Scalable SaaS Indexing Algorithms with Automated Redundancy and Recovery Management

Wei-Tek Tsai¹, Guanqiu Qi¹, and Zhiqin Zhu²

¹(School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, AZ 85281, USA)
²(Automatic College, Chongqing University, Chongqing 400044, China)
Email: {wtsai, guanqiuq}@asu.edu, zhiqin@gmail.com

Abstract Software-as-a-Service (SaaS) is a new software delivery model with Multi-Tenancy Architecture (MTA). An SaaS system is often mission critical as it often supports a large number of tenants, and each tenant supports a large number of users. This paper proposes a scalable index management algorithm based on B+ tree but with automated redundancy and recovery management as the tree maintains two copies of data. The redundancy and recovery management is done at the SaaS level as data are duplicated with tenant information rather than at the PaaS level where data are duplicated in chunks. Using this approach, an SaaS system can scale out or in based on the dynamic workload. This paper also uses tenant similarity measures to cluster tenants in a multi-level scalability architecture where similar tenants can be grouped together for efficient processing. The scalability mechanism also includes an automated migration strategies to enhance the SaaS performance. The proposed scheme with automated recovery and scalability has been simulated, the results show that the proposed algorithm can scale well with increasing workloads.

Key words: software-as-a-service(SaaS); scalability; redundancy management; automated recovery; automated data migration


1 Introduction

Cloud computing plays an important role today, as a new computing infrastructure to enable rapid delivery of computing resources as a utility in a dynamic, scalable, and visualized manner. SaaS (Software-as-a-Service) as a part of cloud computing among PaaS (Platform-as-a-Service) and IaaS (Infrastructure-as-a-Service). SaaS is designed for delivery as Internet-based services. One single code is designed to run for different tenants. SaaS provides frequent upgrades to minimize customer disruption...
and enhance satisfaction. For maintenance, fixing one problem for one tenant also fixes it for all other tenants.

Important SaaS issues include SaaS architecture, customization, multi-tenant architecture (MTA), and scalability. Scalability is an important criterion to evaluate of SaaS as it indicates the ability to handle growing amounts of work while maintaining performance\textsuperscript{[16]}. For instance, an SaaS system can increase its workload while maintaining the delay experienced by users. Theoretically, a system can increase its throughput within a limit by increasing the delay. This has been shown in queuing theory such as throughput/delay for various queuing systems. In traditional software systems, scalability can be addressed using scale-up (or scale horizontally) and scale-out (or scale vertically) methods that involve either using a more powerful processor or using multiple processors to handle the additional load.

Existing SaaS scalability solutions include multi-level scalability architecture, tenant-awareness, automated data migration, workload support, software architecture, and database access. Amazon DynamoDB\textsuperscript{[1]} addresses the scalability issues in database. Amazon Simple Storage Service adds nodes to the system to handle additional load to address scalability. Windows Azure Platform Services uses multitenant storage machines to deal with the data storage. The multitenant machines replicate data to ensure the system works all the time, even if one replica fails. The entity PartitionKey connects to different partitions in different machines.

Most SaaS scalability mechanism such as those used in Salesforce.com and Corentech.com has a two-levels architecture as shown in Fig. 1. While each SaaS operation may differ depending on its special design, a typical SaaS scalability mechanism can be described as follows. At the top level, tenants are distributed to different PoDs (Portal on Demand). A tenant is designated to a specific PoD and all the requests from this tenant are directed to the same PoD afterward. At the 2nd level, the system uses stateless server to serve user requests. Consecutive requests from the same user can be directed to different servers in a PoD, according to the workloads of these servers. First, closely related tenants can be clustered for optimal performance, and the connectivity between tenants can be used to cluster tenants in the two-level scalability architecture. For example, a cluster of bank tenants will share more commonality than a cluster of random tenants. Furthermore, the two-level scalability mechanisms can be extended to multi-level. For example, PoDs can be assigned to different sites, and each site can host multiple PoDs, and thus automated migration and tenant-aware also support scalability. Tenant data in a PoD are stored in data tables. The data tables need to be scanned to read and remove the records belonging to a specific tenant. Each record of the tenant data is uniquely identified by the tenantID field in the record. Each tenant request is associated with a tenantID so that the system knows where to store the tenant data.

A common approach for scalable SaaS architecture is the two-level mechanisms where tenants are allocated to different nodes (or PoDs), and within each node, multiple stateless processes handle requests by sharing a common database within the node. This approach is efficient and effective as tenants in different nodes will not interfere each other, and even tenants within the same node can scale up as each processing services is stateless. The importance of redundancy and recovery management in PaaS has been emphasized in various PaaS systems such as GAE
where each write is done at least three times into different chunks to ensure reliability. Some SaaS systems depend totally on their hosting PaaS for their R&R management. This is good if the SaaS developers are the same as the PaaS developers, or the PaaS system is an open-source system. Unfortunately, many PaaS systems today are not as open and many SaaS developers need to use third-party PaaS systems to host their applications. For example, Salesforce.com has opened up its hosting system for other people to use. It is desirable that an SaaS system has its own R&R mechanisms so that it does not totally depend on the PaaS R&R mechanisms. A PaaS-independent approach has been proposed where a generic SaaS system can be developed, and then customize the SaaS for a specific PaaS system to be deployed and executed in that PaaS. Updating the changed index is crucial.

This paper proposes scalable Index algorithms with automated redundancy and recovery. The index mechanism is based on B+ tree. Data can be easily added, removed, searched using the proposed algorithms. A B+ tree has the following attributes[2,9]:

- The insertion, deletion, and retrieval is efficient;
- The tree can be dynamically scaled according to the number of data in the tree, either upward or downward; and
- The tree is balanced as at the root or any intermediate root, the difference between the numbers of children on the left subtree and the number of the right subtree is bounded in a range.

Instead of storing just one copy of data, this paper proposes to modify the B+ tree so that two copies of data are stored, and they can be processed in parallel, including extracting, loading, and saving. As two copies are available, one copy will be labeled as primary, and the other backup, and the system will automatically save data from the primary to the backup, and restore the primary from the backup in case of failures. Furthermore, automated data migration algorithms have been proposed to move data around to improve the SaaS system performance in a cloud environment.

Data mining techniques will be used to find the best threshold for scalability. In summary, this paper makes the following contributions:
It proposes a \( B^+ \)-tree-based structure method for data storage, data redundancy and recovery management;

- It utilizes data mining techniques to identify the best scalability threshold;
- It uses the redundant \( B^+ \) tree to address automated data migration;
- It proposes a clustering approach based on tenant similarity measures to group related tenant together to improve performance; and
- It evaluates the proposed scheme by simulation and demonstrates the proposed solution is effective.

This paper is structured as follows: Section 2 discusses the related work; Section 3 discusses the scalable SaaS index algorithm for data index storage; Section 4 explores redundancy and recovery mechanisms for SaaS; Section 5 proposes the automated data migration methods and strategies using data mining and time series analysis techniques; Section 6 evaluates the proposed methods by simulation; and Section 7 concludes this paper.

## 2 Related Work

### 2.1 SaaS scalability

Scalability has been addressed in many areas including parallel computing, distributed computing, databases, and Service-Oriented Architecture (SOA). For the design of scalable systems, design principles include divide-and-conquer, asynchrony, encapsulation, concurrency and parsimony\(^8\): Divide-and-conquer means the system should be divided into small subsystems to execute; Asynchrony means the workloads can be assigned to the available resources to execute; Encapsulation means that the components in the system architecture are encapsulated; Concurrency means exploiting the optimal configuration among different parts in the system; Parsimony considers the scalable design from the economic aspect.

Important SaaS scalability factors include levels of scalability mechanisms, automated migration, tenant awareness, workload support, recovery and fault-tolerance, software architecture and database access\(^{11,13}\). Scalability often involves in two-level scalability structure as used in Salesforce.com and Corentech.com. Each user request will be routed first to the designated PoD, and then to a specific server within the PoD. The index table is used to increase the data access efficiency. The distributed Oracle DB handles the recovery and fault-tolerance issues. Yahoo! PNUTS\(^9\) divides data into different "region", where user data storage unit sets to store data partitions. Duplicate data copies are stored in different regions. Once one region’s data fails, the duplicated data in other regions will be used to continue processing. Amazon DynamoDB uses a ring scalability architecture to automatically manage data. The data contents are stored in nodes that compose a ring. The ring structure increases the efficiency for recovery. The failure of a node affects only a few neighboring nodes on the ring. Various tenant data partitioning methods can be to minimize data operation cost for scalability\(^{18,4,17,10}\). SaaS performance and scalability evaluations have been performed\(^4\).
\(B^+\) tree and T-tree is used in the main memory database index structure, that allows concurrent accesses of multiple users\[6\]. A two-tier SaaS scaling and scheduling architecture to save resources at service and application levels and the optimized cluster-based resource allocation algorithm is proposed in a clustered cloud environment\[14\]. A multi-tenant oriented monitoring, detecting and scheduling architecture based on SLA is proposed to solve system resources allocation issues\[9\]. Most existing works of SaaS scalability do not discuss the new factors in SaaS, such as cost and resource utilization. A scalability model to measure the performance of SaaS application scalability in a cloud based on the data mining and statistics techniques\[12\].

The two-level scalability as shown in Fig. 1 often has a Sequential PoD addition algorithm. Specifically, when a new PoD is needed, the new PoD will be added to the end of the list of PoDs, and any new tenant applications will be stored in the new PoD. When the new PoD is filled, another new PoD will be added to the list as shown in Fig. 2. Each PoD also shares common services such as network services, backup services, monitoring services.

2.2 Redundancy and recovery mechanisms

Software and data reliability is critical for SaaS applications, and thus most SaaS systems have multiple redundancy and recovery (R&R) mechanisms. For example, Salesforce.com has a multi-level \(R&R\) mechanisms at the following levels.

- **Data-centers:** It has multiple data centers interconnected by high-speed networks capable of backing up each other in case of a failure of one center.

- **Network-level:** Multiple network carriers with redundant routers, and fail-over configured firewalls, redundant hubs and switches at VLANs.

- **SaaS-Level:** Multiple load balancers at the top-level scalability structure as well as within each PoD. Top-level load balancers route tenant requests to different PoDs, and local load balancers route tenant requests to different
processors in the same PoD. Tenant data are also extensively cached for performance.

- **Database-level:** It uses Oracle RAC EE running on 4 way clustered nodes with excess capacity to carry out the load when a node fails.

- **Storage-level:** It has multiple paths to ensure reliable connecting among four DBMS servers, and alternative paths to storage directors, and the storage systems also have built-in redundancy.

Due to the unique requirements for SaaS systems, many organizations have developed various system to support SaaS operations. For example, RainStor[7] uses a massively parallel processing (MPP) system to process large number of data produced during operation as shown in Fig.3. While data stored in RainStor are logically shared, data are actually partitioned into multiple servers. Furthermore, the workloads are automatically balanced among servers with built-in redundancy to ensure reliability, and the system can scale up when the workload increases.

![RainStor’s architecture for big data](image)

RainStor uses the Hadoop File System (HDFS) manage data. All workloads are shared across multiple serves. Data are landed into separate staging areas specific to each node in the HDFS. Partitions are built in parallel by each node during loading. Service manager analyzes data, de-duplicates and compresses data into multiple partitions. In the data analysis process, data of the same type of data will be put into the same cluster, and then duplicated data will be deleted. When a query is requested, the query is executed in parallel by multiple nodes leveraging the distributed file system.
3 Indexing Algorithm

This paper proposes a new method called Index Ring Tree (IRTree) for storing indices with \(R\&R\) mechanisms. The IRTree originated from \(B^+\) tree that provided efficient, balanced, and scalable operations. However, the IRTree is different from \(B^+\) tree in two key aspects:

1. IRTree has a wrap-round structure for managing the search, insertion and deletion of indices.

2. Each index node has a left and a right child node. The data stored in the right part of a node will have duplicate copy in the left part of the next node. The right part of the last node duplicates the left part of the first node. In this way, each index has at least one duplicate.

Like \(B^+\) tree, it is easy to do insertion, deletion, and search operations in the IRTree.

The IRTree structure is an M-ary tree (\(M > 3\)) like a \(B^+\) tree structure where data are stored at the leaves. At the same depth each node has \(\lceil \frac{L}{2} \rceil\) to \(L\) data. Internal nodes contain searching keys. Each node has \(\lceil \frac{M}{2} \rceil\) to \(M\) children. Each node has \(\lceil \frac{M}{2} - 1 \rceil\) to \((M-1)\) searching keys. For the root, it can be a single leaf, or has 2 to \(M\) children. Each node stores a pointer to its left child, a pointer to its right child. The basic functions, like node insertion, deletion and search, can be implemented in the IRTree. Figure 4 shows the processes of node insertion and deletion.

- **Search** If the IRTree is null, the key does not exist in the tree. If the searching value equals the node, the searching is successful. If the value is less than the node key, search in the left subtree. Else search in the right subtree. Repeating this process until the key is found. The IRTree uses a binary search tree for the searching operations. So the time complexity of IRTree search is \(O(\log N)\),

![Figure 4. Index maintenance mechanism](image-url)
where \( N \) is the size of the list. The advantages of IRTree is efficient in insertion and deletion when the tree is balanced. But it needs some time to maintain the tree balanced, when insertion and deletion operations happen.

**Example 1:** Search 13 in the data storage. The system returns two data storage positions. One is machine 1 position 13, the other one is machine 2 position 3.

- **Insertion** Before insertion, it needs to use the search algorithm to find the insertion position. If the size of insertion position is same or large than insertion data size, it can be inserted directly. Otherwise it needs to create a new chunk to store the insertion data and its copy. The same data will be added into the corresponding two neighboring nodes, the right part of one node and the left part of the other node. The insertion process may need to split the existing chunk for adding new data. The splitting position may involve leaf, internal or root node, that depends on the insertion position. The split left and right part may have different values.

  - When splitting one leaf into two new leaves \( L_{left} \) and \( L_{right} \), \( L_{left} \) has the \( \lfloor \frac{M+1}{2} \rfloor \) smallest keys and \( L_{right} \) has the remaining \( \lceil \frac{M+1}{2} \rceil \) keys.
  
  - When splitting one internal node into two new internal nodes \( N_{left} \) and \( N_{right} \), \( N_{left} \) has the smallest \( \lceil \frac{M}{2} \rceil - 1 \) keys and \( N_{right} \) has the largest \( \lfloor \frac{M}{2} \rfloor \) keys. The \( \lceil \frac{M}{2} \rceil \)th key is not in either node, because \( (\lceil \frac{M}{2} \rceil - 1) + (\lfloor \frac{M}{2} \rfloor) = \lfloor \frac{M}{2} \rfloor + \lceil \frac{M}{2} \rceil - 1 = M-1 \).
  
  - When splitting the root node, \( J \), the parent of \( N_{left} \) and \( N_{right} \), is set to be the root of the tree, because the original root is destroyed by splitting.

---

**Algorithm 1** Index ring search

**Require:**
- Index value

**Ensure:**
- Index position

Start from the root

if an internal node is reached then

  Search index value among keys in that node

  if index value < smallest key then
    follow the leftmost child pointer down

  if index value \( \geq \) largest key then
    follow the rightmost child pointer down

  if \( K_i \leq \) index value < \( K_j \) then
    follow the child pointer between \( K_i \) and \( K_j \)

if a leaf is reached then

  Search index value among the keys stored in that leaf

  if index value found then
    Return the corresponding record

else

  Report not found

---

**Example 2:** Insert two numbers in the data storage, which are 12.5 and 17.5. For 12.5, the system returns machine 1 position 13 and machine \( n+1 \) position 3.
as the data storage positions in Fig. 4 Insertion(a). For 17.5, the system returns
machine n+1 position 17 and machine 2 position 7 as the data storage positions
in Fig. 4 Insertion(b).

- **Deletion** No matter deleting the whole chunk of one node or some parts of one
node, all the same copies will be deleted in different nodes correspondingly.

**Example 3:** Delete 14 in the data storage. The system returns two data
storage positions of the deleted number, which are machine 1 position 14 and
machine 2 position 4 in Fig. 4 Deletion(a). Delete numbers from 10 to 30 in
the data storage. The system returns all data storage positions involving
deleted numbers, that are machine 1 from position 10 to 20, machine 2 from
position 0 to 20, machine 3 from position 0 to 10 in Fig. 4 Deletion(b).

### 3.1 Relaxing constraints to minimize merges and splits

For the efficient SaaS management, IRTree can minimize the splitting and
merging costs by relaxing the constraints of data size, specially, it allows less data in
a node without merge with another node, and more data in a node before splitting a
given node. Specifically, instead of splitting a node when the size is over \( L \), it will
split when the size is over \( L + K \) where \( K \) is a constant to be selected, and it will
merge only if the size is less than \( \frac{(L-K)}{2} \). The larger the \( K \) is, the IRTree will be less
balanced. However, the chance of splitting will be smaller, as then next operation
may be a delete rather than a new insertion. Furthermore, suppose a system is
changing the size frequently around the size of \( L \), then one additional item will split
the current node, but deletion of one item of the splitted node will cause the just
splitted node to merge, and each split and merge can be costly, this is called flip-flop
effect. By relaxing the constraints, the chance of splitting and merge will be
reduced.

### 3.2 Tenant allocation by clustering similar tenants

One tenant optimization is to assign similar tenants into same clusters. As similar
tenants share components, and thus these components are likely to stay in the cache,
rather than stored in the SaaS database. As similar tenant applications will be called
often, most components will stay in the cache, and thus the system is likely to have
better performance. One solution to assign tenant IDs of similar tenants close to each
other. As the tenant IDs are close to each other, the chance of assigning them into
the same node of the IRTree increases. Similarity measurements between two tenants
can be defined as follows:

\[
\text{Similarity}(A, B) = \frac{|\text{shared component}(A, B)|}{|\text{union}(A, B)|}
\]

Using this definition, if \( A = B \), then \( \text{similarity}(A, B) = 1 \), if \( A \) is completely
different from \( B \), \( \text{similarity}(A, B) = 0 \). Given a list of initial tenants, saying from
0 to \( K \), one can first measure their similarities, and use the following algorithm to
assign tenant IDs to be used in IRTree insertion.

1. Set \( \text{rank} = 0; \)
2. Pick one with the highest similarity measuring with another tenant and assign the tenant with $ID = rank$, and $rank = rank + 1$;

3. Pick the next tenant with highest similarity measures with just ranked tenant, set its $ID = rank$, $rank = rank + 1$. In case of a tie, randomly pick one with same similarity measures;

4. Repeat step 3 until all the tenants are assigned with their IDs.

With this arrangement, inserting tenant information into the IRTree is equivalent to running a clustering algorithm as closely related tenants will be clustered together.

3.3 Usage-Based clustering

In the previous scheme, two tenants will be considered as similar if they share many common components. A further optimization can be done if usage information is considered. For example, after running tenant applications extensively, it has discovered that while tenant A and tenant B share one component only, but that component is used 100 times for each user request by both A and B. While tenant B and tenant C share three components, but each shared component is used at most once for each user request. Thus, tenant A should have priority to be clustered with tenant B. The following weighted similarity measures can be used:

$$W_{\text{Similarity}}(A, B) = \frac{\sum \text{usage}(\text{shared components}(A, B))}{\sum \text{usage}(\text{all components}(A, B))}$$

This new definition considers both usage frequency and shared components.

3.4 Workload-Based IRTree

Another method is to build a balanced IRTree is to have a balanced workload. In the previous schemes, the workload for each tenant is assumed to the same, and thus to have a balanced IRTree is to assign more or less the same number of tenants to each part of the tree. But a workload-based IRTree will balance the tree based on the workload of tenants assigned to different parts of the tree. Each tenant can have a weight representing the workload of the tenant.

4 Redundancy and Recovery Mechanisms

The IRTree provides two copies of data as shown in Fig. 5. Two copies can be stored in different servers to provide increased reliability. Only if both copies fail at the same time, the data become unavailable. If the SaaS runs on top of a PaaS such as GAE (Google App Engine), any data write will be written into three different chunks for further reliability, and any data lost in a chunk can be recovered automatically by the PaaS. However, this R&R mechanism may not be directly controlled by the SaaS but by the PaaS, and the mechanism is done at the chunk level.

For this distributed storage structure, it needs to maintain the consistency between two copies. One solution is to keep them synchronized from time to time. If a system crashes before the backup synchronization is done, the system will lose those updates since the previous synchronization. Another approach is to synchronize them all the time, and this can be done by sending changes to both
copies simultaneously and ask both copies to update, and uses a real-time monitoring to watch the behavior of these two copies. If two copies become inconsistent, the monitor will lock both copies to start a resolution mechanism. For example, by having a third party to execute the saved request to determine which of two copies has inconsistent data. The monitor will unlock once the resolution is completed.

4.1 Mirrored trees

Instead of having duplicate copies using IRTree, one can build one or more mirrored trees as redundancy. In this case, it also needs to synchronize these trees. One approach is to perform redundant computation at each site, and the other is to update the mirrored site from the primary site from time to time. The first approach involves duplicated computation, and the second approach may have inconsistent states from time to time until synchronizations.

4.2 IRTree = primary B+ tree plus backup B+ tree

If one labels each node either as a primary or backup, one can see that IRTree is essentially a combination of two trees, primary tree and backup tree, and each tree is a B+ tree. This can be illustrated in Fig. 5.

When the IRTree is being updated, both the primary B+ tree and the backup B+ tree are being updated at the same time, thus the indices are updated at the same time while writing backup data may be delayed. Like the mirrored tree, data synchronization for IRTree can have two choices:

1. The backup will receive the same request as the primary for computation, and thus both the primary and backup perform the same computation but may be at different processors, clusters, or sites;

2. The backup will receive delayed data depending on the update periods. These two choices have the same trade-offs as the choices for the mirrored trees.
4.3 Data allocation and recovery

Potentially, the IRTree mechanism can be used in multiple levels in the 2-level scalability structure.

4.3.1 IRTree at the top-level

IRTree can be used to allocate tenants into different clusters as shown in Fig. 6. Essentially, the IRTree will allocate tenants so that

1. Each tenant will be allocated to two different PoDs for reliability.
2. Each PoD will have a balanced load with respect to the number of tenants (Section 3.3) or the workload of tenants (Section 3.4).
3. Similar tenants can be clustered together (Section 3.2) in the same PoD.
4. Each PoD can perform execution on its own.

Comparing to the sequential PoD addition, PoDs organized by the IRTree can have balanced workloads with automated redundancy management. Table 1 compares this approach with the sequential addition algorithm.

4.3.2 IRTree used in a PoD

Like the RainStor, IRTree can be used to support the virtual file system with built-in redundancy with primary and backup trees stored in different partitions or processors for ensuring reliability as shown in Fig. 7. With additional mechanisms, the system can provide following features:

1. Annotation with tenant information: Each unit of data, depending on the tenant-awareness design such as at the record, table, or chunk level, can be annotated with tenant information. Once a unit is annotated, i.e., self-describing, each unit can be moved asynchronously for scalability migration or automated recovery.
2. Built-in redundancy: The IRTree supports built-in redundancy so that at least two copies will be available and these two copies can be stored in different
Table 1  Comparison between sequential addition and IRTree

<table>
<thead>
<tr>
<th>Data Allocation</th>
<th>Sequential Addition</th>
<th>IRTree Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Allocation</td>
<td>Data are added into PoD sequentially. When the existing PoDs are full, new PoD will be added.</td>
<td>Data are kept sorted in IRTree and allocated in different PoDs.</td>
</tr>
<tr>
<td>Different Operations</td>
<td>Searches, sequential access, insertions, and deletion are allowed. For deletion, the blank spaces will be kept. Stored data may need to be rearranged to occupy contiguous storage locations.</td>
<td>Searches, sequential access, insertions, and deletions are allowed in logarithmic time. For deletion, IRTree readjust the tree structure.</td>
</tr>
<tr>
<td>Load Balancing</td>
<td>Existing PoDs may be heavily loaded while new PoD is lightly loaded.</td>
<td>Balanced workloads are distributed.</td>
</tr>
<tr>
<td>Redundancy and Recovery</td>
<td>Need additional operations for backup copies and recovery.</td>
<td>Automatically keep backup copies and recover failures.</td>
</tr>
</tbody>
</table>

Figure 7. IRTree used in a PoD

...databases or servers. If the IRTree is also used at the top-level, then at least four copies will be made. The top-level IRTree will have two copies at the high-level, and each PoD will contain two copies, so a total of four copies will be made.

3. **Balanced workloads**: The IRTree automatically balances the workload for servers within the PoD.

4. **Scalability**: New servers can be added once the IRTree has determined that the workload of the PoD is high enough, and new servers can be added in a B-tree manner, i.e., all the workload of all the servers will be balanced.

The IRTree application in a PoD provides similar features as the RainStor system including automated workload balancing, built-in redundancy, scalability, and tenant-
awareness as shown in Table 2.

Table 2  IRTree implementation in RainStor architecture

<table>
<thead>
<tr>
<th></th>
<th>RainStor Architecture</th>
<th>IRTree</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Load Balancing</strong></td>
<td>It atomically balances the workload across multiple servers with built-in redundancy.</td>
<td>Balanced workloads of IRTree are assigned to different servers.</td>
</tr>
<tr>
<td><strong>Data Allocation</strong></td>
<td>It de-duplicates and compresses data into multiple partitions.</td>
<td>In virtual file system, primary tree and backup tree will be assigned to different partitions. IRTree can be stored in four locations at minimum, two at the top level and the other two used in a PoD.</td>
</tr>
<tr>
<td><strong>Scale up</strong></td>
<td>It scales up with increasing loads with automated expansion.</td>
<td>IRTree scales up with balanced workload automatically. The system is constantly load balancing and sharing tasks across nodes based on the workloads.</td>
</tr>
<tr>
<td><strong>Tenant Awareness</strong></td>
<td>Self-describing data for self-healing and recovery.</td>
<td>Each unit of data (record, table or chunk depending the design of tenant-awareness) can be annotated with tenant information (self-describing) for easy tenant migration and recovery.</td>
</tr>
</tbody>
</table>

4.3.3  Failure recovery

One issue in using the IRTree is the location to place duplicated data, as the data can be placed in the same processor, a different processor within the same cluster, a different processor in a different cluster, or even a different processor of a different site. If data are lost, they can be recovered from the same processor; if the processor fails, data may be recovered from the cluster. If the cluster fails, backup data may come from a different cluster from the same site for recovery. If the cluster fails, it may recover data from a different cluster; if the site fail, data may be recovered from a different site. The recovery for different allocation schemes are discussed in Table 3. Note that even IRTree has one primary and one backup for each tenant, but potentially the backup can contain multiple entries with different synchronization periods. For example, the backup data in the same processor should be updated more frequently than the backup data at the different site.

4.4  In-Memory DB data management

For in-memory DB, data can have multiple levels ranking items, for example, as hot, medium, cold where hot data are used extensively, and cold data are rarely used recently. Hot items will stay in the memory as well as in the cache, and medium items stay in the memory, and cold items can be swapped out if out of memory space. Continuous updating on all the items follows the above rules. The backup data will also be stored in different speed chunks based on the ranking levels. The hot items are used frequently and the failure possibilities are higher than others. For hot items, the speed of recovery is more important than other items. So their backup data needs to stored in high speed chunks.
Wei-Tek Tsai, et al.: Scalable SaaS indexing algorithms with ...

Table 3 Recovery for different allocation scheme

<table>
<thead>
<tr>
<th>Allocation to</th>
<th>Failures in clusters</th>
<th>Failures in sites</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same cluster</td>
<td>OK, fast recovery</td>
<td>Failure</td>
</tr>
<tr>
<td>Same site but different clusters</td>
<td>OK, medium speed</td>
<td>Failure</td>
</tr>
<tr>
<td>Different sites</td>
<td>OK, but recovery can</td>
<td>Can recovery</td>
</tr>
<tr>
<td></td>
<td>be expensive</td>
<td>be expensive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>everything, but</td>
</tr>
<tr>
<td></td>
<td></td>
<td>expensive</td>
</tr>
</tbody>
</table>

The backup data of medium and cold items are stored in medium and low speed chunks respectively. Once the failures happen, it will be recovered by the backup data. It a complete PoD fails, tenants applications allocated on this PoD may be down until the whole PoD is recovered from the failure.

4.5 Sequential adding with load balance

In tradition, the sequential adding method is used to add new tenant to the current PoD until the PoD is full. Then the system will add a new PoD. This approach is easy to implement but does not address load balancing as the new PoD may have light load comparing to the existing PoDs at least at the beginning. Furthermore, the larger the size of each PoD, the more imbalance the new PoD is.

We propose one $N$-PoDs ($N$ is the natural number, $N \geq 1$) method for tenant storage. If $N$ old PoDs are fully used, the system will add one new PoD. The the system will move $\frac{1}{N+1}$ contents from each old PoD to the new PoD. The empty spaces of the $N$ old PoDs and new adding PoD ($\frac{1}{N+1}$ of PoD) can be used for the future tenants. Then the new tenants are distributed alternatively into all $N + 1$ PoDs. When all the $N + 1$ PoDs are full, the system will repeat the same process again. For example, suppose that at the beginning only one PoD, thus $N = 1$. When the PoD is full, the system adds a new PoD. Then $\frac{1}{2}$ contents of the first PoD will be move to the new PoD. So after moving, both two PoDs have $\frac{1}{2}$ contents and $\frac{1}{2}$ empty space of the capacity. Then new tenants will be added into both two PoDs alternatively. When both PoDs are full, one new PoD will be added. The first two PoDs will move $\frac{1}{3}$ of its own contents respectively to the new PoD. The moving process and other things are same as the first time. Each one of the three PoDs has $\frac{2}{3}$ content and $\frac{1}{3}$ empty space. The processes are illustrated in Fig. 8.

The number $N$ of existing PoDs means that one new PoD is connected to $N$ original fully used PoDs. If $N = 1$, it means that only one new PoD is connected to the original one. If $N = n$, it means that one new PoDs are connected to the $n$ original ones. When $N = 1$, the original PoD only needs to divide the tenant data into two parts. Then the system moves only one part to the new added PoD. But when $N = n$, $\frac{1}{n+1}$ contents of each PoD will be moved to the new adding PoD respectively. The bigger $N$ is, the more complex process is. When $N$ gets bigger, the moving data increases, which starts from $\frac{1}{2}$, $\frac{2}{3}$, $\frac{3}{4}$, ..., to $\frac{n}{n+1}$. When $n$ closes to unlimited value, $\frac{n}{n+1}$ almost equals to one. Almost one PoD contents need to be moved to the new PoD. The data migration process involves more PoDs following the increase of $n$. Since the cost of data migration is high, it is better to decrease the times of data migration. So comparing other numbers (greater than one), $N = 1$ is
Algorithm 2 Revised Sequential Adding Algorithm

Require:
Fully used \( N \) PoDs, one adding PoD

Ensure:
All PoDs containing tenant contents

1: if \( N \) PoDs are full then
2:     if \( N \geq 1 \) then
3:         Move \( \frac{1}{N+1} \) contents of existing PoDs to the new adding PoD
4:     end if
5:     Return \( N + 1 \) PoDs
6: else
7:     Return Null
8: end if

the good number to use.

5 Automated Data Migration

Automated data migration is a key factor of SaaS scalability\(^{[11]}\), and it affects the SaaS performance. For better performance, many issues need to be considered: online migration or offline migration, selection of data for migration, data migration granularity, amount of data for migration, migration strategies based on usage patterns, migration mode (demons, periodic, autonomous, and on-demand)\(^{[11]}\). For example, a multi-level data migration strategy was proposed in Ref.\(^{[11]}\). The following paragraph will discuss when is the best time to do data migration and how to get the best time.

5.1 Threshold for the system efficiency

One important consideration is the threshold for the waiting time to trigger data
Wei-Tek Tsai, et al.: Scalable SaaS indexing algorithms with ...

Table 4 Different PoD management for different tenant data

<table>
<thead>
<tr>
<th>Different Tenant Data</th>
<th>Own Characters</th>
<th>Data Migration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each tenant has a DB with its own schema.</td>
<td>Good for tenant isolation, significant schema design effort.</td>
<td>Due to the own DB, the tenant data can be moved to other PoD.</td>
</tr>
<tr>
<td>Each tenant has a DB, but all tenants use a collection reasonable effort in schema of one application domain.</td>
<td>Although in different domain the schema may change, one tenant has its own DB. The tenant data can also be moved to other PoD.</td>
<td></td>
</tr>
<tr>
<td>Each tenant has a DB but all share the same schema.</td>
<td>Good for tenant isolation, less effort in schema design.</td>
<td>Since each tenant has its individual DB, no matter sharing the same schema, the data can be moved.</td>
</tr>
<tr>
<td>Shared DB, but each tenant has its schema.</td>
<td>Reasonable solution for tenant isolation, significant schema design effort.</td>
<td>Due to the shared DB, the tenant data is not better to move other PoD, which may affect the system efficiency.</td>
</tr>
<tr>
<td>Shared DB and schemas.</td>
<td>Significant effort in isolation.</td>
<td>The same reason as above one, no matter sharing the same schemas or not.</td>
</tr>
<tr>
<td>Each extension is a table.</td>
<td>Lots of join operations may be needed.</td>
<td>It depends on the table contents. Sharing DB contents are not good to move. Others can be moved.</td>
</tr>
<tr>
<td>Sparse columns for tenant information.</td>
<td></td>
<td>The contents in sparse tables can be clustered and moved to different PoDs.</td>
</tr>
<tr>
<td>Hybrid solutions: for example, the SaaS has a hybrid solutions, one key is to collection of schema, critical reduce the number of schemas tenants have their own DB, needed, so to reuse database but some tenants shares the software. same DB with the same schema, and different clusters have different database approaches for their own tenants.</td>
<td>Critical tenants can be moved to other PoDs, since they must be moved, the moving operations should be based on the clusters and database approaches.</td>
<td></td>
</tr>
</tbody>
</table>

migration. The average waiting time time of a user request can be used to set the threshold as the average waiting time is inverse proportional to the user satisfaction. When the average waiting time is longer than the historical mean by say 20%, the system can add additional resources automatically, but may mean existing data in the allocated resource may need to move. From history data, one can predict the future usage. Linear regression, curve fitting and Support Vector Machine (SVM) can be used in linear, curve and hyperplane data type respectively. For instance, based on the linear assumption, the linear regression model analyzes the relationship between variable \( y_i \) and vectors of \( x_i \) in the given data set \( \{y_i, x_{i1}, \ldots, x_{ip}\}_{i=1}^n \). The linear model is like \( y_i = \beta_1 x_{i1} + \ldots + \beta_p x_{ip} + \varepsilon_i = x'_i \beta + \varepsilon_i, i = 1, \ldots, n \), where \( x'_i \beta \) is the inner product of \( x_i \) and \( \beta \). The least squares approach is used to get the fitted linear regression model.
5.2 On-Demand pre-fetching method

It proposes on-demand pre-fetching method to predict the future usages and guides the data migration following the predictions. The main steps of on-demand and pre-fetching method:

- **Analyze the historical data** The expanded time-series information can be got from the historical data sets. There are different types of data, such linear trend, non-linear trend, seasonality problem. By comparing historical data, the upward and downward performance trends can be spotted.

- **Use the analysis results to predict the future usage** It uses statistical methods to estimate the usages at certain specific future times. The relationships among different factors are also considered during the predictive analysis.

- **Build the scheduling model based on the prediction** It mainly involves assigning workloads to different services according to the prediction results. It will be discussed in next subsection.

Here it uses time series models to predict the future usages of computing resources. The models measure a sequence of data points at successive instants spaced at uniform time intervals. The meaningful statistics will be extracted from the time series data analysis. Based on the meaningful results, the model will forecast the future usages. For the time series data, it mainly focuses on the next five features: trends, seasonality, somehow influential data points, a variance that changes because of past observations, non-linearity. Different model is used for different feature data. For the linear trend, it uses $y_t = \alpha + \delta t + u_t$ to model, where $t = 1, 2, ... n$. The growth rate $= \frac{y_t - y_{t-1}}{y_{t-1}} \approx \frac{w_t - w_{t-1}}{w_{t-1}}$.

Here the proposed data migration strategy is designed to use two scaling methods (scaling up and scaling out) to solve the scalability issues. The proposed strategy adjusts the usages of different type data storage following the requests. It means that it not only scales up/out, but also scale down/in based on the usages. The system should classify the data into different type based on the usages. The cold data can be moved to slow speed data storage, and median and hot data can also be moved to corresponding data storage. The data usages may change in different time period. For example, in April of every year, the tax report service will receive a large number of service requests due to tax seasons. The data of tax information is hot during that time. But after that time, maybe the same data will be only accessed for several times. The hot data and cold data can change. So for the data migration strategy, it should reduce the migration times and the amount of data to be migrated as much as possible. At the same time it should also increase the system efficiency, involving computation and recovery. If the data migration costs are higher than computation costs, the data will not migrate.

According to the usage patterns, the unused data will be detected and moved into a backend storage automatically in the SaaS infrastructure. For the large size data, incremental migration is a good way to reduce the workloads of data migration than complete migration. All functions for data migration are controlled by the SaaS
controller. During the migration, the transmitted data will be marked and any access to the data will be blocked until the migration is completed. The different data migration tasks have different priorities, such as data recovery and scalable computation have higher priorities than others. Local migration should be assigned with a higher priority than remote migration. Migration should be done when the system is lightly loaded rather than heavily loaded. Data that results in bottleneck in the application should be migrated first.

6 Evaluation

This section reports the simulation results to evaluate the proposed methods, including the scalability experiment and migration experiment. The experiments were carried out on a desktop computer with 2.40GHz Intel Core2 Duo CPU and 6GB RAM.

6.1 Scalability experimentation

In this experiment, the system randomly generates $2^{10}$ numbers and builds $B^+$ tree for the generated numbers first. Each node in $B^+$ tree is assigned enough space. Insertion, deletion, search, and upgrade are considered in the experiment. For the new node, system will search the appropriate position in the tree. The ordered nodes are easy for doing different operations. After that the system will insert $2^{10}$, $2^{20}$, $2^{30}$, $2^{40}$, $2^{50}$ numbers respectively into the $B^+$ tree to see its performance. For the same number insertion, it will repeat ten times to get the average number. The original $B^+$ tree is used as the control group for comparison purpose. The same experiment is then run on the IRTree.

The experiment results are shown in Fig. 9. For data insertion, the traditional $B^+$ tree is better than IRTree. $B^+$ tree does not handle redundancy, but IRTree does. The scalability recovery experiment results are shown in Table 5. For the data recovery, the traditional $B^+$ tree completely fails as it has no recovery. IRTree can recovery the lost information from its backup data.

![Figure 9. Scalability experiment results](image-url)
Table 5  Scalability recovery experiment results

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Min Computation Time (s)</th>
<th>Max Computation Time (s)</th>
<th>Ave Computation Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{10}$</td>
<td>1.41</td>
<td>1.55</td>
<td>1.47</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>3.65</td>
<td>3.74</td>
<td>3.69</td>
</tr>
<tr>
<td>$2^{30}$</td>
<td>8.47</td>
<td>8.57</td>
<td>8.52</td>
</tr>
<tr>
<td>$2^{40}$</td>
<td>16.76</td>
<td>16.91</td>
<td>16.83</td>
</tr>
<tr>
<td>$2^{50}$</td>
<td>30.55</td>
<td>30.76</td>
<td>30.65</td>
</tr>
</tbody>
</table>

6.2  Migration experimentation

This experiment will simulate the data migration based on the proposed method. Here it supposes all the data are stored in the low speed data storage at the beginning. During running the program, the data will be migrated to the higher speed data storage following the proposed methods. In this experiment high speed data storage is three times faster than intermediate speed data storage and the intermediate speed data storage is two times faster than low speed data storage. Here the threshold of waiting time is 5 seconds for all type data storage. The system also considers the data migration cost. If the data migration cost is higher than waiting cost, the system will not do the data migration. The same program without data migration design is used to do the comparison, in which all data are stored in low speed data storage. The experiment results are shown in Table 6. The same program with data migration design is better than the one without data migration design.

Table 6  Migration experiment results

<table>
<thead>
<tr>
<th>Workload</th>
<th>Throughput with migration (s)</th>
<th>Throughput without migration (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2GB</td>
<td>68.53</td>
<td>101.31</td>
</tr>
<tr>
<td>4GB</td>
<td>136.23</td>
<td>205.41</td>
</tr>
<tr>
<td>6GB</td>
<td>199.47</td>
<td>310.26</td>
</tr>
<tr>
<td>8GB</td>
<td>268.16</td>
<td>414.78</td>
</tr>
<tr>
<td>10GB</td>
<td>330.82</td>
<td>521.63</td>
</tr>
</tbody>
</table>

6.3  Sequential adding with load balance experiment

This experiment uses the proposed sequential adding algorithm to simulate PoD management with load balance. The results are shown in Fig. 10. Each PoD contains 100,000 data. When the old PoD is full, the new PoD will be added. The data have two parts, one is letter part, the other is number part. The range of letter part is from A to Z. The number part is from 1 to 999. The data are randomly generated. The data can be clustered based on the letter. The horizontal coordinator shows PoD numbers and the vertical coordinator shows the adding times in milliseconds. The blue line shows results of sequential adding without pre-adjustment. It means that the data are directly added to each PoD. Then the system will do load balance in each PoD. The data with large size will be kept in the same PoD as much as possible. The data will small size will be moved to other PoD. The red lines shows results of sequential adding with pre-adjustment. It means that the data are clustered before adding to PoD. The new data will be added into PoD based on the existing data in PoD. Based on the results, the pre-adjustment cannot increase the sequential
adding efficiency. But sequential adding with load balance is a good way for PoD management.

7 Evaluation

This paper proposes an SaaS scalability algorithm with automated redundancy management to solve the increasing SaaS scalability issues. The proposed two-level scalability architecture handles tenants and their requests at different level. Furthermore, the two-level mechanism can be extended to multi-level. This paper proposes different solutions to handle SaaS scalability issues involving redundancy and recovery, and data migration. As future work, we will further investigate the SaaS scalability issues and try to propose more intelligent and dynamic solutions.

References


