<http://www.brentozar.com/archive/2011/06/how-design-multiclient-databases/>

<https://sqroot.eu/2012/designing-a-database-for-multiple-clients>

<http://discuss.joelonsoftware.com/default.asp?design.4.319460.16>

At my previous job, we hosted not just one database per client -- in most instances, it was more than that! As I left, there were over 4,500 databases running in one MariaDB cluster, nearly 7,000 in another (ironically smaller) cluster, and 4 "shards" (completely separate, independent web and database servers, even in a wholly separate data center) each hosting 200-500 databases in a single MySQL server. And that company is still growing at a good clip.

The long and the short is that the success of that company proves that such an architecture is indeed feasible. (Caveat: Contrary to the apparent gains in isolation by using separate databases, all data was accessed through a trio of tightly coupled applications that all used the same database user/pass! I suspect performance may have suffered ever-so-slightly if each client had a separate user/pass -- but only slightly.)

From my experiences working closely with the sys admins (technically I was a programmer with the company, but in reality I was the best DBA they had, and the only person they had who knew how to set up a firewall!), performance-related concerns boiled down to concurrent accesses, query complexity/time, index performance, etc. -- all the usual suspects, in other words, and the number of databases on the server played no discernible part, a conclusion affirmed by the highly-paid specialist consultants we consulted regularly.

The bottom line is that you should focus your concerns on your application, on your infrastructure, and not on the number of databases you happen to have. All those other factors will be more than enough to keep you busy resolving performance problems and bottlenecks.

I recommend you read [Multi-Tenant Data Architecture](http://msdn.microsoft.com/en-us/library/aa479086.aspx), a white-paper that discusses the options you have, and pros and cons. To summarize, it gives three options:

* separate DBs
* separate schemas
* shared schema

You are now at separate DBs stage, which offers the best separation (isolation between tenants), but is the most difficult to manage. As you grow into hundreds of tenants you'll realize that the logistics of administering 100s of DBs are far from trivial. Think backup-restore (location of backed up files, jobs, schedules etc). Think how will you monitor and manage file allocation, disk space used and database growth across hundreds of DBs. Think what will be your High-Availability/Disaster-Recoverability scenario in a near future with 1000 tenant? 1000 mirrored DBs, 1000 log shipping sessions? Think what if in 6 months your dev team comes to you and say "I know how give this awesome feature to our product, we'll use Transactional Replication!", what will you say? "sure, let me set up 500 publishers, it will be *fun*"! Is not impossible to manage hundreds of DBs, but if you plan to, you better polish up your [PowerShell](http://technet.microsoft.com/en-us/library/hh245198.aspx) skills and stop using the UI management tools *right now*.

Additionally you need to consider that multiple (hundreds) DBs have measurable impact on performance and cost:

* physical disk space is less efficiently used (every database must have *some* spare room, you'll have that spare room multiplied by the number of DBs)
* There is no way you can create a dedicate log disk for write intensive tasks, you'll will have to move all those LDFs onto one (or more) SSD storage
* Log writes will be less efficient on frequent commits as they spread out across many individual log block records vs. aggregate into one (you'll get underused log blocks). See [what an LSN is: Log Sequence Number](http://rusanu.com/2012/01/17/what-is-an-lsn-log-sequence-number/) to understand what I'm talking about.

Separated DBs come with some advantages though due to isolation, the main advantage being independent backup/restore.

Scenario like yours though are a perfect candidate for [SQL Azure databases](http://www.windowsazure.com/en-us/services/sql-database/). No administration of disk space, no need to provide HA/DR, grow to hundreds/thousands of DBs etc.

There are inherit risks and rewards to both system. I worked for a financial firm that supported roughly 40 clients (national banks) on 1 database. We then purchased another company that sold similar software and that had gone with 1 database per client. Finally, the company went bankrupt and we did have to export all user data. Here is what the people I worked with and I found:

Pros of Single DB:

1. Software updates and bug fixes are easier.
2. Easy to manage and report on all client data.
3. Updating data becomes easier.
4. Easy to create modular functionality that 1 client wants, turn if off for the other clients, and then turn it one when they desire it in the future.

Cons of Single DB:

1. Data integrity - We had 2 or 3 cases where 1 bank's users saw another bank's data. This was a nightmare. Especially because the site users were not just the bank employees but actual account holding customers of the bank! This is by far the biggest issue with 1 database
2. Exporting client data -When we had to this it usually wasn't a big deal. You end up with 1 table that has all of the clients in it and you key off of that table to get your client specific data.

Pros of Multiple DBs:

1. There is no concern of cross client data contamination or breaches
2. Exporting a client’s data is dead easy.

Cons of Multiple DBs:

1. Updates and bug fixes - This was the real nightmare. When you have 20 clients on 20 different databases you quickly run into a case where 1 client wants a bug fixed and another thinks the bug is a feature or doesn't want to risk the update. Furthermore, you will have instances where 1 client wants a game changing enhancement but the other clients don't. When this happens you databases will start to diverge. Suddenly you will have to update clients 1-15 with 1 script 16-19 with another, and 20 with a third. We saw this become such an issue that a bug fix would take 15 to 20 times as long for the company we purchased than for us because they had to run all tests for every client and deal with each client’s special code. Effectively, they needed a new support person for every new client, whereas the parent company needed one for every 5 to 10 clients.
2. DB management - When you get to a large number of clients managing all of the databases becomes a real hassle. You will without a doubt need more DBA time to manage them.

In the end my recommendation having seen and done both is to have "discipline"! I think the multi-db choice is slightly better because it protects you but you cannot ever let you clients make a choice that causes you to add functionality to only them or you will be putting yourself on a path to failure.

I'd have separate database for separate clients. A client might demand this for security reasons - i.e. only their site has access to their data. It also means that if a client want to move their data then it's going to be *much* easier to manage.

It also means that if there is a problem with one client's database it doesn't affect all of the others.

If you want to compare data between clients then you should do that separately.

If you are running out of databases that you can have then perhaps you should be considering changing your host provider.

To add to the pro’s / con's listed up to now:

Pros of multiple databases:

1. Locking issues are avoided; we've got databases where the clients can trigger DDL-changes on some of the tables. For the larger tables (>2m records) this locks the table for a considerable amount of time. The only people at a disadvantage are their own users, so this is sort-of acceptable.
2. Flexibility - some clients have specific wishes regarding data they wish to store; multi-database allowed us the flexibility to alter their database specifically, without having to clutter the data model for the other clients.

Cons:

1. Major con: Joining on other tables is a lot more cumbersome. We've got a Main database which contains most meta-data. The client-specific database users don't have access to this database, so all joins between tables in that database and the client-specific one are handled in the application instead of in the database. You could resolve this by giving the client-specific users access to the main database, but then the app could/might leak information again.

I’ve actually used a hybrid approach where there was a client master database with minimal information stored in it, but the key was the client table that includes the connection string pinpointing where the client information was located. This allowed us to start out having all of our information in a single database (and server), but grow out the number of servers and databases as appropriate (and as we needed to scale). As we added clients to the system, we could pick which database server (and database) to place the client data. This approach does miss out on aggregating data across all databases, but we didn’t have a need to merge the data together.

We also created a system for automating the building of the database deployment scripts. This system would generate the necessary SQL scripts to be run on each database server (it didn’t automatically execute the scripts, but provided the scripts for us to give to the operation guys to deploy). As we scaled, we did plan on automating that script deployment.

We had a debate similar to this about 9 years ago when starting our company, and came down on the “one database per customer” side. Our primary deciding factor which isn’t mentioned in your article is the risk that one slip-up by a developer and you risk exposing Customer A’s data to Customer B. I definitely feel we made the right choice for our company.

Another benefit we enjoy is the ability to stand up N copies of a given customer’s database to create test environments for them. Also, if we really need to dig into a performance problem that one customer is experiencing, we can restore a backup on a different machine and troubleshoot it there, rather than impacting the production environment. It’s much less data to have to move around and restore than if we just had one monolithic database.

We use 1 DB per client, and have a few additional reasons to do it this way.

1) Licensing cost. We use many instances of SQL Express, which is free! Combine with cheaper commodity servers because you scale out.

B) Rolling migrations to new version. We don’t upgrade everyone at once, so we can catch problems with the first few clients, rather than having everyone affected.

iii) Some clients aren’t willing to use a hosted solution, so they can buy a downloaded version of the app, and use the same DB in either place.

The biggest downside in my view is that high availability is much harder.

Personally, I'm not a big fan of this use of schemas. Schemas are for security and to a lesser extent -- avoiding name collisions for large numbers of database objects. I personally prefer putting each set of objects on a different database and using different user logins with default databases to control the name resolution.

Since the clients databases and functionality are diverging, then it means that at one point they will end up being different systems, so in this case I would recommend separate systems since the costs of maintaining the customizations for each client will outweigh the benefits of a single database system.

Single database systems are best for when the changes between different customers are merely configurations but not additional features for each client.

Space is cheap these days, so I'd advise to use one database per application.

Sharing one database for amongst multiple applications has some serious **disadvantages**:

* The more applications use the same database, the more likely it is that you **hit performance bottlenecks** and that you **can't easily scale the load as desired**. SQL Databases don't really scale. You can buy bigger machines but they do not scale well in clusters!
* **Maintenance and development costs can increase**: Development is harder if an application needs to use database structures which aren't suited for the task at hand but have to be used as they are already present. It's also likely that adjustments of one application will have **side effects**on other applications ("why is there such an unecessary trigger??!"/"We don't need that data anymore!"). It's already hard with one database for a single application, when the developers don't/can't know all the use-cases.
* **Administration becomes harder:** Which object belongs to which application? Chaos rising. Where do I have to look for my data? Which user is allowed to interact with which objects? What can I grant whom?
* **Upgrading:** You'll need a version that is the lowest common denominator for all applications using it. That means that certain applications won't be able to use powerful features. You'll have to stick with older versions. It also increases development costs a bit.
* **Concurrency:** Can you really be sure that there're no chronological dependencies between processes? What if one application modifies data that is outdated or should've been altered by another application first? What about different applications working on the same tables concurrently?

Compared to that, data imports/ETL-processes are almost always pretty straightforward and simple. Load the data as often as you need to, space is cheap. You can account for scalability for each application independently, adjust and tweak the structures as you need them and there won't be concurrency issues. Side effects can be traced much easier, too.

**Edit:** I'd like to point out, though, that as @Saeed mentioned, if you can encapsulate data manipulations in a service which is commonly available, then it's easier to share one database with multiple applications. As long as you don't need raw access that is a very good approach.

It may or may not be worth the trade off in your situation, but maintaining data integrity is easier with a single database. In MS SQL Server at least, you cannot foreign key from one database into a different database. You can simulate foreign key behavior with triggers, but it's not particularly elegant.

In addition, creating local copies of the data can be dangerous when writes come into play. If AppA and AppB both have a copy of some shared data and AppA updates it, AppB will still have the old data. Or, you will have to setup triggers to keep the data in sync.

Single Database Pros

* One database to maintain. One database to rule them all, and in the darkness - bind them...
* One connection string
* Can use [Clustering](http://www.mysql.com/products/database/cluster/)

Separate Database per Customer Pros

* Support for customization on per customer basis
* Security: No chance of customers seeing each other’s data

Conclusion

The separate database approach would be valid if you plan to support per customer customization. I don't see the value if otherwise.

1) Having separate databases allows for easier distribution of load on several hosts, it lifts the roof in many ways; disk, memory, locking, cpu, backup-time and so on. If you are serious about putting millions of rows in mysql, it is certainly a good idea with separate databases (not only schemas), and even separate instances, so that the resource-consuming customers won't impose downtime on less resource consuming ones.

2) It is going to be exactly N times harder to manage where N is the number of databases This extra cost you must compare to the cost of using just one db/schema and instead manage separation of customer in code. It's also inherently much harder to manage if you have to call customer support at your hosting company, or even your local grumpy dba, instead of just running a neat script from your console each time you need to update schema or create a new database.

Some databases and persistence frameworks have support for multi-tenancy, Oracle has this and support is beginning to emerge in Hibernate 4.

Even though many arguments point in the direction of separate databases, it *is* generally possible to use just one database as well.

I usually add ClientID to all tables and go with one database. But since the database is usually hard to scale I will also make it possible to run on different database instances for some or all clients.

That way you can have a bunch of small clients in one database and the big ones on separate servers.

A key factor for maintainability though, is that you keep the schema identical in all databases. There will be headache enough to manage the versioning without introducing client specific schemas.

Listen to the Stackoverflow podcast where Joel and Jeff talk about the very same question. Joel is talking about their experience offering a hosted version of their software. He points out that adding client ids all over your DB complicates the design and code (are you sure you didn't accidentally forget to add it to some WHERE clause?) and complicates hosting feature, such as client-specific backups.

In my view, it will depend on your likely customer base. If you could get into a situation where arch-rivals are both using your system, then you would be better off with separate databases. It also depends on how multiple databases get implemented by your DBMS. If each database has a separate copy of the infrastructure, then that suggests a single database (or a change of DBMS). If multiple databases can be served by a single copy of the infrastructure, then I'd go for separate databases.

Think of database backup. Customer A says "Please send me a copy of my data". Much, much easier in a separate database setup than if a single database is shared. Think of removing a customer; again, much easier with separate databases.

(The 'infrastructure' part is mealy-mouthed because there are major differences between different DBMS about what constitutes a 'database' versus a 'server instance', for example. *Add*: The question is tagged 'mysql', so maybe those thoughts aren't completely relevant.)

*Add*: One more issue - with multiple customers in a single database, every SQL query is going to need to ensure that the data for the correct customer is chosen. That means that the SQL is going to be harder to write, and read, and the DBMS is going to have to work harder on processing the data, and indexes will be bigger, and ... I really would go with a separate database per customer for many purposes.

Clearly, StackOverflow (as an example) does not have a separate database per user; we all use the same database. But if you were running accounting systems for different companies, I don't think it would be acceptable (to the companies, and possibly not to the legal people) to share databases.

* **DEVELOPMENT** For rapid development, use a database per customer. Think how easy it will be to backup, restore, or delete a customer's data. Or to measure/monitor/bill usage. You won't need to write code to do it by yourself, just use your database primitives.
* **PERFORMANCE** For performance, use a database for all. Think about connection pooling, shared memory, caching, etc.
* **BUSINESS** If your business plan is to have lots of small customers (think hotmail) you should probably work on a single DB. And have all administrative tasks such registration, deletion, data migration, etc. fully automated and exposed in a friendly interface. If you plan to have dozens or up to a few hundreds of big customers then you can work in one DB per customer and have system administration scripts in place that can be operated by your customer support staff.

The following [screencast](http://wiki.developerforce.com/page/Multitenancy_Webinar) explains how it's done on salesforce.com. They use one database with a special column OrgId which identifies each tenant's data. There's much more to that so you should look into this. I'd go with their approach.

There's another great [article](http://msdn.microsoft.com/en-us/library/aa479086.aspx) about that on MSDN. It explains in depth when you should use a shared or isolated approach. Remember that having a shared DB for all your tenants has some important security implications and if all of them share same DB objects you might want to use [row level security] - depending on the DBMS you use (I'm sure it's possible in MS SQL Server and Oracle, probably in IBM DB2 also). You can use tricks like [row level security in mySQL](http://www.sqlmaestro.com/resources/all/row_level_security_mysql/) to achieve similar results (views + triggers)

Having a database per client generally does not scale well. MySQL (and probably other databases) holds resources open per table, this does not lend itself well to 10k+ tables on one instance, which would happen in a large-scale multitenancy situation.

Of course, if you have some other issue which causes other problems before you get to this level, this may not be relevant.

Additionally, "sharding" a multi-tenant application is likely€ to be the right thing to do eventually as your application gets bigger and bigger.

Sharding does not however mean one database (or instance) per tenant, but one per shard or set of shards, which may have several tenants each. You will need to discover the right tuning parameters for yourself, probably in production (hence it probably needs to be pretty tunable from the outset)

You can start with a single database and partition it as the application grows. If you do this, there a few things I would recommend:

1) Design the database in a way that it can be easily partitioned. For example, if customers are going to share data, make sure that data is easily replicated across each database.

2) When you have only one database, make sure it is being backed up to another physical server. In the event of a failover you can revert traffic to this other server and still have your data intact.

[High Scalability](http://highscalability.com/) is a good blog for scaling SaaS applications. As mentioned, splitting tables across databases as you suggested is generally a bad idea. But a similar concept is sharding, where you keep the same (or similar) schema, but split the data on multiple servers. For example, users 1-5000 are on server1, and users 5000-10000 on server2. Depending on the queries your application uses, it can be an efficient way to scale.

|  |  |
| --- | --- |
| up vote9down vote | My experience (albeit SQL Server) is that multi-database is the way to go, where each client has their own database. So although I have no mySQL or Ruby On Rails experience, I'm hoping my input might add some value.  The reasons why include :   1. data security/disaster recovery. Each companies data is stored entirely separately from others giving reduced risk of data being compromised (thinking things like if you introduce a code bug that means something mistakenly looks at other client data when it shouldn't), minimizes potential loss to one client if one particular database gets corrupted etc. The perceived security benefits to the client are even greater (added bonus side effect!) 2. scalability. Essentially you'd be partitioning your data out to enable greater scalability - e.g. databases can be put on to different disks, you could bring multiple database servers online and move databases around easier to spread the load. 3. performance tuning. Suppose you have one very large client and one very small. Usage patterns, data volumes etc. can vary wildly. You can tune/optimise easier for each client should you need to.   I hope this does offer some useful input! There are more reasons, but my mind went blank. If it kicks back in, I'll update :)  **EDIT:** Since I posted this answer, it's now clear that we're talking 10,000+ tenants. My experience is in hundreds of large scale databases - I don't think 10,000 separate databases is going to be too manageable for your scenario, so I'm now not favouring the multi-db approach for your scenario. Especially as it's now clear you're talking small data volumes for each tenant! |

**Multi Tenants Database Architecture**

It is used to address the problem of SAAS which can serve multiple clients. Multi-Tenants database architecture is very useful when one instance of database is serving to multiple clients. Only one set of hardware resources is needed to fulfill the requirements of all users. Multi-tenant is based on subscriber model, so user has freedom to avail the facility as per business requirement or can turnoff.

There are different approaches to the advantage out of the multi-tenants database. These are

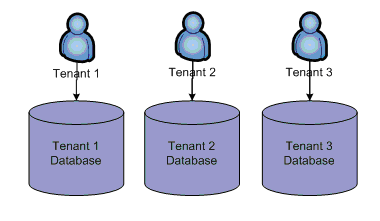
* *Dedicated database*: Separate databases per tenant.
* *Dedicated table and different schema*:  Shared database and separate schema.
* *Share table/schema*: Same database and same table.

Now, it is very important to select appropriate approach for your application depending upon the following factors.

* Size of tenant database
* Number of tenant
* Number of users per tenant
* Growth rate of tenant
* Growth rate of tenant database
* Security
* Cost

**1 . Dedicated database:**

It is straight forward approach where each tenant has its own database. Each tenant has its own set of data that remains logically isolated from data that belongs to all other tenants.

[](http://purikl.files.wordpress.com/2010/01/image1.png)

*Pros:*

* More secure data
* Easy to customize for vendor specific needs
* Easy to maintain e.g. backups, restore etc…

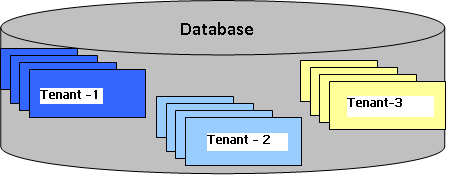
*Cons:*

* Relatively high hardware and maintenance requirements
* This approach tends to lead to higher costs for maintaining equipment and backing up tenant data.

**2. Dedicated table and different schema:**

Serving multiple tenants under same database, where each tenant has its own sets of tables grouped with schema as required by tenant.

|  |
| --- |
|  |
|  |  |

[](http://purikl.files.wordpress.com/2010/01/image2.png)

*Pros:*

* Good for small database application where number of tables per tenant is small.
* Cost is low as compared to dedicated database approach.
* Moderate logical isolation level is there for vendors having security as a concern.

*Cons:*

* Tenant data is harder to restore incase of failure.
* Difficult to manage large database application.

**3. Shared Table/Schema:**

This approach involves using the same database *and* the same set of tables to host multiple tenants’ data. A given table can include records from multiple tenants stored in any order; a Tenant ID column associates every record with the appropriate tenant. Any application accessing the row must refer to this column in every query to ensure that one tenant is not able to see another tenant’s data.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **TenantID** | **Col1** | **Col2** | **Col3** | **Col4** |
| 1 | Abc | .. | .. | .. |
| 21 | Cdw | .. | .. | .. |
| .. | .. | .. | .. |  |
| .. | .. | .. | .. | .. |

*Pros:*

* Lowest hardware cost as compared to other approaches.
* Can serve more tenants per server.
* Ability to update the schema in one place and affect all tenants.

*Cons:*

* More security is required to make sure no one can access cross-tenant data.
* Can affect query performance because of more rows.
* Can only update the schema in one place and thereby affect all tenants.

Extension Table

In case, there is need to increase number of fields as per tenant requirement under approach 3 then?

As all tenants will share same table/schema. It is very difficult to customize the number of fields. One way to avoid these limitations is to allow tenants to extend the data model arbitrarily, storing custom data in a separate table and using metadata to define labels and data types for each tenant’s custom fields.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | **TenantID** | **FN** | **Field** | **TagID** | | 101 | Kim | Trade | 221 | | *202* | *Tim* | *HR* | *433* | | ……………. | | | | | 342 | Rim | Fin | 510 |   **(Data Table)** | |  |  |  |  | | --- | --- | --- | --- | | **TenantID** | **ExtID** | **Label** | **DataType** | | 342 | 3990 | Age | int | | 120 | 3122 | Status | bool | | ………………….. | | | | | *202* | *1200* | *LName* | *string* |   **(Metadata Table)** |
| |  |  |  | | --- | --- | --- | | ***TagID*** | ***ExtID*** | ***Value*** | | *433* | *1200* | *Border* | | 500 | 321 | abc | | …………… | | | | 510 | 3990 | 23 |   **(Extension Table)** |  |

Here, a metadata table stores important information about every custom field defined by tenant, including the field’s name (label) and data type. These fields are created dynamically on front end (GUI) with unique id and value entered by end user corresponds to these fields are stored in different table Extension table.

So corresponding to data table we need to create two new tables *“MetaData”*and *“Extension”.*

This approach allows each tenant to create as many custom fields as necessary to meet its business need. When the end user retrieves a customer record, it performs a lookup in the extension table, selects all rows corresponding to the record ID, and returns a value for each custom field used. To associate these values with the correct custom fields and cast them to the correct data types, the application looks up the custom field information in metadata using the extension IDs associated with each value from the extension table.

This approach adds a level of complexity for database functions, such as indexing, querying, and updating records.

|  |  |  |
| --- | --- | --- |
| |  | | --- | |  | | I do not know the pros-cons of having multiple databases per server (I never had any issue with that) but the many schemas concept [technet.microsoft.com/en-us/library/dd207005.aspx](http://technet.microsoft.com/en-us/library/dd207005.aspx) within the same database, offers both isolation and security. So, you can try this architecture as well |
| |  |  | | --- | --- | | 2 |  | | @Alexandros schema separation offers a little, but it doesn't allow you to use separate recovery models, back up on different schedules, restore one client to a specific point in time, remove one client easily, move one client easily, etc |
| |  |  | | --- | --- | | 3 |  | | I've seen systems with 3,000+ databases (1 per client) on a single server. I wouldn't worry too much - just make sure you plan resources carefully and monitor usage as the client count goes up. |

Managing 100 or 500 databases is really not all that different from managing 5 or 10 - you just have to embrace automation and have a scalability plan in place (and don't plan to use high-cost-per-database features like mirroring across all clients).

At my previous job we used this architecture and I would never once have ever thought of merging two clients into a single database, even though some of the challenges can be "hard."

The big benefits are independent recovery models (a can be simple, b can be full, etc.), the ability to restore to a point in time (or remove entirely) a client without disrupting others, the ability to seamlessly move a resource-heavy client to its own storage or to a completely different server with very little in the way of transparency (you update a config file or table that tells the app where to find that client).

I address several of the objections, and/or how to approach the problems, in these posts:

* [Handling growing number of Tenants in Multi-tenant Database Architecture](http://dba.stackexchange.com/questions/16745/handling-growing-number-of-tenants-in-multi-tenant-database-architecture)
* [Multi-tenant database using SQL Server 2008?](http://dba.stackexchange.com/questions/25719/multi-tenant-database-using-sql-server-2008)
* [Automation of backups of Large Number of Database](http://dba.stackexchange.com/questions/16679/automation-of-backups-of-large-number-of-database/)

That all said, I don't think any of us can tell you the point at which management becomes impractical for you - just know that whatever specific challenges you come across, you can ask about those problems individually.