Int J Software Informatics, Vol.4, No.4, December 2010, pp. 437–471 E-mail: ijsi@iscas.ac.cn
International Journal of Software and Informatics, ISSN 1673-7288 http://www.ijsi.org
 Tel: +86-10-62661040

Data Partitioning and Redundancy Management for Robust Multi-Tenancy SaaS

Wei-Tek Tsai[1,2], Yu Huang[1], Qihong Shao[1], and Xiaoying Bai[2]

[1] (School of Computing, Informatics, and Decision Systems Engineering
Arizona State University, Tempe, AZ, USA)

[2] (Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

Abstract Software-as-as-Service (SaaS) is a new approach for developing software, and it is characterized by its multi-tenancy architecture and its ability to provide flexible customization to individual tenant. However, the multi-tenancy architecture and customization requirements introduce many new issues in software, such as database design, database partition, scalability, recovery, and continuous testing. This paper proposes a hybrid test database design to support SaaS customization with two-layer database partitioning. The database is further extended with a new built-in redundancy with ontology so that the SaaS can recover from ontology, data or metadata failures. Furthermore, constraints in metadata can be used either as test cases or policies to support SaaS continuous testing and policy enforcement.

Key words: SaaS; customization; database partitioning; testing; recovery

Tsai WT, Huang Y, Shao QH, Bai XY. Data partitioning and redundancy management for robust multi-tenancy SaaS. *Int J Software Informatics*, **2010, 4(4): 437–471. http://www.ijsi.org/1673-7288/4/i70.htm**

1 Introduction

Software-as-a-service (SaaS) is software that deployed over the internet and often run on a cloud platform. With SaaS, a software provider licenses an application to customer as a service on demand, through a subscription or a "pay-as-you-go" model. A common SaaS application is CRM (Customer Relationship Management). Notable SaaS applications include Salesforce.com which provides on-demand CRM; Peoplesoft from Oracle which provides SaaS infrastructure for enterprise applications; Google maps and Google apps (such as Google docs, gmail); and Microsoft Office Web Apps. As a proliferating software model, more application developers will embrace the SaaS model. However, it also faces new challenges.

(1) Multi-tenancy architecture (MTA) support: Multi-tenancy refers to a principle where a single instance of the software runs on a server, serving multiple client organizations (tenants), which is often used in SaaS. With MTA, a software

* This work is sponsored by the U.S. Department of Education FIPSE Project P116B060433, U.S. National Science Foundation Project DUE 0942453 and Department of Defence, Joint Interoperability Test Command.
Corresponding author: Yu Huang, Email: yu.huang.1@asu.edu
Received 2010-09-03; revised 2010-12-20; accepted 2010-12-25.

application is designed to virtually partition its data and configuration, so that each client works with a customized virtual application instance. Although all tenants share the same software, they feel like they are the sole user of the software. A maturity model for SaaS with four levels is proposed in Ref.[9] with, the highest level of SaaS being configurable, scalable, and having MTA. A configurable SaaS is often achieved by customization and a scalable SaaS is often achieved by duplicating software to meet the increased load. The MTA requires SaaS providers to support a huge number of applications. In 2009, Salesforce.com[55] reported that it is supporting 100,000 distinct applications using 10 databases running on top of 50 servers in two mirror-sites with just one software base. Due to the unique features of MTA, a realistic SaaS application needs to address the following issues:

- Scalability: The algorithm should adjust according to the changing load of system. Specifically, if the workload increases, resources (such as processors, memory, and disk space) should be allocated to handle the task, and if the workload decreases, resources should be re-allocated to other tasks. In this way, resources can be dynamically allocated and re-allocated at runtime. Ideally, the increase (decrease) in resources should be proportional to the increase (decrease) in workload, while keeping the performance of each task at an acceptable level;
- Database partitioning and consistency: Tenant data need to be partitioned well in the back-end database to support real-time high performance computing;
- Fault-tolerant computing: The failures of processors or storage should not affect the operation or data because they may represent business transactions that will bring in revenue for the company.
- Security and fairness: Tenant data should be isolated from each other, and tenants of the same priority should receive the same level of services and resources as multiple tenants will share the same software and possibly also the same database in MTA;
- Parallel processing: It is highly desirable that the tasks can be processed in parallel such as done by Map-Reduce to take advantage of the massive number of processors, memory and storage units available in a cloud environment;
- Isolation: Any changes in a tenant's data should not affect any other tenants;
- Performance and availability: As one SaaS program potentially needs to serve hundreds of thousands or even millions of tenants or applications, it is critical that SaaS software can provide real-time performance and availability with automated data migration, backup and restoration and isolation.

Note that cloud computing has changed computing significantly due to massive number of processors will be used to support real-time applications that require high availability and reliability. Data must be reliably stored and restored in case of data failure. A cloud need to allocate resources to a computing task in case of raising workload and re-allocate resources when the workload decreases. In cloud computing, system availability, performance, reliability and security will be more important

than minimizing system resources. To appreciate the scale scope of cloud computing, one can take look of the existing cloud infrastructure. For example, Google data center[12] reports that it can host 45K servers in 45 containers in one single data center as reported in 2009. Microsoft[24] also reported their data center version 4.0 recently where there will be no side walls so that processors can be exposed to air, and containers of processors can be added to the data center to meet the increased workload. These massive numbers of servers will be used to provide real-time computing with automated reconfiguration and recovery mechanisms.

In Ref.[51], we proposed a four-layered architecture for SaaS customization. This paper further extends the four-layer architecture, with an additional data layer, to provide a scalable framework for SaaS. This paper discusses the possible database partitioning choices for SaaS in the data layer. For each choice, this paper discusses what types of application is suitable for taking advantage and their trade-offs. This paper also proposes the related load-balancing algorithms to address the scheduling problem when data are partitioned. More details will be discussed in Section 4.

(2) Fault-tolerant support: SaaS is usually deployed on a cloud infrastructure to handle large volume of requests. Failures, such as disk failure and power outage, are common in a cloud. Common cloud platforms often provide fault-tolerance and recovery mechanisms in the PaaS (platform as a service) level. Providing these mechanisms at the PaaS has the advantage that these mechanisms will be available for all kinds of applications including SaaS or non-SaaS applications. However, such mechanisms will not be optimal because SaaS often has its own software architecture that is unique. For example, a well-know PaaS GAE (Google App Engine) provides fault-tolerance mechanisms by writing almost every data into at least three chunks with each chunk 64 MB, and the system will recover any failure of a chunk by retrieving data from the other two chunks. This is a good solution, but this solution is applied to all kinds of applications running on top of GAE regardless of the application architecture. Furthermore, any recovery will be at the level of chunks rather than at the software level as the recovery mechanisms at the PaaS level do not have any knowledge of SaaS runs on top of it. Because the solutions provided by the fault-tolerance and/or recovery mechanisms are available at the SaaS level, redundant data can be managed to fit the SaaS components such as data, metadata, indexing, and customization, and recovery mechanisms understand the relationships among the same SaaS components.

This paper proposes a framework to store redundant information in SaaS components, so that a failure of any SaaS component will be stored in ontology, metadata and data to support SaaS recovery. Specifically, any information in one aspect, such as ontology, metadata and data, can be used to recover data in other aspects. For example, ontology information can be used to recover metadata, and metadata information can be used to recover ontology, and data can be used to recover ontology and metadata. This ensures that information in SaaS will not be lost easily, and this also provides support for the storage layer (layer 5) in the framework. Note that this approach can be used on top of a fault-tolerant PaaS where data are automatically written into at least three chunks to provide additional assurance that data will be lost in cloud computing. More details will be discussed in Section 5.

(3)SaaS testing with new challenges: Testing SaaS is different from testing software services as SaaS involves customization, configuration, and scalability while

services involves only calling and responding with QoS constraints. Currently the testing of SaaS software often uses the traditional software testing practices. As only a single copy of the software is maintained for multiple sharing tenants and each having different requirements. Furthermore, a new tenant may be added into SaaS, and bring new requirements after SaaS deployment, and thus SaaS testing may need to continue after deployment. In fact, as the SaaS being deployed, it needs to be continuously verified to support the SaaS. This feature has been used by Google Chrome OS where continuous verification is a key feature. As SaaS being developed, it also needs to be tested and evaluated, and thus SaaS requires a continuous testing model throughout its entire lifecycle. iTKO[25] is a new SaaS testing enterprise that uses the continuous testing when building up their DevTest Cloud. iTKO's continuous validation service (CVS) feature orchestrates the testing and validation aspects of IT, Integration workflows and SOA Governance, to ensure reliability and instill trust throughout the lifecycle of the application.

This paper proposes a built-in continuously testing SaaS testing framework from ontology and metadata to support QoS (Quality of Services) and SLA (Service Level Agreement). Besides continuous testing, testing SaaS software can also be collaborative by nature since it is usually developed with a service oriented architecture. In this paper, a collaborative testing environment by generating test scripts in a collaborative manner. The schema integrates continuous testing with the storage layer, by leveraging database triggering rules. Integrating testing and intelligent testing can also be conducted within the framework. More details will be discussed in Section 6.

(4) Ontology-based analysis and development: Ontology can have a significant role in service applications and SaaS development, and it can be used for specification, references, reasoning, and even customization[51]. In the customization framework, each layer is supported with its own ontology for discovering similar templates and conducting intelligent mining. This paper further leverages ontology to provide a quadpartite recovery scheme to make the framework self-recoverable as current SaaS design just uses metadata and data for assurance and recoverability. The proposed scheme adds ontology to support recovery, and also update data and metadata so that they can be reconstructed in case of failures from ontology. In other words, the system has a built-in redundancy so that each part can be recovered.

As a summary, cloud computing and SaaS provide new requirments for scalability, and robustness. Knowing the key differences between cloud computing and traditional computing can help to better understand the issues in SaaS. Comparing with traditional computing, cloud computing has several major differences, as shown in Table 1: In terms of scalability, cloud computing provides on-demand resource scaling, which allows the cloud users to scale their applications dynamically according to application workloads. It is the most cost-effective way for users to scale since they do not need to worry about the management of resources and hardware costs. While in traditional computing, users need to support scalability either by scaling up (upgrading the configuration of the hardware) or scaling out (buying and adding more compute nodes into the system). This is not cost-effective when what the users want to deal with is only temporary workload changes. In traditional computing, usually one issue is considered when conducting the computing. For example, fault-tolerant computing concerns more about availability, while real-time computing concerns more

about application performance. However in cloud computing, all the issues, such as fault-tolerance, reliability, application performance, resource management, need to be taken into consideration at the same time. This is a new type of software engineering. In traditional computing, people concerns more about efficient resource utilization and management, while in cloud computing, the reliability of the cloud system becomes a serious concern. The importance of the read/write operations is also changed. Write operations are considered to be more important in cloud computing because a write update may represent a customer order[18], while it is the opposite case in traditional computing where efficient execution is the primary concern.

	Traditional Computing	Cloud Computing
Scalability	Provides scalability by scaling-up or scaling-out. Need to buy / update hardware to scale to satisfy increasing demand. Resource might be wasted as idle.	Provides "on-demand" scalability to users of the cloud. The resources can be dynamically allocated and released. User can scale with low hardware costs.
Software Architecture	Usually focuses on a single facet, such as fault-tolerant computing, real-time computing. Traditional software engineering principles can be applied.	A new kind of software engineering: Many separate issues, such as fault-tolerant computing, real-time computing, database, are all involved together in the cloud.
Frequency and priority of operations (Read/Write)	Read operations are considered more important than write operations	Write operations are considered more important than read operations
System Metrics	Resources are considered to be more important. The efficient management of resources is highly valued.	Reliability is considered to be more important than resources. Efficient reallocation of the resources is highly valued.

Figure 1. Major differences of cloud computing and traditional computing

The contribution of this paper is five-fold:

- This paper tackles the scalability problem in SaaS framework by extending the previous four-layer SaaS customization framework with additional data layer.
- This paper investigates a two-layer partitioning schema with effective index to support scalability in SaaS applications.
- This paper proposes a scheme to embed the recoverability inside the proposed SaaS framework.
- This paper incorporates the feature of continuous testing in the proposed SaaS framework, which provides embed testing support.
- This paper exploits the usage of ontology in providing support for customization, recovery and continuous testing in SaaS.

This rest of the paper is structured as follows: Section 2 discusses the related works. Section 3 presents the SaaS framework with data layer to support scalable

SaaS. Specific database partitioning schemes and related issues are discussed in Section 4. Section 5 discusses the quadpartite recovery scheme with ontology. Section 6 presented how to embedded the support of continuous testing. Section 7 concludes this paper.

2 Related Work

This paper is related to several perspectives, including SaaS customization, database partitioning and SaaS recovery and testing. The following sections will discuss these topics in turn.

2.1 SaaS customization

Existing SaaS customization has mainly addressed data-level customization, specifically data schema design. Specifically, the data architecture of multi-tenant[9] is identified as three distinct approaches (Separate Databases, Shared Database Separate Schemas and Shared Databases Shared Schema). Reference [53] proposed a new schema-mapping technique for multi-tenant data named Chunk Folding. But they have not touched other layers in customization. Another group of researches touched service level, for example, Zhang[3] proposed a novel SaaS customization framework using policy through design-time tooling and a runtime environment. Mietzner[32] described the notion of a variability descriptor which defines variability points for the process layer and related artifacts in a service-oriented manner. Li[30] considered the multi-layer and cross layer relationship, and used multi-granularity models to compose customization tasks. Essaidi[20] presented an open source infrastructure to build and deliver on-demand customized business intelligence services.

These existing projects either discussed solutions in certain layer of customization or did not investigate the relationship cross layers based on ontology information. In our previous work[51], we addressed the problem of customizable SaaS with MTA by providing a four-layer architecture. This paper further extends the four-layer architecture, with an additional data layer, to provide a scalable framework for SaaS.

Existing industry partners support SaaS customization in their own ways. For instance, Google App Engine (GAE)[22] offered the customization at the program layer using a deployment description file in which users can change the parameters such as servlet, URL paths, jsps, and security methods. GAE also supports service layer customization in which users can set up service names and domains, control the billing budgets. However, workflows and date are not customizable at GAE. Amazon EC2[2] offers similar customization capabilities as GAE, but it does not offer the workflow layer. Salesforce.com[39] offers a flexible customization framework, in which users can customize the UI, workflow and data inside their framework, but semantic information, e.g., ontology is not integrated into customization.

2.2 Scalability and database partitioning

Data partitioning is a well-studied problem in database systems (e.g., Refs.[31, 48, 14, 29, 19]). In the literature, many partitioning schemes have been studied; e.g., vertical partitioning vs. horizontal partitioning, round-robin vs. hashing vs. range partitioning[15]. Past work has noted that partitioning can effectively increase the scalability of database systems, by parallelizing I/O[29] or by assigning each partition

to separate workers in a cluster[31]. H-Store[44] presents a framework for a system that uses data partitioning and single-threaded execution to simplify concurrency control. G-Store[17] extend this work by proposing several schemes for concurrency control in partitioned, main memory databases, concluding that the combination of blocking. The discussion of vertical partitioning, e.g. column based database has been introduced recently, including MonetDB[8], C-Store[38], which can provide performance improvement on read-intensive analytical processing workload, such as in data warehouse.

To support scalability of cloud, data partitioning becomes a widely accepted solution. Parallel DBMSs with emerging cloud data management platforms is provided by industry partners, such as Refs.[4, 16, 18, 34] (such as efficient data partitioning, automatic fail over and partial re-computation, and guarantees of complete answers). In the database community recent work compared the performance of Hadoop versus the more traditional (SQL-based) database systems[35] which focuses on read-only, large scale OLAP workloads, and Refs.[27, 28] focused on OLTP workloads. Berkeley's Cloudstone[47] specifies a database and workload for studying cloud infrastructures and defines performance and cost metrics to compare alternative systems.

In this paper, a novel two-layer model for partitioning is provided, which first partitions horizontally by tenants, and then vertically partitions by columns. The model can benefit both read and update operations comparing with horizontal-partitioning only or vertical-partitioning only methods. Effective indexes, DHT and B-tree are used at each layer respectively to help with the load balancing and scheduling.

2.3 Multi-tenancy support

In the new SaaS applications, multitenancy architecture (MTA) is highly valued and heavily employed in build these applications. There are several benefits brought by using MTA. First, by employing MTA, only single copy of the code based is maintained to support tenants from different domain. SaaS application developers do not need to develop different copies of customized application and maintain all of them separately. Second, each tenant can customize their own GUI, workflow, service and data schema according to their unique application needs and the application appears to them as if they are the sole tenant.

However, this comes with a price. The complexity of building a SaaS application using MTA is high. And to scale the SaaS application to support thousands of users is very challenging. Force.com has successfully provides a on-demand Customer Relationship Management (CRM) SaaS application using MTA. The several key architectural principles used in their system are stateless AppServers, highly customized and optimized Database system, tenant based table partitioning, creative denormalization and indexing. In their design, they have three types of tables, which is responsible for storing metadata, data and the index respectively. The index table is called pivot table, which stores specialized indices for unique fields, relationships, most-recently-used objects and so on. These pivot tables effectively accelerated the access to the metadata and data tables design specifically to support MTA. All data and metadata structures are partitioned to improve performance and manageability. The tables are hash partitioned by tenant / organization. The application tier is also dynamically partitioned by tenant / organization.

Among the key design principles they employ in their system, the database partitioning scheme and indexing scheme are related to the techniques developed in this paper. As will be discussed in this paper, besides the horizontal tenant based table partitioning, this paper also proposes vertical partitioning of the relation for read-intensive applications. By combining the two-level partitioning scheme and the duplication of the relational tables, the application will be able to achieve high performance whether it is read-intensive or write-intensive.

2.4 Recovery mechanism

Traditionally, data recovery is achieved by adding / storing some redundant information so that whenever the data is corrupted / lost, it can be reconstructed using the redundant information. For example, in different file systems, checksum is usually computed for each data block to verify that the operation is performed correctly. In RAID 1, it simply uses mirroring to store additional copy of the data. In RAID 2, 3, 4, 5 and 6, they use different kinds of parity as redundant information so that the failure of one disk can be recovered. However, there are disadvantages in these approaches. First, the file system approach cannot tolerate the failure of the entire disk. Second, the hardware is expensive. Most of the cloud computing platform use cheap commodity machines as nodes for computation, and thus RAID is usually not available on these commodity machines. Third, the redundant information is centralized. If power outage happens to the node having the RAID array, all the data will be unavailable.

To provide better availability, replication is a frequently adopted technology in the cloud. However, such recoverability is provided outside of the SaaS framework. In some cases it might be desirable that recoverability can be embedded in the SaaS framework. This paper proposes a solution which can recover different data type with the assist of ontology information.

2.5 Continuous testing

Continuous testing is a natural fit for the cloud, which is a feature introduced by Google Chrome OS, a new operating system. iTKO also uses the continuous testing when building up their DevTest Cloud. Distributed enterprise applications are naturally evolving over time, as they leverage highly interdependent and changing services and technologies, which are assembled to build the finished application at runtime. Continuous integration is moving from a best practice to a do-or-die IT tactic.

iTKO[25]'s continuous validation service (CVS) feature orchestrates the testing and validation aspects of IT, Integration workflows and SOA Governance, to ensure reliability and instill trust throughout the lifecycle of the application. It conducts live regression, functional and performance monitoring of critical business workflows on a continuous basis, providing an actionable way to enforce that expected business policies are being met. The CVS functionality can be used as a shared provider of both scheduled and even-based regression and performance test suites. When a change to an underlying application is made, when a new service is promoted for use, or if an unexpected error or performance expectation is not met, it will communication with the enterprise's stakeholders, or report this activity to an SOA governance, ALM

collaboration tool or IT monitoring platform.

However, it is not discussed that how the continuous testing / integration feature can be embedded into an existing software. It is not hard to imagine that this process can be extremely complex and the cost-effectiveness is a question having no answers. In this paper, we proposed to embed the continuous testing feature within the SaaS framework rather than posting it from outside. More details will be discussed in Section 6.

3 SaaS Customization Framework

A multi-layer customization framework *OIC* is proposed in Ref.[51]. In the paper, all the aspects for an application can be configured through a platform, and tenant specific customization of a SaaS application affects all layers, from functional requirements in Graphic User Interface (GUI), customized business processes to database schemas design. The customization process is assisted by domain ontology[56], that specifies domain vocabulary and their relationships. All layers have their own ontology information, thus data ontology, service ontology, business process (workflow) ontology, and GUI ontology, that describes concepts and relations in that layer. *OIC* allows users to search for objects (data, service, workflows) in a repository, and then reuse, include or modify them as needed when designing new ones, so that the design phase will be easier and be shortened comparing with designing new ones from scratch. To deal with the commonality of tenants, a set of templates (standard) objects is provided for designers to assist SaaS customization. The template objects are stored at different repositories at all layers (including data repository, service repository, workflow repository and GUI repository). Given ontology information for a particular application domain, *OIC* uses template objects as an initial starting point, and support customization in a cost effective way. Also the recommendation engine can provide a list of candidates according to tenant's profiling.

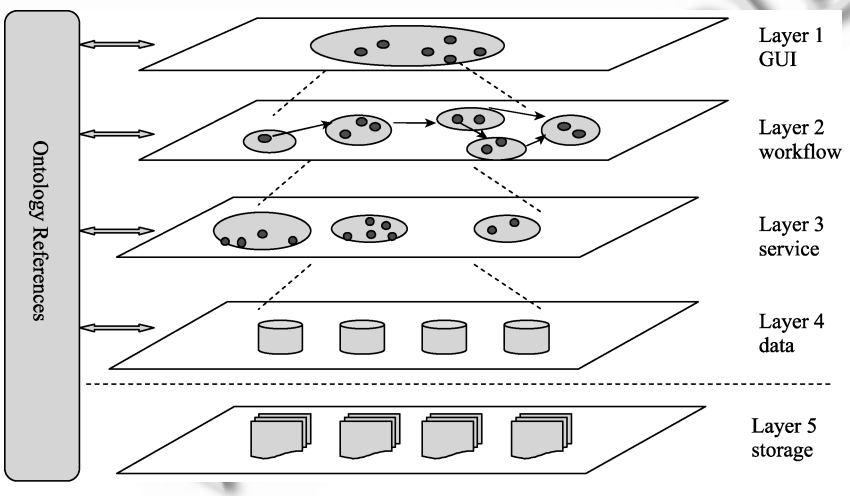

Figure 2. 5-layered architecture for SaaS customization

This paper further extends *OIC* with an additional storage management layer, as shown in Fig.2, responsible for database partitioning, load balancing, and scheduling.

Based on the five-layer model, this paper proposes a new solution for recovery in Section 5.

3.1 Ontology driven meta-data customization

In traditional database design, objects and fields are defined to provide abstractions of the real-world entities that they represent. Separate database tables are created for each type of object represented. Specific attributes are represented by fields within the tables. Object instances are represented by rows within the tables. Actual data is placed into a database by inserting rows into the database tables. Relationships are represented by fields in one table referring to a key field in another table.

To support MTA, a metadata-driven database operates somewhat differently. Objects and their fields are mapped to metadata tables. Actual data is stored in either in a single data table, or, for large text objects such as documents, in a separate character large object storage (Clobs) area. A series of index tables is created to make accessing the data within the single data table more efficient. To support multiple tenants, the object and field metadata contains information about the fields, and also about the tenants. The comparison of metadata driven databases and traditional database designs are shown in Fig.3.

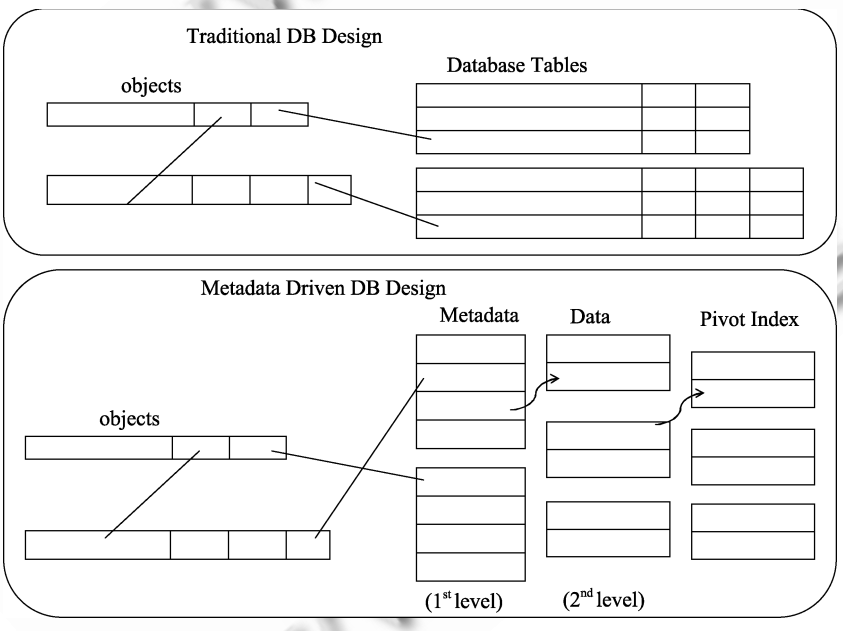

Figure 3. Metadata driven database design

In details, three types of data in MTA with diverse features[54]:

- Metadata: Objects and their fields are mapped to metadata tables.
- Data: Actual data is stored in either in a single data table, or, for large text objects such as documents, in a separate character large object storage (Clobs) area.

- Pivot index: make accessing the data within the single data table more efficient. To support multiple tenants, the object and field metadata contains information about the fields, and also about the tenants.

Ontology semantic information can be matched to database logic designs and help metadata generation. The domain objects can represent a large proportion of meta-data that are serialized into the data repository. Multiple database schemas can be used in MTA[1], such as XML, sparse table, views. Tenants can choose any database schemas as needed.

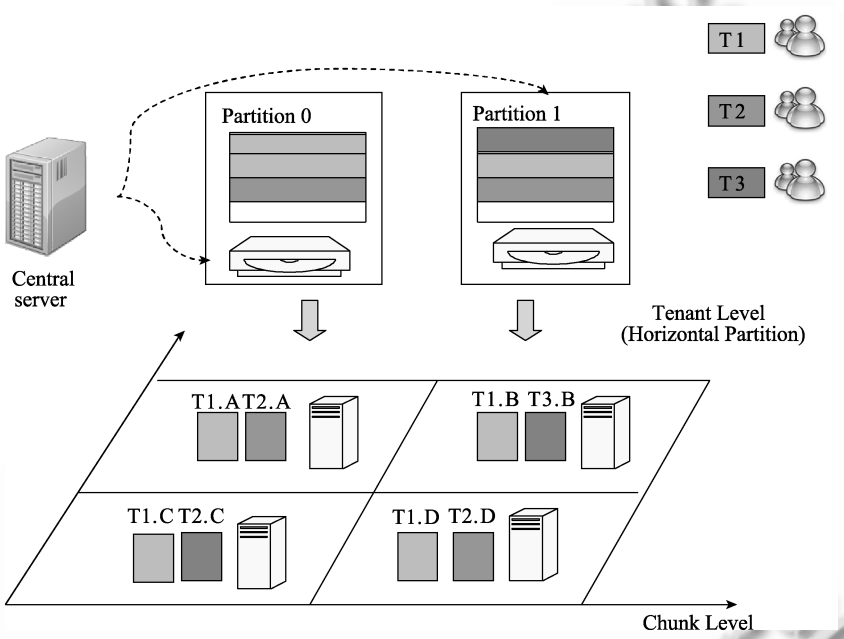

Figure 4. Tow layer partitioning model

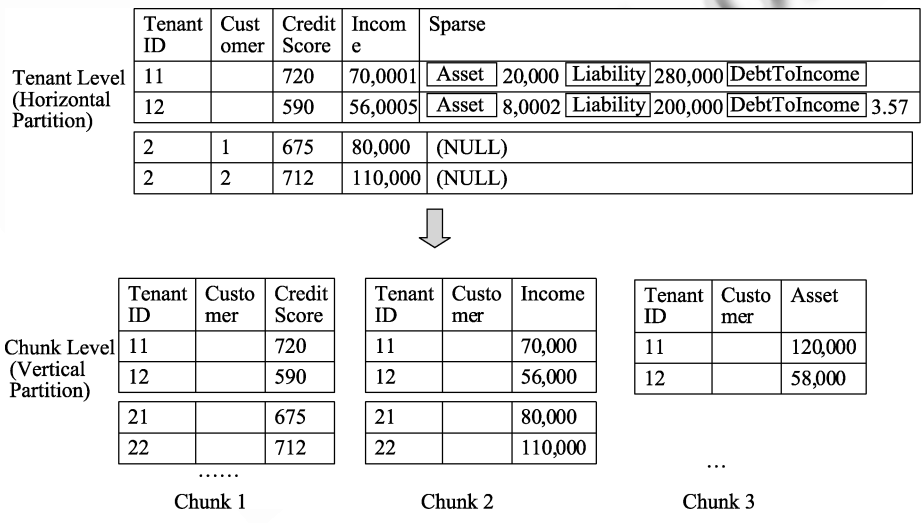

Figure 5. Example for two layer partitioning model for Fig.4

4 Scalable SaaS with Database Partitioning

In MTA, the shared database architecture design[9] calls for effective scalability support. In the ideal case, the maximum number of tenants should be proportional to the increase of resources, while keeping the performance metrics of each tenant at an acceptable level. There are two types of scaling: scale-up and scale-out. The scale-up or vertical scaling is done by adding additional resources, such as CPUs, memory, and disks into a single node in a clustered system. In this way, a node becomes more powerful by having more resources. The scale-out or horizontal scaling is done by adding additional nodes to an existing clustered system. For example, instead of a cluster of thirty nodes, the system may have fifty nodes instead. The scale-up is easy to use but may not provide linear scalability increase due to the overhead in resource management. The scale-out provides a more cost-effective way, where it can incrementally extend the system by adding more resources to a low-cost hardware set. Furthermore, it can improve the reliability and availability of the system due to the redundancy. In the scale-up scenario, one can create more than one database partition on the same physical machine, while in the scale-out scenario, partitions can be created in multiple physical machines, and each partition has its own common memory, CPUs, and disks.

With the increase of tenant's traffic, SaaS application can be easily scaled out by adding new instances, but database server becomes the bottleneck of the system scalability[17]. While most traditional database systems (e.g., DB2, Oracle 11, SQL Server, MySQL, Postgres) uses traditional data structures (e.g., dynamic programming, B-tree indexes, write-ahead logging), the differences in the implementation of SaaS are immense.

Database Partitioning[54] can improve the system performance, scalability and availability of a large database system in a multi-tenant way. For example, given a tenant's information, the query optimizer only has to access the partitions containing the tenant's data rather than the entire table or index, using "partition pruning". Data partitioning is a proved technique that database systems provide to physically divide large logical data structures into smaller and easy manageable pieces (chunks). The data inside a database can be distributed across one or more partitions. A distribution key is the column used to determine the partition in which a particular row is stored. Instead of having one database server controlling the whole system, the database is logically partitioned and each of them can be controlled by a separate server. Indexes play an important role in improving overall performance together with partitioning. Different types of indexes are built to provide efficient query processing for different applications.

4.1 Database partitioning choices

Many partitioning schemes have been studied; e.g., vertical partitioning vs. horizontal partitioning, round-robin vs. hashing vs. range partitioning[15]. Two most widely used methods are horizontal partitioning and vertical partitioning.

Row stores and horizontal partitioning Key-value stores (row stores) is inherent to be the preferred data management solutions in cloud, such as Bigtable[11], PNUTS[16], Dynamo[18], and their open-source HBase[23]. These systems provide various key-value stores and are different in terms of data model, availability, and

consistency guarantees. The common property of these systems is the key-value abstraction where data is viewed as key-value pairs and atomic access is supported only at the granularity of single keys. This single key atomic access semantics naturally allows efficient *horizontal data partitioning*, and provides the basis for scalability and availability in these systems.

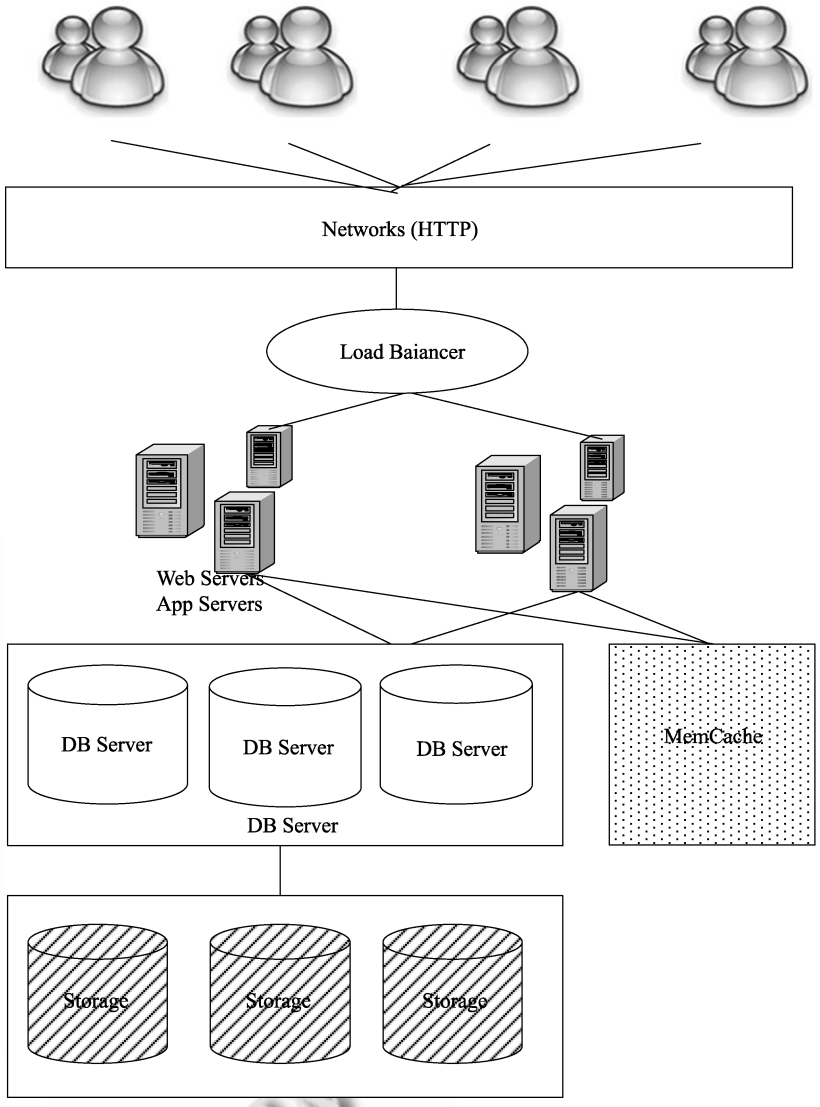

Figure 6. Scheduling system architecture

Horizontal partitioning is widely used in existing cloud computing products, such as IBM DB2 V9[50], Force.com[21] and etc. Two horizontal database partitioning approaches are available: application-based distribution keys (choosing one or more attributes as a distribution key according to domain knowledge) and tenant-used distribution keys(stores each tenant's data in a single partition).

Update in row partition is simple and supported as follows: the storage key (SK), for each record is explicitly stored in each partition. A unique SK is given to each "insert" of a tuple in a Table.

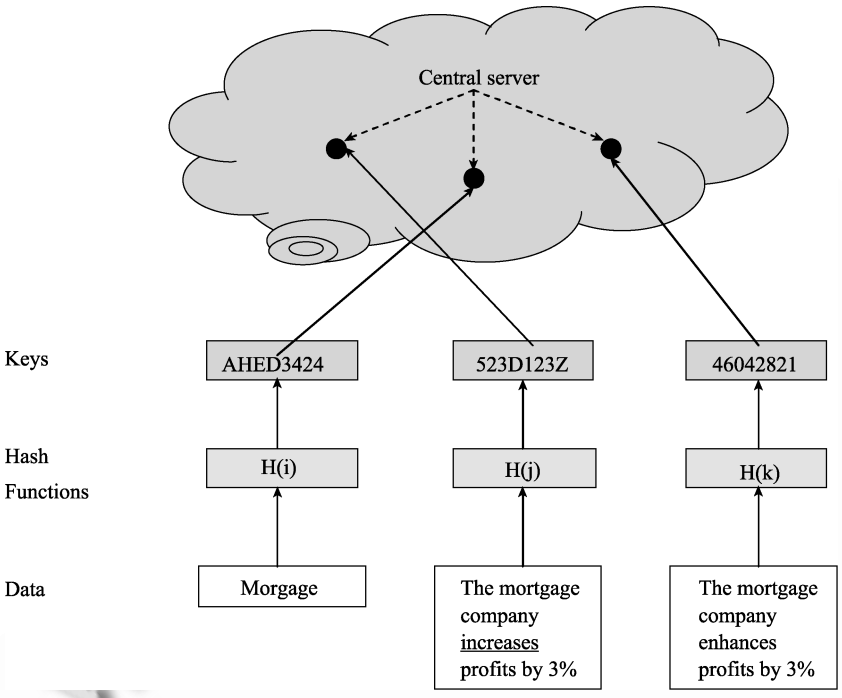

Figure 7. Sample of DHT (distributed hash tables)

Column store and vertical partitioning Column store is a read-optimized solution, any fragment of projections can be broken into its constituent columns, and each column is stored in order of the sorted key for the projection. There are several possible encoding schemas considering the ordering and proportion of distinct values it contains, including:

1. Self-order, few distinct values: represented using triple $(val, 1^{st}, occur)$ such that val is the value stored in the column, 1^{st} is the position where val first appears, $occur$ is the number of occurrence of val in the column.

   Clustered B-tree indexes can be used over this type of columns. With large disk blocks (e.g. 128k), the height of this index can be small.

2. Foreign-order, few distinct values, represents as (val, bmp) such that val is the value stored in the column, bmp is a bitmap index, which can indicate the positions the value is stored. Each bitmap is sparse, one can run length encode to save space. To find the i^{th} value, "offset indexes" B-tree can be used to map values contained in the column.

3. Self-order, many distinct values: represent *delta* value of the previous value in the column. The first entry of every block is a value in the column and its associated storage key.

4. Foreign-order, many distinct values: not necessary to encode.

Update in column store is more complicated, one has to join values cross columns, in which join indexes are used to connect various projection in the same table.

As a summary, row store and horizontal partitioning is writeable operation preferable, while column store and vertical partitioning is optimal for read operations. This paper proposes a hybrid approach as SaaS involves both read and write operations.

4.2 P^2: Two-Layer partitioning model

The hybrid two-level scheme combines both read-optimized column store and an update oriented writeable store as shown in Fig.4. At the top level, there is a partition for each tenant, which can support high performance inserts and updates. At the lower level, a larger component for column partitions are supported, which can optimize for reading and batching with the tenant's attribute level. As one can see, tenant A, B, C share same physical databases, and each of them has its own physical chunks associate with it respectively.

A sample mortgage data of the three tenants with P^2 are shown in Fig.5. Two tenants are represented with ID 1 and 2 respectively, and each of them has two customers. Using P^2 model, the horizontal partition splits the data into two at tenant level according to the tenant ID; then at the vertical partitioning, similar attribute for different tenants are clustered together into different chunks, for example, credit score of tenant 1 and 2 are clustered into chunk 1 while income of tenant 1 and 2 are clustered into chunk 2 accordingly. Specially, tenant 1 has its own unique attribute asset, which is clustered into chunk 3.

Originally, a master server is used to maintain the global index. All queries are sent to the master server to search the global index and then forwarded to corresponding servers. The size of the global index is proportional to the size of the data and concurrent requests, the master server risks being a bottleneck, hence one can further distributes the global index across servers. Each server only maintains a portion of the global index. The distributed approach improves scalability and fault tolerance. The global index is build on top of the local indexes. To search local data efficiently and make the local balancing, B-tree is used for local chunks. In the global index, a DHT index is used to make the uniform distribution among servers.

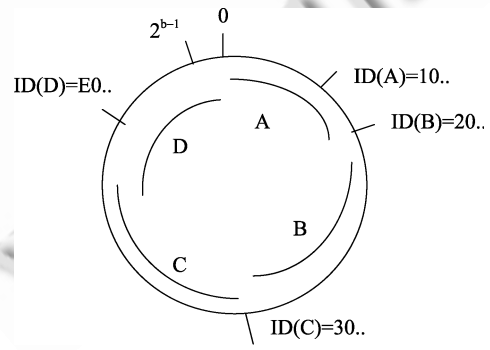

Figure 8. Balanced range allocation

DB Schema

OrgID	TableName	ColumnName	Type	Length	IsIndexed	MaxValue	MinValue
1	Customer	Income	Integer	64	0		10,000
1	Customer	CreditScore	Integer	32	1	1000	0
2	Customer	DebtToIncome	Float	64	0	100.0	0.0
2	Customer	Liability	Integer	64	1	1,000,000	1000
…	…	…	…	…	…	…	…

Triggers

Trigger_Schema	Trigger_Name	Created	Action_Condition	Action_Statement	Definer
Customer	Test_Income	08/01/10	Income < 10,000	Warning	Paul
Customer	Test_CreditScore	07/25/09	CreditScore < 0	Warning	Tim
Customer	On_Insert_Liability	06/30/10	Insert	Check_Liability	Tim
…	…	…	…	…	…

Figure 9. The metadata table

4.3 Scheduling and load balance

To do better load balancing among partitions to optimize the overall database performance, an effective algorithm is highly desirable, that can migrate, distribute and duplicate tenants among partitions through monitoring the load. Most cloud scheduling algorithms and database solutions address their problems independently. However, most of cloud components and functionalities are interconnected. Specifically, a task scheduling algorithm needs to consider database partitioning to provide an efficient solution for performance and scalability. More specific, a task assigned to a processor should host the appropriate data partitions otherwise data updates and migration among caches and processors can be expensive.

The most scalable MTA requires an SaaS scheduler that can dispatch tasks to multiple copies of the same software in a data center[10]. As the same version of the software is used, user customization must be stored in databases, and thus an integrated solution must address both scheduling and database partitioning together as shown in Fig.6.

Different strategies have been adopted to allocate data partitions in the cloud. One allocation strategy permits a single copy of the database to be stored in the network, non duplication. The partitions are allocated to the nodes to minimize the overall system communication cost, query response time, and other criteria depending upon the objective of the designer[14,48]. Another strategy is to store multiple copies of all or a part of duplications. Although this reduces transmission cost and improves response time, it increases data redundancy, storage costs, and update costs to keep data consistency. To solve this problem, sharing everything among tenants provides a solution. This paper adapts a sharing everything framework to support scheduling and load balancing in a cost effective way.

4.4 Two-Layer index for P^2

To scheduling requests and balance loads using our P^2 model in Section 4.2, a correspondingly two-layer index mechanism is proposed as follows:

DHT at tenant partitioning level: DHT(Distributed Hash Tables) can be adopted in the upper layer of partitioning for nodes among tenants.

Given a key, DHT can map the key onto a tenant's data block as shown in Fig.7. Inherent from consistent hashing[28,46] to assign keys to blocks, the consistent hashing supports balance load, since each node received roughly the same number of keys, and involves relatively little movement of keys when add or delete chunks from the system. Several good features are maintained in DHT, e.g. balances load with high probability (all nodes receive roughly the same number of keys), minimize maintain cost (when an N^{th} node added/deleted, only $O(1/N)$ faction of the keys are moved to a different node). Each node maintains information only about $O(logN)$ other nodes, and a lookup takes $O(logN)$ time.

Mortgage Company A

Customer	Credit Score	Income
1	675	80,000
2	712	110,000

Mortgage Company B

Customer	Credit Score	Asset	Liability	Income	DebtToIncome
1	720	120,000	280,000	70,000	4
2	590	58,000	200,000	56,000	3.57

Multi-tenancy Data Schema

Tenant ID	Customer	Credit Score	Income	Sparse
1	1	675	80,000	(NULL)
1	2	712	110,000	(NULL)
2	1	720	70,000	Asset 120,000 Liability 280,000 DebtToIncome 4
2	2	590	56,000	Asset 58,000 Liability 200,000 DebtToIncome 3.57

Figure 10. The data table

One can use unsigned integers to match to the output of cryptographic hash function. It is convenient to visualize the key space as a ring of values, support b bit in the ring, starting at 0 and increase clockwise until they get to $(2^b - 1)$ and then overflow back to 0. Figure 8 shows a ring representation of Pastry-style routing[36], in which key space is divided into evenly sized sequential ranges, each node has one range, and ranges are assigned in the order of nodes, sorted by hash ID. Hence data are uniformly distributed among the nodes.

B-tree index at chunk partitioning level To allocate and schedule chucks at the second level at least the following approaches can be applied:

(a) Allocate tenant's data with fixed partitions periodically or asynchronously: given the number of tenants k in a cloud, and the number of partitioning blocks at each tenant, it can partition the available database chunks into groups based on resource constraints and user requests. This method will decrease the contention, and each partition will be allocated to a certain copy of the software. As the workload changes, a re-allocation needs to be done. One way is to perform the update periodically, whenever the changes or the rates of changes exceed certain thresholds, or when the system slows down significantly due to unbalanced workload among different tenants.

(b) Flexible partitioning and re-partitioning. Unlike the previous approach, the partitions will be dynamically maintained as the workload changes. For example, one may use a scheme similar to B-tree to organize data partition accordingly. A B-tree allows a congested partition to double the resource, and a lightly loaded partition to reduce its resources by half, and it may be served together with another light partition

in the same processor. In this way, a busy tenant can have its needs met, and a light tenant will not occupy idle resources by sharing with fellow light tenants. By using this approach, the resource can be automatically maintained and balanced.

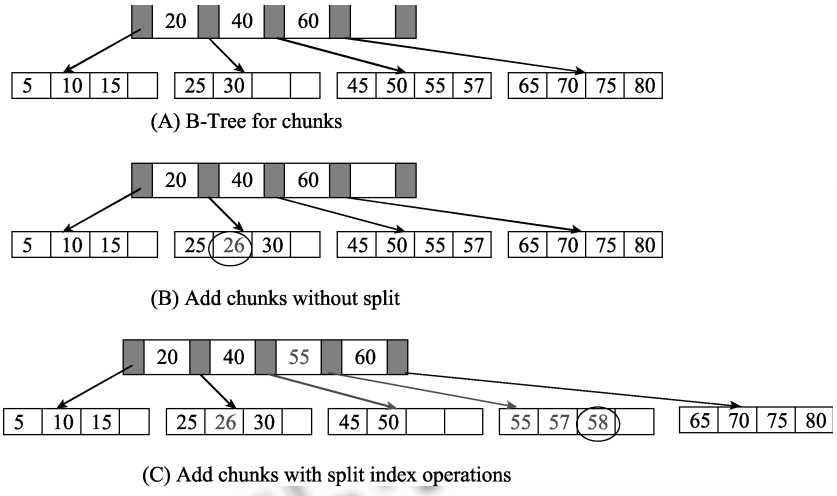

Figure 11. Sample B-tree for chunks

Example: A sample B-tree for chunk partitioning is shown in Fig.11. The B-tree is used to maintain all chunks. At the beginning, each tenant can be allocated 20 chunks, and more chucks can be allocated to them when necessary. To add a chuck, either simple add operations (in Fig.11(B)) or split operations on the index page in B-tree (in Fig.11(C)) are needed.

4.5 Performance analysis for P^2

To analyze the performance of P^2, this paper first analyzes the performance of row-store and column-store. Then it further compares the performance of P^2 with these two schemes.

Operations composition based analysis: The performance of a storage mechanism can be evaluated using the access patterns. There are three types of access pattern: read mostly, write (update) mostly, and a hybrid of read and write operations.

To have a better understanding of P^2, one can start from the simple cases, and analyze the performance of row-store and column-store. The comparison of row-store and column-store is easier. At the beginning, all are update operations with no read operations (read = 0%, update = 100%), row-store has a shorter average response time than column-store, since update operations are only influence certain tuples in row-store, while column-store needs to update multiple columns in different chunks with join operations. As the read operation percentage increases to an extreme degree, all read operation and no update (read = 100%, update = 0%), column store beats row store due to its easy access to certain columns, especially when queries are focus on some specific attributes, e.g. credit score. When the distributions of read and update operations are close to each other, there would be some points, row-store and column-store have similar performances.

The case for two-layer is more complicated:

1. Mostly write-only operations: When update operations dominated the whole set of operations, the first layer of row-partitioning data will be used to store update changes. To ensure data consistency at the chunk level, similar as Dynamo's "always writeable" strategy, one can be maintained and postponed the chunk level update to back-end. At the meanwhile, the query operations are supported by two-level indexes, which can easily find out matching data, hence the average response time for the whole operation sets(both read and update operations) are shorter than row-store only, even much better than column-store.

2. Mostly read-only queries: When read operations dominate the whole operation sets, two-level indexes can support better query response time than others, and considering the mix of column store's high update costs, the average of two-layer model is better than column-store, even much better than row-store.

3. Mix of read and write operations: write operations are few but important, while read operations are frequent but less important. To satisfy the priority requirements of write operations, one can optimize write operations by adapting to some database transaction isolation models, to change the execution order of read/write operations without violating the consistency constraints.

Hence, using the proposed two-layer partition model with two-level indexes, one can improve the overall system performance.

Specially, some interesting observations here: suppose the total number of write operations is $|w|$ and the total number of reading operations is $|r|$, it is easy to see that $|w| \leqslant |r|$, since most of operations for tenants are getting data and few update to save their utility costs. After commit the initial data, the tenant may seldom update the data when necessary. On the other hand, the priority of write operations can be represented as $p(w)$, and the priority of read operations is $p(r)$, as we discussed in Section 1, since write operations have a high priority than read operations, one can get $p(w) > p(r)$. There is a conflict between the two arguments, as shown in Eq.(1), in another words, write operations are few but important, while read operations are frequent but less important. Balancing the frequency and importance of these two types of operations is an interesting problem. Readers may notice that P^2 is a natural solution for this conflict: the first level horizontal partitioning can fit for the priority requirement of $p(w) > p(r)$, since update operations are favored at this level, hence one can write easily. While the second level of vertical partitioning works for $|w| \leqslant |r|$, in which read operations dominate the system in most of the time.

$$Conflict(occur, p) = \begin{cases} |w| \leqslant |r| \\ p(w) > p(r). \end{cases} \tag{1}$$

To satisfy the priority requirements of write operations, one can optimize write operations by adopting to some database transaction isolation models, to change the execution order of read/write operations without violating the consistency constraints. New business requirements in cloud applications force service providers to loosen the rigid constraints and adopt a more relaxable approach in transaction isolation. It is important to ensure that the relaxation of isolation does not cause difficult-to-find problems.

The write priority optimization algorithm is suitable for cloud applications for the following reasons: In some circumstances, read operations are not necessary to get the most recent updates or it can even read data which are entered later. For example, in an online shopping application, reading customer's product list will not affect the shopping cart data. Hence one can move write operations forwards, in another words, change the order of read/write operations to adapt to high priority write operations.

On the other hand, there are specific read operations could not be postponed, due to the isolation affect. For example, a customer may want to know many books have been ordered in the shopping cart, and he/she can use "read" before issuing another write, but the write can move forward, and the read will read the most recently updated data. Another example, if a customer found that an order has been placed (for a read operation issued before the write) or an order has been removed, the customer can re-issue the read for double confirmation. This type of read as "double-confirmation-read" instead of read. Hence, three kinds of operations existing in cloud applications, including read, double-confirmation-read, and write. It is easy to see that write operations can move forward before any read operation, but not before double-confirmation-read. As we do not have too many double-confirmation-read, the optimization algorithm can be easily developed. Double-confirmation-read can be easily identified by checking whether a read operation is issued after the write operation from the same user on the same data.

When adjust the execution order of read/write operations, one can explore the traditional database concurrency issues[37] and design an optimization algorithm for write-priority operations without violating certain constraints accordingly. The three concurrency issues includes: Dirty Reads (one transaction reads data written by another uncommitted transaction), Non-repeatable Reads (one transaction read the same data twice and one write operation modify the data in between the two reads, which cause the 1st read operation got the non-repeatable value) and Phantom Reads (when one read operation gets a range of data more than once and a write operation inserts/deletes rows that fall within that range between the first transaction's read attempts, hence "phantom" rows appear/disappear).

Usage view analysis: There are two types of usage views for P^2, one is tenant-specific view for each customers, the other is cross-tenant view by cloud service providers, such as system monitoring, auditing, and performance control.

For the cross-tenant system level view, the three types of operations co-exist as well, and read operations dominate the system in most of the time. Most of operations for service providers are monitoring, and auditing, hence the system can get the statistic information of any chunks easily using P^2's partitioning model.

For the tenant-specific operations, as we discussed earlier, there are three types of operations including read/write/mix operations, and one can find out the benefit of using P^2 easily.

5 Quadpartite Recovery Model

The quadpartite recovery model (QRM) uses four major components: OC (ontology component), MC (metadata component), DC (data component) and IC (index component). In a traditional cloud environment, data and metadata are at least

triplicated into different chunks to ensure reliability and availability. This is good as long as at least one chunk is available in case of failures, while data lost can be troublesome, metadata lost can cause significant issues for a cloud. This paper extends the traditional approach by having redundant metadata information among the four components so that metadata can be recovered in case all the metadata chunks are lost.

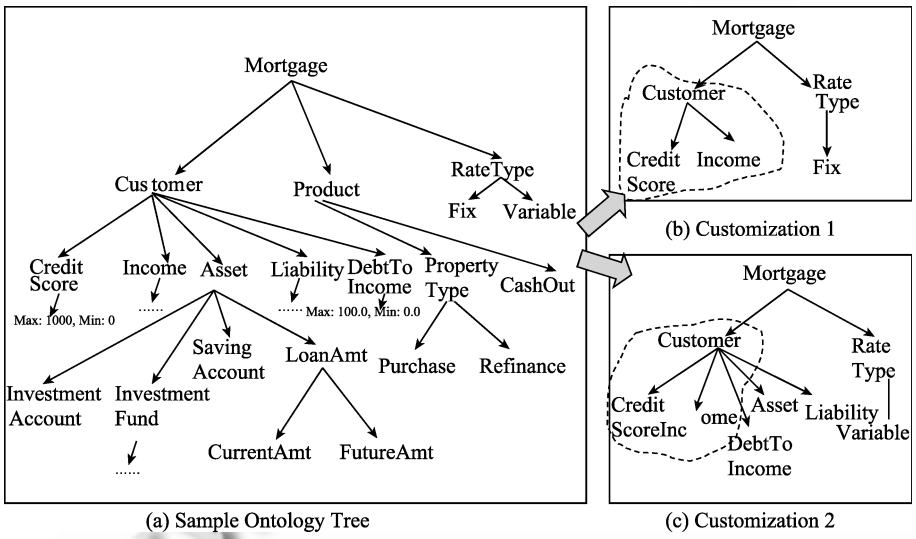

Figure 12. The mortgage ontology

Definition 1. OC contains information such as entity, attributes, relations, classicalists and constraints which include those information(e.g. entity, attribute, relational and classification constraints). Note that constraints can be mapped into metadata in MC.

Definition 2. MC contains the schema description, the data dictionary that stores the constraints, triggering rules, and other relational constraints.

Definition 3. DC contains the actual data, but also metadata in the MC.

Definition 4. IC contains the indices generated for the actual data. Indices are generated only for the data stored in the local node so that the overhead of the distribution of redundant data can be minimized.

The indices of the data, which can be considered as a summary of the existing data, can be used to guide the recovery of the data. Besides, the indices are also used to continuously validate the recovered data to ensure the correctness of data recovery. Also, the indices can be effectively used to detect the inconsistency of data stored in different nodes in the cloud. Any information stored in each component will have redundancy. Note that currently, a cloud environment does not have an OC, but it can be added for customization, service specification, classification, and now metadata and data recovery.

Note that in most current SaaS design, the DC will contain just data, but QRM adds metadata into the DC. This increases the redundancy of metadata as a failure in metadata may cause significant troubles and thus extra redundancy for metadata. In addition, IC is used to help to accelerate the recovering process as well as validating the recovered components.

Algorithm 1 Recover metadata table from ontology

RecoveryMetadata(OC oc)
INPUT: Ontology Trees T_1, T_2, $\cdots$, T_i, describing E
OUTPUT: The metadata table

1: $T_c = T_0$
2: **for** $j = 1$ to i **do**
3: Compare tree T_c with T_j
4: Extract the common tree T_{cj}
5: $T_c = T_{cj}$
6: **end for**
7: Convert tree T_{cj} to E's metadata table
8: **for** $j = 0$ to i **do**
9: Compare tree T_j with T_{cj}
10: Extract the difference between T_j and T_{cj} as T_k
11: Convert tree T_k to E's metadata table
12: **for** all constraint in tree T_k **do**
13: Store constraint into constraint table
14: **end for**
15: **end for**

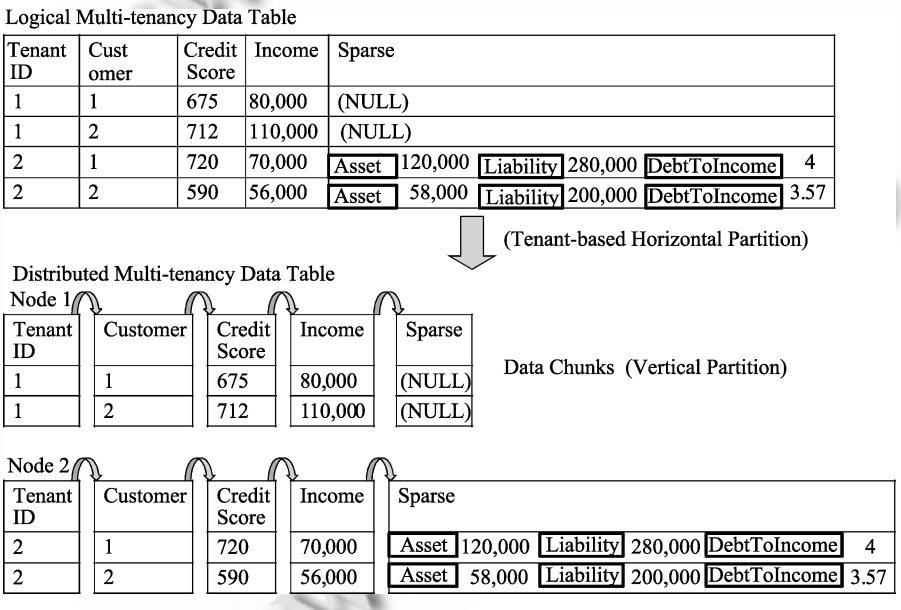

Figure 13. The distributed data table

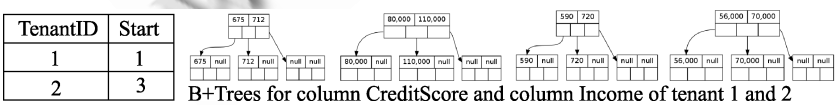

Figure 14. Some indices of the tenant data tables

Information stored in an OC will be at least triplicated into different nodes or chunks like GFS. Furthermore, each entry in the OC will be mapped to the corresponding entries in the MC and DC to ensure completeness. In other words, whenever an entry is added or changed in the OC, the corresponding entries in the MC and DC need to be updated. In this way, in case that when parts of or the entire OC failed, the OC can be reconstructed using information in MC and DC. Note that entities, attributes, and relations can be easily mapped into data tables, and classification and constraints such as entity, attribute, relational, and classification constraints can be mapped into metadata. Furthermore, some constraints can be additionally mapped into triggering rules associated with data. Thus, the entire OC information may be reconstructed using information in DC and MC.

To avoid the transmission of IC when DC are duplicated, the IC is only generated from DC locally. Whenever the DC is updated, IC will continuously be updated to reflect the change in DC. In this way, the traffic between nodes are minimized, and the consistency of DC can be quickly checked by using IC. IC also provides the ability of cross- checking in different granularity. To provide a lightweight cross checking, only the IC needs to be examined. And a heavyweight cross checking will be to conduct a comparison between the DC.

Due to the distributed nature of cloud computing, an OC may be stored in different chunks for reliability. Thus, if there is any cross reference, the 2-way relation will be stored in two places in the OC. For example, if a customer is related to a product, the relationship will be stored in both customer node and product node. The name of the relationships, attributes of the relationships, and the names of the parties will all be stored in both parties. In this case, whenever one of them fails, it can be recovered from the other.

For each chunk of data, mechanisms such as parity and checksum can be used to indicate its integrity, as are done in RAID. Additional benefits of this approach are the acceleration of the lookup operations because the relationship information is stored locally. Figure 15 shows the OC, MC, DC and IC in QRM. Whenever, one of the component fails, it can be recovered from the other two components.

The QRM recovery mechanisms can be illustrated through a mortgage example below. Figure 12 represents the OC in the mortgage example. The corresponding MC and DC are shown in Fig.9 and Fig.10. Figure 13 shows the mortgage data partitioned according to the hybrid partitioning proposed. Figure 14 shows the index of the mortgage data.

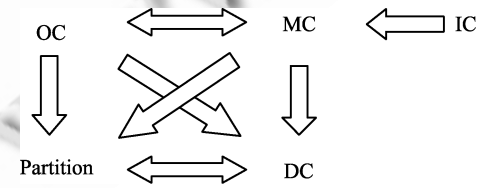

Figure 15. Quadpartite

	Information	Redundancy
OC	Entities, attributes, relations, classifications, constraints, including entity, attribute, relational and classification constraints.	1. Ontology information can be duplicated and stored in different chunks.
MC	Schema description, data dictionary, triggering rules, and relational constraints.	1. Metadata are duplicated and stored into different chunks; 2. Metadata can also be expressed as triggering rules associated with MC.
DC	Data and their metadata.	1. Data are duplicated and stored into different chunks; 2. Metadata store together with their data in the same table, and as data are duplicated metadata are also duplicated into the corresponding chunks; 3. Metadata are also expressed as triggering rules associated with DC.

Figure 16. Summary of the OC, MC, DC, and IC

	Sample Recoveries
Failure of the OC	For example, if the customer node in the ontology tree as shown in Figure 10 is lost, one can recover it by: a) Retrieving the customer metadata and data stored in DC and reconstruct the customer node; b) Constructing the customer node information using metadata from the MC. Both a) and b) can use using Algorithm 1, and the result from these two constructions can be cross verified.
Failure of the MC	For example, when the metadata of the customer table is lost, the customer metadata can be recovered by: a) Retrieving the customer node from the OC, and construct the metadata using Algorithm 2; b) Retrieving the duplicated copy of customer metadata stored in the DC. The result generated by alternative a) can be cross verified with that retrieved from the DC.
Failure of the DC	For example, when the data in the customer table are lost, the DC can be recovered by the duplicated chunks of customer table stored in other nodes and the metadata can be recovered by retrieving them from the MC or from ontology and they can be cross verified. The recovery of the DC can be accelerated by using the existing IC. The consistency of the DC can be cross checked by the IC.
Failure of the IC	For example, when the indices in the customer table are lost, the IC can be generated while accessing the DC, as it is a byproduct of accessing the DC.

Figure 17. Sample recovery

In case of an MC failure, one can recover the MC by using the information in the OC. As can be seen in Fig.12 and Fig.10, the OC used in the mortgage domain stores the relationship between the entities. If the customer table is lost, one can recover it from the OC. From the OC, one can know that the credit score and the income are

Algorithm 2 Recover ontology from data and metadata

```
RecoverOntology(Node n)
INPUT: Data tables describing entities E_1, E_2, ..., E_i
INPUT: Data tables describing entities R_1, R_2, ..., R_i
INPUT: Metadata tables containing constraints M_1, M_2, ..., M_i
OUTPUT: The ontology tree
 1: T = n
 2: for all E_i related to n do
 3:    Append child node m to n
 4:    for all R_i related to m do
 5:       Append child nodes from R_i to m
 6:       if m represent a column then
 7:          Lookup the constraints in M_1, M_2, ..., M_i
 8:          Append the constraint to m
 9:       end if
10:    end for
11: end for
12: for all child nodes m of n do
13:    RecoverOntology(m)
14: end for
```

common for the tenants by comparing the two customized ontology trees in Fig.12. Then one can restore the shared column "Customer", "Credit Score" and "Income" in the MC. For the rest of the columns, one knows that it belongs to the sparse part in the MC. Ontology tree merge / comparison operation need to be conducted to accomplish the reconstruction. Algorithm 1 describes the process of recovering the MC and OC. Using the mortgage example as shown in Fig.10, one can retrieve tree T_1 and T_2 customized by the tenants. For the common part of the two trees (the customer, credit score and income columns), one can fill the metadata table with the corresponding information available in the OC. After that, one can fill the description for the sparse columns according to the difference between T_1 and T_2.

The MC failure can also be recovered by using the information in the DC. As the DC contains the same metadata information, except the metadata will be stored together with the data, and some metadata may be mapped to enforcement rules to be triggered when the corresponding data are changed.

In the cases of an OC failure, the OC can be reconstructed using the MC, in a reverse manner as the process of recovering the MC. In this process, one can also use the DC. For each node in the ontology tree, all data table describing the node will be collected. Then starting from the root of the tree, one can append the nodes to the ontology tree using the data table describing the relationships between entities. After that, the data tables are used to append the child leaves to the ontology tree. Algorithm 2 describes this process.

In case of a DC failure, one can resort to the duplicates of the DC for data recovery. The metadata in the DC can be recovered from the MC and OC. The recovery process will be different because now metadata are stored together with data, and some meta information such as constraints need to be mapped into enforcement rules to be triggered when the data are changed. Besides resorting to duplicates

Algorithm 3 Recover DC with IC

RecoverData(IC ic)
INPUT: Index tables related to data table T
OUTPUT: Data table T to be recovered

```
1: canRecover = true
2: for all column c in table T do
3:    if c is not covered in ic then
4:       canRecover = false
5:       break
6:    end if
7: end for
8: if canRecover == true then
9:    RecoverDataByIC(ic)
10: else
11:    data = RecoverDataByDuplicates()
12:    CrossCheckingWithIC(data, ic)
13: end if
```

Algorithm 4 Recover DC using IC

RecoverDataByIC(IC ic)
INPUT: Index tables related to data table T
OUTPUT: Data table T to be recovered

```
1: Sort all columns in ic by tenant ID
2: for all column c in ic do
3:    Construct record r with c
4:    Insert r into table T
5: end for
```

of DC, one can also leverage the IC to accelerate the DC recovery process. As a summary of the DC, the IC can be used to re-construct the DC. Different ICs for the different part of the DC can re-construct the DC "collaboratively" because they contains different information from the DC. Note that after a data table recovery, the recovered data might not be consistent with the existing constraints (metadata). In this case, recovering the constraints using OC and/or MC may be necessary. And the IC can be also be used for cross checking.

Example 5.1. To recover the DC from the IC, consider the same mortgage data in Fig.13 and the indices in Fig.14. Suppose one tries to recovery data of tenant 1, the algorithm first retrieves the Bitmap indices of tenant 1. Then using the bitmap for column income, one can identify the row of the data. Then the system can write the value retrieved from the bitmap continuously to the DC.

Example 5.2. To recover the OC and MC, consider the same mortgage data in Fig.14. Suppose in the ontology, the attribute "DebtToIncome" of is missing due to node failure. By looking at the metadata table in Fig.13, the algorithm can discover that a branch "DebtToIncome" is missing. Thus the branch is added according to the information retrieved. If any other attribute is missing in the ontology, they can be recovered in the same way. The MC can be recovered in a similar manner if they are lost due to node failure.

Algorithm 5 Cross checking DC by IC

RecoverDataByDuplicates(Duplicates d, IC ic)
INPUT: Index tables related to data table T
INPUT: Duplicates d of table T
OUTPUT: Data table T to be recovered

```
1: T = d
2: for all column c in table T do
3:   if c is covered in ic then
4:     Checking the validity of c with ic
5:     if Recovered c is not valid then
6:       Cross checking c with other duplicates
7:     end if
8:   end if
9: end for
```

Example 5.3. Recovering the IC is relatively easy because the index is produced by accessing the data, as this is the way index constructed. To recovery the index as shown in Fig.14, one can just scan through the data table in Fig.13 and generate the indices.

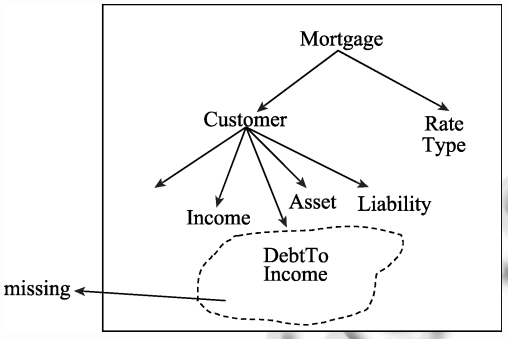

Figure 18. Recovering OC

Figure 17 summaries the process and the required components in the proposed recovery mechanism. Note that one significant advantage of the QRM is that even if the replication are lost, one can still recover metadata by re-construction from other components. Algorithm 1 and Algorithm 2 demonstrated how the sample examples can be resolved as listed in Fig.17. More discussion about the mapping between metadata and ontology can be found in Refs.[42, 58, 13].

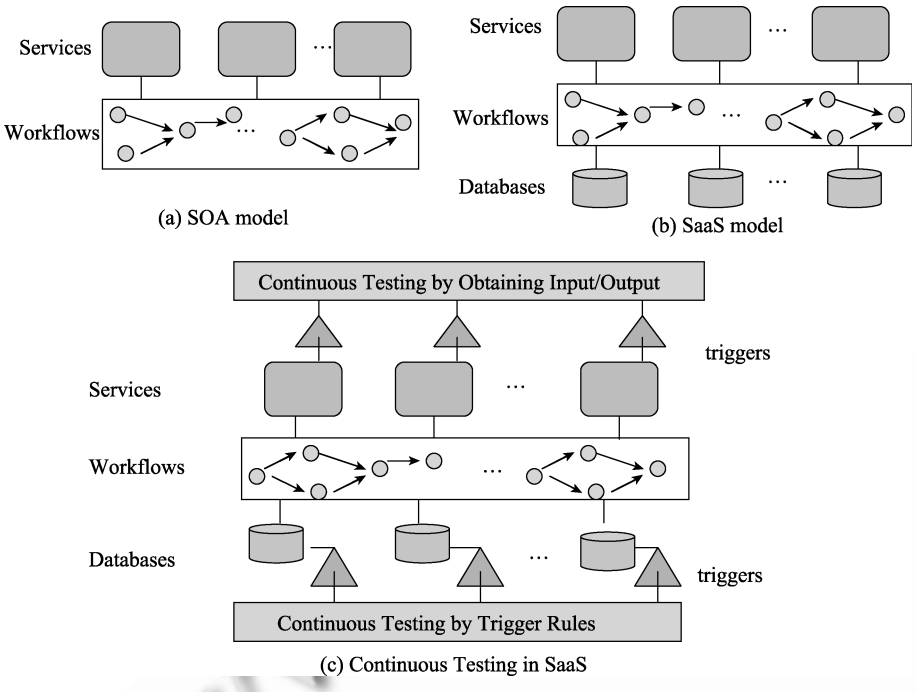

Figure 19. Continuous testing model in SaaS

6 Built-In Continuous Testing for SaaS

A challenging problem faced by software developed in the SaaS model is the testing of SaaS software. Since SaaS software have to adopt MTA requirement, the number of users is increasing dramatically. Requirements for Diverse features of SaaS software is fast evolving as new users are entering the system, as well as old users are requiring new functionalities. Thus the evolvement of the SaaS software need to be fast, while the quality of software needs to be ensured in an evolvement way.

Traditional testing practices need to conduct testing activities after all development activities are completed. To test the modified part of a software can cost longer time, as well as costs more labors. Such a sequential develop-test process is insufficient to satisfy the requirement fast involvement posted by the MTA and SaaS model. Therefore, this paper proposes an embedded capability of *continuous testing* in the SaaS framework, to address the testing challenges introduced by SaaS model.

6.1 Continuous testing

An effective model of automated testing is continuous testing. It can also be part of the TDD (Test-Driven Development) process. Continuous testing implements continuous processes of applying quality control - small pieces of effort applied frequently, in the process of software development. Continuous testing has been proposed and can be applied in various aspects in software development. For example, as proposed in Ref.[45], tests run 24 hours a day, 8 days a week and the results of these testings are efficiently processed. While in Ref.[41], continuous testing is integrated into eclipse as a tool for continuous code verification when source code changes. It uses excess

cycles on a developer's workstation to continuously run tests in the background, providing rapid feedback about test failures as source code is edited. A radical design choice in the Google Chrome OS is its incorporation of continuous verification. Given the extensive usage of continuous testing, its desirable that the SaaS framework also provides built-in continuous testing capability.

Besides continuous testing, testing SaaS software can also be collaborative by nature since it is usually developed with a service oriented architecture. In the framework proposed in this paper, we proposed to embed built-in testing capability in the SaaS framework. We provide a collaborative testing environment by generating test scripts in a collaborative manner. We integrate continuous testing with the storage layer, by leveraging database triggering rules. We also propose algorithms so that integrating testing and intelligent testing can be conducted within our framework. Figure 19 show the evolution of different models, from SOA, SaaS to continuous testing model.

6.2 Test cases generation from metadata

Test cases can be generated by examining metadata, e.g. Income length of customer must be 64 bits or so, hence some simple test cases will be randomized with 64 bits. One can generate a collection of customers of 64 bits, another collection with 128 bits or any other bits.

Random number from 0 $2^{64}-1$, e.g. another set is negative numbers, and greater than $2^{64}-1$, so we have three set of values, one valid and other two are invalid. The boundary value test cases can be generated from {-2, -1, 0, +1, +2} around boundary of the constraints, specifies by the metadata. For example, according Fig.9, credit score > 0 is an invalid testing put, credit score $= 0$, 1 are boundary test values. Several test case generations from ontology including constraints have been proposed[6,5] and can be used in our framework.

Based on the WebStra's framework, test cases can be ranked[52], and based on the importance, and history, test result oracle can be established by voting[7], test case dependency can be automatically analyzed using the test results based on statistic techniques. without canalizing software structures, a large collection of test cases can be constructed, ranked and evaluated on a continuous bases.

6.3 Collaborative testing

Testing SaaS software can be collaborative by nature since it is usually developed with a service oriented architecture. A collaborative testing environment can generate test scripts in a collaborative manner as shown in Fig.20. Test scripts can be contributed by different parties or automated generated in a multi-tenancy way.

6.4 Intelligent testing

SaaS data (such as code bases, execution logs, mailing lists, and bug databases) is a good wealth of information about an application's lifecycle. Using data mining techniques, one can fully explore the potential of this valuable data, and manage their projects in a cost effective way, produce higher-quality software systems with less bugs. Two types of information are available as data resources as shown in Fig.21: (1) Historical repositories: including source control repositories, bug repositories, and

communications records of project evolution and etc. It captures dependencies between project artifacts (e.g. functions, documentation files, and configuration files). Not only handling static or dynamic code dependencies, one has to consider implicitly dependency, e.g. change of writing data may require reading data code change implicitly. Also it can be used to track the history of a bug or a feature, determine the expected resolution time according to previously closed bug resolution history. (2) Real-time repositories: including deployment logs with execution information and system usage logs from multi-tenancy. By monitoring the execution, one can find out the dominant execution or usage from logs, and tune the system performance accordingly. Similarly, one can mine the dominant APIs usage patterns by monitoring code repositories.

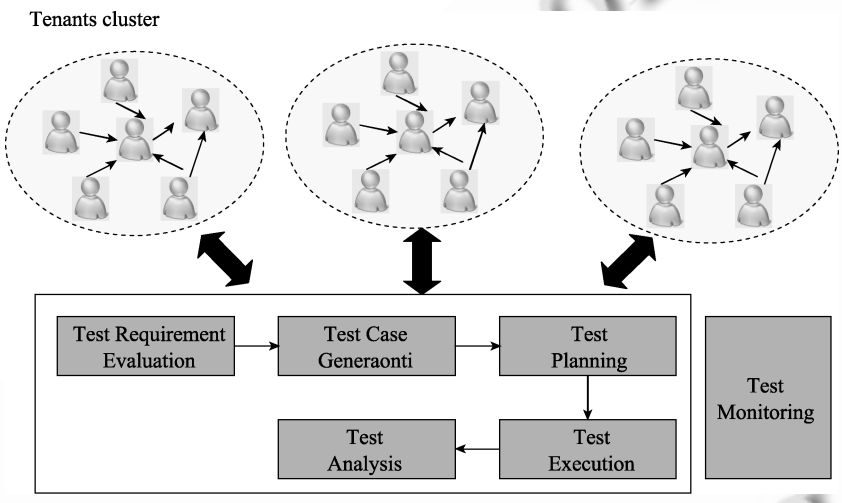

Figure 20. Collaborative testing

6.5 Policy enforcement

Policies represents the expected software behavior, which are enforced at runtime to ensure that the software execution conforms to the requirements. They are derived from business goals and service level agreements(SLA) in enterprises, which are "rules governing the choices in behavior of a system"[42]. Policies includes obligation policies(event triggered condition-action rules), authorization policies(define what services or resources a subject can access) and etc. This paper is focus on obligation policy to manage SaaS testing process. The *Obligation Policy (OP)* defines a tenant's responsibilities, what activities a subject he must (or must not) do. In general, obligation policies are event-condition-action rules (ECA) as trigger rules, in the format of

On *Event* ***If*** *Condition* ***Do*** *Action*

The event part specifies when the rule is triggered; the condition part determines if the data are in a particular state, in which case the rule fires; the action part describes the actions to be performed if the rule fires. ECA systems receive inputs (mainly in the form of events) from the external environment and react by performing actions that change the stored information (internal actions) or influence the environment itself (external actions).

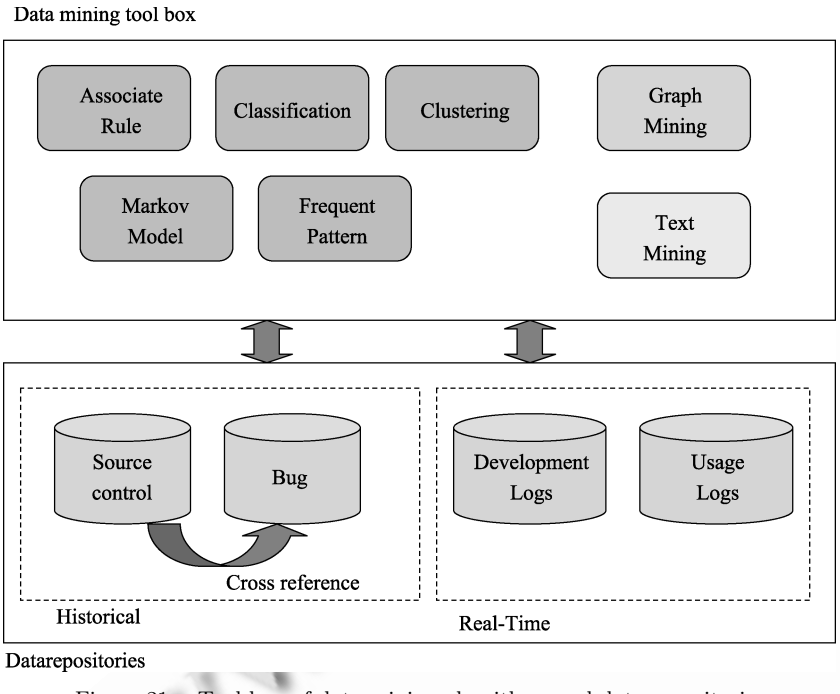

Figure 21. Tool box of data mining algorithms and data repositories

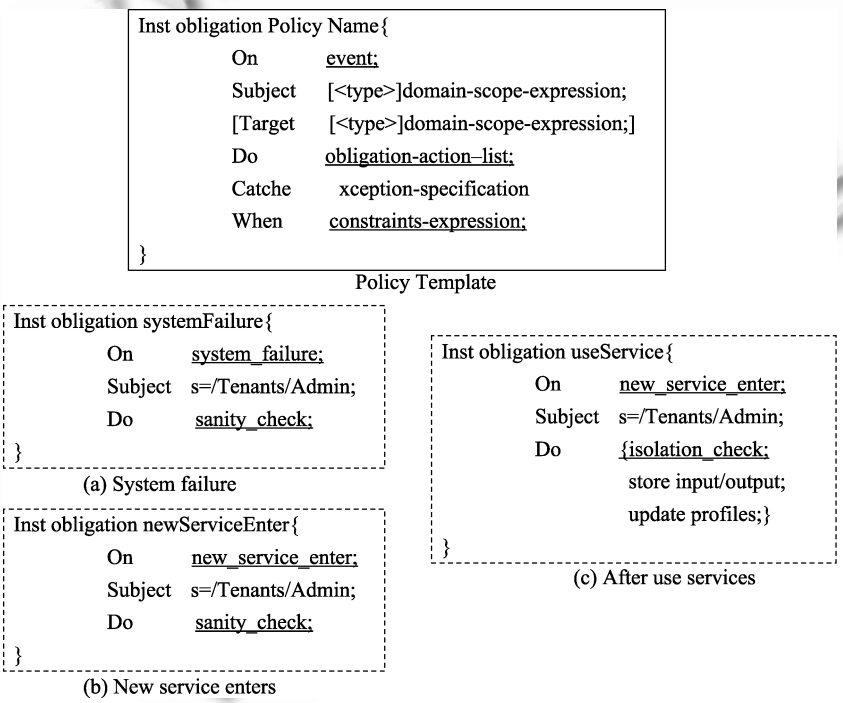

Figure 22. Sample policy specifications

Two general ways to address the faults using trigger rules, one emphasized prevention, e.g. developing a formal trigger rule to ensure the decidability and complete-

ness of a trigger rule system, which prevents anomalies. The second is to design a mechanism for handling various faults or failures during the execution of trigger rules, e.g. develop sophisticated plans for any possible results, which either eliminate the adverse effects or minimize the bad effects. This paper uses the second method, and proposes policy enforcement framework, which not only uses trigger rules, but also contingency plans.

SaaS are applied to increasing complicated, non-conventional application areas with real-time constraints, the probability of faults during the execution of trigger rules increase greatly. A trigger service in SaaS become increasingly complicated in handling the faults, such as failures and aborts, which may occur during the execution of SaaS customization. This paper models failures, aborts and other fault situations as events in the ECA paradigm, hence the contingency plans for handling fault events can be modeled as trigger rules.

6.6 Policy enforcement triggering rules

Policies are often enforced in service application when a service is been involved, for example, WS policy, XACML[33] and other policy standards, however, the policy used in the paper are derived from constraints, in the metadata and they may need to be enforced whenever data are changed, other than a service is involved[49], and also in the SaaS environment, multiple threads and services may be active at a given time, and may cause multiple data to be access or updated concurrently, and thus one needs different policy triggering rules, other than the traditional service invocation events. The following events are selected sample of policy enforcement triggering events.

1. There is a failure in the system somewhere, this is for sanity check; (Fig.20(a))
2. Before a service will be used (to ensure that the service is in a good shape, this is similar to acceptance testing); (Fig.20(b))
3. After a service has been just used (to ensure that the running does not affect the software), and store the input/output pair, to update the profile; (Fig.20(c))
4. Whenever a new service with the same service specification arrives;
5. Whenever a new application is created to specify that it intends to use the service (this is equivalent to testing during development)
6. If the service is replaced by another one as the previous one has some bugs or performance issues;
7. Certain time period has passed. For example, one week has passed, and the system is not sure that something is wrong, this is more like a sanity checking;
8. Whenever the new resources are added into SaaS during execution; new resources may cause issues, and need to work on scalability issues (scale out);
9. Whenever an existing resources is removed from the SaaS during execution, a reduce resource may cause issues, and need to work on scalability issues (scale in);

10. Whenever the cloud platform has a change in configuration: to ensure scalability issues (scale up and down), to maintain performance and so on

11. Whenever the output produced does not match with the predicted output;

12. Whenever a new input that has not occurred before arrive, and the new input may reveal new bugs not known;

Some sample rules can be specified as shown in Fig.20.

6.7 Other SaaS testing techniques

Many other possible testing techniques can be applied to SaaS, for example collaborative testing scripts generation, in which test scripts can be contributed by different parties or automated generated in a multi-tenancy way.

Integration testing, in which an integration testing script can be considered as a service testing by fixed the other services, and uses a series of test cases, and just test the specific service for regression testing. Once one collects enough information, he can have a good set of regression test scripts for the service.

According to previous test results, if one saves all the history information, more interesting mainlining algorithm, e.g. Markov model[52], similarity based prediction, collaborating profiling and etc can be applied to further improve the testing performance. Also Test scripts and test cases can be ranked based on test run, and dependency of test cases can be analyzed.

7 Conclusion

SaaS is characterized by its multi-tenancy architecture and its ability to provide flexible customization to individual tenant, which brought up various challenging problems, such as the testing of software developed with the SaaS model and built-in recoverability. This paper presents a unified and innovative multi-layered customization framework supporting continuous testing and recoverability. Different database partitioning strategies are offered for customization. Ontology is used to derive customization and deployment information to tenants and to support continues testing and recoverability. In the future, more testing techniques will be investigated to further improve the robustness of SaaS framework. A simulation of two-layer partitioning model will be investigated to further evaluate the proposed model performance.

References

[1] Aulbach S, Grust T, Jacobs D, Kemper A, Rittinger J. Multi-tenant Databases for Software as a Service: Schema-mapping Techniques. SIGMOD'08. ACM. New York, 2008. 1195–1206.
[2] Amazon. Amazon elastic compute cloud (amazon ec2). 2010. http://aws.amazon.com/ec2/.
[3] Zhang K, Zhang X, Sun W, Liang HQ, Huang Y, Zhen LZ , Liu X Z . A Policy-Driven Approach for Software-as-Services Customization. ICEBE. 2007. 123–130.
[4] Brantner M, Florescu D, Graf D, Kossmann D, Kraska T. Building a Database on S3. SIGMOD '08. ACM. New York, NY, USA. 2008. 251–264.
[5] Bai XY, Lee SF, Tsai WT, Chen YN. Ontology-Based Test Modeling and Partition Testing of Web Services. ICWS'08. IEEE Computer Society. Washington, DC, 2008. 465–472.
[6] Bai XY, Liu YL, Wang LJ, Tsai WT, Zhong P D. Model-Based Monitoring and Policy Enforcement of Services. SERVICES I. 2009. 789–796.

[7] Bai XY, Wang YB, Dai GL, Tsai WT, Chen YN. A Framework for Contract-based Collaborative Verification and Validation of Web Services. CBSE'07. Springer-Verlag. Berlin, Heidelberg. 2007. 258–273.
[8] Boncz PA, Zukowski M, Nes N. MonetDB/X100: Hyper-Pipelining Query Execution. CIDR. 2005. 225–237.
[9] Chong F, Carraro G. Architecture Strategies for Catching the Long Tail. 2006.
[10] Chong F, Carraro G, Wolter R. Multi-Tenant Data Architecture. June 2006.
[11] Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE. Bigtable: a Distributed Storage System for Structured Data. OSDI'06, USENIX. Association. Berkeley, CA, 2006. 15–15.
[12] Google Data Center. http://www.youtube.com/watch?v=zRwPSFpLX8I.
[13] Cullot N, Ghawi R, Ytongnon K. DB2OWL: A Tool for Automatic Database-to-Ontology Mapping. SEBD 2007. Citeseer. 2007. 491–494.
[14] Ceri S, Navathe S, Wiederhold G. Distribution Design of Logical Database Schemas. IEEE Trans. Softw. Eng., 1983, 9(4): 487–504.
[15] Ceri S, Pelagatti G. Distributed Databases Principles and Systems. McGraw-Hill. Inc., New York, NY, USA. 1984.
[16] Cooper BF, Ramakrishnan R, Srivastava U, Silberstein A, Bohannon P, Jacobsen H, Puz N, Weaver D, Yerneni R. PNUTS: Yahool's Hosted Data Serving Platform. Proc. VLDB Endow. 2008, 1(2): 1277–1288.
[17] Das S, Agrawal D, El Abbadi A. G-Store: a Scalable Data Store for Transactional Multi Key Access in the Cloud. SoCC '10. ACM. New York, NY, 2010. 163–174.
[18] DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W. Dynamo: Amazon's Highly Available Key-value Store. SOSP'07. ACM. New York, NY, 2007. 205–220.
[19] Eadon G, Chong EI, Shankar S, Raghavan A, Srinivasan J, Das S. Supporting Table Partitioning by Reference in Oracle. SIGMOD '08: Proc.of the 2008 ACM SIGMOD International Conference on Management of Data. ACM. New York, NY, 2008. 1111–1122.
[20] Essaidi M. ODBIS: Towards a Platform for On-demand Business Intelligence Services. EDBT'10: ICDT Workshops. ACM. New York, NY, 2010. 1–6.
[21] Force.com. http://force.com/.
[22] Google. Google app engine. 2010. http://code.google.com/appengine/.
[23] Apache Software Foundation. Hbase: Bigtable-like structured storage for hadoop hdfs. 2009. http://hadoop.apache.org/hbase/.
[24] Microsoft Building New Data Center in Quincy. http://www.datacenterknowledge.com/archives/2010/05/19/microsoft-building-new-data-center-in-quincy/.
[25] iTKO. itko lisa. http://www.itko.com/default.jsp.
[26] Kraska Tim, Hentschel M, Alonso G, Kossmann D. Consistency Rationing in the Cloud: Pay Only When it Matters. Proc. VLDB Endow. 2009, 2(1): 253–264.
[27] Kossmann D, Kraska T, Loesing S. An Evaluation of Alternative Architectures for Transaction Processing in the Cloud. SIGMOD'10. ACM. New York, NY, 2010. 579–590.
[28] Karger D, Lehman E, Leighton T, Panigrahy R, Levine M, Lewin D. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. STOC'97. ACM. New York, NY, 1997. 654–663.
[29] Livny M, Khoshafian S, Boral H. Multi-disk Management Algorithms. SIGMETRICS Perform. Eval. Rev. 1987, 15(1): 69–77.
[30] Li HB, Shi YL, Li QZ. A multi-granularity customization relationship model for SaaS. WISM'09. IEEE Computer Society. Washington, DC, USA. 611–615.
[31] Mehta M, DeWitt DJ. Data Placement in Shared-nothing Parallel Database Systems. The VLDB Journal, 1997, 6(1): 53–72.
[32] Mietzner R, Leymann F. Generation of BPEL Customization Processes for SaaS Applications from Variability Descriptors. SCC '08. IEEE Computer Society. Washington, DC, USA. 2008. 359–366.
[33] Moses T. eXtensible Access Control Markup Language TC v2.0 (XACML), February 2005.

[34] Olston C, Reed B, Srivastava U, Kumar R, Tomkins A. Pig latin: a Not-so-foreign Language for Data Processing. SIGMOD '08. ACM. New York, NY, USA. 2008. 1099–1110.
[35] Pavlo A, Paulson E, Rasin A, Abadi DJ, DeWitt DJ, Madden S, Stonebraker M. A Comparison of Approaches to Large-scale Data Analysis. SIGMOD'09. ACM. New York, NY, USA. 2009. 165–178.
[36] Rowstron A, Druschel P. Pastry: Scalable, Decentralized Object Location, and Routing for Large-scale Peer-to-peer Systems. MIDDLEWARE. 2001. 329–350.
[37] Ramakrishnan R, Gehrke J. Database Management Systems. 2007.
[38] Stonebraker M, Abadi DJ, Batkin A, Chen XD, Cherniack M, Ferreira M, Lau E, Lin A, Madden SR, O'Neil EJ, O'Neil PE, Rasin A, Tran N, Zdonik SB. C-Store: A Column-Oriented DBMS. VLDB. Trondheim, Norway. 2005. 553–564.
[39] Salesforce.com. Salesforce. 2010. http://www.salesforce.com/.
[40] Saff D, Emst MD. Continuous Testing in Eclipse. 2nd Eclipse Technology Exchange Workshop (eTX). Barcelona, Spain. March 2004.
[41] Sloman M. Policy Driven Management For Distributed Systems. Journal of Network and Systems Management, 1994, 2: 333–360.
[42] Sung S, McLeod D. Ontology-Driven Semantic Matches between Database Schemas. 2006. 6–6.
[43] Stonebraker M, Madden S, Abadi DJ, Harizopoulos S, Hachem N, Helland P. The End of an Architectural Era: (It's Time for a Complete Rewrite). VLDB'07. VLDB Endowment. 2007. 1150–1160.
[44] Smith E. Continuous Testing. Proc. of the 17th International Conference on Testing Computer Software. 2000.
[45] Stoica I, Morris R, Liben-Nowell D, Karger DR, Kaashoek MF, Dabek F, Balakrishnan H. Chord: a Scalable Peer-to-peer Lookup Protocol for Internet Applications. IEEE/ACM Trans. Netw., 2003, 11(1): 17–32.
[46] Sobel W, Subramanyam S, Sucharitakul A, Nguyen J, Wong H, Klepchukov A, Patil S, Fox O, Patterson D. Cloudstone: Multi-platform, Multi-language Benchmark and Measurement Tools for Web 2.0, 2008.
[47] Sacca D, Wiederhold G. Database Partitioning in a Cluster of Processors. ACM Trans. Database Syst., 1985, 10(1): 29–56.
[48] Tsai, WT, Chen YN, Paul R, Zhou XY, Fan C. Simulation Verification and Validation by Dynamic Policy Specification and Enforcement. Simulation, 2006, 82(5): 295–310.
[49] Mary Taylor and Chang Jie Guo.Data Integration and Composite Busi- ness Services, Part 3: Build a Multi-tenant Data Tier with Access Control and Security. http://www.ibm.com/developerworks/data/library/techarticle/dm- 0712taylor/index.html.
[50] Tsai WT, Shao Q H, Li W. OIC: Ontology-based Intelligent Customization Framework for SaaS, 2010.
[51] Tsai WT, Zhou XY, Paul RA, Chen YN, Bai XY. A Coverage Relationship Model for Test Case Selection and Ranking for Multi-version Software. HASE. 2007. 115–112.
[52] Weissman CD, Bobrowski S. The Design of the Force.com Multitenant Internet Application Development Platform. SIGMOD'09. New York, NY, USA. ACM. 2009. 889–896.
[53] Weissman C. Behind the Scenes: Salesforce.com Chief Technology Officer on Cloud Architecture, 2009.
[54] Wikipedia. Ontology in computer science and information science. 2010. http://en.wikipedia.org/wiki/Ontology (information science).
[55] Xu ZM, Zhang SC, Dong YS. Mapping between Relational Database Schema and OWL Ontology for Deep Annotation. WI'06. 2006. 548–552.