SQLite Development with Zumero

SQLite Development with Zumero

Copyright © 2013 Zumero LLC

Table of Contents

1. Introduction 1
  1.1. What is Zumero? 1
  1.2. Why Sync? 1
  1.3. About this guide 1
  1.4. Registering Zumero with SQLite 2
2. Getting Started 3
3. Zumero tables vs. regular SQLite tables 4
  3.1. Creating tables 4
  3.2. Identifiers 4
  3.3. Conflict clauses 4
  3.4. Foreign Keys 4
  3.5. Tables are permanent 5
  3.6. Optional feature: Strict Type Checking 5
  3.7. Limitations of SQLite virtual tables 5
    3.7.1. Indexes 5
    3.7.2. Adding columns to an existing table 6
    3.7.3. Incremental blob I/O 6
    3.7.4. Triggers 6
4. The Zumero Server 7
5. Conflict Resolution 8
  5.1. Row conflicts 8
    5.1.1. Rules: Situations and Actions 8
    5.1.2. Column Merge 9
    5.1.3. Large text fields 10
    5.1.4. Defaults 10
    5.1.5. Example Snippets 11
    5.1.6. Audit Trails 11
  5.2. Constraint Violations 12
    5.2.1. Example: CHECK 12
    5.2.2. Example: Foreign keys 13
    5.2.3. Example: UNIQUE 13
    5.2.4. UNIQUE constraints: An ounce of prevention 14
    5.2.5. SQLite "INTEGER PRIMARY KEY" columns 14
6. Quarantined Packages 18
7. Security 19
  7.1. Authentication 19
  7.2. Permissions 20
    7.2.1. Access Control Lists 20
    7.2.2. ACL Entries 21
    7.2.3. Searching an ACL 22
  7.3. Internal Auth 22
  7.4. Securing a newly installed server 23
  7.5. Example: Free Private Wikis 24
8. History 28
  8.1. How History is Stored 28
9. The Server Log 29
  9.1. Permission to access zumero_log 29
  9.2. Columns in the log table 29
  9.3. Sample zumero_log queries 30
10. Reference: Functions 32
  10.1. Network activity 32

10.1.1. zumero_sync() 32
10.1.2. zumero_pull_without_history() 33
10.1.3. zumero_internal_auth_create() 34
10.1.4. zumero_internal_auth_add_user() 35
10.1.5. zumero_internal_auth_add_alias() 35
10.1.6. zumero_internal_auth_set_password() 36
10.1.7. zumero_internal_auth_set_acl_entry() 37
10.1.8. zumero_get_storage_usage_on_server() 38
10.2. Functions without side effects 38
10.2.1. zumero_internal_auth_scheme() 38
10.2.2. zumero_auth_scheme() 39
10.2.3. zumero_named_constant() 39
10.3. Convenience 41
10.3.1. zumero_define_acl_table() 41
10.3.2. zumero_define_audit_table() 41
10.4. Adding conflict resolution rules 42
10.4.1. zumero_add_row_rule() 42
10.4.2. zumero_add_column_rule() 42
10.5. Misc 43
10.5.1. zumero_alter_table_add_column() 43
10.5.2. zumero_adopt_existing_table() 43
10.6. Quarantine 44
10.6.1. zumero_quarantine_since_last_sync() 44
10.6.2. zumero_restore_quarantine() 44
10.7. Destroying things 44
10.7.1. zumero_purge_history() 44
11. Reference: Error Messages 45
12. Frequently Asked Questions 46
Glossary 53

Chapter 1. Introduction

1.1. What is Zumero?

To describe Zumero, we first describe SQLite.

SQLite[1] is a lightweight (but surprisingly powerful) implementation of a SQL database.

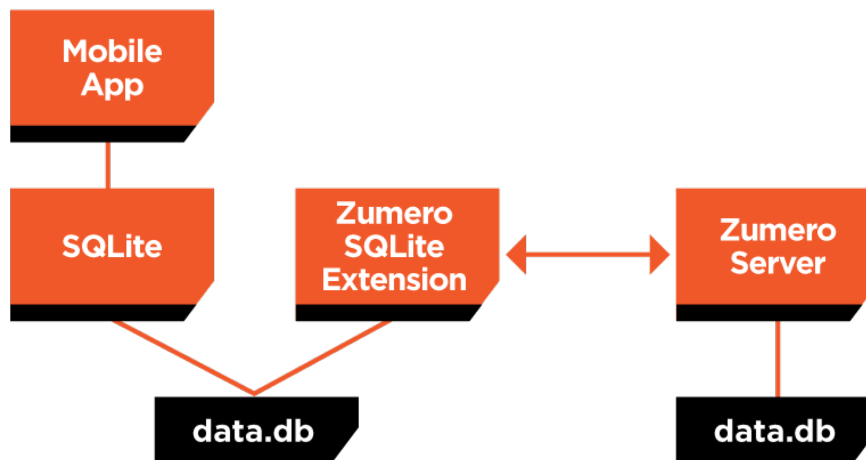
SQLite is the standard database software for iOS, Android, and Windows RT. It is installed on over a billion mobile devices.

But like any other computer, and perhaps more so, a mobile device is not isolated. It needs to share data with a server.

And SQLite has no synchronization capabilities.

Zumero solves that problem.

Zumero is "sync for SQLite".



1.2. Why Sync?

Many mobile apps are built on the assumption that the network is reliable and always available. Whenever the app needs to access the database, it makes a REST call to the cloud. Most user operations require network activity.

Zumero enables a "replicate and sync" model for mobile database apps. The device has its own copy of the database. The app can interact with the database without involving the network, so queries and updates are faster and more reliable. Synchronization with the server can be handled entirely in the background without the user waiting. When no network connection is available, the app continue to work offline.

1.3. About this guide

This document is primarily focused on explaining how to use Zumero from within SQLite statements.

[1]http://www.sqlite.org/

Familiarity with SQLite development is assumed. Furthermore, this document does not attempt to cover the wide variety of ways that SQLite can be used from with different languages and platforms. We do not talk about how to execute SQL statements with SQLite. Rather, we talk about how to construct SQL statements that make use of Zumero features. This document is applicable regardless of which platform or language you are using.

When this document discusses Zumero features that are planned but not yet implemented, the word "LAT-ER" is used.

This document was generated 2013-06-26 11:24:35.

1.4. Registering Zumero with SQLite

Whatever platform you are using, before you can use the new features provided by the Zumero SQLite extension, you need to *register* it.

The native SQLite API is C-based[2]. At the C level, you need to call zumero_register(), passing it the SQLite database connection handle[3]:

```
#include "sqlite3.h"
#include "zumero_register.h"

sqlite3* db = NULL;

sqlite3_open_v2(path, &db, SQLITE_OPEN_READWRITE|SQLITE_OPEN_CREATE, NULL);
zumero_register(db);
```

Alternatively, the Zumero client library may have been provided as a dynamically loadable extension:

```
SELECT load_extension('zumero.dylib');
```

In this case, zumero_register() is automatically called for you when the dynamic extension is loaded.

[2]http://www.sqlite.org/c3ref/intro.html
[3]http://www.sqlite.org/c3ref/sqlite3.html

Chapter 2. Getting Started

If you already know how to develop for SQLite, you can start using Zumero by doing just three things:

1. To create a Zumero table instead of a regular table, do this:

```
CREATE VIRTUAL TABLE foo USING zumero (...);
```

2. To create indexes on a Zumero table, prefix the table name with 'z$':

```
CREATE INDEX bar ON z$foo (...);
```

3. To synchronize your Zumero tables with the server, call the zumero_sync() function:

```
SELECT zumero_sync(
    'main',
    'https://my_zumero_server/',
    'my_dbfile_name',
    ...
    );
```

Essentially, that's it.

In most ways, a Zumero table works just like a regular SQLite table. You can SELECT, INSERT, UPDATE and DELETE, exactly like you normally would. (A complete explanation of the differences appears in Chapter 3, *Zumero tables vs. regular SQLite tables*).

The rest of this document is filled with all kinds of other information we hope you find helpful, but if you stopped reading now, you and Zumero could still get a lot of work done .

Chapter 3. Zumero tables vs. regular SQLite tables

Zumero tables are implemented as SQLite virtual tables[1]. This is the same mechanism which is used to implement SQLite's full-text search module[2].

In most ways, a Zumero table works just like a regular SQLite table. You can SELECT, INSERT, UPDATE and DELETE, exactly like you normally would.

However, there are other differences, so here is the complete list, roughly sorted from "essential" to "arcane":

3.1. Creating tables

If you create a regular SQLite table like this:

```
CREATE TABLE t (column definitions);
```

You can create the equivalent Zumero table like this:

```
CREATE VIRTUAL TABLE t USING zumero (column definitions);
```

3.2. Identifiers

A Zumero table cannot have any name which contains a dollar sign ($).

A Zumero table cannot be named "*" (a single asterisk).

The table name "z_acl" is reserved for use as an Access Control List.

The table name "z_audit" is reserved for use as the conflict resolution audit trail.

In a Zumero table, the column names "z_rv", "z_recid", and "z_txid" are reserved for internal use.

3.3. Conflict clauses

Zumero has partial support for SQLite conflict clauses[3] (which, by the way, are unrelated to the Zumero's notion of conflict resolution during sync, as explained in Chapter 5, *Conflict Resolution*): All five SQL conflict policies (ABORT, FAIL, ROLLBACK, IGNORE, REPLACE) *are* supported as clauses for the INSERT and UPDATE statements, but they are *not* supported within column definitions for CREATE TABLE statements.

3.4. Foreign Keys

Foreign keys are supported with the following restrictions:

[1]http://www.sqlite.org/vtab.html
[2]http://www.sqlite.org/fts3.html
[3]http://www.sqlite.org/lang_conflict.html

- Foreign key constraints may only appear using the REFERENCES keyword as column constraint, not as a table constraint.
- The foreign table must also be a Zumero table.
- The ON DELETE and ON UPDATE clauses are not supported.
- The constraint is always DEFERRABLE INITIALLY DEFERRED.

3.5. Tables are permanent

Zumero tables cannot be renamed or dropped.

3.6. Optional feature: Strict Type Checking

SQLite doesn't do type checking. Regardless of the type used to declare a column, you can insert any type of value into it.

In cases where a SQLite db file is being synchronized against a "big SQL" (like MSSQL or PostgreSQL) on the server, Zumero wants SQLite to be stricter. To facilitate this, Zumero tables can optionally do strict type checking.

To activate this feature, we add a little bit of extra syntax to the CREATE TABLE statement. Here's an example of a regular Zumero table:

```
CREATE VIRTUAL TABLE foo USING zumero
(
  a INT,
  b TEXT,
  UNIQUE (a,b)
);
```

Here is the same table with strict type checking turned on:

```
CREATE VIRTUAL TABLE bar USING zumero
(
  * WITH_STRICT_TYPES,
  a INT,
  b TEXT,
  UNIQUE (a,b)
);
```

The following statement will fail with a constraint violation, because 'hello' is not an integer:

```
INSERT INTO bar (a) VALUES ('hello');
```

3.7. Limitations of SQLite virtual tables

There are a few features which SQLite does not support for virtual tables. Each of these is explained below, with a provided workaround for use with Zumero.

3.7.1. Indexes

SQLite does not support the CREATE/DROP INDEX syntax for virtual tables. But you can achieve the same thing by managing indexes on a underlying table, which has "z$" prefixed to the Zumero table name.

Under the hood, each Zumero table stores its rows in a regular SQLite table of the same name, prefixed with z$. For example, if the name of the Zumero table is FOO, the underlying table is z$FOO.

SELECT queries on the Zumero virtual table are simply passed through to the z$ table. Therefore, adding indexes to the z$ table will do what is intended.

For example, since SQLite will not allow this:

```
CREATE INDEX bar ON foo (col);
```

Do this instead:

```
CREATE INDEX bar ON z$foo (col);
```

Note that creation and dropping of indexes is "local" to the current SQLite db file. Zumero does not keep track of indexes. Changes to the indexes will not be propagated to the Zumero server during sync.

3.7.2. Adding columns to an existing table

SQLite does not support the ALTER TABLE ADD COLUMN syntax for virtual tables, but Zumero provides a function specifically for this purpose. Instead of:

```
ALTER TABLE main.foo ADD COLUMN whatever;
```

do this:

```
SELECT zumero_alter_table_add_column('main', 'foo', 'whatever');
```

'whatever' must be a column definition which is compatible with SQLite's restrictions for ALTER TABLE ADD COLUMN[4].

After adding a column to a Zumero table, it is necessary to close the SQLite db handle and reopen it.

3.7.3. Incremental blob I/O

SQLite does not support the incremental blob I/O routines[5] on virtual tables. For Zumero, the workaround is the same as it is for CREATE INDEX. You can work with the blob in the "z$" table instead.

3.7.4. Triggers

SQLite does not support triggers on virtual tables. Zumero does not support them either. However, it is possible to create a view on a zumero table and then create a trigger on that. Both the view and the trigger will be "local" to that SQLite db file. Neither will be propagated during Zumero sync operations.

[4]http://www.sqlite.org/lang_altertable.html
[5]http://www.sqlite.org/c3ref/blob_open.html

Chapter 4. The Zumero Server

The Zumero server provides:

- Synchronization services (with conflict resolution) for mobile devices.
- A centralized, authoritative copy of your data.
- Security features such as authentication and access control lists.

When you perform a zumero_sync() operation, you are managing two copies of a SQLite database file, one on the client, and the other on the server. We can refer to these two copies as two *instances* of the same SQLite database.

On the client, all of your interaction with the SQLite database file takes place using the SQLite API (or some wrapper around it). You can place the file wherever you want, and the path of that file is the identifier you use to access it. When you open the file (using, say, sqlite3_open_v2() from the SQLite C API), you pass it the path.

The Zumero server can manage a [possible large] collection of SQLite database instances. For the purpose of identifying which one is which, each one is given a unique name. Unlike the filesystem abstraction presented on a client, the server's abstraction is a flat list, not a hierarchy.

The Zumero server is very particular about how each SQLite database instance is named. The name may contain only lower-case letters, digits, or underscores. The first character must be a lower-case letter. Any dbfile name that begins with "zumero_" is reserved for internal use.

When we speak of a SQLite database file, we often use the term "dbfile", especially when the context is the Zumero server. Essentially, "dbfile" is a synonym for "SQLite database", but with the additional connotations arising from Zumero's involvement.

It is worth noting here that a dbfile on the server might not actually be represented as a SQLite file. The Zumero server can support synchronization with other SQL implementations such as PostgreSQL.

Chapter 5. Conflict Resolution

When a client invokes zumero_sync(), it sends a package to the server. This package contains all changes to the client's copy of the database which have been made since the last time that client was synchronized.

If the server's copy of the database has not been changed in the meantime by other clients, the changes contained in the incoming package will simply be added to the database.

However, if some other client has already sent a package of changes, it is possible that there will be conflicts. The Zumero server is responsible for automatically resolving these conflicts using a set of rules. These rules can be customized.

There are two basic kinds of conflicts that can happen:

- Row conflicts. The incoming package is trying to modify a row which has already been modified by another client.
- Constraint violations. The incoming package causes the violation of a SQL constraint because of a change that has already been received from another client.

5.1. Row conflicts

Example: Consider a scenario involving two clients, John and Paul, who are sharing a database on the server.

- We start the example with John, Paul and the server all at version 1 of the database, which contains a row named "foo" that has a value of 42.
- John modifies "foo", changing its value from 42 to 13.
- Paul deletes "foo".
- Both John and Paul perform a zumero_sync().

What should happen to "foo"? The server cannot accept both John's change and Paul's change. The two changes are in conflict.

To answer this question, we first observe that the two zumero_sync() operations will not actually happen simultaneously. One of them will happen before the other. The first one will occur without incident. The second one will be identified as a conflict.

Let's suppose that John's sync was completed first. The change from his incoming package is based on version 1 of the database. Since the server's copy of the database is also still at version 1, John's change is incorporated without difficulty, resulting in version 2.

Now the server considers Paul's request, which was also based on version 1. However, things have changed since Paul's last sync. The server's copy of the database is now at version 2. So the server examines each change in Paul's request to see if there are any conflicts. And of course, it discovers that the current version of "foo" is not the same as the one Paul is asking to delete.

5.1.1. Rules: Situations and Actions

This example is just one of three conflict situations that can happen to a row:

- situation_del_after_mod

  The incoming package is trying to delete a row which has been modified.
- situation_mod_after_del

The incoming package is trying to modify a row which has been deleted.

- situation_mod_after_mod

  The incoming package is trying to modify a row which has been modified.

For the sake of completeness, note that "del after del" is not included here because it is not considered a conflict. If everbody wants the row to go away, it does.

A row-level conflict resolution rule is a pairing of a situation and an action. There are three basic actions:

- action_accept

  The incoming package wins. The current state of the row is overwritten in favor of the change requested by the incoming package.

  In the Paul and John example, in version 3 of the database, "foo" would not be present.

- action_ignore

  The incoming package loses. The current state of the row is unchanged, and the change requested by the incoming package is lost.

  In the Paul and John example, in version 3 of the database, "foo" would have the value 13.

- action_reject

  The entire incoming package is rejected, causing the sync operation to fail.

  In the Paul and John example, the database would stay at version 2, and Paul would be notified of an error.

Use of action_reject will mean that the mobile device wasted precious network bandwidth to upload a package which was discarded. There are no conflict situations for which action_reject is the default.

5.1.2. Column Merge

In 'situation_mod_after_mod', one other action is available:

- action_column_merge

  Instead of resolving this conflict for the row as a whole, examine each column individually, and try to merge the two versions of the row on a column-by-column basis.

Let's illustrate this with another example, involving two clients named Ringo and George.

- The Zumero table contains three columns, defined as:

```
CREATE VIRTUAL TABLE foo USING zumero (a text PRIMARY KEY, b integer, c integer);
```

- We start the example with Ringo, George and the server all at version 1 of the database, which contains one row:

```
INSERT INTO foo (a,b,c) VALUES ('bar', 17, 13);
```

- Ringo does:

```
UPDATE foo SET b=289 WHERE a='bar';
```

- George does:

```
UPDATE foo SET c=169 WHERE a='bar';
```

- Ringo does:

```
SELECT zumero_sync(...)
```

  The server's database goes to version 2, and the row is ('bar', 289, 13).

- George does:

```
SELECT zumero_sync(...)
```

When George tries to sync, a 'situation_mod_after_mod' conflict will occur. Assuming the rules do not not specify 'action_reject' as the action, version 3 of the database will be constructed in one of the three ways, depending on which action is used to resolve the conflict:

- action_accept

  George's version of the row overwrites Ringo's.

  Version 3 of the database: ('bar', 17, 169)

- action_ignore

  George's version of the row is ignored.

  Version 3 of the database: ('bar', 289, 13)

- action_column_merge

  Since Ringo and George changed different columns, both of their changes can be accepted.

  Version 3 of the database: ('bar', 289, 169)

In this example, there was a conflict at the row level but not at the column level. If Ringo and George had both modified the same column, then a column-level conflict would have occurred, which would once again require looking at the conflict resolution rules to decide how it should be resolved.

Just as with row-level conflicts, a rule for resolving a column-level conflict can specify 'action_accept', 'action_ignore' or 'action_reject' as the resulting action.

5.1.3. Large text fields

If the type of the column is text, one additional action is available for resolving a column-level conflict:

- action_attempt_text_merge

  Try to automatically merge the two changes using a line-oriented 3-way merge. The technique is identical to what version control tools do when attempting to automerge two changes to a text file.

  This action is used by performing a bitwise OR with another action.

5.1.4. Defaults

The default action for each of these situations is:

- situation_del_after_mod -- action_ignore
- situation_mod_after_del -- action_accept
- situation_mod_after_mod -- action_column_merge

5.1.5. Example Snippets

Add a row-level rule which applies to all tables:

```
SELECT zumero_add_row_rule(
    'main',
    NULL,
    zumero_named_constant('situation_mod_after_mod'),
    zumero_named_constant('action_column_merge'),
    NULL
    );
```

Add a row-level rule which applies only to the table called 'foo':

```
SELECT zumero_add_row_rule(
    'main',
    'foo',
    zumero_named_constant('situation_mod_after_mod'),
    zumero_named_constant('action_column_merge'),
    NULL
    );
```

Add a column-level rule which says that in the table called 'wiki', for the column called 'content', attempt to automatically merge the text, and if that fails, just accept the change from the incoming package.

```
SELECT zumero_add_column_rule(
    'main',
    'wiki',
    'content',
    zumero_named_constant('action_attempt_text_merge')
        | zumero_named_constant('action_accept'),
    NULL
    );
```

5.1.6. Audit Trails

Whenever Zumero uses rules to resolve a conflict, it alters data. Optionally, you can preserve an audit trail of all such changes. This information can be used to verify that your conflict resolution rules are behaving in the manner you expect.

The audit trail is stored as a Zumero table called "z_audit". Its definition looks like this:

```

CREATE VIRTUAL TABLE z_audit USING zumero(
    tbl TEXT NOT NULL,
    ancestor TEXT NOT NULL,
    already TEXT,
    incoming TEXT,
    result TEXT
    );
```

If you want to capture the audit trail, you must create this table. A convenience function exists for this purpose:

```
SELECT zumero_define_audit_table('main');
```

If you do not create the z_audit table, no audit trail will be kept.

During conflict resolution, if the z_audit table exists, the Zumero server will add a row to z_audit for each row conflict resolved.

The audit trail table is designed only for the purpose of having a place for the Zumero server to keep a chronicle of any changes it makes during conflict resolution. It will INSERT rows, but it does not UPDATE or DELETE. You should not INSERT or UPDATE anything in z_audit. You may, however, DELETE rows if you want to save space.

Each row of the z_audit table contains the name of the Zumero table plus four versions of the conflicting row, each one serialized in JSON.

Column	Description
`tbl`	the name of the Zumero table
`ancestor`	the row as it appeared before the conflict happened
`already`	the row modified by the first client
`incoming`	the row as modified by the second client
`result`	the row with its conflict resolved

5.2. Constraint Violations

A constraint violation can happen on the server during sync even when no such problems happened when those changes were originally committed on their respective clients. The following sections show some examples of how this can happen.

5.2.1. Example: CHECK

Consider a scenario involving two clients, Nancy and Ann, who are sharing a database on the server.

- The database contains one Zumero table, defined as:

  ```
  CREATE VIRTUAL TABLE foo USING zumero (a int, b int, c int, CHECK (c > (a + b)));
  ```

  And one record:

  ```
  INSERT INTO items (a,b,c) VALUES (10, 20, 50);
  ```

- We start the example with Nancy, Ann and the server all at version 1 of the database.
- Nancy does:

  ```
  UPDATE items SET a=25;
  ```

  The row is now (25, 20, 50). Since 50 > (20 + 25), the CHECK constraint is satisifed.
- Ann does:

  ```
  UPDATE items SET b=35;
  ```

  The row is now (10, 35, 50). Since 50 > (10 + 35), the CHECK constraint is satisifed.
- Ann does:

  ```
  SELECT zumero_sync(...)
  ```

  The server's database goes to version 2.
- Nancy does:

```
SELECT zumero_sync(...)
```

Nancy's sync will cause a violation of the unique constraint, because column merge will result in a record (25, 35, 50), which now violates the CHECK constraint, because 50 is not > (25 + 35).

5.2.2. Example: Foreign keys

Consider a scenario involving two clients, Harold and Don, who are sharing a database on the server.

- The database contains two Zumero tables, defined as:

```
CREATE VIRTUAL TABLE foo USING zumero (a text PRIMARY KEY);
```

```
CREATE VIRTUAL TABLE bar USING zumero (b text REFERENCES foo (a));
```

- We start the example with Harold, Don and the server all at version 1 of the database, which contains the following row:

```
INSERT INTO foo (a) VALUES ('hello');
```

- Harold does:

```
INSERT INTO bar (b) VALUES ('hello');
```

- Don does:

```
DELETE FROM foo;
```

- Don does:

```
SELECT zumero_sync(...)
```

  The server's database goes to version 2, and the foo table is now empty.

- Harold does:

```
SELECT zumero_sync(...)
```

Harold's sync will cause a violation of the foreign key constraint, because the row he inserted into bar is referencing a row in foo which no longer exists, because Don deleted it.

5.2.3. Example: UNIQUE

Consider a scenario involving two clients, Phil and Lew, who are sharing a database on the server.

- The database contains one Zumero table, defined as:

```
CREATE VIRTUAL TABLE foo USING zumero (
    a text,
    b integer,
    c double,
    UNIQUE (a,b)
    );
```

- We start the example with Phil, Lew and the server all at version 1 of the database, which contains no rows.

- Phil does:

```
INSERT INTO foo (a,b,c) VALUES ('rose', 16, 3.14159);
```

- Lew does:

```
INSERT INTO foo (a,b,c) VALUES ('rose', 16, 1.41421);
```

- Lew does:

```
SELECT zumero_sync(...)
```

  The server's database goes to version 2, with his row inserted into the foo table.

- Phil does:

```
SELECT zumero_sync(...)
```

Phil's sync will cause a violation of the unique constraint.

5.2.4. UNIQUE constraints: An ounce of prevention

Whenever possible, the best way to deal with constraint violations during sync is to carefully design your app to avoid them happening in the first place.

In practice, the most common situation where a SQL constraint is violated on the server (when it succeeded on the client) is a UNIQUE constraint.

With Zumero, any time you INSERT or UPDATE data in a column which has a UNIQUE constraint (including PRIMARY KEY columns), try to make sure that data is likely to be unique.

5.2.5. SQLite "INTEGER PRIMARY KEY" columns

TL;DR -- By default, Zumero handles SQLite's special "INTEGER PRIMARY KEY" columns by automatically adjusting the values during sync to ensure uniqueness. For most applications, this provides the desired behavior. In cases where this approach will not work, the behavior can be disabled, requiring the app to manage things itself.

5.2.5.1. Explaining the problem

An "INTEGER PRIMARY KEY" in SQLite is a special column[1], defined like this:

```
CREATE TABLE foo (x INTEGER PRIMARY KEY, y text);
```

When you declare column x this way, you can omit x from an INSERT statement:

```
INSERT INTO foo (y) VALUES ('hello');
```

and column x will automatically get an integer value which is one greater than the largest value in the table prior to the INSERT[2], or if the table was empty, 1.

But in a decentralized environment like Zumero, this approach is likely to cause unique constraint violations on the server during sync. When SQLite chooses the next available integer for column x, it is only

[1]http://www.sqlite.org/lang_createtable.html#rowid
[2]http://www.sqlite.org/autoinc.html

concerned with uniqueness *within that particular SQLite database file*, which can be achieved by simply adding 1 to the previous maximum value in the column. If this happens, say, on multiple mobile devices, they will all choose the same number, and they will all conflict when attempt to sync.

So, it would be accurate to say that this approach does not merely make constraint violations more likely -- it essentially guarantees they will happen.

For this reason, SQLite's integer primary key columns require special handling for Zumero.

5.2.5.2. Available compromises

We start by observing that an INTEGER PRIMARY KEY has three attributes:

1. It is unique.
2. It is a sequential integer.
3. It doesn't change after it is set.

In a "replicate and sync" architecture, we simply can't have all three of these attributes.

And the first one is not negotiable. If a PRIMARY KEY does not provide an unambiguous way of referencing exactly one row, then it is not a PRIMARY KEY.

So we have to decide which of the other two attributes we are willing to give up. We have two choices:

1. Allow Zumero to change the values during sync to make sure they are unique.

   This is Zumero's default behavior, and is further explained in Section 5.2.5.3, “Fixing INTEGER PRIMARY KEY values during sync”.
2. Stop using sequential integers. This choice can be further broken down into two sub-choices, both of which involve using randomness instead of sequential-ness:
   a. Stop using integers. Use a Universally Unique Identifier (UUID) instead.

      This approach would require you to change the column type from INTEGER to TEXT. See Section 5.2.5.4, “UUIDs as Primary Keys” for more information.
   b. Continue to use integers, but make them random.

      This approach allows you to keep the column type as INTEGER, but requires you to specify the values rather than allowing SQLite to choose them for you. See Section 5.2.5.5, “Maybe 64 bits are enough?” for more information.

5.2.5.3. Fixing INTEGER PRIMARY KEY values during sync

Zumero's default behavior is to adjust the values of all INTEGER PRIMARY KEY columns during sync to ensure that they are unique. This approach is compatible with most applications. Specifically, it should work for you as long as you are complying with the following guidelines for your INTEGER PRIMARY KEY column:

- Don't insert specific values. Allow SQLite to automatically create the value on INSERT.
- Once the value is set on INSERT, don't change it.
- Don't store the value somewhere else unless you declare it as a foreign key.
- Don't assume the value won't change. Until the value is synchronized, it may change. After synchronization, Zumero will not change it again.

If these guidelines are not appropriate for your app, then you should disable this Zumero feature (explained below) and change your app to manage things in some other way.

To instruct Zumero to stop adjusting values of an INTEGER PRIMARY KEY during sync, give the PRIMARY KEY constraint a name of "z_ipk_no_change_on_sync".

```
CREATE VIRTUAL TABLE foo USING zumero (
    x INTEGER
    CONSTRAINT z_ipk_no_change_on_sync PRIMARY KEY,
    y text
    );
```

5.2.5.4. UUIDs as Primary Keys

If you just need each row to have some kind of a unique ID, we can accomplish the goal by using numbers that are random instead of sequential.

If the notion of using random numbers as keys seems odd, rest assured that the practice is quite common.

The typical solution is something called a "universally unique identifier" (UUID)[3], which have become very widely used in database systems.

A UUID is a 128 bit number. In the original version of the UUID specification, those bits were determined by combining the network address with a timestamp. However, this practice has declined in favor of the "version 4" UUID, which is simply a 128 bit random number.

Although it may seem unintuitive, when using a high quality random number generator, and with a 128 bit range, it is astonishingly unlikely to get the same number twice. How unlikely? If you generate a million numbers per second for 82,000 years, it is less than 1% likely that you will have found a duplicate.

Fortunately, SQLite has a built-in random number generator which is of cryptographic quality. We can use it to generate 16 bytes of random data and convert that to hex. The resulting string is essentially the same as a UUID [4].

So, in order to use UUIDs as a primary key instead of "INTEGER PRIMARY KEY", we can define column x like this:

```
CREATE TABLE foo (x TEXT PRIMARY KEY NOT NULL
    DEFAULT ( lower(hex(randomblob(16))) ), y text);
```

With this approach, there is no need to concern ourselves with resolving unique constraint violations during sync, since that will "never" happen.

The main problem with using UUIDs is that they are 128 bits wide. In SQLite, integers are only 64 bits wide, so a UUID will not fit, which means we have to store it in a column of type text instead.

5.2.5.5. Maybe 64 bits are enough?

If we could get by with UUIDs that are 64 bits instead of 128 (which, by the way, means we should not call them UUIDs anymore), that would be really nice. Integer columns take up less space and are more efficient.

Depending on your application, it is actually possible that 64 bits are enough. A 64 bit number can hold 1.8×10^{19} different possible values. That's 1 billion times 1 billion, plus some more. Is it enough?

Ask yourself how many rows your table is going to have.

- A table with 190 million rows would have a 0.1% chance of a collision. Most apps on mobile devices are not likely to have that many rows in a table.

[3] http://en.wikipedia.org/wiki/Universally_unique_identifier
[4] http://www.sqlite.org/lang_corefunc.html

- A table with 6 million rows would have a 0.0001% chance of a collision. For some mobile apps, a table with that many rows would still be considered enormous.

Then ask yourself how bad the consequences would be for your app if a collision did happen. If a unique constraint fails, can you just generate another ID and try again?

If we decide that a 64 bit random number is appropriate, we have two ways to proceed:

- Keep the column defined as "integer primary key".

  But instead of omitting the column on INSERT, specify a random value.

  ```
  INSERT INTO foo (x,y) VALUES (random(), 'hello');
  ```

  The advantage of this approach is that column x remains an alias for SQLite's rowid column.

- Add DEFAULT (random()) to the column definition.

  This approach allows you to continue to just omit column x from INSERT statements.

  Ideally, we could implement this idea by simply adding the DEFAULT clause to the column definition, like this:

  ```
  CREATE TABLE foo (x INTEGER PRIMARY KEY DEFAULT ( random() ), y text);
  ```

  However, this doesn't work. For special "integer primary key" columns, SQLite ignores the DEFAULT clause.

  So we need to do something like this:

  ```
  CREATE TABLE foo (x INT PRIMARY KEY DEFAULT ( random() ), y text);
  ```

  Because we are specifying the type as "int" instead of the full word "integer", this is now a regular column. So the DEFAULT clause works as expected.

Chapter 6. Quarantined Packages

A quarantined package is a collection of changes which have been removed from the database and placed in a waiting area. Typically, the reason something is quarantined is because of conflicts. In most cases, there is no need to quarantine anything. Zumero is designed to manage conflict resolution automatically. However, in some cases, it is appropriate to "undo" some changes to the client instance of the db.

There are two ways that something can get quarantined:

- You quarantined something intentionally by calling zumero_quarantine_since_last_sync().

  The primary use case for this function is to remove changes from the client db because the server refused to accept them during a zumero_sync() operation.

- A zumero_sync() operation had to quarantine something because of a conflict that could not be resolved.

  The only way this can happen is when your app makes changes to the database in another thread while the zumero_sync() operation is happening, AND those changes conflict with new changes received from the server, AND the conflict cannot be resolved by the rules.

  In order to ensure that apps can provide the best experience to users, while zumero_sync() is waiting for the server to respond, the client database file is not locked. Once the server has responded, zumero_sync() checks to see if the db was changed during the network request. If so, it quarantines any such changes. Then the db is updated with any changes from the server. Finally, zumero_sync() tries to restore the quarantined package. If there are no unresolvable conflicts or constraint violations, this will succeed, and the quarantined package will be removed, and you need never know that the quarantine ever happpened. However, if the restore of the quarantined package fails, it stays in the quarantine area.

  For the best user experience, we recommend that you perform zumero_sync() in a background thread. Users of a Zumero app should never have to wait for the network. However, it is also a good practice to avoid performing a zumero_sync() while the user is actively using the app. If you only sync when the user is idle, then zumero_sync() will not need to quarantine anything.

When a package is quarantined, it is stored in a housekeeping table (named t$q). The rowid of that table is called the "quarantine id". This id can be used to reference the quarantined package for various operations. For zumero_quarantine_since_last_sync(), the quarantine id is returned as the result of the function. If something gets quarantined during zumero_sync(), the quarantine id will be included in the return value.

The quarantine feature is an advanced aspect of Zumero. If you are careful to avoid conflicts during sync, you won't need to use it.

Chapter 7. Security

The Zumero server supports security features which can be used to control who has permission to read or write dbfiles.

- Every request from a client can optionally include credentials for authentication.
- Each dbfile can have an Access Control List containing entries which allow or deny access to an item based on the effective identity resulting from authentication.
- The Zumero server can support multiple authentication schemes.
- A built-in authentication scheme called "internal auth" allows a Zumero dbfile to contain user/password pairs which can be used for authentication.

7.1. Authentication

Every request from a client can optionally include credentials for authentication. A set of credentials includes three things:

- scheme
- username
- password

The scheme defines the scope in which the username and password exist. It describes how and where the username/password pair is to be validated.

Under the hood, the scheme is a JSON string. The "scheme_type" key must always be present, and it must contain a recognized name for the type of scheme being described. The JSON object may also contain any additional name/value pairs which are appropriate for that kind of authentication scheme.

For example:

```
{
    "scheme_type" : "telephone",
    "number" : "719-555-1234"
}
```

This scheme string refers to an absurd (but conceptually valid) authentication scheme called "telephone". Authenticating a with this scheme would require the Zumero server to call a phone number (which is provided in the scheme string) and ask if the username/password pair is valid.

Let's say a client wants to present authentication credentials based on this scheme. It would submit three items with its request:

- a scheme string: { "scheme_type" : "telephone", "number" : "719-555-1234" }
- a username: Madonna
- a password: holiday

When the Zumero server receives credentials with a scheme string containing an unrecognized "scheme_type", it returns an "authentication failed" error.

However, if the Zumero server is configured to recognize the authentication scheme type named "telephone"[1], then it will pick up the nearest phone and call the provided number and ask, "Do you recognize user 'Madonna' with password 'holiday'?"

[1]The Zumero server will never support the "telephone" authentication scheme type.

If the person on the other end of the phone call says "Nope!", then the server will return an "authentication failed" error back to the client.

Otherwise, the authentication succeeds, thus establishing the effective identity for this request.

The effective identity for a request is the pairing of the scheme string and the user name. For this example, the effective identity pair would be:

- { "scheme_type" : "telephone", "number" : "719-555-1234" }
- Madonna

By the way, it is worth noting that the effective identity above is completely different from this one (note the different phone number):

- { "scheme_type" : "telephone", "number" : "312-555-1234" }
- Madonna

Somebody named Madonna in Colorado (area code 719) is quite likely to have different permissions than somebody named Madonna in Chicago (area code 312).

Also worth noting: If the client provides no credentials, the server continues to process the request without authentication. The effective identity is null. We use the word "anonymous" in describing this situation.

Anyway, the server will now proceed to figure out if the effective identity actually has permission to do whatever the client is requesting.

7.2. Permissions

Every request from a client will be denied unless the effective identity has been granted the necessary permission(s).

Note that simply being authenticated grants no permission to do anything. Even after successful authentication, the effective identity of "Madonna in Colorado" will not be able to do anything unless that identity has been granted permissions.

7.2.1. Access Control Lists

Every dbfile can optionally have an Access Control List (ACL) which grants or denies permission to do operations on that file. The ACL is a Zumero table with a specific set of columns.

Conceptually, you could add an ACL table by doing this:

```
-- for illustration purposes only
-- don't do this
-- use the zumero_define_acl_table() function instead

CREATE VIRTUAL TABLE IF NOT EXISTS z_acl USING zumero (
    scheme TEXT NOT NULL,
    who TEXT NOT NULL,
    tbl TEXT NOT NULL,
    op TEXT NOT NULL,
    result TEXT NOT NULL
    );
```

It is important that the ACL table have exactly those five columns with exactly those names. For convenience, Zumero provides a function which creates a properly-formed ACL table:

```
SELECT zumero_define_acl_table('main');
```

Under the hood, this function executes a CREATE VIRTUAL TABLE statement which is somewhat more complicated than the one above because it includes extra constraints for error-checking purposes.

7.2.2. ACL Entries

An ACL entry (a row in an ACL table) has 5 columns:

1. scheme
2. who
3. tbl
4. op
5. result

The first four are used to determine whether the entry matches. The result is either 'allow' or 'deny'.

7.2.2.1. who and scheme (effective identity)

The who column can have four possible values:

- zumero_named_constant('acl_who_anyone') -- Match anyone, whether they are authenticated or not. In this case, the scheme column is unused and should be an empty string.
- zumero_named_constant('acl_who_any_authenticated_user') -- Match any authenticated user from the provided scheme. The scheme column is mandatory.
- zumero_named_constant('acl_who_specific_user') -- Match a specific user, the name of which is concatenated after the named constant. The scheme column is mandatory.
- zumero_named_constant('acl_who_specific_group') -- Match any user who is a member of a group, the name of which is concatenated after the named constant. The scheme column is mandatory. The meaning of a group is defined by the authentication scheme.

7.2.2.2. tbl

For acl_op_tbl_add_row, acl_op_tbl_modify_row, and acl_op_tbl_add_column, the tbl column can be either the name of a Zumero table or '*', a wildcard which will match any table.

For all other ops, the tbl column must be an empty string.

7.2.2.3. op

The op column can be either '*', a wildcard which will match any operation, or one of the named constants below. If the op column is '*', then the tbl column must be an empty string.

- acl_op_pull -- Retrieve all or part of a dbfile from the server to the client

  Permission to pull a dbfile is an all-or-nothing proposition. Either a user has permission to see every table and row in the dbfile or they do not.
- acl_op_create_table -- Create a table

  The tbl column should be an empty string.
- acl_op_tbl_add_row -- Add a row to a table

- acl_op_tbl_modify_row -- Update or delete a row in a table
- acl_op_tbl_add_column -- Add a column to an existing table
- acl_op_add_rule -- Add a conflict resolution rule
- acl_op_auth_add_user -- Add a user to an internal auth db
- acl_op_auth_set_password -- Set a user's password in an internal auth db
- acl_op_auth_set_acl_entry -- Set an ACL entry for an internal auth db
- acl_op_create_dbfile -- Create a dbfile

7.2.3. Searching an ACL

When searching an ACL for a matching entry, the inputs are:

- The effective identity
- The operation to be performed
- The table involved in the operation (if any)

ACL entries are searched in order from most-specific to least-specific. As soon as a match is found, the search is done. The result of the ACL check (allow or deny) is determined by the first ACL entry which matches.

When checking permissions for an operation, the Zumero server looks for any Access Control List entries which match the effective identity.

If there are no matching ACL entries, then the following defaults apply:

- If the dbfile involved in this operation was created by an authenticated user, check to see if the effective identity for the current request is the same as that user. If so, allow. Otherwise, deny.
- Alternatively, if the dbfile was created by 'anonymous', allow.

That last rule bears repeating: a dbfile created by anonymous is accessible to anyone unless there is an ACL entry to deny. If you don't want this to happen, then don't allow anonymous to create anything, or make sure that explicit ACL entries are in place..

7.3. Internal Auth

Obviously, the running example using the telephone scheme_type is fake.

A more realistic example might look like this:

```
{
    "scheme_type" : "LDAP",
    "server" : "127.0.0.1",
}
```

(LATER) Note that the Zumero server doesn't actually support LDAP authentication yet.

One authentication scheme that is currently supported is the built-in scheme called "internal auth". This scheme allows a dbfile to contain a list of users with passwords.

The scheme string for an internal auth dbfile would look this:

```
{
    "scheme_type" : "internal",
    "dbfile" : "foo"
}
```

where "foo" is the name of the dbfile which contains the user/password entries.

One installation of the Zumero server can have multiple internal auth dbfiles.

Note that an internal auth db is just a regular Zumero dbfile with a 'users' table which has a specific schema. However, for security reasons, we don't want anyone to synchronize this dbfile down to their client. In this particular case, a "replicate and sync" design would risk exposing information we prefer exist only on the server. Therefore, Zumero provides several functions which allow clients to work with internal auth remotely.

- To create an auth db, use zumero_internal_auth_create().
- To add a user to an auth db, use zumero_internal_auth_add_user().
- To change a password, use zumero_internal_auth_set_password().
- To add an *alias*, use zumero_internal_auth_add_alias(). An alias is a username which, instead of a password, has a "pointer" to another dbfile/username.

  For example, suppose we have two dbfiles named "admins" and "users". The "admin" dbfile contains a user named Chet, but Chet is not just an admin -— he is also a user. If we added a Chet user to both dbfiles, then Chet would have two passwords. Instead, we add an alias to the "users" dbfile. When authenticating users/Chet/password, Zumero will see the alias and verify password against admins/Chet instead.

7.4. Securing a newly installed server

> All Zumero servers provided as part of our cloud hosted service have already been configured as described in this section.
>
> During signup, you were asked to provide a password for the 'admin' user. You should think of your 'admin' user like 'root' on a Unix system, or an Administrator account for Windows. It is mostly used for administration, initial setup, and to create other users that will be endowed with fewer privileges.
>
> The 'admin' user is created in an internal auth dbfile named 'zumero_users_admin'. You should only add new users to this dbfile if those users are intended to have administrator privileges.

Depending on how a Zumero server is installed, it may be initially configured with permissions that allow anyone to create a dbfile.

This permission is configured by creating a dbfile called "zumero_config" containing an ACL with an entry for acl_op_create_dbfile.

If the "zumero_config" dbfile was not created for you when your Zumero server was installed, you will probably want to create it before you do anything else.

Example:

First we need an internal auth db to list people who are allowed to create a dbfile. Then we create zumero_config and add an acl which points to the auth db we just created.

The SQL commands for this operation look something like this:

```
SELECT load_extension('zumero.dll');

SELECT zumero_internal_auth_create(
    'https://server',
    'zumero_users_admin',
    NULL,
    NULL,
    NULL,
    'admin',
    'you shall not pass',
    zumero_internal_auth_scheme('zumero_users_admin'),
    zumero_named_constant('acl_who_any_authenticated_user'),
    zumero_internal_auth_scheme('zumero_users_admin'),
    zumero_named_constant('acl_who_any_authenticated_user')
    );

BEGIN TRANSACTION;

SELECT zumero_define_acl_table('main');

-- Don't let anyone do anything with this dbfile
INSERT INTO z_acl (scheme,who,tbl,op,result) VALUES (
    '',
    zumero_named_constant('acl_who_anyone'),
    '',
    '*',
    zumero_named_constant('acl_result_deny')
    );

-- except admins
INSERT INTO z_acl (scheme,who,tbl,op,result) VALUES (
    zumero_internal_auth_scheme('zumero_users_admin'),
    zumero_named_constant('acl_who_any_authenticated_user'),
    '',
    '*',
    zumero_named_constant('acl_result_allow')
    );

-- explicitly mention create_dbfile, for clarity of this example
INSERT INTO z_acl (scheme,who,tbl,op,result) VALUES (
    zumero_internal_auth_scheme('zumero_users_admin'),
    zumero_named_constant('acl_who_any_authenticated_user'),
    '',
    zumero_named_constant('acl_op_create_dbfile'),
    zumero_named_constant('acl_result_allow'));

COMMIT TRANSACTION;

SELECT zumero_sync('main','https://server', 'zumero_config', NULL, NULL, NULL);
```

If you paste the commands above into a text file called "my_install.sql" (making the appropriate edits for your server URL and username/password), then you can execute them with the SQLite command line shell[2]:

```
sqlite3 :memory: -init my_install.sql
```

7.5. Example: Free Private Wikis

Suppose we want to provide a cloud-hosted wiki service with the following requirements:

[2]http://sqlite.org/sqlite.html

- Anyone can create a wiki.
- The creator of the wiki can control who is allowed access to it.

Implementing this service using Zumero:

- We need an internal auth db to list the wiki owners. Create a temporary SQLite file and do this:

```
SELECT zumero_internal_auth_create(
    'https://server',
    'owners',
    zumero_internal_auth('zumero_users_admin'),
    'admin',
    'my admin password',
    NULL,
    NULL,
    '',
    zumero_named_constant('acl_who_anyone'),
    NULL,
    NULL
    );
```

  We use the admin user to perform this operation. Note that 'owners' is configured to allow anyone to add a user, so the app can support self-registration.

- Now we configure the server so that only people in dbfile "owners" can create a dbfile. In another SQLite file (a temporary or memory db will suffice), do this:

```
SELECT zumero_define_acl_table('main');
INSERT INTO z_acl (scheme,who,tbl,op,result) VALUES (
    zumero_internal_auth_scheme('owners'),
    zumero_named_constant('acl_who_any_authenticated_user'),
    '',
    zumero_named_constant('acl_op_create_dbfile'),
    zumero_named_constant('acl_result_allow'));

SELECT zumero_sync('main' ,'https://server', 'zumero_config', ...);
```

- The app has a "Create Wiki" feature which allows someone to create their own wiki. This feature prompts the user for a password and creates a user in the 'owners' dbfile.

```
SELECT zumero_internal_auth_add_user(
    'https://server',
    'owners',
    NULL,
    NULL,
    NULL,
    'Aretha',
    'respect'
    );
```

  Now we want another internal auth db to store the members of this wiki. We configure it so that only the wiki owner can add users:

```
SELECT zumero_internal_auth_create(
    'https://server',
    'my_wiki_members',
    zumero_internal_auth('owners'),
    'Aretha',
    'respect'
    NULL,
    NULL,
    zumero_internal_auth_scheme('owners'),
    zumero_named_constant('acl_who_specific_user') || 'Aretha',
    zumero_internal_auth_scheme('owners'),
    zumero_named_constant('acl_who_specific_user') || 'Aretha'
    );
```

Now we want to create the wiki itself:

```
BEGIN TRANSACTION;

CREATE VIRTUAL TABLE wiki USING zumero (
    title text NOT NULL UNIQUE,
    content text NOT NULL
    );

-- Use text automerge for wiki pages
SELECT zumero_add_row_rule(
    'main',
    'wiki',
    zumero_named_constant('situation_mod_after_mod'),
    zumero_named_constant('action_column_merge'),
    NULL
    );
SELECT zumero_add_column_rule(
    'main',
    'wiki',
    'content',
    zumero_named_constant('action_attempt_text_merge')
        | zumero_named_constant('action_accept'),
    NULL
    );

-- The wiki is private.  Members only.
SELECT zumero_define_acl_table('main');
INSERT INTO z_acl (scheme,who,tbl,op,result) VALUES (
    zumero_internal_auth_scheme('my_wiki_members'),
    zumero_named_constant('acl_who_any_authenticated_user'),
    '',
    '*',
    zumero_named_constant('acl_result_allow'));
INSERT INTO z_acl (scheme,who,tbl,op,result) VALUES (
    '',
    zumero_named_constant('acl_who_anyone'),
    '',
    '*',
    zumero_named_constant('acl_result_deny'));

COMMIT TRANSACTION;

SELECT zumero_sync(
    'main',
    'https://server',
    'my_wiki_pages',
    zumero_internal_auth_scheme('owners'),
    zumero_named_constant('acl_who_specific_user') || 'Aretha',
    'respect'
    );
```

Chapter 8. History

When Zumero is doing synchronization, it sends packages of incremental changes between client and server. For example, the client pushes a package which contains only the changes which have been made since the last sync. There is no reason to send anything that happened prior to the last sync, since the server already knows about it.

In order to facilitate this, Zumero keeps track of every change you make to the database. With a regular SQLite table, after you INSERT/UPDATE/DELETE something, the state of the database prior to that operation is forgotten. But every time you modify a Zumero table, in addition to doing what you asked, Zumero stores a *delta*, a representation of what operation you performed.

The information contained in a delta is complete. It describes exactly how the database *after* your modification is different from how it was before. In theory, we could undo committed transactions by applying the deltas in reverse.

Here is another way to say this: With a Zumero table, nothing ever *really* gets deleted.

Here is yet another way to say this: With Zumero, all changes to the database are additive. When you delete a row, what you are really doing is adding a state wherein that row does not exist.

For some apps, the retention of this historical information is a good thing. The next section describes how the history is stored and made available for apps to use.

For other apps, all that history is just taking up precious space on the mobile device. The zumero_purge_history() function can be used to reclaim space.

8.1. How History is Stored

When you create a Zumero table called FOO, under the hood, your rows are actually stored in a regular SQLite table called z$FOO (the prefix z$, concatenated with the name of your table).

The z$FOO table contains exactly what FOO would contain if it were a regular table instead of a Zumero table. When you DELETE a row from FOO, it really does get deleted from z$FOO.

Zumero also maintains another underlying table called zoldFOO (the prefix zold, concatenated with the name of your table). This table contains rows (or versions of rows) that are not being used anymore.

When you DELETE a row from FOO, what actually happens is that Zumero removes it from z$FOO and adds it to z$old$FOO.

When you UPDATE a row in FOO, before updating the row in z$FOO, Zumero copies it to zoldFOO. The row ends up existing in two versions. The current version is in z$FOO. The old version of the row is archived in zoldFOO.

Both of these tables contain all of your columns. The main difference between them is simply that one contains all your current stuff, and the other contains all of your old stuff.

If you want to access history, you can SELECT from the zold table.

The zumero_purge_history() function deletes the old rows from all the zold tables in your database.

For the initial sync to retrieve a dbfile from the server to an empty SQLite file on the client, you can use the zumero_pull_without_history() function instead of zumero_sync().

Chapter 9. The Server Log

The Zumero server logs all client requests into a dbfile called "zumero_log". This dbfile can be synchronized down to a client (perhaps a client on a server or workstation rather than a mobile device) for inspection, analytics, or reporting purposes.

9.1. Permission to access zumero_log

By default, the ACL table in zumero_log is configured to disallow all access. To gain permission to pull zumero_log, you must request with the credentials of an authenticated user in the auth db called zumero_users_log[1]. (If you need to add a user to zumero_users_log, you will need to zumero_internal_auth_create() it (if it doesn't exist), and then zumero_internal_auth_add_user().)

9.2. Columns in the log table

Inside zumero_log is a table called "log" which contains the following columns:

Column	Description
`dbfile`	The name of the dbfile involved in the client request
`url`	The path portion of the URL requested by the client. • /pull • /push • /auth_create • /auth_add_user • /auth_add_alias • /auth_set_password • /auth_set_acl_entry
`ip_address`	The IP address of the client making the request
`unix_time`	The time of the request as *unix time*[a]
`scheme`	The scheme string from the credentials provided by the client
`username`	The username from the credentials provided by the client
`request_size`	The compressed size of the package being pushed from the client, in bytes
`response_size`	The compressed size of the response package being sent back down to the client, in bytes
`status`	The HTTP status code of the response • 200 -- successful request, response package sent back to the client • 204 -- successful request, no response package sent back • 304 -- /pull, but nothing new to send back • 401 -- authentication failed

[1]Members of zumero_users_admin are also allowed.

Column	Description
	• 403 -- permission denied • 406 -- database constraint violation
`elapsed`	The amount of time necessary to process the request, in milliseconds
`constraint_details`	Details about any constraint violation that occurred, in JSON
`full_request_size`	Same as request_size, but uncompressed
`full_response_size`	Same as response_size, but uncompressed
`txid_max`	(internal use)

[a]http://en.wikipedia.org/wiki/Unix_time

- Each time the server process is launched, an extra log entry is added with url=SERVER_STARTUP.
- Whenever a new dbfile is created as part of a push request, an extra log entry is added with url=CREATE_DBFILE. The entry for the push operation is included as well.
- For url=/pull, if full_response_size is 0, the response package came from the server cache.

9.3. Sample zumero_log queries

Average elapsed time for each kind of client request:

```
sqlite> .headers ON
sqlite> SELECT url, avg(elapsed) avg_time FROM z$log GROUP BY url ORDER BY avg_time DESC;
url|avg_time
SERVER_STARTUP|46.0
/push|20.0091743119266
/auth_set_password|17.0
/auth_add_user|15.8928571428571
/auth_create|13.125
/auth_set_acl_entry|10.625
/pull|7.26923076923077
/auth_add_alias|6.0
CREATE_DBFILE|0.0
```

Average size of a package pushed from a client:

```
sqlite> SELECT avg(request_size) FROM z$log WHERE url='/push';
avg(request_size)
1664.90825688073
```

Total outgoing bandwidth used:

```
sqlite> SELECT sum(response_size) FROM z$log;
sum(response_size)
228613
```

Total incoming bandwidth used:

```
sqlite> SELECT sum(request_size) FROM z$log;
sum(request_size)
181475
```

List of people who tried to do something that didn't have permission to do:

```
sqlite> SELECT DISTINCT scheme,username FROM z$log WHERE status=403;
scheme|username
|
aragorn|{"scheme_type":"internal","dbfile":"t07c82fec97b103c9c2420be9"}
aragorn|{"scheme_type":"internal","dbfile":"t562430406c732c3fbb9bc2be"}
aragorn|{"scheme_type":"internal","dbfile":"tc3e0861a6584104341739233"}
frodo|{"scheme_type":"internal","dbfile":"t287a4cab223bd0399ef4cb"}
frodo|{"scheme_type":"internal","dbfile":"tbb3cdadf930fee621bdc97"}
gandalf|{"scheme_type":"internal","dbfile":"t07c82fec97b103c9c2420be9"}
saruman|{"scheme_type":"internal","dbfile":"tbb3cdadf930fee621bdc97"}
strider|{"scheme_type":"internal","dbfile":"tbb3cdadf930fee621bdc97"}
```

Average compression percentage for packages pushed by clients:

```
sqlite> SELECT 100 - avg(request_size * 100 / full_request_size)
    FROM z$log
    WHERE full_request_size > 0;
100 - avg(request_size * 100 / full_request_size)
83.6574074074074
```

Chapter 10. Reference: Functions

Most of the Zumero functions should be thought of as procedures. They are designed to be called to achieve side effects, not to return a value for use in a query expression.

Nonetheless, the only way to execute a function in SQLite is during the evaluation of an expression, so calling a Zumero function is accomplished by a SELECT statement:

```
SELECT zumero_do_something(...)
```

Some of SQLite's own built-in functions use this pattern as well. For example, load_extension().

10.1. Network activity

10.1.1. zumero_sync()

Synchronize the SQLite database with a dbfile on the server.

```
zumero_sync(
    attached,
    server_url,
    dbfile,
    credentials_scheme,
    credentials_user,
    credentials_password,
    temp_dir
    )
```

Parameter	Description
`attached`	name of the attached database, usually 'main'
`server_url`	https://wherever
`dbfile`	name of the dbfile on the server
`credentials_scheme`	(optional) credentials for this sync. auth scheme string.
`credentials_user`	(optional) credentials for this sync. user name.
`credentials_password`	(optional) credentials for this sync. password.
`temp_dir`	(optional) path for the directory to be used for temporary files

Return value: a string, containing a semicolon-separated list of integers:

1. partial -- If the local database is now up-to-date with the server dbfile, this value is 0. If the app needs to call zumero_sync() again to retrieve more data, this value is >0.
2. quarantine -- If it was necessary to quarantine something during sync, this value is the id of the quarantined pacakge. Otherwise 0.
3. bytes_up_full -- The size of the uncompressed package which was pushed to the server, in bytes.
4. bytes_down_full -- The size of the uncompressed package which was received from the server, in bytes.
5. bytes_up_compressed -- The size of the compressed package which was pushed to the server, in bytes.

6. bytes_down_compressed -- The size of the compressed package which was received from the server, in bytes.
7. elapsed_ms_net -- The amount of time spent waiting for the server to respond, in milliseconds.
8. elapsed_ms_total -- The amount of time required for the entire sync operation, in milliseconds.

Notes:

- This function may be called with 3, 6, or 7 parameters. The attached db, server_url and dbfile parameters are always required. The authentication credentials are optional, but if you specify any of them, you must specify all three. The temp_dir parameter is optional. If you need to specify parameters you don't actually need in order to specify a later parameter, use NULL.
- This function involves network activity and will block until the sync operation is complete. Best practice is to call this function in a background thread.
- To perform the sync without authenticating, omit or pass NULL for scheme, user, and password.
- It is an error to call zumero_sync() within an explicit transaction.
- If a large amount of information needs to be pulled from the server, this function may need to be called more than once. Check the return value to see if the sync operation was complete or not.
- If the corresponding dbfile on the server does not exist, it will be created (unless the local db file is empty).
- A SQLite file should be synced against only one remote dbfile.
- A dbfile name must begin with a lower-case letter and must contain only lower-case letters, digits, or underscores. Any dbfile name that begins with "zumero_" is reserved for internal use.

10.1.2. zumero_pull_without_history()

NOTE: zumero_pull_without_history() is currently available for testing only, in the client SDK and the Development Server [http://zumero.com/dev-center/]. Support for live applications, synching with zumero.net, will be available in the very near future.

Retrieve a copy of a dbfile from the server with its history already purged.

```
zumero_pull_without_history(
    attached,
    server_url,
    dbfile,
    credentials_scheme,
    credentials_user,
    credentials_password,
    temp_dir
    )
```

Notes:

- This function is nearly identical to zumero_sync(), with one major difference: When zumero_pull_without_history() is used as the first sync to retrieve a dbfile from the server, it will be retrieved with its history already purged.
- This function is handy for situations where you want to keep the history purged from the client-side instance of a dbfile for the purpose of saving space. If this function did not exist, you would need to do a full zumero_sync(), followed by a zumero_purge_history(). By calling this function instead, you avoid retrieving a large amount of information from the server that you plan to immediately purge.
- It is an error to call this function on a client dbfile that is not empty. This function may only be called once for a given client dbfile. All future sync operations on that file must use

zumero_sync(), following by a zumero_purge_history() call if you want to continue purging history as the dbfile grows.

- All parameters to this function are identical to the parameters of zumero_sync().
- The return value of this function is identical to the return value of zumero_sync(). If the 'partial' field is non-zero, you should call zumero_sync() to retrieve more.
- Like zumero_sync(), this function involves network activity and will block until the sync operation is complete. Best practice is to call this function in a background thread.

10.1.3. zumero_internal_auth_create()

Setup an internal auth dbfile on a server.

```
zumero_internal_auth_create(
    server_url,
    dbfile,
    credentials_scheme,
    credentials_username,
    credentials_password,
    first_username,
    first_password,
    allow_add_scheme,
    allow_add_who,
    allow_mod_scheme,
    allow_mod_who
    )
```

Parameter	Description
`server_url`	https://wherever/
`dbfile`	name of the dbfile on the server
`credentials_scheme`	credentials for this operation. auth scheme string.
`credentials_username`	credentials for this operation. user name.
`credentials_password`	credentials for this operation. password.
`first_username`	name of a first user to be created. optional (can be null)
`first_password`	password for the first user to be created. optional (can be null)
`allow_add_scheme`	scheme string for an ACL entry for acl_op_auth_add_user
`allow_add_who`	who for an ACL entry for acl_op_auth_add_user
`allow_mod_scheme`	scheme string for an ACL entry for acl_op_set_acl_entry
`allow_mod_who`	who for an ACL entry for acl_op_set_acl_entry

Return value: NULL

Notes:

- Permission to perform this operation depends on the credentials provided and the 'acl_op_create_dbfile' entry.
- For security reasons, the ACL for this dbfile is configured to disallow pull.
- A dbfile name must begin with a lower-case letter and must contain only lower-case letters, digits, or underscores. Any dbfile name that begins with "zumero_" is reserved for internal use.
- This function makes no changes to the SQLite database attached to the connection in which it is executed.

10.1.4. zumero_internal_auth_add_user()

Add a new user to an internal auth dbfile on a server.

```
zumero_internal_auth_add_user(
    server_url,
    dbfile,
    credentials_scheme,
    credentials_user,
    credentials_password,
    new_user,
    new_password
    )
```

Parameter	Description
`server_url`	https://wherever/
`dbfile`	name of the dbfile on the server
`credentials_scheme`	credentials for this operation. auth scheme string.
`credentials_user`	credentials for this operation. user name.
`credentials_password`	credentials for this operation. password.
`new_user`	name of the user to be created
`new_password`	password for the user to be created

Return value: NULL

Notes:

- This function is provided because an internal auth db is normally not allowed to be pulled down to a client.
- Permission to perform this operation depends on the credentials provided and the 'acl_op_auth_add_user' entry within the ACL table of the dbfile.
- Attempting to add a user name which already exists will result in a 'unique_constraint_violation' error.
- A dbfile name must begin with a lower-case letter and must contain only lower-case letters, digits, or underscores. Any dbfile name that begins with "zumero_" is reserved for internal use.
- This function makes no changes to the SQLite database attached to the connection in which it is executed.

10.1.5. zumero_internal_auth_add_alias()

Add an alias to an internal auth dbfile on a server.

```
zumero_internal_auth_add_alias(
    server_url,
    dbfile,
    credentials_scheme,
    credentials_user,
    credentials_password,
    new_username,
    other_dbfile,
    other_username
    )
```

Parameter	Description
server_url	https://wherever/
dbfile	name of the dbfile on the server
credentials_scheme	credentials for this operation. auth scheme string.
credentials_user	credentials for this operation. user name.
credentials_password	credentials for this operation. password.
new_username	name of the user to be created
other_dbfile	name of the dbfile containing the underlying username. optional (may be null)
other_username	name of the underlying username. optional (may be null)

Return value: NULL

Notes:

- This function is provided because an internal auth db is normally not allowed to be pulled down to a client.
- Permission to perform this operation depends on the credentials provided and the 'acl_op_auth_add_user' entry within the ACL table of the dbfile.
- Attempting to add a user name which already exists will result in a 'unique_constraint_violation' error.
- If other_dbfile is null, simply check for other_username in the same dbfile.
- If other_username is null, simply check for new_username in the other dbfile.
- Either other_dbfile or other_username (or both) must be non-null.
- The permissions on the alias are distinct from the permissions on the underlying user.
- A dbfile name must begin with a lower-case letter and must contain only lower-case letters, digits, or underscores. Any dbfile name that begins with "zumero_" is reserved for internal use.
- This function makes no changes to the SQLite database attached to the connection in which it is executed.

10.1.6. zumero_internal_auth_set_password()

Change the password for a user in an internal auth db on the server.

```
zumero_internal_auth_set_password(
    server_url,
    dbfile,
    credentials_scheme,
    credentials_user,
    credentials_password,
    username,
    password
    )
```

Parameter	Description
server_url	https://wherever/
dbfile	name of the dbfile on the server
credentials_scheme	credentials for this operation. auth scheme string.

Parameter	Description
credentials_user	credentials for this operation. user name.
credentials_password	credentials for this operation. password.
username	name of the user
password	new password for the user

Return value: NULL

Notes:

- This function is provided because an internal auth db is normally not allowed to be pulled down to a client.
- Permission to perform this operation depends on the credentials provided and the 'acl_op_auth_set_password' entry within the ACL table of the dbfile.
- A dbfile name must begin with a lower-case letter and must contain only lower-case letters, digits, or underscores. Any dbfile name that begins with "zumero_" is reserved for internal use.
- This function makes no changes to the SQLite database attached to the connection in which it is executed.

10.1.7. zumero_internal_auth_set_acl_entry()

Add an ACL entry to an internal auth db.

```
zumero_internal_auth_set_acl_entry(
    server_url,
    dbfile,
    credentials_scheme,
    credentials_user,
    credentials_password,
    scheme,
    who,
    tbl,
    op,
    result,
    )
```

Parameter	Description
server_url	https://wherever/
dbfile	name of the dbfile on the server
credentials_scheme	credentials for this operation. auth scheme string.
credentials_user	credentials for this operation. user name.
credentials_password	credentials for this operation. password.
scheme	Section 7.2.2, “ACL Entries”
who	Section 7.2.2, “ACL Entries”
tbl	Section 7.2.2, “ACL Entries”
op	Section 7.2.2, “ACL Entries”
result	Section 7.2.2, “ACL Entries”

Return value: NULL

Notes:

- This function is provided because an internal auth db is normally not allowed to be pulled down to a client.
- This function is intended only for use when an internal db, not a regular dbfile that can be synced to a client.
- Permission to perform this operation depends on the credentials provided and the 'acl_op_set_acl_entry' entry within the ACL table of the dbfile.
- A dbfile name must begin with a lower-case letter and must contain only lower-case letters, digits, or underscores. Any dbfile name that begins with "zumero_" is reserved for internal use.
- This function makes no changes to the SQLite database attached to the connection in which it is executed.

10.1.8. zumero_get_storage_usage_on_server()

Request information from the server regarding the size of dbfiles. Store the results in a temporary table.

```
zumero_get_storage_usage_on_server(
    server_url,
    credentials_scheme,
    credentials_user,
    credentials_password,
    temptable
    )
```

Parameter	Description
server_url	https://wherever/
credentials_scheme	credentials for this operation. auth scheme string.
credentials_user	credentials for this operation. user name.
credentials_password	credentials for this operation. password.
temptable	the name of the temporary table to be created for the results

Return value: NULL

Notes:

- Permission to perform this operation depends on the credentials provided and the 'acl_op_get_storage_usage' entry within the zumero_config dbfile.
- The temporary table must not already exist.
- The temporary table will have two columns: dbfile (the name of the dbfile) and size (in bytes).

10.2. Functions without side effects

10.2.1. zumero_internal_auth_scheme()

Construct an "auth scheme string" for an internal auth db with a given name.

```
zumero_internal_auth_scheme(dbfile)
```

Parameter	Description
dbfile	name of the dbfile on the server

Return value: an authentication scheme string in JSON.

Notes:

- This function constructs a well-formed JSON authentication scheme string for an internal auth db with a given name.

```
INSERT INTO z_acl (scheme,who,tbl,op,result) VALUES (
    zumero_internal_auth_scheme('people_we_dislike'),
    zumero_named_constant('acl_who_any_authenticated_user'),
    '',
    '*',
    zumero_named_constant('acl_result_deny'));
```

10.2.2. zumero_auth_scheme()

General-purpose function to construct an "auth scheme string".

```
zumero_auth_scheme(scheme_type, key,value, key,value, ...)
```

Parameter	Description
scheme_type	required. the name of an authentication scheme type.
key	optional. the name of a key to place in the JSON.
value	optional. sort of. the value for the previous key argument.

Return value: an authentication scheme string in JSON.

Notes:

- This function must have an odd number of arguments. The initial scheme_type argument. After that, the key/value arguments must be in pairs.
- This function constructs a well-formed JSON string for an arbitrary authentication scheme.
- The following function:

```
zumero_internal_auth_scheme('foo'),
```

  will return exactly the same result as this one:

```
zumero_auth_scheme('internal', 'dbfile', 'foo'),
```

- The farcical example:

```
zumero_auth_scheme('telephone', 'number', '719-555-1234'),
```

10.2.3. zumero_named_constant()

Conceptually similar to using C #define in SQL. Included for the purpose of making queries more readable.

This function allows you to write:

```
SELECT zumero_add_row_rule(
    attached,
    NULL,
    zumero_named_constant('situation_mod_after_mod'),
    zumero_named_constant('action_column_merge'),
    NULL
    );
```

instead of:

```
SELECT zumero_add_row_rule(
    attached,
    NULL,
    3,
    8,
    NULL
    );
```

Available named constants:

- Conflict resolution actions:
  - action_default
  - action_accept
  - action_ignore
  - action_reject
  - action_column_merge
  - action_attempt_text_merge
- Conflict resolution situations (row level):
  - situation_del_after_mod
  - situation_mod_after_del
  - situation_mod_after_mod
- ACL who:
  - acl_who_anyone
  - acl_who_any_authenticated_user
  - acl_who_specific_user
  - acl_who_specific_group
- ACL op:
  - acl_op_pull
  - acl_op_auth_add_user
  - acl_op_auth_set_password
  - acl_op_auth_set_acl_entry
  - acl_op_create_table
  - acl_op_tbl_add_row
  - acl_op_tbl_modify_row
  - acl_op_tbl_add_column
  - acl_op_add_rule
  - acl_op_create_dbfile

- ACL result:
    - acl_result_deny
    - acl_result_allow

10.3. Convenience

10.3.1. zumero_define_acl_table()

Convenience function to add an ACL table to the current db.

```
zumero_define_acl_table(attached)
```

Roughly, this is equivalent to:

```
CREATE VIRTUAL TABLE IF NOT EXISTS attached.z_acl USING zumero (
    scheme TEXT,
    who TEXT,
    tbl TEXT,
    op TEXT,
    result TEXT
    );
```

Return value: NULL

Notes:

- The table is defined with strict type checking.
- In addition to creating the table, this convenience function also inserts an ACL entry into it, prohibiting anyone from adding columns to z_acl itself.

```
INSERT INTO z_acl (scheme,who,tbl,op,result) VALUES (
    '',
    zumero_named_constant('acl_who_anyone'),
    'z_acl',
    zumero_named_constant('acl_tbl_op_add_column'),
    zumero_named_constant('acl_result_deny'));
```

10.3.2. zumero_define_audit_table()

Convenience function to add an audit trail table to the current db.

```
zumero_define_audit_table(attached)
```

Roughly, this is equivalent to:

```
CREATE VIRTUAL TABLE attached.z_audit USING zumero(
    tbl TEXT NOT NULL,
    ancestor TEXT NOT NULL,
    already TEXT,
    incoming TEXT,
    result TEXT
    );
```

Return value: NULL

Notes:

- Do not INSERT or UPDATE anything in the z_audit table.

10.4. Adding conflict resolution rules

10.4.1. zumero_add_row_rule()

Add a row-level conflict resolution rule

```
zumero_add_row_rule(
    attached,
    tbl,
    situation,
    action,
    extra
    )
```

Parameter	Description
attached	name of the attached database, usually 'main'
tbl	name of the tbl to which this rule applies. to make this rule apply to any table, pass NULL.
situation	one of zumero_named_constant('situation_*')
action	one of zumero_named_constant('action_*')
extra	for future use. pass NULL.

Return value: NULL

10.4.2. zumero_add_column_rule()

Add a column-level conflict resolution rule. These only get used when the row-level situation was 'situation_mod_after_mod' and the row-level action was 'action_column_merge'.

```
zumero_add_column_rule(
    attached,
    tbl,
    col,
    action,
    extra
    )
```

Parameter	Description
attached	name of the attached database, usually 'main'
tbl	name of the tbl to which this rule applies. to make this rule apply to any table, pass NULL.
col	name of the column to which this rule applies. to make this rule apply to any column, pass NULL
action	one of zumero_named_constant('action_*')
extra	for future use. pass NULL.

Return value: NULL

10.5. Misc

10.5.1. zumero_alter_table_add_column()

```
zumero_alter_table_add_column(
    attached,
    tbl,
    col_def
    )
```

Add a column to a Zumero table. This function exists because SQLite does not pass "ALTER TABLE ADD COLUMN" down to a virtual table implementation.

Parameter	Description
`attached`	name of the attached database, usually 'main'
`tbl`	name of the Zumero table
`col_def`	column definition

Return value: NULL

Notes:

- After adding a column to a Zumero table, it is necessary to close the sqlite db handle and reopen it.
- 'whatever' must be a column definition which is compatible with SQLite's restrictions for ALTER TABLE ADD COLUMN[1].

10.5.2. zumero_adopt_existing_table()

```
zumero_adopt_existing_table(
    attached,
    tbl
    )
```

Convert a regular SQLite table into a Zumero table.

Parameter	Description
`attached`	name of the attached database, usually 'main'
`tbl`	name of the SQLite table

Return value: NULL

Notes:

- It is an error to call this function within an explicit transaction.
- The regular SQLite table is converted in place to become a Zumero table.
- This function uses the same rules as "CREATE VIRTUAL TABLE foo USING zumero", which are somewhat stricter than the rules used by SQLite itself during a regular CREATE TABLE statement. For a Zumero table, if you are going to use a SQLite keyword as a column name, you must make it a quoted identifier by putting double quotes around it. If the regular SQLite table being adopted was originally created by taking advantage of SQLite's relaxed quoting rules, this function will fail.

[1]http://www.sqlite.org/lang_altertable.html

10.6. Quarantine

10.6.1. zumero_quarantine_since_last_sync()

Quarantine and remove all changes since the last sync.

```
zumero_quarantine_since_last_sync(attached);
```

Return value: integer, the id of the quarantined package

This function allows you to non-destructively revert changes that the server has rejected.

If zumero_sync() fails due to permission_denied, then the client's dbfile can never be sync-ed again unless one of two things happens:

- The ACL is altered to allow the sync
- The disallowed changes are reverted

Similarly, if zumero_sync() fails due to package_rejected, then the client's dbfile can never be sync-ed again unless one of two things happens:

- The conflict rule is altered to allow the sync
- The rejected changes are reverted

10.6.2. zumero_restore_quarantine()

Try to restore a quarantine.

```
zumero_restore_quarantine(attached, qid);
```

Parameter	Description
`qid`	quarantine id

Return value: NULL

Notes:

- The quarantine id is returned by zumero_quarantine_since_last_sync() or zumero_sync().
- If the contents of the package cause a constraint violation, this function will cause an error.

10.7. Destroying things

10.7.1. zumero_purge_history()

Purge rows that don't exist anymore.

```
zumero_purge_history(attached);
```

Chapter 11. Reference: Error Messages

When a failure occurs during execution of a SQLite statement, Zumero may store additional information in the error message string. At the C level, this text is obtained by calling sqlite3_errmsg()[1].

When this happens, the error message string will be of the form "zumero:identifier". For example:

```
zumero:permission_denied
```

Identifier	Description
authentication_failed	The Zumero server said the authentication credentials were invalid.
permission_denied	The Zumero server denied a request due to insufficient permissions.
package_rejected	The Zumero server rejected a package because a conflict resolution told it to do so. None of Zumero's default conflict resolution rules specify 'action_reject' as an action.
column_definition_mismatch	During a sync operation, the incoming package tried to define a column which was already defined, and the new column definition did not match the old one.
unique_constraint_violation	An unresolved SQL unique constraint violation occurred on the server.
foreign_key_constraint_violation	A unresolved SQL foreign key constraint violation occurred on the server.
check_constraint_violation	A unresolved SQL check constraint violation occurred on the server.
http_400	The client received HTTP status code 400 from the server, indicating that the client's request was bad in some way.
http_406	The client received HTTP status code 406 from the server, but no further information is available.
http_500	The client received HTTP status code 500 from the server, but no further information is available.
http_other	The client received an unsuccessful HTTP status code from the server, but no further information is available.
network_connection_failed	The client was unable to make a network connection to the server.
invalid_argument	An argument passed to a Zumero function was invalid.
invalid_dbfile_name	The dbfile name provided was invalid. /^[a-z][a-z0-9_]+$/
syntax_error	There was a syntax error found when parsing a column definition.
conflict_clauses_unsupported	Zumero does not support ON CONFLICT clauses.
table_rename_unsupported	Renaming a Zumero table is not allowed.
table_drop_unsupported	In the current version of Zumero, dropping a Zumero table is not allowed.
unrecognized_named_constant	zumero_named_constant() was called with a name that was not recognized.
no_dollar_sign_in_table_name	Zumero table names may not contain a dollar sign ($).
invalid_auth_scheme_string	The auth scheme string passed to a Zumero function was invalid.

[1]http://www.sqlite.org/c3ref/errcode.html

Chapter 12. Frequently Asked Questions

Should I turn on SQLite foreign keys when using Zumero?

Yes. Zumero's housekeeping tables use foreign keys to ensure data integrity. Using SQLite with foreign keys turned ON is highly recommended.

Can I put other stuff in my sqlite db?

Yes, but Zumero will not synchronize it.

Can I sync only some of the Zumero tables in a single file?

No. Sync happens at the granularity of one SQLite dbfile.

My Zumero server has lots of dbfiles. Do I have to sync them all down to every device?

No. You can sync only the ones you want.

Can I modify the z$ table directly?

No. You should not modify the z$ table. All modifications should happen through the Zumero virtual table.

Can I read data from the z$ table directly as long as I don't modify anything?

Yes. You'll get exactly the same results doing SELECT FROM z$foo as you get doing SELECT FROM foo.

Is there any way to use a sqlite file containing Zumero tables without registering the Zumero extension?

As long as you are not modifying data, yes. You can SELECT directly from the z$ table.

How do I create a new dbfile on the server?

When you perform a zumero_sync(), if you provide a dbfile name which does not exist yet on the server, it will be created.

I have a dbfile on the server but it does not exist yet on the client. How do I copy it down?

Create an empty SQLite db on the client and call zumero_sync().

How do I remove a conflict resolution rule?

Add one with 'action_default'. This will override the previous rule.

Why can't I declare a column in a Zumero table which has a foreign key reference to a non-Zumero table?

Because the non-Zumero table is not synchronized to the other instances of the dbfile, so the constraint is guaranteed to be violated.

How does the Zumero server store things?

Normally, each SQLite dbfile is actually stored in SQLite. However...

I want a Zumero server that integrates with my existing system. Does the Zumero server have the ability to sync data into a some sort of a "big SQL" database instead of SQLite?

Yes. Contact us for more information.

Can I retrieve new changes from the server (pull) without sending the unsynced changes on the client (push)?

No. Each call to zumero_sync() will submit any and all pending changes from the client, after which it will download changes from the server.

Does my SQLite file on the client need to have the same name as the dbfile on the server?

No. Zumero doesn't care (or even know) about the name of the SQLite file on the client.

Can I drop a column from a Zumero table?

No. This is a limitation of SQLite itself. SQLite doesn't support ALTER TABLE DROP COLUMN.

Can I change the type of a column in a Zumero table?

No. This is a limitation of SQLite itself. SQLite doesn't support ALTER TABLE ALTER COLUMN.

What is that "t$tx" table I see in my SQLite file?

Zumero creates several housekeeping tables that are used to store the additional information necessary to support synchronization. All of them are prefixed with t$ or z$.

Can I modify the stuff in those t$ tables myself?

Bad idea.

Why is Zumero so fussy about names of dbfiles?

For maximum compatibility with different platforms on which the server may be running.

Can I sync one dbfile on the server with multiple SQLite files on the client?

Yes.

Can I sync one SQLite file on the client with multiple different dbfiles on server(s)?

No. Bad idea.

What permissions are configured on a newly-installed Zumero server?

Until a dbfile named 'zumero_config' exists, anyone can create a dbfile. See Section 7.4, “Securing a newly installed server”.

How do secure my server right after it is first installed?

Create a dbfile named 'zumero_config' containing an ACL table which specifies who is allowed to create a dbfile or not. See Section 7.4, “Securing a newly installed server”.

Can I store my internal auth users table in the same dbfile as other data?

Well, yes, but you probably want to protected the internal auth db from anyone pulling it, which means nobody could pull the other data either.

How do I set permissions on the ACL table?

An ACL table protects itself. It makes little sense to have an ACL table which anyone can modify. It is good practice to add entries specifically designating who is allowed to add rows to the ACL table.

How do I accidentally open the permissions on my ACL table?

One good way is to insert an ACL entry for acl_op_tbl_add_row while specifying '*' as for the tbl. This means anybody matching that entry can modify any table, including the ACL table.

What happens if I push a package that contains 50 transactions and only one of them causes a conflict resulting in action_reject?

The entire package is rejected. The zumero_sync() operation has no effect except to return an error code. Sync and conflict resolution happen atomically for the package being pushed. If you want finer granularity, then sync more often.

Why is my SQLite db so big?

Because when you DELETE a row from a Zumero table, it doesn't completely go away. See Chapter 8, *History* for more information. You can use the zumero_purge_history() function to reclaim space.

I ran zumero_purge_history() but my database file didn't shrink. Why not?

The zumero_purge_history() function merely DELETEs things. This results in free pages within the SQLite file, available for use. To actually shrink the file, you would still need to do a SQLite VACUUM operation.

If I zumero_purge_history(), does the history get deleted in the server's copy of the database as well?

No. On the server, each database is kept with complete history information. There is no way to purge that history on the server.

Can't I just configure Zumero to stop keeping history on the client?

No, but you can call zumero_purge_history() after every sync if you want.

Also, for the initial sync to retrieve a dbfile from the server to an empty SQLite file on the client, you can use the zumero_pull_without_history() function instead of zumero_sync().

What does zumero_register() actually do?

The following things:

- Setup the Zumero virtual table using sqlite3_create_module_v2()
- Setup the Zumero functions using sqlite3_create_function_v2()
- Register internal functions with sqlite3_commit_hook() and sqlite3_rollback_hook()
- PRAGMA recursive_triggers=1;

What does it mean if zumero_register() returns SQLITE_MISUSE?

Zumero requires SQLite version 3.7.11 or higher. If the Zumero library is linked with an older version of SQLite, zumero_register() returns SQLITE_MISUSE.

Can I use SQLite's special "integer primary key" columns with Zumero?

In a word, yes. In far more words, see Section 5.2.5, “SQLite "INTEGER PRIMARY KEY" columns”.

How do I delete a dbfile on the server?

(LATER) Currently, you can't.

Does the Zumero SQLite extension support UTF-16?

(LATER) Currently, no, it only supports UTF-8.

Does Zumero internal auth support the concept of "groups" within a single auth dbfile?

No.

What's this "zumero_config" dbfile I found?

It contains settings for your Zumero server.

Can I give my dbfiles names that start with "zumero_"?

No. Any dbfile name that begins with "zumero_" is reserved for internal use.

Are permissions enforced on the client?

No.

How do I convert a regular SQLite table into a Zumero table?

Use zumero_adopt_existing_table().

Is Zumero open source?

No.

How are passwords stored in an internal auth db?

bcrypt[1]

How can I delete a user from an internal auth db?

For security reasons, you can't. If you delete user Eddie and somebody later creates user Eddie, that new Eddie would inherit the permissions from all the ACL entries referencing the old Eddie.

So can I deactivate a user, or somehow make sure it cannot be used for login anymore?

Yes. Just set its password to something impossible to guess, and then forget the password.

```
SELECT zumero_internal_auth_set_password(
    'https://server',
    'auth_dbfile_name',
    credentials_scheme,
    credentials_username,
    credentials_password,
    'name of user to deactivate',
    lower(hex(randomblob(16)))
    );
```

[1] http://en.wikipedia.org/wiki/Bcrypt

Why the name "internal auth"?

The Zumero server was designed to make it easy to integrate with existing authentication systems like LDAP. In those cases, Zumero is not managing the the directory information -- it is simply contacting the authentication provider to validate credentials. In contrast, the "internal auth" scheme is called "internal" because it is a simple way of keeping user information inside Zumero itself.

Is Zumero compatible with my version of SQLite?

Zumero requires SQLite 3.7.11 or later, which is preinstalled with iOS 6, Android Jelly Bean, and Windows RT. For older releases of these mobile operating systems, you should build a more recent version of SQLite with your app.

At the time of this writing, the front page of the official SQLite website (sqlite.org) says: "Current Status: Version 3.7.15.2 of SQLite is recommended for all new development. Upgrading from all other SQLite versions is recommended." We concur with this recommendation. Whenever possible, use the latest version of SQLite with Zumero.

Why doesn't (CREATE VIRTUAL TABLE USING zumero) accept column names in single quotes like SQLite does?

Even the SQLite developers wish they had never done this. In SQL, single quotes are for string literals and double quotes are for identifiers. SQLite's parser does accept string literals as identifers in certain cases, but this is not standard SQL. Zumero does not duplicate this undesirable behavior.

In the SQLite shell app, why does .dump look so funny when using Zumero tables?

A Zumero table is a virtual table implemented by several regular SQLite tables underneath. The .dump command treats virtual tables specially, providing only the instructions necessary to setup the virtual table. For each of the underlying regular SQLite tables, .dump does what it would normally do. The result is that for a Zumero table named foo, its records show up in .dump output as z$foo.

How are Zumero's conflict resolution features related to SQLite's "ON CONFLICT" clauses?

These two things are completely unrelated. With Zumero, "conflict resolution" refers to the reconciliation of multiple changes during synchronization. The SQLite "ON CONFLICT" clauses refer to the way SQLite performs error handling on certain kinds of constraint violations.

Why do several of the Zumero functions accept a first argument that always seems to be 'main'?

That argument specifies the name of the SQLite attached database where the operation should take place.

Background: SQLite allows multiple database files to be attached to a single database connection. Each one has a name. In addition to any databases you might attach explicitly (using the ATTACH command), there are two databases that are always available: 'main' (the main database) and 'temp' (the database used for temporary tables).

In most SQL statements (like INSERT, UPDATE, etc), you can explicitly reference a table within a specific database by using "database.table" syntax. Or, you can omit the database qualifier and SQLite will search the attached databases, in the order in which they were attached, for a matching table, accepting the first match it finds. In practice, if you omit the database name, and if there is a match in 'main', then that match will always be found, since 'main' was attached to the database connection before any other databases.

The Zumero functions which need to know a database name require you to specify it explicitly.

Can I ROLLBACK a zumero_sync()?

No. In fact, zumero_sync() may not be called from within an open transaction. It will manage its own transaction to ensure the sync process is atomic.

Do I have to hard-code the password for Zumero server into my mobile app?

No. Definitely not.

When your server was installed, you were asked to provide a password for the 'admin' user. You should think of your 'admin' user like 'root' on a Unix system, or an Administrator account for Windows. It is mostly used for administration, initial setup, and to create other users that will be endowed with fewer privileges.

What credentials should users of my mobile app use when contacting the server for sync?

That's your decision, but a typical approach would be to allow users to self-register by creating a new user name within an internal auth dbfile, the members of which are given limited privileges.

- Create an internal auth dbfile. Let's call it "people".
- Configure the ACL on "people" to allow anyone to create a user within it. This is the only operation that can be performed by an unauthenticated client of your server.
- Assign limited privileges to members of "people" as needed.

Why doesn't zumero_sync() work? I know there are changes on my server that zumero_sync() isn't getting.

Sometimes zumero_sync() returns only part of the changes you need, in which case, you need to call it again. The return value from zumero_sync() is a semicolon-separated list of integers. The first integer is the one you need to check. If it's zero, your sync was complete. If it's non-zero, you should call zumero_sync() again to retrieve more.

What is 'zumero_users_admin'?

An internal auth dbfile which is created during installation of a Zumero server.

Should I add all my users to 'zumero_users_admin'?

No. This internal auth dbfile should be used only for users who need administrative privileges. To create other users, create a new dbfile (name it whatever you want) and add users there. Then add privileges for those users by inserting ACL entries wherever is appropriate.

Glossary

accept	A possible action for a data conflict: Accept the change from the incoming package in favor of whatever change was previously there.
action	An instruction which tells the Zumero server how to resolve a conflict.
authentication scheme	Describes a specific provider of authentication. Represented as a JSON object which contains a "scheme_type" key plus any other name/value pairs necessary for that scheme type.
column merge	A possible action for a row-level mod_after_mod conflict: Attempt to resolve the conflict on a column-by-column basis.
dbfile	Essentially a synonym for "SQLite database file".
ignore	A possible action for a data conflict: Ignore the change from the incoming package.
internal auth	A type of authentication scheme where a Zumero dbfile can be used for authentication by the Zumero server.
package	A set of changes being sent between the client and the server during sync.
reject	A possible action for a data conflict: Reject the entire package, causing the sync operation to fail.
situation	One of the three main possible scenarios for a row-level conflict: del_after_mod, mod_after_del, mod_after_mod.