Relational Database Guidelines For MMOGs

Jay Lee

Imagine you're building an MMOG, and after conducting some research, you decide that a relational database is the ideal solution to the needs of game state persistence, back office functionality (such as account management and billing), and even building game logic with a data driven approach. Congratulations, you have made an excellent choice! But before you get too carried away feeling good about yourself, be aware that there are many challenges to implementing your game on top of a database. Designing an effective schema for an online game is not a trivial task.

In this article I will provide a foundation for making some of the key decisions that you'll encounter in regards to your database, based upon my years of experience in this field, most recently at NCSoft. Hopefully your development effort will culminate with shipping a game built on top of a robust database solution, and your organization will be poised to take advantage of everything that it has to offer.

Where appropriate, I will share specific comments related to Microsoft SQL Server 2000, the database product in use where I am employed. This should not be interpreted as a pitch for the product, nor an indication that other competing products are inferior. It just happens to be the software I am currently utilizing, and thus most familiar with.

It could very well be that the same reasons we chose MS SQL Server would make a different product the right choice for your game. It really depends on your specific needs, and no one knows these better than you. The information presented here is designed to supplement your own analysis, to the extent that it applies to your situation. My goal is to provide information to help you make the right decisions in this regard.

While I try to include examples wherever possible, this article is not intended for someone who is a novice with databases. The assumption is that you have at least worked with a relational database in some capacity before, or that you have

someone on your staff with sufficient background to help you work through the content.

The information is organized by topic. Tackle the topics in any order you wish, as there are no topics that are prerequisite to others.

Naming Standards

Establish naming standards for all the objects that are created in the database. Use as many characters as needed to be both clear and descriptive when naming each object. Name objects so that they are easy to remember and can be guessed correctly most of the time by anyone on the team.

For example, if during discussion about the subsystem that generates creatures for the game, the terms "Creature" and "Spawner" are frequently used, then there should be tables called Creature and Spawner in the database.

It is not worth getting too creative when naming new tables that result from the association of existing tables. It's simply easier to concatenate the existing names together. So the association of Creature and Spawner becomes a table called SpawnerCreature, or CreatureSpawner if you prefer, making it very obvious which two (or more) entities are being related.

When naming columns on a table, the goal should be to pick names that make it very obvious what each column is being used for. This is especially important on games with long development schedules and lifetimes like MMPs. It's all too easy to look at a table after a few months and have no clue what a column is used for. Good naming goes a long way towards preventing this from happening.

Examples of column names one could expect to see: CurrentlyUsedFlag on the Spawner table to indicate if it is currently in use or HitPointAmount on the Creature table to store the hit point value of each creature captured in the table. Notice that in these names, the data type of the column is implied; one would expect a flag to have an on or off value, or that an amount would be some type of integer.

Microsoft SQL Server allows the ability to create user-defined types in terms of existing types. This can be leveraged to further standardize data type assignment. Suppose that each in-game object is identified by a bigint, which is the equivalent of a 64-bit value. One might create a user-defined type called 'gameObjectId'

which is in fact a bigint. Then wherever you see a 'gameObjectId' column, it immediately reminds you of its domain.

Because you will be creating a lot of stored procedures, it's worth reiterating the naming mantra. Choose names that will allow grouping stored procedures with similar functionality. Stored procedures that return the entire of contents of a table might be prefixed with the word 'Contents', such as 'ContentsCreature' which returns the entire contents of the Creature table. CharactersByUserId is understood to be a select statement that returns multiple rows of character related data given a UserId as an argument.

You can overcome the character limit on names in some databases by establishing a consistent set of abbreviations that everyone is able to reference. A stored procedure that you would like to name RetrieveItemsByInventoryCategory could be called RtrvItmByInvCtg and still meet the naming goals.

Tables

Almost without exception, each table in the database should be created with a key. A key is one or more columns that can be used to uniquely identify a single row on a table. If at all possible, have keys be system assigned. This allows the database to generate the next unique value for a new row when created on its primary table.

To illustrate: the Creature table is given a primary key called CreatureId that is system assigned. When the Bugbear creature is added to the Creature table, it is assigned the value 15 for its CreatureId. Then whenever a creature has a CreatureId value of 15 in the game, it is guaranteed to be the Bugbear creature.

It is also useful to retain the name of a primary key column when it appears as a foreign key on another table. Applied consistently, one can be assured that the CreatureId column on the CreatureSpawner table represents a creature associated with a particular spawner. If the value is 15, it further means that a Bugbear is associated with the spawner identified by the given SpawnerId.

Keys with numerical data types are preferable because they make for a more efficient index compared to other data types. Additionally, no one really cares that the value 15 represents a Bugbear, but one might need to change 'Bugbear' to 'BuggBear' for some reason, and a changing value violates the purpose of a key.

When picking the numerical data type for a system assigned key, pick one that is large enough that it will never overflow in the lifetime of the game. Assume dramatic growth for the game beyond your wildest dreams (hey, why not dream?), and assign the data type accordingly, even if it feels like space is being wasted. Disk space is cheap compared to reworking a system that does not adequately anticipate growth.

Creating a 'smart' key is never acceptable. A smart key is present when one is able to look at its value alone and determine things about the row it is identifying. The world abounds with smart keys, and the associated expense of dealing with changes, as growth can no longer be accommodated. The classic example is the Vehicle Identification Number (VIN) that identifies each vehicle. Various positions in the VIN represent different attributes, such as country of origin. Because the VIN is a smart key, the IT departments at automakers and state transportation agencies have suffered whenever expansion was required to accommodate new requirements. Don't bring the headache of smart keys into your game; use system assigned keys instead.

When a table has a key with multiples columns they should be ordered based on their cardinality value from high to low. This allows for the creation of a more effective index structure in the database. Cardinality is the number of unique values in the domain of a particular column. For example, a flag has a cardinality of 2. It can be 0 or 1. Other cardinalities may not be so obvious. While you may choose a 32-bit integer for the SpawnerId, its cardinality is only the actual number of unique spawners in the game. Typically though, the ordering of columns will be easy to determine, based on familiarity with the system requirements and how the table will be utilized.

There is one caution when using system assigned keys. If the value of a key has to be synchronized across multiple database instances, be wary of the possibility of undesirable results. You may need to plan for a centralized assigner mechanism shared by each database instance, or the partitioning of the number space from which assignments may occur. For example, if every user has a unique identifier across the game universe, that identifier must be centrally allocated, or each database instance must make provision for allocating identifiers from a pre-defined and non-overlapping range.

It is enormously helpful to interact with the database in terms of related groups of tables, such as all the tables supporting a particular game system. MS SQL Server has a great tool that provides this feature. The Database Diagram, a database object that can be manipulated from Enterprise Manager allows WYSIYG table creation, modification and association to other tables. Multiple diagrams can exist in a database instance, and can contain any combinations of tables. This is superb for partitioning related tables into logical groups from a game system perspective, and is one tool I would hate to do without!

Tables Indexes

The general approach to creating indexes on tables should be "wait and see if it's necessary". It is simple to add an index when it becomes obvious it is needed; it's much harder to anticipate that one is going to be needed until the table is utilized and a particular query performs poorly because an index is missing. Since the maintenance of additional indexes on a table can impact performance, only use them when there is an obviously verifiable need.

I usually only create the index associated with the primary key of a table. I will consider an additional index if I know for a fact that there will be a query that is not going to utilize the primary key to access the table, and that there will be enough data in the table to slow down performance. Databases usually store data with some uniform page size for efficient access. This page size is typically large enough to contain the entire contents of small tables with a few rows, allowing them to be retrieved with a single read.

Before adding an index, do some homework to verify that the reason you are having poor performance of a query is truly the reason. Just as a profiler reveals hotspots in code, almost all databases provide the ability to analyze the execution plan of a query. Use this mechanism to profile your queries, both before any modifications to understand what needs to be done, and after the modifications to see that they had the intended effect.

Schema Structure

Relational tables can be designed to meet an idealized structure that has no redundancy and maximum flexibility in handling changes to the system. However, in typical database applications, this structure is often compromised to achieve performance goals. The act of re-factoring the 'normalized' structure of a

database to speed data access and manipulation is known as 'de-normalization'. De-normalization can be a black art, and the gyrations required to achieve particular performance targets while balancing ease of use is not for the faint of heart.

The following is my approach to de-normalization, and it goes against conventional wisdom. Don't do it. Admittedly there are some caveats to this perspective, so let me explain.

Unless you are in a position to throw ever-bigger hardware at database performance issues, you will be spending development resources on an implementation that alleviates the need to block processing when accessing data from the database.

There is little question that to get sufficient performance in the game loop, you will have to implement some combination of either a data caching scheme, an asynchronous data access mechanism or a separate data access thread. By isolating the game loop from sensitivity to data access time, the requirement to optimize data access speed through de-normalization is significantly reduced.

Moreover, if the game design can be 'nudged' into embracing a few guidelines, then we are free to maintain a normalized data structure in the schema to support the game. The following are these guidelines.

1. All data that remains static during game play such as data for game systems and representation of the game world should be pre-loaded into memory at server startup. Even if this segment of the schema involves hundreds of tables, this is likely to be achievable in a matter of minutes in the worst case.

2. Ensure that every in-game object can be retrieved in its entirety via a single query that returns a single row, even if it requires joining multiple tables. For example, if an object in the world is a weapon that has magical abilities, ensure that the single query retrieves all the data that makes it a weapon, and all of the data that makes it magical. Once loaded as a single row, the weapon should have everything it needs to function in the world; it must not wait on any subsequent queries to complete its definition.

3. When data is complex, such as a player character and everything that it owns, defer loading that data until it is actually needed and do so in an asynchronous

manner. A player expects an up front startup time when he first logs into the game, so this is a perfect time to load all the information about his character. By accessing the database asynchronously, the impact to other players in the game is minimized, while the overall processing hit is amortized over a relatively long period of time.

4. Avoid hierarchical or nested structures in data that may change at run time. For example if a player has a container such as an inventory, make it a flat container - don't allow containers within containers. Stacks of items of the same type are good as long as only a single stack of a particular type is allowed. Rethink any data that required multiple database round trips or nested stored procedures to resolve.

5. Keep the design of the schema as clear and accessible as possible; avoid the temptation to build a 'database within a database'. This is characterized by the storage of non-human-readable binary data within columns, or through the creation of game attribute to value lookup tables. The latter is where the first column of a table defines an in-game attribute and the second stores its value. Either of these approaches makes the database more opaque and harder to deal with. It is a road fraught with problems, and it would be preferable to make a somewhat less data driven game than to settle on either of these approaches. The power of the database is in making it easy to access data, and both of these 'solutions' have the opposite effect. The benefit of going with a clear and accessible approach is that your game remains fully configurable via data entry, and that the data can be leveraged in many ways via the power of SQL queries.

Stored Procedures

The stored procedure must be central to any strategy for using a database on an MMP game. Stored procedures exhibit many desirable attributes. They are reusable and remotely callable by code with little network overhead. They are efficient because their execution plan to data is determined ahead of time, and are configurable at run time with arguments. Finally, they can return result sets from queries while hiding the underlying data structure being utilized.

Because you will be developing a lot of stored procedures, it is necessary to determine some strategies for their implementation.

The first thing to decide is to determine is how to specify the columns returned in a result set. A SELECT statement written with the shorthand '*' conveniently returns every column from each table the query. Specifying '*' when selecting from the Creature table with 2 columns, CreatureId and CreatureName, returns both columns in the result set, as shown below.

SELECT *FROM Creature

Output:

CreatureId Creature Name---------- ----------15 Bugbear16 King Bugbear

Using this shorthand also eliminates the need to modify the stored procedure when a third column, say HitPointAmount, gets added. Unfortunately, depending on how the code is written that issues the call to the stored procedure, you may be forced to make code changes to avoid breakage.

The alternative approach is to have the stored procedure explicitly select each column to return from the query. This makes for a little extra work when creating the stored procedure, but insulates the code when a column gets added. Of course, when the code is ready to handle the change, you have to remember to select the additional column in the stored procedure.

Stored procedures can take multiple arguments, supporting any of the data types native to the database. If your database supports specifying default values for arguments, be sure to take advantage of this feature. When a new column is added to a table, you can add the additional argument with a default value to stored procedures manipulating that table and maintain compatibility with calling code. Unfortunately, stored procedures generally do not handle arrays of types as arguments. This means that a stored procedure has to be called repeatedly to get the effect of adding multiple rows to a table.

Stored procedures support a single return value similar to functions in programming languages. While this is convenient, not all database access libraries support this feature, so be prepared to work around the limitation. The good news is that every library supports returning the results of queries and one can return the stored procedure 'result' value via a result set.

For example, given the following stored procedure snippet:

```
if @value = 1beginreturn 1end

return 0
```

You should be able to substitute:

```
if @value = 1beginSELECT 1returnend

SELECT 0
```

This achieves the same net result by retrieving the return value as a result set with a single row. Naturally, if the stored procedure contains a query with a separate result set, you will have to determine if the interface library you use supports retrieving multiple result sets. The moral of this story is to discover the capabilities and limitations of your selected interface library before proceeding with heavy development on stored procedures that may end up having to be redone.

It should go without saying that if you are going to take the time develop stored procedures with return values that you should be checking the return values in the calling code to ensure that everything is running as expected.

On the topic of return values, every SQL statement will generally set an error code after it executes. Be sure to check these return codes! It may be the case that the 'problem' can be handled gracefully.

For example, suppose the data on a row needs to be updated. When the stored procedure runs, the update fails because the row does not exist. If appropriate, the specific 'does not exist' error can be trapped and handled silently with code that actually creates the row. This alleviates the need for the calling code to understand ahead of time whether the row exists or not. It always calls the combination 'update, insert if not present' stored procedure.

MS SQL Server has a quirk with its error code and other status variables. These variables only retain their values until the next statement is executed in the stored procedure, even if that statement is not a query. This requires that the error code

be checked immediately, or that its value gets stored into a local variable for use later in the stored procedure. This is illustrated below:

```
-- Either check immediatelyUPDATE TblName WHERE ColA = 1

if @@error <> 0

-- Or save value for checking laterdeclare @errVal int

UPDATE TblName WHERE ColA = 1

set @errVal = @@error
```

Write stored procedures to use the most efficient SQL possible. With respect to general efficiency, SQL statements can be ranked in the following order: SELECT, UPDATE/DELETE, INSERT. That is to say, that given the options of solving the same problem using one of these statement types, use the ranking is a rough guide of the order of preference.

A simple illustration: given a requirement to record the last time a player logged into a game server, the determination of which SQL statement to use might proceed as follows.

A SELECT statement can be dismissed because it is a read only operation.

The next candidate is either UPDATE or DELETE. DELETE is obviously ruled out, but UPDATE would work. A single player row could be updated each time they logged in with the current time and the requirement would be met. In addition the operation would be very efficient, since each Player row is accessible via a single column primary key, suggesting a very efficient index.

Another workable option is to INSERT a new row each time the player logs in. While this would gain us a history of every log in performed by a player, it is less efficient since INSERT operations need to establish a new row and potentially update indexes that exist for that table. If this history is not important, the UPDATE is clearly the most efficient option.

Transactions

A feature that is extremely valuable to have in your database software is transactions. Transactions provide the ability to group modifications across

multiple tables into a single unit of work. When a transaction successfully completes, all of the outstanding activity is committed, or written, to the database. However if an error occurred at any time during the transaction, a rollback is issued, reinstating each table to its original state prior to the beginning of the transaction. This is best demonstrated with an example.

begin transaction

INSERT INTO TableAvalues(…)

if @@error <> 0begin-- an error occurred, back out any changes to this-- point and exitrollbackreturnend

-- everything good to this point, insert the other rowINSERT INTO TableBvalues(…)

if @@error <> 0begin-- an error occurred, back out any changes to this-- point and exitrollbackreturnend

-- everything is good, commit this work, which makes -- updates to both tables permanentcommit

You should be using a transaction in every stored procedure where multiple tables are being updated. This convenient mechanism is much cleaner to read and maintain than code that tries to manually undo prior activity if an error occurs.

The MMP space has several cases where transactions are useful such as trading between players, players selling items to an NPC vendor or removing an item from a publicly accessible container.

In MS SQL Server, transactions may be nested. But be forewarned that they do not work in an obvious manner. One might expect for a nested transaction, that if that transaction's unit of work is committed, it remains committed independent of the outer transaction.

The actual implementation is that any nested transactions take on the result of the outer transaction. So if the outer transaction is rolled back, the inner transaction gets rolled back even if it was committed during execution.

Joins

The power of a relational database is realized when associating various tables together by performing joins on related columns. For example, to find all the names of creatures spawned by various spawners, we would need to join the SpawnerCreature table to the Creature table via the CreatureId column on each table.

Joins are more expensive performance wise than querying a single table, but storing creature names on the SpawnerCreature table is not only redundant, it makes changing names of creatures or adding new creatures expensive activities compared to doing the same with a normalized schema. As described elsewhere, we will tend to keep our tables normalized, but aid our overall performance in following ways:

1. Perform joins on columns that are system assigned keys with good cardinality values2. Add an index that verifiably speeds access if a join column is not the primary key on a table.3. Keep the number of different tables in the join down to the minimum, particular if the query will return a large number of rows from one or more tables.4. Offload the query to server startup time when we can better afford slower than optimal performance.

There are two types of join operations available: inner and outer joins. The easiest way to distinguish between them is to ask the question if the result of the query should return a result only if there are matching rows on all tables involved in the join (inner join) or if there needs to be row is returned from the main table regardless of whether there are matches on the other tables involved (outer join).

Let's continue with our Creature and Spawner example to illustrate. If the Creature and Spawner tables are joined on CreatureId, do we need to see a row for every creature even if it has not been assigned to a spawner? If the answer is yes, then an outer join is required. If the answer is no, an inner join will suffice.

Note that there is no 'correct' answer; it depends on the dataset needed to meet our requirements. In the case of an outer join, the value NULL will appear for all columns from the joined table without one or more matching rows.

Null Value

The concept of the NULL value is worthy of its own topic because of the confusion it can create. The NULL value in a column should be read as "unknown", unlike the C programming concept where NULL has an actual value.

Any operations attempted against a NULL value will give NULL as the result. Rows with NULL values on join columns will not match each other, because NULL does not equal NULL in a comparison. NULLs cannot be allowed on any column that is part of a primary key. However, it is possible to test a column to determine if it has a NULL value, and columns that allow NULLs still honor any constraints in place when changed to a non-NULL value.

Used appropriately, NULL is quite useful. Designate a column to allow NULL if the value in that column is not always applicable. For example, if a creature may be optionally assigned a Spell, the SpellId column on the Creature table would accept NULL. If a creature currently does not have a Spell, then its SpellId column would be NULL. To return every creature without a spell, you could write the following query:

SELECT *FROM CreatureWHERE SpellId is NULL

NULL columns are overwhelmingly preferable to synthetic "not applicable" values, which are the moral equivalent of magic numbers in code.

Referential Integrity

In my opinion, the use of referential integrity (RI) within a database is not optional. One cannot afford to let inconsistencies in data creep into the game. RI may reveal bugs in game logic or even cause code to break. But either of these is much more desirable than having errors lurk undetected until who knows when. RI can and should be used to prevent exploits by players, such as item duplication. Think of RI as an extra set of eyes that is always checking to ensure data consistency and correctness as the game runs. It is much more expensive to find out after the fact that things have gone bad, so use RI as a prevention mechanism.

Because it is essential that data be kept internally consistent, databases typically provide numerous mechanisms for implementing RI. These features usually encompass the following three major areas:

1. Foreign Key Relationships/Unique Constraints 2. Column Constraints3. Triggers

You should take advantage of key based constraints by striving to establish a unique key for every table in the system. This prevents duplication of the key value and removes any ambiguity as to which row in a table is the 'correct' one. Every time a primary key shows up on another table, assign it a foreign key relationship with delete prevention. This way, any primary key value that shows up on a separate table will be prevented from being deleted. For example, on the CreatureSpawner table, CreatureId is a foreign key from the Creature table. With RI in place, one cannot delete a creature if it is referenced on the Spawner table.

Databases typically offer the option to cascade the delete of a key through to all its related associations. Exercise caution with this option since it works silently and you have little indication when something unintended is happening. Instead, you might find that some data has 'gone missing' because of the effects of cascading deletes.

If supported by the database, definitely take advantage of column constraint features. These allow for simple rules to be added to tables that limit the valid domain of values for individual columns. They are very easy to implement, and should be added as at table creation time. As bugs are discovered, add new constraints if they will help prevent their recurrence in the future.

To constrain the HitPointAmount column on the Creature table to the accepted range, one would add a rule to Creature table stating "HitPointAmount between 5 and 1500". The database will guarantee from that point forward that an out of range value is rejected, regardless of its source.

The last major referential integrity feature is the trigger. Triggers are snippets of code associated with a table that run when data is manipulated on the table. They are somewhat more difficult to implement than the other methods of RI. However they are the only mechanism by which referential integrity can be maintained

across multiple tables when a foreign key constraint is insufficient for the task. For example, to ensure the HitPointAmount of any creature always exceeds the minimum base damage amount of any weapon in the game requires a trigger. The following is an example of the implementation of that trigger:

Create Trigger CheckCreatureVsWeapon On CreatureFor Insert, Update AsDeclare @maxBaseDamage int, @hitPointAmount int

SELECT @hitPointAmount = HitPointAmountFROM INSERTED

SELECT @maxBaseDamage = MAX(BaseDamageAmount)FROM Weapon

If @hitPointAmount <= @maxBaseDamageBeginRaiseError('HitPointAmount of creature does not exceed max base weapon damage')Rollback TransactionEnd

The time spent developing triggers will pay off handsomely in overall bug prevention. Humans are prone to mistakes, and triggers are there to ensure that those mistakes are caught and don't do any harm to the game.

The last thing to be aware of is that the creation of these RI mechanisms will occur at any time in the life of a game. Be sure to take advantage of the features which validate that any existing data conforms to the new rules being added. Failure to do so will allow bad data to remain in place until a subsequent change is made to the violating value(s), which is something we obviously want to avoid.

In-Game Logging

One of the more compelling uses for a database with an MMP to record all important player actions and chat text so that there is a detailed record of what transpired in game. The logs are invaluable because they:

1. Provide information needed to discover exploits 2. Simplify understanding and tracking down bugs3. Arm customer support with information needed to explain 'what happened' to a customer4. Record when players abuse or cheat other players5. Impartially report the 'facts' for dispute resolution.

The following should be considered when creating a logging system:

1. Log all transactions that have true game play impact. The location where something happened can be very useful, but it is unnecessary to record every player movement in the world. Doing so will generate an overwhelming amount of data and make it difficult to keep the system responsive. Instead, if necessary, record the location of the player at the time they perform an action that impacts game play.

2. Logging is still going to generate a vast amount of data. Consider a separate logging process that writes to the database independent of the normal game persistence mechanism, avoiding undue impact to the game loop. Also consider hosting the logging process on a separate server to avoid resource contention while the game is running.

3. The process of querying logging data from the database while the game is running can impact performance. Consider that the logging database be separate from the game database so that interacting with the logging data does not impact game performance.

4. Plan in advance if there is any inclination to consolidate logs across game instances. The sheer

duration of performing such consolidation across servers of a popular game can get prohibitive. Avoid this problem by making the logging directly to a single central database shared across games, consolidating on the fly. But be sure to have plenty of disk storage on hand!

Security

Modern databases have robust security features that may be used to secure and partition access to the database schema by user. If desired, security can be as specific as allowing a user to only view the values of individual columns in a table!

Having to manage security during development when objects are being rapidly added and modified can become a cumbersome task that is difficult to keep up with. In practice, it may be easier to create 2 security profiles. The first would be a user that is granted the ability to do any development action desired: create and drop tables, add stored procedures, etc. The second profile would be the "read only" user. This user has access to view everything in the database, but is prevented making from any modifications. These two types of users should be sufficient during development.

Once the game is ready to go live, it becomes important to implement a more robust security policy. Live data must be protected because of the expense of accidental modification or even malicious tampering of player data. Restoring from backups can help resolve issues, but preventing the problem from ever happening is always better, right?

Perform a careful analysis of the type of access needed by various people in the organization. Keep data opaque to those people without a need for access to perform their duties. At a minimum, provide individual logins for those that do require access so that there will be an audit trail of every modification to the database originating from outside the game. This is extremely important, and with the security mechanisms available, there is little excuse for not keeping people accountable for modifications made to valuable company data.

While there are some significant challenges to implementing a database on an MMP game, potentially the biggest hurdle to overcome is the cost factor. This topic is presented so that you are not blind sided by all the potential costs that may crop up when using a database.

The first potential expense is licensing the database software. One way to deal with this is to go the open source route. There is at least one competent relational database solution that is open sourced, so avoiding potentially high licensing fees is not out of the question. Do not however make the selection solely on the basis that it is 'free'. Instead, do a head to head comparison with commercial offerings to ensure all the requirements of your game are met satisfactorily by the chosen software.

When choosing the commercial database route, you will quickly come face to face with licensing considerations. The commercial offerings tend to be licensed on the basis of CPU configuration, but this does vary among vendors. Pricing changes based on CPU count on the database server, and there can be big jumps when moving between configurations. Vendors also typically charge a premium for "enterprise" setups, which typically allow unlimited user connections to meet the needs of large corporations.

It is worth the time to be studied up on your selected vendor's licensing; perhaps the architecture that you plan to implement does not require the enterprise level offering that can be so expensive. While a monolithic, 32-CPU centralized database server may be desirable because of its simplicity, what is the impact to the service if that machine has problems? Having a redundant fail over just sitting around will be costly, and would need to be accounted for in the overall architecture.

Alternatively, would it be preferable to run a single, much smaller (and less expensive) database server per game world instance? How would the architecture handle data that is shared across game world instances, such as player login? All these considerations require a delicate balancing act between cost and ease of implementation.

Figuring out an appropriate licensing strategy will be a challenge. If you have any doubts, then check out the extensive online FAQ sheet and White Paper that

Microsoft has devoted to the subject.

One thing is always true. You will do yourself a favor by treating the database as valuable resource that needs to be used utilized efficiently and creatively, as opposed to something that gets thrown onto bigger hardware every time there is a performance problem.

If you are part of an organization that is large enough to have a dedicated IT department, you should immediately investigate what they are using for their database software and leverage that knowledge. If they already have a suitably robust solution, you will want to jump on that bandwagon because of the experience and knowledge that they have developed in house. The opportunity for savings in terms of acquiring expertise, understanding licensing issues, and so forth is worth its weight in gold, especially if your own experience with databases is light.

The next cost related item to consider is performance versus ease of use. The leading database vendor, Oracle, has a reputation for achieving the highest performance, but at the expense of ease of use. It provides the ability to tweak and configure every nuance of performance, but doing so is enough of a black art that Oracle expertise commands top dollar in the open market. It may be worth sacrificing some performance for a solution that is easier for mere mortals to administer, such as MS SQL Server.

You may need to account for software costs related to working with your chosen database software. Does the vendor provide competent tools to administer and utilize the product, or is it necessary to procure third party offerings to avoid misery? It is not a given that a product will be easy to use out of the box, or even that tools are provided gratis by the vendor. While vendors have come a long way in developing friendlier tools with graphical user interfaces, there remains a thriving market for third party database software. You may be surprised if you presume to have no expense in this area.

The use of vendor provided tools would almost certainly lead to an outcome that could be undesirable for your organization. For obvious reasons, vendor tools will be proprietary in nature and incompatible with competing database offerings. Any significant development with a vendor tool or library will tend to "lock" you into their product, because that investment is sunk should you desire to migrate to another database. Third party tools are much more likely to give you the ability to

switch out, since it is beneficial to their business to support multiple database platforms.

Will you want a tool that provides you the ability to maintain the entire database schema for the game within an easy to use graphical interface? More than likely, you will have to pay for the convenience of such a tool. Fortunately, using MS SQL Server, we benefit in that the Enterprise Manager application that comes bundled with the database provides us this functionality in a competent manner.

How do you plan to interface the rest of the game with the database? Are you going to utilize the vendor provided libraries, or do you utilize a language (such as Python or Java) that a third party excels at supporting? If the latter, you will need to account for licensing the third party libraries. On my project, we have licensed mxODBC, which is a very simple-to-use multi platform Python database library built on top of ODBC, an industry standard database independent C based API. This has given us the flexibility to develop our database server on Windows, but later migrate to running on Linux. To talk to MS SQL Server from Linux, we licensed the Easysoft ODBC Bridge software. This allows us to continue leveraging our company investment in MS SQL Server, while running servers on Linux, where the vast majority of my team's server expertise resides.

The last significant area of expense in software is that of data entry or query tools. It is great to have the game built on a database, but without tools to enter data and generate reports, a significant piece of the puzzle is missing.

For small tables, one can likely get by with the standard tools from the vendor to perform data entry and querying. That quickly becomes impractical when working with tables with large amounts of data. On the Windows platform, products such as MS Access and Visual Basic can be used to create custom forms that make it easy for designers to perform data entry. Of course, resources have to be allocated on the project to build these custom forms and reports.

Database software has to run on hardware of course and the more robust the better. The database will perform better on machines with fast and large hard drives - think SCSI and RAID here. The software will require enterprise quality operating system software; you'll be running some version of Windows NT Server or variant of Unix. The software performs better with multiple processors on board, with each processor having as much on board cache as possible, and the software will need a very fast network connection as close to its clients as

possible, with potentially multiple network cards to maximize bandwidth. In other words, procure the fastest, biggest, and fattest that the budget allows. No bare bones $1500 PC allowed here!

Finally, you should account for personnel related costs. Development budget must be allocated to administer the database, design and create forms and reports, and implement the database server architecture and stored procedures needed to support the game.

In addition, if there is no one with database expertise in house, you will need to provide training to your staff, or bring in a consultant/contractor. Having in house expertise is very valuable and worth the extra hit on the budget to acquire or develop, especially when problems crop up.

If you are a developer with an outside publisher, look into whether they have some database expertise to offer. Most experienced MMP publishers already understand the benefits of a database and will likely be very willing to help you get started in the right direction. Also don't forget your internal IT department, if your company has one. You may be surprised with who you find lurking there. And no doubt working on a database for a game will be quite an interesting draw to them!

Hopefully all this cost related discussion hasn't caused you to question your decision to go with a database. As expensive as it can all turn out to be, it will likely not match up to the cost of not using a database on your MMP game.

Architecture

Currently, most MMOGs run multiple game server instances, in which each instance is a unique world for players. Even if that is how you plan to proceed, it is not immediately obvious how the database should be factored in.

You could have a large monolithic database box that supports all of the game instances from a single point. In a global marketplace, this is likely to be a regional or country server, since you'll want the game servers and database to be in close proximity for the best connection possible.

If proceeding with a monolithic database box, how does the database get partitioned so that it looks unique to each game world instance? Should there be a single database instance, or a database instance for each world instance? This will

require some extra strategizing, as will planning for the impact of outage to the hardware. To ensure that all worlds are not all down at once, the game servers will have to be prepared to switch over to a backup at a moment's notice. Achieving this without interruption to gameplay will undoubtedly be quite challenging.

Implementing a separate database server for each world instance may seem like the easiest and logical solution. But it can get complicated if there are requirements for data to be shared across game world instance, such as a player reputation, or account name and password. How does one keep this data unique across servers? And with separate servers, there is the cost of consolidating data when it is desirable, such as for data mining.

In addition to the overall architecture, one has to consider the internal architecture for how the game talks to the database. Should each game server have a separate thread for performing database actions, or should they communicate to another process that interacts with the database? If you implement a separate database server process you'll have to consider where that process is going to reside. Does it reside on the database server machine itself, or does it co-exist with the other game servers?

Should data be cached? Caching will definitely help overall performance and reduce bottlenecks due to constant database access. But if data is cached, an unexpected interruption in service will cause any activity that hasn't been written to the database to be lost. Is this something the game can afford? How upset will this make players?

On the other hand, failure to cache data likely leads to a high volume of activity in the database. Will the database be able to stay responsive when the load gets very heavy? A database can easily be overload if one is lax in applying good judgment in table design and building queries.

If a game world is partitioned and there are definite boundaries between these partitions, deploying a database server process per partition becomes a viable option. Since players cannot be in two partitions at once, transactions are guaranteed to be atomic, and the overall load is divided among multiple database server processes.

Backup and Restore

One of the most useful features of a relational database is the ability to perform a backup without service interruption, also known as a 'hot backup'. Yes, there is likely to be some performance impact during a hot backup, but it can be kept to a minimum by selecting the appropriate backup device(s). For instance, make backups to a fast SCSI based RAID device as opposed to tape. Once the backup is created, it can always be independently moved to tape for archival purposes.

MS SQL Server also supports the concept of snapshot backups for databases requiring high availability. These are services that used in conjunction with third party offerings that can perform backups of large databases in seconds with little or no impact on the database server. This is a more expensive option than the standard backup mechanism, but may prove desirable for your situation.

It may be obvious, but is worth stating anyways: bad things can and will go wrong. The more recent a backup you have of the game state, the less unhappy your player base is likely to be when you have to restore from a backup.

Perform a hot backup as often as you can get away with without overly degrading game responsiveness. Players don't like to have to put up with any kind of loss, but the smaller the amount of "time warp" a player has to experience when a restore is necessary, the more palatable it will be.

Sometimes, circumstances make a "time warp" unavoidable. In those instances, you may choose to refer to the game logs and return items or other in game earnings lost by players because of the restoration of a backup. Fortunately, doing this is relatively simple using the power of queries in the database.

It is not feasible to restore the database while the game is running; instead the game will have to be taken off line. Most of the time, the need to restore is forced on you by some catastrophic event, so it's not likely you "scheduled" this down time. Instead, scheduled maintenance down times will be reserved for modifications to bring new content online or fixes for exploits or cheats that are hurting the game.

Other Departments and the Database

The operations staff will of course be primarily handling database backup, restore and maintenance activity. They will very much appreciate your decision to rely on a database for the game. It makes this part of their jobs a well understood quantity

and there are a lot of external resources available for help should the need come about.

The database allows customer support to easily correct issues that are reported by players or uncovered through internal investigation. The only requirement of the player is that they not be in the game when fixes are applied. The player then sees any corrections made on their behalf the next time they login.

This is a certainly a powerful capability, and has the potential for abuse. However, support employees are a lot less likely to give in to temptation when they know that the database is recording every transaction that occurs, and that it is extremely easy to track down the source of suspicious or illegal activity.

Having the support staff utilize the database to manage the game also potentially frees development from having to create a "super client" or other proprietary mechanisms for administering the game when goes live. If desired, a suite of simple web based reports and forms can be built on top of the database to remove the need for the majority of the support staff to be conversant in SQL. I strongly suggest that you cannot have too many people in house with database skills, and that you should encourage those that are interested in learning.

Customer support can also perform data analysis to discover exploits and cheats. While referential integrity will help cut down on bugs that are exploited, what is 'correct' data in the database can in fact be problems when someone asks the right questions.

Triggers or queries should be written to report when data is legal yet falls out of 'normal ranges' so someone can take a look into why the variances exist. For example, the database can reveal if players are accumulating a rare item at a rate that is statistically out of bounds. It can also identify that players are accumulating in game currency at significantly faster rates than expected. Not every anomaly reveals a cheat or exploit, but with a database it is trivial to casually analyze the integrity of the game without any player impact.

One side note: a lot of analysis activity involves running complex queries against live data. However the queries need not run in the current live database. Instead, to minimize any impact to the running game, such analysis should be performed on a very recent backup restored to a separate database instance.

Marketing will want to perform data analysis to aid their efforts. More than likely, they will want summary reports. These reports can take a long time to run, and could potentially impact overall game performance. However, just as you would for support data analysis, provide Marketing with read access to a recent backup from the live game. When developing the game schema, you may want to consider that Marketing will likely want to generate reports for the entire game universe; as opposed to generating these per game world instance should that be your chosen architecture.

The community team will be able to leverage in game data to provide additional player content that does not require the player to be connected in the game. For example, one of my employer's products, Lineage, offers web pages that allow players to see the items they have in their inventory and to make it publicly viewable if they desire. The web content also reports who currently owns the castles in various regions of the game world, and their current taxation rate. Having information accessible from a database makes providing such value added features embarrassingly simple.

Of course, it should go without saying that interacting with these various departments while the game is still in development affords you the best opportunity to meet any specific needs they have. They know where their challenges occur, and with the database you have a robust mechanism to aid in alleviating these.

Schema Maintenance

During development, adding new objects to the database is quite straightforward. However as a schema grows over time, changes to the game design or bug fixing will call for modifications to tables andstored procedure that are already utilized by code. Applying the desired modifications will potentially break this code, causing delays for the team as it waits for the code fixes to occur.

Both because it is a beneficial activity, and keeps your colleagues happier, make every effort to minimize the impact of schema and stored procedure modification. Build code that is resistant to change by understanding the types of changes that can typically occur. It will be common to add and remove column(s) on tables, or add or remove columns from a query result set. The data types of columns may change, as can the names of columns, tables or stored procedures. With this in mind, determine a strategy for how the code will handle these types of

modifications. If it is not possible for the code to continue to run, at least structure it so that it can be easily modified to accommodate changes as the need arises. Finally, if an operational interruption is unavoidable, use email to announce it ahead of time to the team. This will help everyone maintain their sanity!

Have QA perform testing on a separate snapshot of the database and code. This isolates development from QA and vice versa, and both parties will be better off with this scenario. When QA has tested everything, they can 'bless' the build, approving both the code and the compatible version of the database.

Maintenance issues get magnified in scope and importance on a game that is live. The good news is that there will be regularly scheduled service windows when the game is offline to perform database maintenance. The bad news is that any maintenance has to be applied in such a way as to retain all of the players' progress in the game up to that point.

Performing this task can be daunting, especially if the maintenance is a new content publish. It's almost a given that something will go wrong if a human being has to make the changes manually.

Fortunately, there are third party tools available that will perform both a structural and data level comparison between database instances and generate the scripts necessary to make one database look like the other. Trust me; this is money that is well worth spending. Use such a tool to create the scripts and verify they perform correctly against the most recent backup of the live database. Then during the service window, apply the same scripts to the live database and you should be free of any surprises.

Don't forget to make a backup of the database at the beginning of the service window after players are forced to exit the game. That way, if the planned modifications fail for some reason, you are able to quickly restore service using the backup without any loss to players. This buys time to resolve the issue independent of the pressure to get the game back on line.

Conclusion

If you are serious about the overall integrity of your game, and you want to provide the highest level of service to your players, you really have little choice but to utilize a database solution. Consider this as well: If you do forge ahead

with a database, you will have a significant competitive advantage over those in the industry that decide against it. And perhaps more importantly, you won't be at a significant disadvantage compared to those in the industry that are already on board.

Without a doubt, there are many issues to consider when choosing to implement your MMP game on top of a relational database. But with careful consideration to the information presented here, I am confident you can avoid many of the pitfalls in the process. In the end, the benefits reaped from a robust database implementation will far outweigh the negatives, especially when the game goes live.