Application level replication in Joomla! using graphs

Sam Moffatt, Joomla!

September 13, 2009

1 Introduction

Joomla! is an open source PHP/MySQL based content management system that has a flexible extension interface that allows users to install multiple third party extensions.

Typically a Joomla! user isn't an organisation that maintains its own web presence within a local data centre to the organisation. The user might not even have a colocation in a remote data centres. Typically the user isn't even on their own dedicated server and is using a shared hosting environment.

MySQL as a database system provides replication between database systems in a relatively useful model using multiple techniques to handle scaling. MySQL works well when this is configured for it to work however for the typical Joomla! user who is on shared hosting this method of replication is unavailable. MySQL replication also falls over for weakly connected nodes in a replication tree such as a laptop which might contain a copy of the system however it may only wish to replicate with the master system occasionally.

As we can't utilise MySQL's replication techniques we need to design our own method for handling replication. The system should also be capable of handling conflicts potentially as well as maintaining referential integrity between nodes when an update in a given table occurs in one place or another between any given replication sequence. Any given system might be at various stages of synchronisation with different items.

2 Scope limitations

What is not covered in this implementation is handling multiple masters. For this system, one device has to be nominated as the master to retain stuff like keys across the system which is important for linking. This makes this system inappropriate for real time replication between master and multiple slaves where they might be each accessed independently in an authoritative manner. The replication technique described here is ostensibly aimed at providing a method for offline editing of the site or working copies where you might wish to push or revert updates from the system.

Because of this the only consistent version of the data is stored on the master copy of the system. All other versions should be considered inconsistent if they have any changes. All subservient copies will have themselves reindexed upon receiving updates from the primary system. Since we might have a situation where two tables that are relationally dependent could have their keys altered so to maintain referential integrity as mentioned earlier.

An issue that could potentially arise is a patch being applied to the master without the change being noted locally. At this point we assume that receiving the update could potentially work in conflict resolution as well. A potential solution that requires further examination is that of implementing patchset stores. Any system could store patchsets that they receive or patchsets that they send. Each patchset could have a unique identifier (UUID) which would permit applications identifying changes that originate from themselves to permit automatic merging.

3 Problem

So the problem that is presented here is building an application level replication system that handles transferring data between different nodes in a manner that maintains the referential integrity of the system.

Replication between systems should be portable: it might use HTTP as the transport or it might be the distribution of a graph within a file format that is transmitted using other means.

The primary issue that can occur is that the referential integrity of the system can be compromised if a graph is not applied in such a way where each dependent edge is updated properly when linked nodes have their keys altered.

Design wise Joomla! helps us here with a strong usage of MySQL's auto_increment feature that means that integral values are used extensively as the primary keys for multiple tables. This style is propagated through the framework with this being the default key storage method used for the framework's data storage API's.

4 Solutions

To solve the issue we have presented we are going to use a graph to handle transferring the information between different systems. In constructing the graph we need to obtain a set of dependent edges to construct the graph between the changed nodes. If the database system and Joomla! properly modelled its foreign key dependencies we could easily build this graph by examining the database structure for the foreign key constraints and use them however this is not the case so to resolve this we will implement a set of files that a developer can opt in to use.

Interestingly Joomla! stores references to foreign key within regular columns in its table (e.g. in a field which stores the same value) but it also stores some references within a special field typically called 'params'. The params field is something used to store semi-structured data related to an individual item and can also store extra information within it. This is used in tables to store stuff for configuration items ostensibly which aren't known at database creation time as they are added by the third party extensions.

XML Dependency File:

```
<?xml version="1.0" ?>
<replication>
<table name="#__sample_table" comparison_fields="test,catid">
<column name="category" table="#__categories" column="id" />
</table>
<table name="#__modules" where="module='mod_sample_table'">
<param field="params" name="catid" table="#__categories" column="id" />
</table>
</replication>
```

So what this permits is the ability to update links within fields themselves but also support for referential integrity within the params fields. This example works on the modules table adding a rule to link the param 'catid' back to the categories table. It also introduces a constraint to ensure that we don't pick up entries we don't care about or don't actually have those referential integrity constraints.

So at this point we now have enough information to build a graph of dependencies if we wanted to replicate the system. What we need to do now is to generate the graph to handle the update.

Consider the following master system:

#__categories	
id	name
1	Beef Pies
2	Apple Pies

#__sample_table		
id	test	catid
1	A	2
2	C	2
3	B	1

#__modules		
id	module	params
1	mod_sample_table	catid=2

Both tables also feature a column that handles timestamps to enable updates to be processed properly which is left out of this example. A child system adds an extra entry to the categories table called “Pumpkin Pies” whilst the master system adds an entry called “Chicken Pies”. At this point we have a collision as they will have the same ID in each system. At this point we don’t have any dependants relying on the primary key so when the child applies its update to the master, the child’s key is updated (at this point it will be ‘4’) and the new entry from the master server will be applied down to the client.

If before the replication the child site adds a new module which refers to the new category the child has added as well as an entry in the sample table that refers back as well. At this point if we merely update the key of the category then both the modules table and the sample_table table will be inconsistent because they will end up pointing to the “Chicken Pies” entry in the category table instead of the correct ”Pumpkin Pies”. So we need to create a dependency graph that permits us to update parts of the system.

Due to the design of Joomla!, building the graph can be taken into two parts: the initial node detection and allocation and then the graph link creation. In reality node detection can occur only on the client whilst graph link creation can occur on any location that has the XML dependency files for the relevant table. However for performance the system will build the graph within the system that is creating the ’patch’ which can then be cached in a way that can easily load the data up into the destination system freeing it to handle collision resolution and then updating the graph as relevant.

So to construct our graph we split this into building the initial set of directed edges and determining the root node. The node with the least amount of outward edges and if we have multiple nodes that satisfy this criteria then they each become the root nodes of the graph. In our example our root node would be the categories table because it has only inward edges and no external dependencies, however it might be not be the case. We could end up with a co-dependency with our root nodes so at this point we prefer the node with the least amount of inward edges and if this still results in more than one item we pick any of the options at this point.

Once the root nodes of the graph have been determined it permits two operations: unapplying the patch from the modified non-master system (this effectively results in removing any of the changes and deletes them reverting auto_increment numbers) and applying the graph to the destination system. In applying the graph the system can also choose to partially apply updates presuming all outward dependencies of committed nodes are met. So in our example above where we have changes in all three tables, we could apply the update to the ’categories’ table without applying the updates from the other two tables. So partial application of a graph permits partial patching of any system within the dependency constraints.

Deletion of data presents issues as well however deleted information will exist as an orphan as it isn’t updated it won’t be replicated. However this means that a deleted item

will only disappear from the system it was deleted from using the present model.

The final aspect of the system is conflict management. Earlier it was discussed that with the new category that it would be added to the database however there might be a conflict due to the fact that the item in fact already exists on the system. In the initial dependency XML file there is an attribute for table's called "comparison_fields" which enables the developer to nominate a few quick fields to easily compare and look for potential duplicates. When looking at applying a conflict the system can then request the user resolve the issue providing a meaningful alternative to merge (keeping in mind that selecting an existing item as the primary key behaves the same as creating a new primary key as this could cause the graph to update its links). This may be ranked as a level of importance however any existing entry might be valid. All of this is related purely to master system as all child systems accept changes from the master system without question.

5 Outstanding problems

In the event that a change can be applied in a non-conflicting manner it will be automatically applied. If for some reason 'duplicate' entries with different non-conflicting primary keys are committed then in effect we will have disjoint database. Since we have access to the relationships the tool could potentially be designed to 'refactor' entries into a single entry. A tool for handling merges between different entries so that a conflict isn't a left or right style proposition (e.g. take existing value or create new value) enabling people to merge new information into an existing row. This can be used with the refactor tool as well which would permit graph wide updates in an intelligent manner. Composite keys aren't handled by the system however typically Joomla! doesn't utilise such keys.

6 Implementation

In building this system we use a combination of tables to handle data storage. Since PHP doesn't handle large structures well and what we're working on is quite potentially large, the data will be backed into the database for storage whilst the edges are constructed. Fortunately edge construction can be done progressively by examining simple rules which leaves two implementation level problems: determining the least most connected node and updating nodes progressively.

Support Tables The system has three major support tables:

- #__graphrepl_graphs
  - Graph ID
  - Graph Name
  - Graph Timestamp
  - Graph Source
- #__graphrepl_nodes
  - Node ID
  - Graph ID
  - Node Table
  - Node Table ID
  - Node Data
- #__graphrepl_edges

- Graph ID
- Source Node
- Destination Node
- Link Type
- Link Details

The graphs table stores entries for the graph construction, the nodes table stores the nodes of a given graph and the edges table stores the edges of a given graph. Simple construction. So say we take our example from the above system. Utilising the common master documented above, we have System A and System B with changes. These systems look like the following:

System A

```
#__categories:
id | name
---+-----------
1  | Beef Pies
2  | Apple Pies
3  | Chicken Pies

#__sample_table:
id | test | catid
---+------+------
1  | A    | 2
2  | C    | 2
3  | B    | 1

#__modules:
id | module           | params
---+------------------+--------
1  | mod_sample_table | catid=2
```

As you will recognise, this is the same as the proposed update to the master system - and this will be committed first to the master. Now consider System B:

```
#__categories:
id | name
---+-----------
1  | Beef Pies
2  | Apple Pies
3  | Pumpkin Pies

#__sample_table:
id | test | catid
---+------+------
1  | A    | 2
2  | C    | 2
3  | B    | 1
4  | D    | 3

#__modules:
id | module           | params
---+------------------+--------
1  | mod_sample_table | catid=2
2  | mod_sample_table | catid=3
```

So we can see we've got a conflicting update to #__categories and two apparently non-conflicting changes to #__sample_table and #__modules. So quickly we have the following changes: - Added Category ID "3" with name "Pumpkin Pies" - Added Sample Table entry "4" with test value of "D" and category ID (catid) of "3" - Added module ID 2 with module "mod_sample_table" and params of "catid=3"

Sounds quite simple however the category entry is dependent in both cases, so lets see what this would look like in the nodes table (ignoring the 'graph' table and graph ID for the purposes of the example and the contents of the data field for brevity):

```
#__graphrepl_nodes
ID | Table           | Table ID
---+-----------------+---------
1  | #__categories   | 3
2  | #__sample_table | 4
3  | #__modules      | 2
```

Looks pretty simple, its a list of tables and primary keys - the real table would also store a serialised copy of the data as well in the node data field. So lets see what the edges looks like (graph ID again absent):

```
#__graphrepl_edges
Source | Dest | Type  | Details
-------+------+-------+--------
2      | 1    | Field | catid
3      | 1    | Param | params/catid
```

Again quite simple, the source and destination fields are foreign keys to the nodes table, the type is the method the link takes place and the details field stores the details about the link, for field types its the field name and for param types its the field name of the param field followed by the name of the param.

So at this point we're ready to apply the graph to the system. To start with, we find the node that is the source of the least most edges. This is a combination of two queries: the first is a simple statement which selects any node that doesn't have a source (e.g. it isn't dependent on any other node) and if this fails to return an entry we search for the node with the least number of source entries. If we find two entries that have the same number, we try to pick the one with the least amount of edges and at that point if we have a tie we pick the first entry from the list returned. At this point we apply the node to the new tree and update any dependent nodes if the key changes. So for example if we applied the categories change we could see that the table ID changed from "3" to "4" so we would then traverse the edges to update the sample_table and modules copy of the field. Once we've applied the node, we can work on applying the next node in the graph. We use the same selection process as before to determine the node and then apply it. Interestingly graphs can be partially applied presuming all dependent nodes of the nodes being applied are also applied. So in this case we can apply the new category without applying the others presuming that those other entries are appropriately updated.