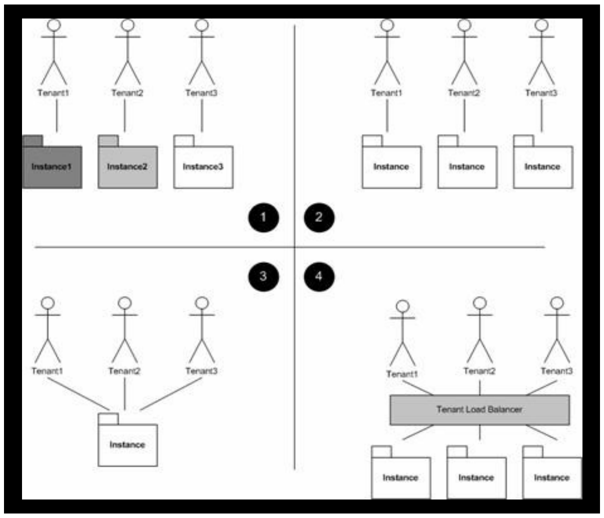
SaaS
Design Issues

SaaS Requirement

Customization

Configurable (but single tenant based)

Physical or Virtual Isolation



Networked, Centralized, Configurable, Multi tenant

Networked, Centralized, Configurable, Multi tenant, Scalable

SaaS and Traditional Software

		Software as a Product	Software as a Service
√	**Delivery**	Installed	Hosted
√	**Development**	Longer cycle, "big bang"	Short, continuous cycle
√	**Pricing**	Perpetual license + maintenance	Subscription (all inclusive)
	Allocation	Capitalized	Expensed
	Additional Costs	Installation, maintenance, customization, & upgrades	Configuration
√	**Platform**	Multi-version	Single Platform
√	**Updates**	Larger, less-frequent	Shorter, frequent
√	**Sales Focus**	Close the deal	Prove value in first 90 days
	Feedback Cycle	Long	Short
	Profits	Initial sale	Ongoing
	Success	New license revenue	Lack of churn

SaaS Design Issues

- SaaS multi-tenancy support
  - Multi-tenancy database design issues
  - Tenant isolation
- SaaS itself
  - How to convert single-tenant system to multi-tenant
    - New terminology: SaaSification
  - SaaS configurability issue
    - Tradeoff with performance
- Access control models for SaaS

Multi-Tenancy Database Design

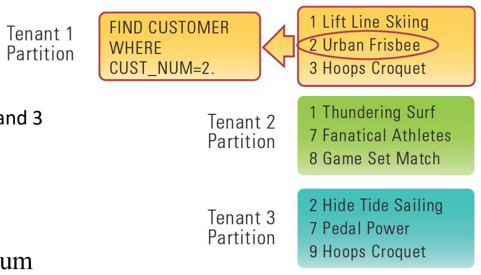
- DBs for SaaS
  - Always need to maintain data for all users
    - The tenants, the users of each tenant, and other basic user info
    - Tenant/User specific configurations for the SaaS
    - SaaS application data
      - Including system wide data and user-specific data
  - How to organize the data?
    - Multiple tables, one for each tenant
      - Better isolation
      - Performance impact
    - Single table for all tenants

Multi-Tenancy Database Design

- DBs for SaaS
  - Performance impact with multiple tables
    - When there are too many tables (too many tenants) in a DB, performance degrades
    - Resources in DBs are allocated in a per-table basis
      - E.g., IBM DB2 V9.1 allocates
        - » 4 KB of memory for each table
        - » Buffer pool (cache) allocated for each table
    - ⇒ Cause resource contention when the number of tenants grows
  - Single table design breaks isolation, what issues are to be considered?

Multi-Tenancy Database Design

- DB for SaaS
  - Requirement: indexing/sequencing preserving
  - User query is always toward its own tenant domain
    - User query should be translated to include (tenant-id = x)
  - Indexing
    - For the query example
      ⇒ "Urban Frisbee"
      Even though both Tenants 1 and 3 both have cust-num = 2
  - Sequencing
    - E.g., User of Tenant 3 accessing "lowest" cust-num
      ⇒ 2 (Even though lowest cust-num in the global table is 1)

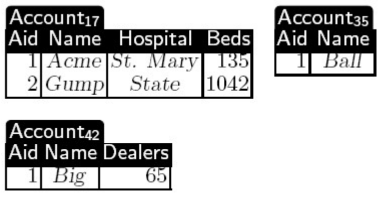


Multi-Tenancy Database Design

- DBs for SaaS
  - Configurability
    - Allow each tenant to keep a different user table schema
      - With some common and some different attributes
    - Each tenant may keep a different SaaS application data schema
  - When the schema for all tenants is the same
    - Only need to consider indexing and sequencing preservation
  - When the schema for each tenant is different
    - Different table design can result in different performance tradeoffs
    - Private tables
    - Universal table
    - Other improved designs

Multi-Tenancy Database Design

- Private tables
  - One table for each tenant
    - Can also have more than one tables for each tenant
  - Retain the individual table logic, provide good extensibility
    - No overhead for integrating new tenants with new schemas
  - Provide good isolation
  - Performance concern



Account17

Aid	Name	Hospital	Beds
1	*Acme*	*St. Mary*	135
2	*Gump*	*State*	1042

Account35

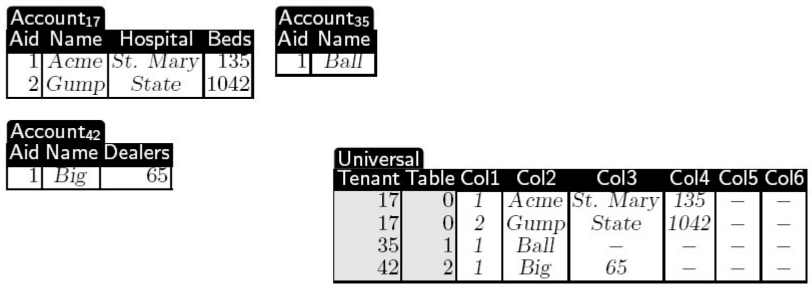
Aid	Name
1	*Ball*

Account42

Aid	Name	Dealers
1	*Big*	65

Multi-Tenancy Database Design

- Universal table
  - Wide rows, DB has to handle many null values
    - Commercial DB can handle null values fairly efficiently, but will require additional memory
  - Can only have a generic data type for each column



Account17

Aid	Name	Hospital	Beds
1	*Acme*	*St. Mary*	135
2	*Gump*	*State*	1042

Account35

Aid	Name
1	*Ball*

Account42

Aid	Name	Dealers
1	*Big*	65

Universal

Tenant	Table	Col1	Col2	Col3	Col4	Col5	Col6
17	0	*1*	*Acme*	*St. Mary*	*135*	–	–
17	0	*2*	*Gump*	*State*	*1042*	–	–
35	1	*1*	*Ball*	–	–	–	–
42	2	*1*	*Big*	*65*	–	–	–

Multi-Tenancy Database Design

- Pivot table
  - Schema
    - Table index, row#, column#, value
    - Use mixed data types
      - ⇒ Only one pivot table
        - #rows in the pivot table
          = Σ #rows in private tables
          * average #columns per private table
        - #rows in a universal table
          = Σ #rows in private tables
    - To retain data types
      - ⇒ One table per data type

Account_{17}

Aid	Name	Hospital	Beds
1	*Acme*	*St. Mary*	135
2	*Gump*	*State*	1042

Account_{35}

Aid	Name
1	*Ball*

Account_{42}

Aid	Name	Dealers
1	*Big*	65

Pivot_{str}

Tenant	Table	Col	Row	Str
17	0	1	0	*Acme*
17	0	2	0	*St. Mary*
17	0	1	1	*Gump*
17	0	2	1	*State*
35	1	1	0	*Ball*
42	2	1	0	*Big*

Pivot_{int}

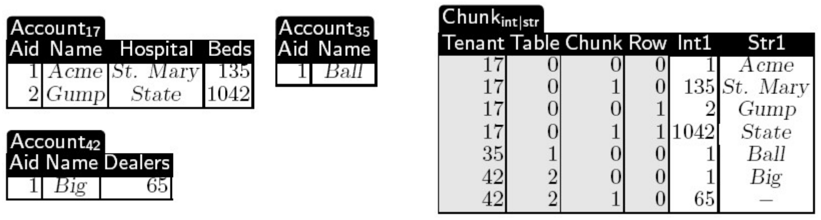
Tenant	Table	Col	Row	Int
17	0	0	0	1
17	0	3	0	135
17	0	0	1	2
17	0	3	1	1042
35	1	0	0	1
42	2	0	0	1
42	2	2	0	65

Multi-Tenancy Database Design

- Pivot table
  - Potential problems
    - Higher overhead on metadata
      - One cell requires 4 metadata columns (2 additional)
    - Performance depends on the use case scenario
      - Performs well if retrievals frequently retrieve a single column
      - Need to retrieve multiple rows if an entire row of the original table has to be retrieved
    - Expensive to reconstruct an original logical table (private table)
      - Note: Queries are issued toward the original logical table
      - If the query is complex ⇒ Need to reconstruct the logical table
        - » Select col1, col2, col3, … Where relation(col1, col2, col3, …)
      - Requires (N–1) joins to reconstruct an N column logical table

Multi-Tenancy Database Design

- Chunk table
  - Columns are categorized into chunks
    - Use chunk id, instead of column id (in this example, 2 chunks)
    - Can easily retain data types of the columns by proper design
  - Reduced overhead
    - This is similar to pivot table, but with a reduced number of rows
      - Reduce metadata overhead and private table reconstruction cost
        - » Reduction = # rows reduced = average #columns per chunk



Account_{17}

Aid	Name	Hospital	Beds
1	*Acme*	*St. Mary*	135
2	*Gump*	*State*	1042

Account_{35}

Aid	Name
1	*Ball*

Account_{42}

Aid	Name	Dealers
1	*Big*	65

$\text{Chunk}_{int|str}$

Tenant	Table	Chunk	Row	Int1	Str1
17	0	0	0	1	*Acme*
17	0	1	0	135	*St. Mary*
17	0	0	1	2	*Gump*
17	0	1	1	1042	*State*
35	1	0	0	1	*Ball*
42	2	0	0	1	*Big*
42	2	1	0	65	–

Multi-Tenancy Database Design

- Chunk table
    - Universal table and pivot table are existing approaches to deal with table consolidation, they are not specific for multi-tenancy
      - Pivot table was designed to handle DB for semi-structured data
        - » Such as XML based data or OODB
      - Pivot table has been used in Force.com PaaS architecture
    - Chunk table is designed specifically for multi-tenancy scenarios
  - Benefits
    - Chunk table can control the (#columns per chunk) parameter to balance the metadata overhead and overhead in #null entries
      - Balance between pivot table and universal table
      - Also balance the cost for logical table reconstruction

Multi-Tenancy Database Design

- Other multi-tenant DB design considerations
  - Perform cross tenant operations ⇒ Use a super tenant
  - Consider users who does not belong to any tenant want to use the SaaS individually ⇒ Add a default tenant for them
  - Hierarchical tenancy
    - E.g., in an enterprise, each branch is a tenant, and the enterprise is a higher level tenant
    - Potential solution for DB accesses
      - Use role-based concept, define roles within each individual tenant (e.g., X1, X2), as well as within the group (e.g., G)
      - If a user assumes role Xi, access is to the local tenant table
      - If a user assumes role G, access scope includes all tables of all tenants in the group

Tenant Isolation

- SSO isolation
  - Single sign-on is desirable
    - A user may integrate the SaaS with services in her/his private domain ⇒ Should not require the user to sign on multiple times
    - How to achieve SSO with isolation?
  - Keep a user table at the tenant site
    - = SaaS tenant private table
    - Maintain the complete user table can be a problem
  - Build a trust relation
    - Should not allow one user access private domain of another tenant
    - The trust relations should map tenant ids to tenant domains
  - How about SSO among multiple SaaSs?

Tenant Isolation

- Access isolation
  - A user of one tenant should not access the data of another tenant without the proper permission
  - Implicit filter based control
    - A user can never access the data of another tenant
    - If such access is desired, has to be delegated to a super user
  - Explicit filter based control
    - Explicitly assign the permission using a special access control list for any cross tenant access

Tenant Isolation

- Information isolation
  - Private tables: better isolation
  - Private tables are consolidated for performance gain
    - Encrypt critical tenant data to assure information isolation
      - May backfire performance due to decryption for each access, should only use "good enough" encryption
- Performance isolation
  - Bad SaaS performance of one tenant should not impact the SaaS performance of another tenant
    - Paper: Performance isolation and fairness for multi-tenant cloud storage

Tenant Isolation

- Performance isolation
  - Existing solutions
    - Use VMs for processing tenant requests
    - One VM dedicated to process the request of one tenant
      - Generally one VM for one type of SaaS service
      - A tenant can have multiple VMs, for multiple SaaS services
      - For each VM, some specific system or tenant data needed by the VM should be distribute with the VM
    - Use fair sharing or hard cap on the VMs ⇒ Too restrictive + Only local fairness, not global fairness cross all VMs of each tenant
  - In this paper
    - Use partitions instead of VMs (but we still use VMs)
    - Use several methods to assure global fairness without being too restrictive

Tenant Isolation

- Performance isolation
  - Weight and share assignment
    - Each tenant is assigned a global weight
      - Based on its estimated total demand and SLA level (⇒ payment)
    - Each VM of the tenant is assigned a share according to
      - Estimated resource demand (request rate * demand per request)
      - AND: Total shares on all VMs of a tenant = weight of the tenant
  - VM placement
    - Place the VMs of all tenants using a simple greedy algorithm, based on resource demands (shares) of the VMs

Tenant Isolation

- Performance isolation
  - How to enforce shares on each server
  - ⇒ Use fair queuing
    - In every round, the server allocates tokens to each tenant according to each tenant's local shares
    - Server maintains one queue for each tenant
    - When a request arrives at a server, add it to tenant's queue
      - If the tenant has used up its token ⇒ Requests wait in the queue
    - Use weighted round robin in each scheduling rounds
      - Retrieve from the queue with more tokens more frequently
      - E.g., A has 10, B has 15; Retrieve with probability 2/5, 3/5 for A, B
    - At the end of a round, sever refills tokens for all tenants
      - Otherwise, tenants can consume excess resources left idle by the other tenants without penalty

Tenant Isolation

- Performance isolation
  - Global fairness
    - Fair queuing is for local fairness ⇒ too restrictive
    - Initially estimated demand may fluctuate ⇒ Should regularly adjust share allocation among VMs on each server
      - D: resource demand; R: resource share allocation
      - VM_1: $E_1 = D_1 – R_1 >$ threshold
        - » E: Demanded-allocated mismatch measure
      - Choose VM_2 with $R_2 – D_2 > 0$ (has extra resources)
      - Take shares from VM_2 to VM_1 by amount: $((R_2–D_2) + (D_1–R_1)) / 2$
    - But this makes the sharing unfair
      - Consider fairness globally ⇒ Share swap
        - » Take at one server, give at another
        - » Only allow adjustment by swapping shares

VM1: D=5, R=3, newR=4.5
VM2: D=2, R=3, newR=1.5
(1+2)/2 = 1.5

SaaSification

- Developing SaaS software
  - Manual approach with the assistance of PaaS
    - Good for developing new SaaS
    - Still will require a lot of efforts to SaaSify an existing application
  - Tools for SaaSification
    - What entities in an application system need to be modified?
    - What are the major entities in a typical SaaS application?
      - UI
      - Business logic
      - Databases

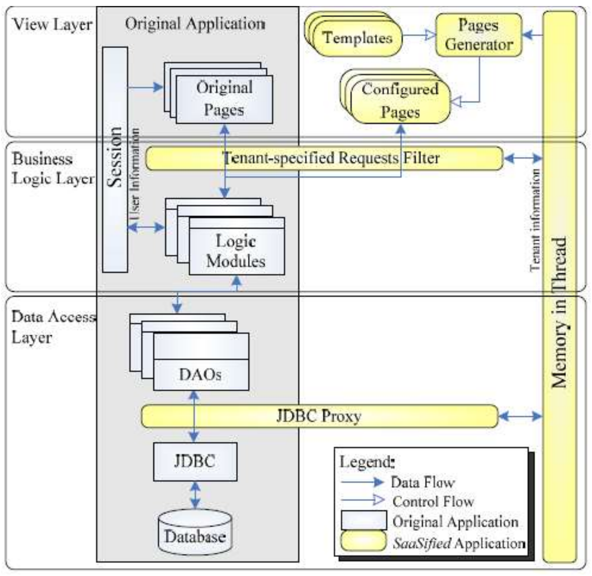
SaaSification

- Major steps for SaaSification
  - Make UI configurable
    - Allow each tenant to have the desired look and feel for the SaaS UI
      - E.g., Use CSS style sheets
  - Modify single-tenant DB for multi-tenancy
    - Modify the database using any design discussed earlier
    - Use a DB proxy to convert queries
      - Even in application tenancy, still need to use the DB proxy to correctly access the tenant-specific DB

SaaSification

- ❖ Major steps for SaaSification
  - ➢ Partition SaaS functions (business logic)
    - ■ Identify the user level functions and group them into categories
    - ■ Provision based on the categories
      - ● A tenant may have only paid for a subset of the SaaS functions
    - ■ Modify DB to maintain tenant accessible functions
      - ● Tenant configuration DB
    - ■ Modify UI to incorporate differentiated provisioning
      - ● UI should not show the functions a customer cannot access
      - ● On the fly generate the tenant-specific UI
        - » Classify UI pages based on the function categories
        - » Generate tenant-specific dispatching pages

A SaaSification Framework



Request filter takes user login info, find tenant id, create tenant context, and put it in memory of the request processing thread

JDBC proxy converts query and responses
It accesses tenant context for all conversions

SaaSification

- ❖ Major steps for SaaSification
  - ➢ Identify isolation points and perform isolation
    - ■ Resources in the single-tenant application that need to be isolated for different tenants
    - ■ Example: Pet store SaaS code piece

```
After login
  visitor-count++;   new (visitor);
  if (x-dog in visitor.cart) then gift = bone;
  payment = sum (visitor.cart);
  checkout …
  total-sales-today += payment;
```

DB is also an isolation point
⇒ Multi-tenant DB

- ● Which variables are isolation points and which are not
- ● Isolation point => one for each tenant (e.g., “tenant-id.visitor-count”)
- ● Variable like “gift” is not an isolation point

SaaSification Challenge

- ❖ How to identify isolation points in an application
  - ■ Besides the functions and DBs
  - ➢ A very challenging task in automated SaaSification
    - ■ Use knowledge base and other AI methods
    - ■ Semi-automated instead of fully automated
      - ● Can have false positives, and let user reject incorrect ones
      - ● But cannot have false negatives, which implies that the developer has to anyway go through the entire program to identify the missing isolation points (then automation is not of much help)
      - ● In the program, eliminate instances that are impossible to be the isolataion points, present all the rest for user to make the decisions
  - ➢ Once identified ⇒ Adapt the code to multi-tenancy
    - ■ Much easier

SaaSification for Business Logic

- Business logic as policies
  - Instead of fixed coding of business logic, use policies to accommodate different business logic
    - Principle: Same software, different data
  - Example: In a travel agency SaaS
    - Case 1: If a reseller or a client exceeds his credit limit, all his upcoming purchases will be rejected
    - Case 2: If a reseller or a client exceeds his credit limit for three weeks without any payment, he will be blacklisted
    - …

SaaSification for Configurability

- How to offer more configurability?
  - Allow each tenant
    - To create custom extensions to standard data objects or entirely new custom data objects ⇒ Need tenant-specific code to handle these tenant-specific data objects
    - To customize the SaaS application program itself
  - Developing such application programs can be very difficult
  - Software becomes hard to maintain
  - System performance may get hindered
    - Due to the execution of the cumbersome software with many branches into customer-specific choices

SaaSification for Configurability

- ❖ SaaSification solution for configurability
  - ➢ SaaS becomes a runtime engine
    - ■ To generate a specific version of the application from tenant configuration data that are about the application itself
    - ■ Code generation and compilation done on the fly
    - ■ Specialized code modules become "data objects"
      - ● Suitable for complex SaaS
  - ➢ SOA based solution
    - ■ Provide specialized functionalities as services
    - ■ Support workflow specification (like modifying application itself)
      - ● Tenant-specific execution flows is stored as workflows in DB
      - ● Tenants can integrate their own services into the workflow
    - ■ SaaS is a runtime engine to execute tenant-specific workflows

API Access Control in SaaS

- ❖ Accesses to services in SaaS
  - ➢ Tenant level
    - ■ A tenant may have subscribed a subset of the SaaS services
    - ■ SaaS system can use an ACL to decide for each service, which tenants have the access rights to it
      - ● Or, services can be grouped together and access rights are define on service groups (e.g., services that have to be accessed together can be in one group)
  - ➢ Intra-tenant
    - ■ Each tenant may have their own policies for service accesses
    - ■ Can be role based or ACL based (or other models)
  - ➢ SaaS system
    - ■ First enforce tenant level ACL, then run intra-tenant access control based on intra-tenant policy

Data Access Control in SaaS

- Access to data in SaaS
  - Two layers
    - Intra-tenant access control model
    - Inter-tenant access control model
    - Need to consider consistency between intra-tenant and inter-tenant access control models
- Access isolation
  - Use existing models for access control within each tenant
  - Absolutely no cross tenant access
  - What if sharing among tenants is needed?
- Consider sharing
  - ACL-based and role-based approaches

Access Control for SaaS

- RBAC for SaaS
  - Role based access control within each tenant
    - Each tenant specifies its role hierarchy and permissions
      - Tenant user table has a role column
      - Tenant resources have permission assignments
        - Can define the resources at different granularity
      - Each query has tenant/role contexts
  - Cross tenant accesses
    - Use role mapping technique
      - Map the roles of tenant A to the roles of tenant B
      - User of tenant A assumes the mapped role of tenant B for all accesses of tenant B's resources
    - Who assigns the mappings

Access Control for SaaS

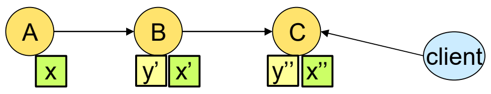
- ❖ Role mapping for cross tenant accesses
  - ➢ Each domain has an "issuer" who can assign mappings
  - ➢ Mapping tenant A's roles to tenant B's roles
    - B should make the mapping decision
      - Because it involves accesses to resources of B
    - A needs to expose the role hierarchy to B
        - » Will A be willing to reveal the role hierarchy (RH)?
      - A and B first build a trust relation before role mapping
      - If A trusts B for revealing its RH, then proceed
      - Some consider private roles, which should not be revealed even after trust relationship is established
- ❖ Problem with role mapping in SaaS
  - ➢ All involved tenants should have used RBAC

AIFC in SaaS

- ❖ Use MLS or lattice models
  - ➢ Easy to implement if interoperable
  - ➢ Consistency between intra-tenant and inter-tenant?
    - Need to use the same IFC model
    - Need to align security classes and object categories if they are not the same in the involved tenants
  - ➢ Inflexible
    - In a single domain, these models are already inflexible
    - In multi-tenant systems, members from different tenants may not have the rigid security levels
      - Especially the problem with confidentiality and integrity conflict

AIFC in SaaS

- Two-level AIFC
  - Track information flow
    - How do we know x” is derived from x ⇒ When x’ is created, record its relation to x; when x” is created, record its relation to x’ ⇒ With such tracking, we know x” is derived from x
    - This information can be called provenance information
  - High level AIFC policy
    - Decide how to determine access rights based on the provenance info and the access control policies in individual tenants
    - E.g., All tenants who had contributed to the data should approve the access to the data
      - Tenants A, B, C all have contributed to x” ⇒ client needs to have access rights for x in A, x’ in B and x” in C in order to access to x”



Readings

- Access control models for SaaS
  - Access control
    - Multi-tenant data architecture, https://msdn.microsoft.com/en-us/library/aa479086.aspx
    - Multi-tenancy authorization models for collaborative cloud services
    - SERAT : Secure role mapping technique for decentralized secure interoperability
  - AIFC
    - Multi-tenant Access and Information Flow Control for SaaS