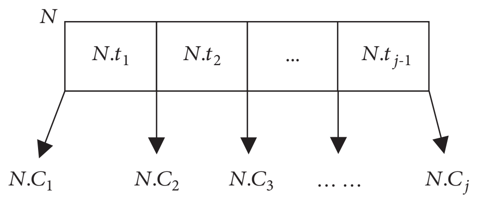


FIGURE 4: An interior node of B++-tree.

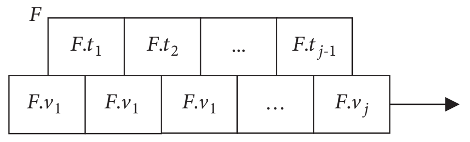


FIGURE 5: A leaf node of B++-tree.

4.1. Structure of B++-Tree. Different from B+ tree, we add an addition attribute for each leaf node of B++-tree, which is used to store the concurrency of each interval. Note that the interior nodes keep the same structure as the B+ tree and do not have the additional attribute. In this way, the concurrency of all intervals is stored in leaf nodes, and we can only traverse leaf nodes to obtain the final result, which improve the performance of our algorithm efficiently.

In the B++-tree, each node can hold up to c timestamps. If the number of timestamps stored in the node exceeds c, we should call *split* (Section 4.4) process to split the overflowed node into two new nodes. Actually, two adjacent timestamps in a node represent an interval. The interval is a criterion which can help us determine the child node that we need to traverse when we need to insert or delete a new record. The detailed structures of interior nodes and leaf nodes are as follows.

Interior Node. The structure of an interior node is shown in Figure 4. Each interior node contains j ($j \leq c$) intervals, $N \cdot I_1, \ldots, N \cdot I_i, \ldots, N \cdot I_j$. $N \cdot C_1$ to $N \cdot C_j$ are the corresponding pointer to the child nodes. $N \cdot I_i$ is associated with $N \cdot C_i$.

$N \cdot I_i = [N \cdot t_{i-1}, N \cdot t_i]$ is the ith interval. There are two special conditions:

(1) The start time of $N \cdot I_1$ is $-\infty$, if node N is the root node or it is the first child. Otherwise, the start time of $N \cdot I_1$ is $N' \cdot t_{k-1}$, where N' is the parent node of N and $N' \cdot C_k$ points to N.

(2) The end time of $N \cdot I_j$ is $+\infty$, if this node is the root node or it is the last child. Otherwise, the end time of $N \cdot I_j$ is $N' \cdot t_k$, where N' is the parent node of N and $N' \cdot C_k$ points to N.

Leaf Node. Figure 5 shows the structure of a leaf node. Compared with interior nodes, leaf nodes have additional attributes which store the concurrencies of intervals. The definition of leaf nodes' intervals is the same as interior nodes. Moreover, each leaf node has a pointer which points to its next

```
Input: Interval $I$.
Output: B++-tree after inserting $I$.
(1) Start from the root node $N$
(2) for each interval $N \cdot I$ in $N$. do
(3)     if interval $N \cdot I$ intersects with $I$
(4)         if $N$ is the leaf node
(5)             Set the common interval of $N \cdot I$ and $I$ as the new
                interval, add $v$ of $N \cdot I$ and $v$ of $I$ to the new
                interval
(6)         else
(7)             search into the child node $N'$ of $N$, insert $I$ to
                $N'$, back to step (2).
(8)         end if
(9)     end if
(10) end for
```

Algorithm 1: Insert operation.

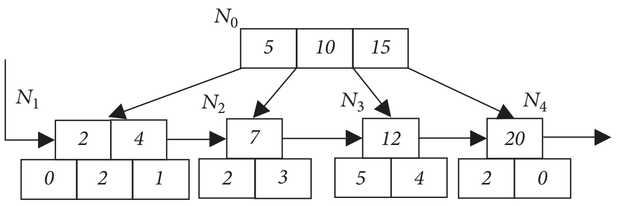


Figure 6: An example of B++-tree.

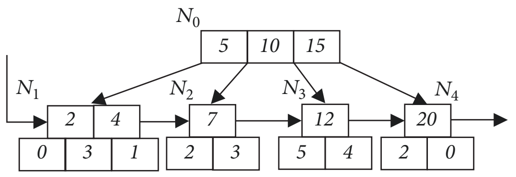


Figure 7: The B++-tree after inserting $\langle [2,4], 1\rangle$.

sibling leaf node. Specially, the last leaf node has no pointer. In addition, there is a header pointer pointing to the first leaf node.

For example, Figure 6 plots the example of B++-tree. The first interval of N_0 is $[-\infty, 5]$ and the last interval of N_0 is $[15, +\infty]$, because it is a root node. The first interval of N_1 is $[-\infty, 2]$, because it is the first child of its parent, and the concurrency of the interval is 0. The first interval of N_2 is $[5, 7]$, and the concurrency of the interval is 2.

4.2. Insertion

Main Idea. In this section, we introduce the insert operation. We define the procedure insert $(\langle I, v\rangle, N)$ as an insert operation, where I indicates the user's usage interval, such as $[5, 10]$, v indicates the user's concurrency, and N indicates the node that to insert. Insertion will be firstly processed from the root node N_{root}. We firstly traverse the root node and find the intervals which intersect with I. If $N_{\text{root}} \cdot I_i$ intersects with I, we search in the $N_{\text{root}} \cdot C_i$ and traverse it in the same way until the leaf nodes. When we traverse to a leaf node N_l, we find the intervals which intersect with I and add the v to its corresponding concurrency.

Description. Algorithm 1 shows the pseudo code of insertion. The input is the interval I, and the output is the B++-tree after insert I. Suppose that now we want to insert the record $\langle I, v\rangle$ into the tree. Start from the root node N (line (1)), for each interval $N \cdot I_i$ of N:

(1) If $N \cdot I_i$ intersects with the time interval I and N is a leaf node, and if interval I contains $N \cdot I_i$, then add v to $N \cdot I_i$ directly (line (5)). If interval I does not contain $N \cdot I_i$ but only intersects with it, then we take their intersecting interval as a new interval and add v to the new interval. The original interval keeps unchanged (line (2)~(5)).

(2) If $N \cdot I_i$ intersects with the time interval I but N is an interior node, then call insert $(\langle I, v\rangle, N \cdot C_i)$, i.e., regarding $N \cdot C_i$ as N, executed the algorithm start from the second step (line (6)~(7)).

Example. For example, we want to insert $\langle [2,4], 1\rangle$ to the tree in Figure 6. As for N_0, only interval $(-\infty, 5]$ intersects with interval $[2, 4]$, so we insert $\langle [2,4], 1\rangle$ to its child node. Then we find the second interval of $N_1 [2, 4]$, which is contained by the interval that we want to insert. So we add v to $N \cdot I_i$ directly, as shown in Figure 7.

As a slightly more complicated example, suppose that we want to insert $\langle [5,8], 4\rangle$ to the tree in Figure 6. Firstly we start from the root node N_0. We can see that only interval $[5, 10]$ intersects with the interval $[5, 8]$, so we should still call insert $(\langle [5,8], 4\rangle, N_2)$. Now N_2 is the leaf node and the interval $[5, 7]$ and $[7, 10]$ both intersect with the interval $[5, 8]$. As for $[5, 7]$, it is contained by $[5, 8]$, so add v directly. As for $[7, 10]$, $[5, 8]$, they have an intersecting interval $[7, 8]$. So we create a new interval $[7, 8]$ and its concurrency is $3+4 = 7$. Other intervals keep unchanged. Then we complete the insertion of $\langle [5,8], 4\rangle$, as shown in Figure 8.

```
Input: Node $N$ which is overflowed;
Output: Node $N_1$ and $N_2$ which are split from $N$.
(1) create new node $N_1$ and $N_2$
(2)   $N_1$ retains the first half intervals of the original node
      $N$; $N_2$ retains the rest of intervals
(3)     if $N$ is the interior node
(4)        $N_1$ retains the first half pointers, $N_2$ retains the
           rest of pointers
(5)     if $N$ is the leaf node
(6)        $N_1$ retains the first $\lceil n/2 \rceil$ values, $N_2$ retains the rest values
(7)     if $N$ is the root node
(8)        create a new root node $N'$, make it be the
           parent node of $N_1$ and $N_2$. Make $N'$ be the new
           root node
(9)     else if $N$ is not the root node
(10)       suppose $N'$ is the parent node of $N$, make
           $N'$ be the parent of $N_1$ and $N_2$
(11)      end if
(12)    If $N'$ is overflowed
(13)       Split $N'$
(14)    end if
(15)  end
(16) end
```

Algorithm 2: Split operation.

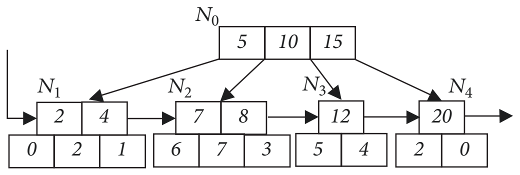


Figure 8: The B++-tree after inserting $\langle [5,8], 4 \rangle$.

4.3. Deletion. The deletion operation is similar to the insertion operation. We can regard the deletion operation as an opposing operation to the insertion. Specifically, if we want to delete the record $\langle I, v \rangle$, we can insert a record $\langle I, -v \rangle$. Therefore, deletion operation can be easily understood without extra explanation.

4.4. Split

Main Idea. As records are inserted, the number of leaf nodes' intervals gradually increases. As mentioned above, each node can hold up to c intervals. Therefore, when a node N becomes overflowed, i.e., its number of intervals exceeds c, we should split N into two nodes N_1 and N_2. The first half intervals of N are assigned to N_1 and the remaining intervals of N are assigned to N_2. Suppose N' is the parent node of N. Because of the split of N, the numbers of interval in N' will be added 1. If N' also become overflowed, split N'.

Description. Algorithm 2 shows the pseudo code of split. The input is the node N which is overflowed and the output is nodes N_1 and N_2 which are split from N. Suppose that node N is overflowed and currently stores n intervals. We define the procedure as split (N). Specific operations are as follows:

(1) Node N splits into N_1 and N_2. Node N_1 retains the first half intervals of the original node N; i.e., N_1 retains the first $\lceil n/2 - 1 \rceil$ time points. If N is an interior node, N_1 also retains the pointer from $N \cdot C_1$ to $N \cdot C_{\lceil n/2 \rceil}$. If N is a leaf node, N_1 also retains the first $\lceil n/2 \rceil$ values from $N \cdot v_1$ to $N \cdot v_{\lceil n/2 \rceil}$. Node N_2 retains the rest of intervals, which means N_2 retains the time points from $N \cdot t_{\lceil n/2 \rceil + 1}$ to $N \cdot t_{n-1}$. If N is the interior node, N_2 also retains the pointer from $N \cdot C_{\lceil n/2 \rceil + 1}$ to $N \cdot C_n$. If N is the leaf node, N_2 also retains the value from $N \cdot v_{\lceil n/2 \rceil + 1}$ to $N \cdot v_n$ (lines (1)~(6)).

(2) If node N is the root node, then create a new root node N', which is the parent node of N_1 and N_2. Make $N' \cdot t_1 = N \cdot t_{\lceil n/2 \rceil}$, $N' \cdot C_1 = N_1$, and $N' \cdot C_2 = N_2$ (line (7)~(8)).

(3) If node N is not the root node, suppose that N' is the parent node of N, and $N' \cdot C_j = N$. Keep the first $j-1$ intervals of N' unchanged. Then starting from the $j + 1$th interval to the end, move them to the right one. Make $N' \cdot t_j = N \cdot t_{\lceil n/2 \rceil}$, $N' \cdot C_j = N_1$, and $N' \cdot C_{j+1} = N_2$. If node N' is overflowed, call split (N') (lines (9)~(13)).

Example. For example, Figures 9-10 show us how the nodes split. Suppose that the capacity of the tree is 4. This means if the number of node's intervals exceeds 4, the node needs to be split. The node N_3 in Figure 9 has 5 intervals, $[30, 35]$, $[35, 40]$, $[40, 45]$, $[45, 50]$, and $[50, +\infty]$, whose intervals are bigger than 4, so the node N_3 should be split.

According to the split rules, node N_3 split into N_4 and N_5. N_4 keeps the first 3 intervals: $[30, 35]$, $[35, 40]$, and $[40, 45]$. N_5 keeps the remaining intervals: $[45, 50]$, $[50, +\infty]$. Because

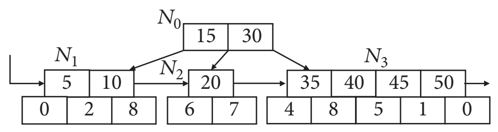


Figure 9: The B++-tree before splitting operation.

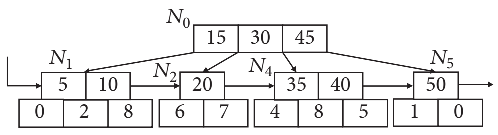


Figure 10: The B++-tree after splitting operation.

the node N_3 is leaf node, so N_4 keeps the first 3 values and N_5 keeps last 2 values. Move $N_3 \cdot t_{\lceil n/2 \rceil}$ to N_0. The result of the split is shown in Figure 10.

4.5. Query for the Maximal Concurrency. When the index building is complete, we can traverse all leaf nodes to acquire each interval and corresponding concurrency. Since all leaf nodes are connected by pointers, we can easily traverse these leaf nodes sequentially without traversing any interior node.

For example, in Figure 6, traversing the leaf node we can get the result: $\langle [2,4],2\rangle$, $\langle [4,5],1\rangle$, $\langle [5,7],6\rangle$, $\langle [7,8],7\rangle$, $\langle [8,10],3\rangle$, $\langle [10,12],5\rangle$, $\langle [12,15],4\rangle$, $\langle [15,20],2\rangle$. It is easy to see that the maximum number of concurrent users is 7.

When we get the maximal concurrency, we get the cost of cloud resource provider. Then we can use our pricing model to charge each user.

5. Experiment

In this section, we provide experimental evaluation of our algorithm. We simulate five datasets which contain 10,000, 100,000, 500,000, 1 million, and 2 million records, respectively. Each record in the dataset contains the user's name, time interval, and the concurrency. For example, "u1--> [2017-07-31 11:46:15, 2017-07-31 21:02:56], 4" is a record. "u1" is the user's name, and his time interval is from "2017-07-31 11:46:15" to "2017-07-31 21:02:56"; the concurrency is 4. We design a series of experiments in the construction of the B++-tree and the performance of operations, like query, insertion, and deletion. There are two explicit factors in our experiments, which are the data size and the capacity c. Thus, we conduct two sets of experiments by changing the data size and the capacity. For the one set of experiments, we change the data size while fixing the value of capacity. For the other set of experiments, we change the capacity while fixing the data size.

5.1. Construction of the B++-Tree. Firstly, we test the performance of the construction of B++-tree. In this experiment, we change the data size and capacity to compare with the time needed to build a B++-tree.

In Figure 11(a), we vary the data size from 10,000 to 2,000,000, while fixing the capacity to 50. We denote the time of constructing a B++-tree as CT, which means the construction time. As we can see, when the data size is small, for example 10,000 and 100,000, their CTs are very small and similar. But with the growing of data size, CT is also increasing. The greater the size of data is, the more CT consumed. Because with the increment of data size, the structure of B++-tree will be more complicated. Therefore, the construction operation consumes more time.

In Figure 11(b), we vary the capacity of B++-tree from 10 to 100, while fixing the data size to 1,000,000. It is easy to find that CT is the shortest when the capacity $c = 35$. If the capacity is too small, the tree will be very high. If the capacity c is too big, the node will store too many intervals. Neither of these conditions contributes to the construction of the B++-tree. Therefore, the capacity $c = 35$ is the most suitable for the construction of B++-tree, while the data size is 1,000,000.

5.2. Performance of Operations. In this subsection, we test the performance of operations, which includes the query performance, the insertion performance, and the deletion performance. Then we analyze the reason of the performance and give the conclusion of experiments.

Query. Firstly we test the performance of query. Figure 12 gives the result of the experiment. We change c and the data size separately to conduct comparative experiments. We denote the time of traversing the leaf nodes as TT, which means the traversal time.

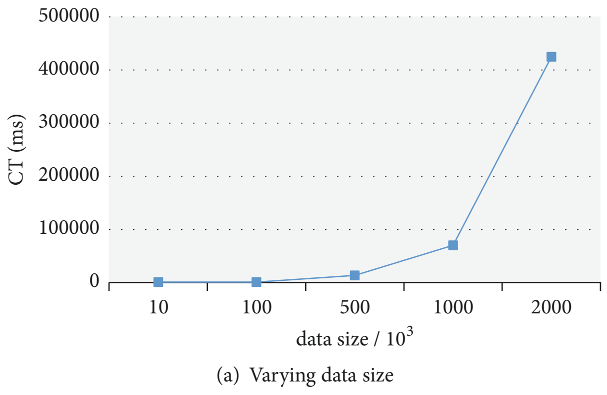
In Figure 12(a), we vary the data size from 10,000 to 2,000,000 while fixing the capacity to 50. From the figure, we can see that the greater the data size is, the bigger the TT is. Because the larger the data size is, the more leaf nodes the B++-tree has. Therefore, it takes more time to traverse the B++-tree.

In Figure 12(b), we vary the capacity c from 10 to 100 while fixing the data size to 1,000,000. The trend of TT is decreasing first but increasing again. Because when the capacity c is small, there will be too many leaf nodes in B++-tree, which cost much more time to traverse the B++-tree. When the capacity c is too big, the B++-tree will has too many leaf nodes. So when $c = 35$, the query performance is the best, while the data size is 1,000,000.

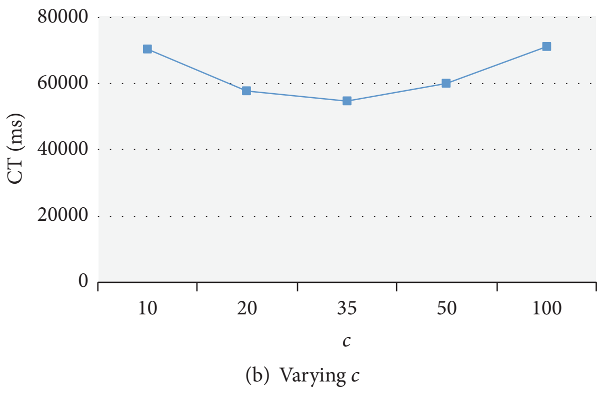
But the most important is that TT is very small all the time, which means query speed is very fast. Even the data size is 2 million; the query time is less than a second. This shows that the B++-tree we proposed is very suitable for the query operation.

Insertion. Secondly we investigate the insertion performance of B++-tree. Data insertion is the most common operation in constructing a B++-tree. We still test the insertion performance by changing the data size and capacity. We compare the time of inserting a bunch of data and the time of inserting a piece of data. The time of inserting data is called IT, which means the insertion time.

In Figure 13, we compare the time of inserting a bunch of data. In Figure 13(a), we vary the data size from 10,000 to 2,000,000 while fixing the capacity to 50 and the inserting data size to 5000. From the figure, we can see that with the data size increasing, the IT increases as well. Because

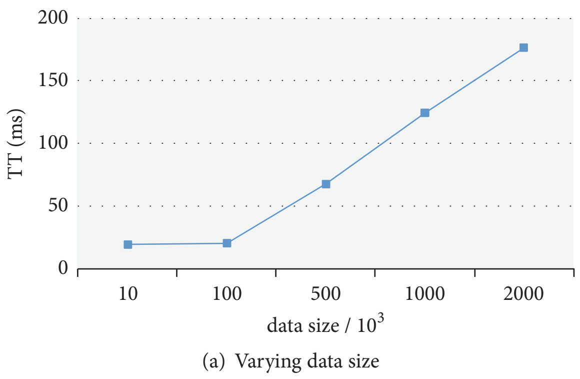


(a) Varying data size

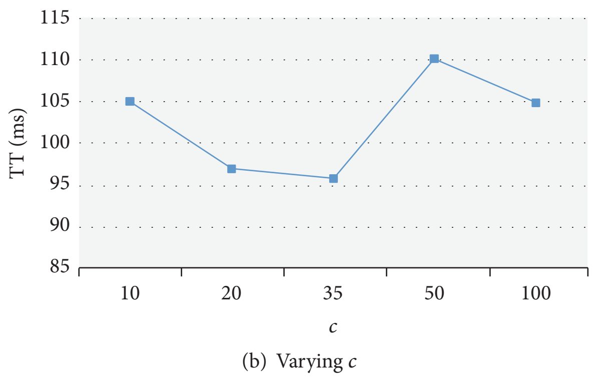


(b) Varying c

Figure 11: Experimental results of CT.

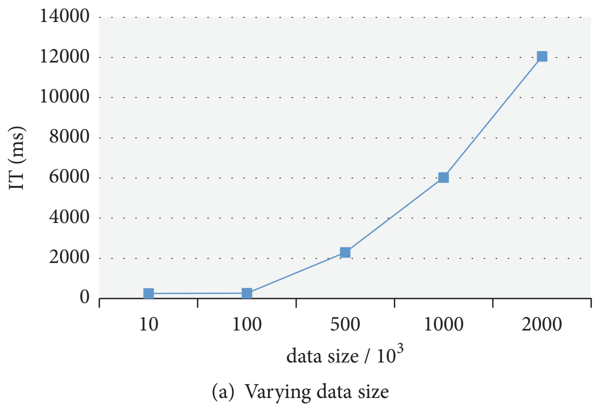


(a) Varying data size

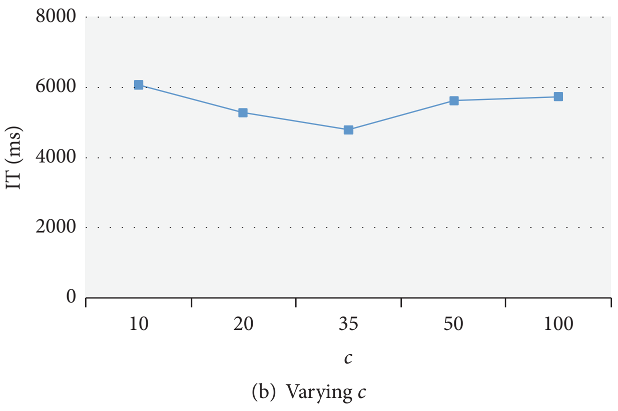


(b) Varying c

Figure 12: Experimental results of TT.



(a) Varying data size



(b) Varying c

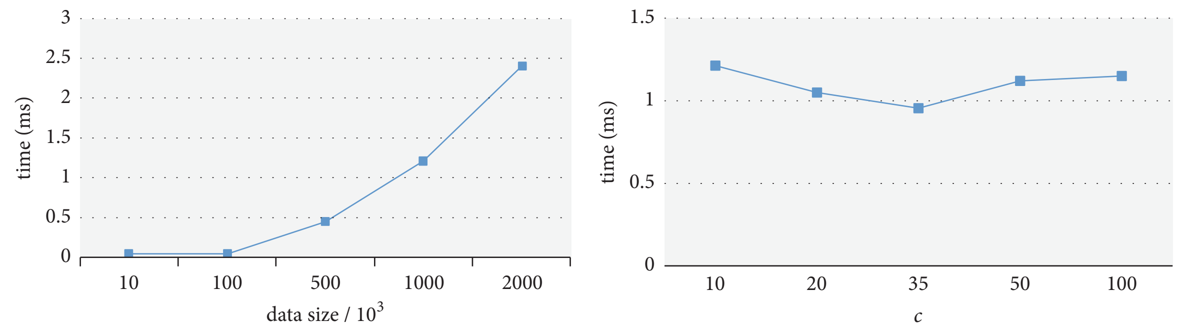
Figure 13: Experimental results of IT.

with the data size increasing, the B++-tree becomes more complicated. We need traverse more nodes to insert a record.

In Figure 13(b), we vary the capacity from 10 to 100 while fixing the data size to 1,000,000 and the inserting data size to 5000. Same as the previous experiment of query, IT is the shortest when $c = 35$, because the structure of the B++-tree is the best at this capacity. When the capacity $c = 35$, B++-tree will not have too many nodes or too many intervals in a leaf node.

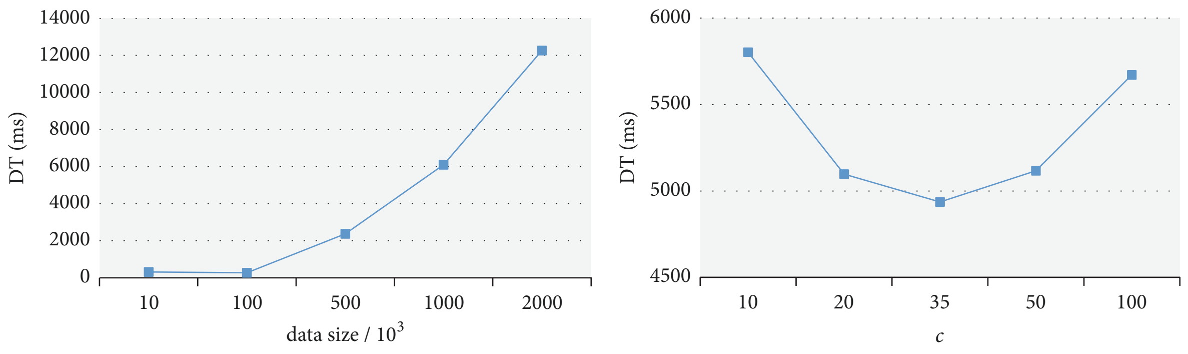
In Figure 14, we compare the time of inserting a piece of data. In Figure 14(a), we vary the data size from 10,000 to 2,000,000 while fixing the capacity to 50. We can see that the trend of Figure 14(a) is similar to Figure 13(a). It only takes 2 ms to insert a piece of data when the data size is 2,000,000.

In Figure 14(b), we vary the capacity from 10 to 100 while fixing the data size to 1,000,000. The time of inserting a piece of data is only about 1 ms. It is obvious that B++-tree is very efficient in data insertion.



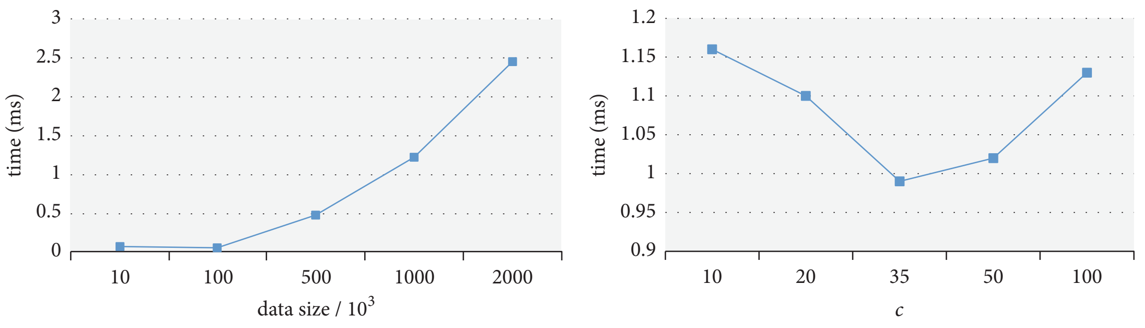
(a) Varying data size (b) Varying c

FIGURE 14: Experimental results of inserting a piece of data.



(a) Varying data size (b) Varying c

FIGURE 15: Experimental results of DT.



(a) Varying data size (b) Varying c

FIGURE 16: Experimental results of deleting a piece of data.

Deletion. Thirdly we investigate the deletion performance of B++-tree. Actually the principle of deletion is the same as the principle of insertion. Like the data insertion, we also test the deletion performance by changing the data size and capacity. We compare the time of deleting a bunch of data and the time of deleting a piece of data. The time of deleting data is called DT, which means the deletion time.

In Figure 15, we compare the time of deleting a bunch of data. In Figure 15(a), we vary the data size from 10,000 to 2,000,000 while fixing the capacity to 50 and the deleting data size to 5000. The result of Figure 15 is similar to Figure 13. With the increment of data size, the time of deletion increases as well, because of traversing more nodes.

In Figure 15(b), we vary the capacity from 10 to 100 while fixing the data size to 1,000,000 and the deleting data size to 5000. Same as Figure 13(b), DT is the shortest when $c = 35$. And the reason is the same as in Figure 13(b).

In Figure 16, we compare the time of deleting a piece of data. Figure 16 shows the same result as Figure 14. In Figure 16(a), we vary the data size from 10,000 to 2,000,000 while fixing the capacity to 50. The time of deleting a piece of data is very similar to the time of inserting a piece of data.

In Figure 16(b), we vary the capacity from 10 to 100 while fixing the data size to 1,000,000. The time of deleting a piece of data is only about 1 ms. The result shows that B++-tree is suitable for data deletion.

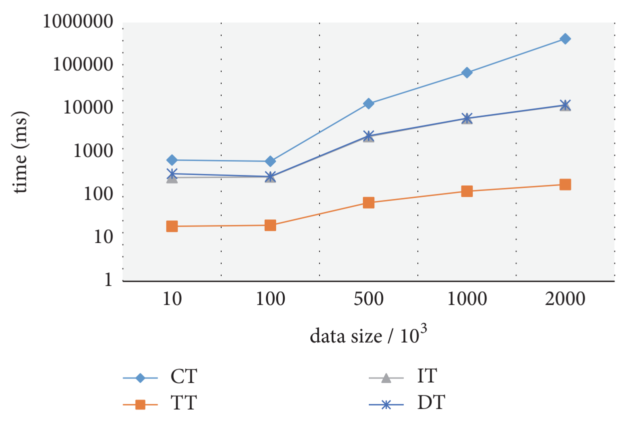


FIGURE 17: The trend of CT, TT, IT, and DT when varying the data sizes.

Finally we give the conclusion of experiment. We combine the results of the previous experiments and put them into Figure 17.

For ease of viewing, we set the ordinate of Figure 17 to a logarithmic scale of 10. From the figure we can see IT is almost the same as DT, because those two operations are the same in principle. With the increment of data size, TT, CT, IT, and DT increase at the same time. But the growth rate of the TT is not very big. Because according to the description, the query operation only needs to traverse leaf nodes, which is very fast and efficient. Besides, we can see from Figure 17 that the time of query is very short and less than a second.

As we can see from our experiments, the B++-tree we proposed in this paper is well suited for calculating the maximal concurrency for our pricing model.

6. Conclusion

In this paper, we propose a dynamic pricing model, which takes into account using time, resource consumption, and maximum concurrency to make the price of cloud resources more reasonable for both users and providers. In order to calculate the maximal concurrency of all the users, we propose a new data structure, named B++-tree, which extends from B+tree and has additional information in leaf nodes. Besides, we introduce the insertion, deletion, split, and query operation of B++-tree, which can calculate the maximal concurrency.

Finally, we performed extensive experiments to study the performance of the construction, query, insertion, and deletion operations with different data sizes and capacities of B++-tree. The result of the experiments shows that the B++-tree we proposed in this paper is well suited for calculating the maximal concurrency for our pricing model. We can complete the query operation on 10 million data in only 0.2 seconds. For the future work, we plan to find a much more reasonable pricing model which can take more factors into account.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was partially supported by the National Natural Science Foundation of China (nos. 61602411 and 61572437) and Key Research and Development Project of Zhejiang Province (no. 2015C01034).

References

[1] J. Lee, "A view of cloud computing," *Communications of the Acm*, vol. 53, no. 4, pp. 50–58, 2013.

[2] K. Sowmya and R. P. Sundarraj, "Strategic bidding for cloud resources under dynamic pricing schemes," in *Proceedings of the International Symposium on Cloud and Services Computing (ISCOS '12)*, pp. 25–30, IEEE, Mangalore, India, December 2012.

[3] A. Zhou, S. Wang, Q. Sun, H. Zou, and F. Yang, "Dynamic virtual resource renting method for maximizing the profits of a cloud service provider in a dynamic pricing model," in *Proceedings of the 2013 International Conference on Parallel and Distributed Systems (ICPADS '13)*, pp. 118–125, Seoul, Korea (South), December 2013.

[4] S. Wang, T. Lei, L. Zhang, C.-H. Hsu, and F. Yang, "Offloading mobile data traffic for QoS-aware service provision in vehicular cyber-physical systems," *Future Generation Computer Systems*, vol. 61, pp. 118–127, 2016.

[5] S. Wang, A. Zhou, C.-H. Hsu, X. Xiao, and F. Yang, "Provision of data-intensive services through energy-and QoS-aware virtual machine placement in national cloud data centers," *IEEE Transactions on Emerging Topics in Computing*, vol. 4, no. 2, pp. 290–300, 2016.

[6] Y. Ma, S. Wang, P. C. Hung, C. H. Hsu, Q. Sun, and F. Yang, "A highly accurate prediction algorithm for unknown web service QoS value," *IEEE Transactions on Services Computing*, vol. 9, no. 4, pp. 511–523, 2016.

[7] A. Zhou, S. Wang, Z. Zheng, C.-H. Hsu, M. R. Lyu, and F. Yang, "On cloud service reliability enhancement with optimal resource usage," *IEEE Transactions on Cloud Computing*, vol. 4, no. 4, pp. 452–466, 2016.

[8] E. Mykletun and G. Tsudik, "Aggregation queries in the database-as-a-service model," in *Proceedings of the IFIP Conference on Data and Applications Security and Privacy*, vol. 4127, pp. 89–103, Springer Berlin Heidelberg, 2006.

[9] H. Wang, Q. Jing, and R. Chen, "Distributed systems meet economics: pricing in the cloud," in *Proceedings of the USENIX Conference on Hot Topics in Cloud Computing*, 2010.

[10] C. Kilcioglu and J. M. Rao, "Competition on price and quality in cloud computing," in *Proceedings of the International Conference on World Wide Web*, pp. 1123–1132, April 2016.

[11] M. Al-Roomi, S. Al-Ebrahim, S. Buqrais et al., "Cloud computing pricing models: a survey," *International Journal of Grid & Distributed Computing*, vol. 6, no. 5, pp. 93–106, 2013.

[12] H. Xu and B. Li, "Dynamic cloud pricing for revenue maximization," *IEEE Transactions on Cloud Computing*, vol. 1, no. 2, pp. 158–171, 2013.

[13] S. Gu, Z. Li, C. Wu, and C. Huang, "An efficient auction mechanism for service chains in the NFV market," in *Proceedings of the IEEE INFOCOM 2016—IEEE Conference on Computer Communications*, pp. 1–9, San Francisco, Calif, USA, April 2016.

[14] W.-Y. Lin, G.-Y. Lin, and H.-Y. Wei, "Dynamic auction mechanism for cloud resource allocation," in *Proceedings of the 10th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid '10)*, pp. 591-592, May 2010.

[15] R. Zhou, Z. Li, C. Wu, and Z. Huang, "An Efficient Cloud Market Mechanism for Computing Jobs with Soft Deadlines," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 793–805, 2017.

[16] W. Shi, L. Zhang, C. Wu, Z. Li, and F. C. M. Lau, "An online auction framework for dynamic resource provisioning in cloud computing," in *Proceedings of the 2014 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '14)*, pp. 71–83, June 2014.

[17] L. Zheng, W. Tarneberg, and K. Maria, "Using a Predator-Prey Model to Explain Variations of Cloud Spot Price," *Journal of Intelligent & Fuzzy Systems: Applications in Engineering and Technology*, vol. 28, no. 6, pp. 2679–2689, 2017.

[18] Z. Zhang, Z. Li, and C. Wu, "Optimal posted prices for online cloud resource allocation," *ACM SIGMETRICS Performance Evaluation Review*, vol. 44, no. 1, pp. 60-60, 2017.

[19] N. Kline and R. T. Snodgrass, "Computing temporal aggregates," in *Proceedings of the 1995 IEEE 11th International Conference on Data Engineering*, pp. 222–231, March 1995.

[20] B. Moon, I. F. V. Lopez, and V. Immanuel, "Scalable algorithms for large temporal aggregation," in *Proceedings of the 2000 IEEE 16th International Conference on Data Engineering (ICDE'00)*, pp. 145–154, March 2000.

[21] B. Moon, I. F. V. Lopez, and V. Immanuel, "Efficient algorithms for large-scale temporal aggregation," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 3, pp. 744–759, 2003.

[22] J. Yang and J. Widom, "Incremental computation and maintenance of temporal aggregates," *The VLDB Journal*, vol. 12, no. 3, pp. 262–283, 2003.

[23] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," in *Proceedings of the 1970 ACM SIGMOD Workshop on Data Description, Access and Control (SIGFIDET '70)*, pp. 107–141, November 1970.

[24] G. Graefe and H. Kuno, "Modern B-tree techniques," in *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*, pp. 1370–1373, April 2011.

[25] P. Yao, H. Zhang, Y. Xue et al., "Segment-tree based cost aggregation for stereo matching with enhanced segmentation advantage," in *Proceedings of the 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '17)*, pp. 2027–2031, New Orleans, LA, USA, March 2017.