FrontBase® 8.x

Notifications
A way to keep clients in sync

Released 2019-07-21 GBC

How to Contact FrontBase:

USA and International:

FrontBase Inc.
26741 Portola Pkwy. Suite 1E #414
Foothill Ranch, CA 92610
USA

World Wide Web	http://www.frontbase.com
Technical Support	support@frontbase.com
Information	info@frontbase.com
Sales & Marketing	sales@frontbase.com
Licensing	license@frontbase.com
Document Feedback	doc-feedback@frontbase.com

Table of Contents

1. Notifications....................4

1.1 Producing Notifications....................4

1.2 Receiving a Notification....................5

1.3 Notification Output Format....................6

2. Notifications and sql92 11

3. GET NOTIFICATION - Error Situations 12

1. Notifications

A notification is a way for a FrontBase database server (called server below) to tell clients that some client has made one or more changes to one or more tables, all within the context of a transaction. If a given client executes any combination of INSERT, DELETE and UPDATE SQL statements in a transaction, the server will collect the information necessary for generating a notification. Once the client executes a successful COMMIT, a notification is generated and distributed by the server to all other clients waiting for such a notification.

The notification mechanism in FrontBase suggests that a client application have a dedicated connection to the database listening for notifications, i.e. a connection in addition to any normal connection(s). The two connections are below labelled "producer" and "consumer", both typically running in separate threads. The "producer" will tell the server that notifications are to be produced after each successful COMMIT. The "consumer" will wait for notifications and whenever a notification is received, the "producer" part of the client application can be informed appropriately.

A notification can include values for PRIMARY KEY columns, but will otherwise not include column values. The latter point is mainly to avoid having the server perform SELECT privilege checks for each "consumer", but will in general also reduce the overhead the server incurs for managing the notification system.

A "consumer" implies a normal database connection, preferably using the same login details as the "producer" and can as such request the server also to perform any other SQL statements. After a "consumer" receives a notification it can decide to, pending on the nature of the notification, select e.g. the new column values for an UPDATE before passing on the notification to its "producer" companion.

1.1 Producing Notifications

A "producer" needs first to tell the server that notifications are to be produced and distributed after each successful COMMIT:

```
SET NOTIFICATION OUTPUT TRUE
     [WITH PRIMARY KEY] [WITH SCHEMA] [USER '<string>'];
```

After executing the statement above, the server will during a transaction collect the necessary information required to produce a notification. If a subsequent COMMIT is executed successfully, the notification is produced and distributed to all "consumer"s.

The optional WITH PRIMARY KEY is to indicate that primary key information for a given row is to be included in the generated notification output. The default is to not include the primary key information in the notification output.

The optional WITH SCHEMA is to indicate that the name of the schema of a table is to be prefixed to the table name in the generated notification output. The default is to not include the schema name in the notification output.

The optional USER is to allow generated notifications to contain an identification any "consumer" can use and/or display as it see fit.

If a "producer" for whatever reason want to skip generating notifications (could be bulk INSERT, UPDATE or DELETE statements), it needs to tell the server like this:

```
SET NOTIFICATION OUTPUT FALSE;
```

1.2 Receiving a Notification

Any connection to a database can receive notifications. The first step is to tell the server that "this" connection wants to become a "consumer", i.e. to receive notifications:

```
SET NOTIFICATION GET TRUE [EXCEPT OWN];
```

The optional EXCEPT OWN is to indicate that if "this" connection also can produce notifications, such notifications are not to be returned to "this" connection.

Receving notifications can be turned off anytime:

```
SET NOTIFICATION GET FALSE;
```

Receiving a notification:

```
GET NOTIFICATION [TIMEOUT <double value>];
```

The issuing connection will block until a notification is received or if a timeout value has been specified and reached.

The notification output is returned as a property list (see 1.3) and as part of the meta data the server always returns when a statement has been executed. The "stmt" property has the value "NOTIFICATION" and the "msg" property has the property list as value.

1.3 Notification Output Format

The notification output returned to a client application is formatted as an "old style" property list, which means simple dictionaries and arrays with the separators being {}()";,

A quick brush-up on the syntax:

```
Dictionary:    { <key> = <value>; ... }
Array:         ( <value>, ...);
```

<key> and <value> are string values, often wrapped in double-quotes, but not always. This implies that a double-quote in a <key> or <value> is escaped as \". The dictionary and array components can be nested arbitrarely deep.

Example: A property list with a single dictionary mapping KEY1 to an array and KEY2 to a string:

```
     {
          KEY1 = (1001, "1.23"); KEY2 = "Just a string";
     }
```

Example: An array with 3 values:

```
     (
          1001, "1.23", "Just a string",
     )
```

A normal distribution of FrontBase contains a client side library called FBCAccess (libFBCAccess-<os version>.a) that includes a C based property list reader called FBCPlist that can interpret the notification output and provide access to the various dictionary, array and string components.

The notification output generated by the server is a top level dictionary with multiple key/value pairs. The values for the INSERT, DELETE and UPDATE keys will always be dictionaries.

Example:

```
     { INSERT = { ... };
       UPDATE = { ... };
       DELETE = { ... };
       USER = "<string>";
     }
```

The USER key is mapped into the string value specified as part of the "SET NOTIFICATION OUTPUT TRUE ..." statement.

The server will collect relevant information during a transaction and upon a successful COMMIT, the property list is generated and forwarded to all "consumer"s that have requested to receive notifications with the "SET NOTIFICATION GET TRUE ..." statement (see x.y).

The INSERT, DELETE and UPDATE sub-dictionaries are identical in structure:

```
{ "<table name>" = { ... }; ... }
```

Each sub-directory holds one or more <table name> keys each mapped into another sub-directory. If the optional WITH SCHEMA is supplied with the SET NOTIFICATION OUTPUT TRUE statement (see 1.2), a <table name> is actually a string including names for both the schema and the table:

```
"<schema name>.<table name>"
```

The keys for a sub-directory which denotes the value for a <table name> key are:

```
{    ROW_INDEXES = ( ... );
     PK_COLUMN_NAMES = ( ... );
     PK_COLUMN_VALUES = ( ... );
     UPDATE_COLUMN_NAMES = ( ... );
}
```

If the table in question has no PRIMARY KEY, the PK_COLUMN_NAMES and PK_COLUMN_VALUES keys are not included. Applies also if the optional WITH PRIMARY KEY isn't supplied with the SET NOTIFICATION OUTPUT TRUE statement (see 1.1).

The UPDATE_COLUMN_VALUES key is only included for tables in the UPDATE sub-directory.

The value for the ROW_INDEXES key is an array of 64-bit integers (formatted as strings):

```
ROW_INDEXES = (1001, 1017, 123456789);
```

Each row index is a unique value denoting the row in the table that has been INSERTed, DELETEd or UPDATEd. The row index allows for an extremely fast way of SELECTing rows:

```
SELECT ... FROM <table> WHERE INDEX = <row index>;
     or
SELECT ... FROM <table> WHERE INDEX IN (<row index>, ...);
```

The value for the PK_COLUMN_NAMES key is an array of column names:

```
(<column name>, ... );
```

If there is only a single column in the PRIMARY KEY constraint, a string value is returned instead of the sub-array.

The value for the PK_COLUMN_VALUES key is an array with a sub-array for each affected row:

```
( (<pk1>, ....), ...);
```

If there is only a single column in the PRIMARY KEY constraint, a string value is returned for each affected row instead of the sub-array.

The value for the UPDATE_COLUMN_NAMES key is an array with a sub-array for each row affected:

```
( (<col1>, ....), ...);
```

If only a single column is updated, a string value is returned for each affected row instead of the sub-array.

The cardinality of the array values for the ROW_INDEXES, PK_COLUMN_VALUES and UPDATE_COLUMN_NAMES keys are identical.

Example (assuming that auto commit has been disabled):

```
SET NOTIFICATION OUTPUT TRUE;
CREATE TABLE AT0(C0 INT PRIMARY KEY,
                 C2 INT,
                 C3 VARCHAR(32));

INSERT INTO AT0 VALUES (1, 1001, 'string 1'),
                       (10, 1002, 'string 2'),
                       (100, 1101, 'string 3');
DELETE FROM AT0 WHERE C0 = 10;

UPDATE AT0 SET C2 = 1003;
UPDATE AT0 SET C3 = 'string 4' WHERE C0 = 100;
UPDATE AT0 SET C2 = 5, C3 = 'Just a string';
COMMIT;
```

The resulting notification output property list:

```
{
     INSERT = {
          AT0 = {
               "ROW_INDEXES" = (0, 1, 2,);
               "PK_COLUMN_NAMES" = C0;
               "PK_COLUMN_VALUES" = (1, 10, 100,);
          };
     };
     UPDATE = {
          AT0 = {
               "ROW_INDEXES" = (0, 2, 2, 0, 2,);
               "PK_COLUMN_NAMES" = C0;
               "PK_COLUMN_VALUES" = (
                    1,
                    100,
                    100,
                    1,
                    100,
               );
               "UPDATE_COLUMN_NAMES" = (
                    C2,
                    C2,
                    C3,
                    (C2, C3,),
                    (C2, C3,),
               );
          };
     };
     DELETE = {
          AT0 = {
               "ROW_INDEXES" = (1,);
               "PK_COLUMN_NAMES" = C0;
               "PK_COLUMN_VALUES" = (10,);
          };
     };
}
```

2. Notifications and sql92

The command line tool called sql92 has been augmented with a new command:

```
     SHOW NOTIFICATION [FOREVER];
```

It is a cover for sending the SET NOTIFICATION GET TRUE and GET NOTIFICATION statements to the server.

sql92 will receive the notification, interpret the property list (using the FBCPList functions found in the FBCAccess library included with all FrontBase versions) and convert the result back into a string which is output on the console.

Example:

```
demodb@localhost#3> show notification;
2019-06-21 15:13:20.186892
{ "INSERT" = { "t0" = { "ROW_INDEXES" = ("19");
"PK_COLUMN_NAMES" = "c0"; "PK_COLUMN_VALUES" = ("110"); }; };
"USER" = "Myself"; }
```

The optional FOREVER is to indicate that sql92 will enter an infinite loop where it will be repeating the "GET NOTIFICATION + interpret property list + convert property list back to string" sequence forever. This forever loop can, however, be interrupted by a Ctrl-C.

A wait caused by a single GET NOTIFICATION statement can similarly be interrupted by a Ctlr-C. In case a GET NOTIFICATION is interrupted, the server will return an error message (see next section).

3. GET NOTIFICATION - Error Situations

If a timeout value has been specified and reached, the server returns an error message:

```
GET NOTIFICATION wait did timeout
```

If "this" connection in the database is closed by some other connection, the server returns an error message:

```
GET NOTIFICATION wait was stopped, new connection is required
```

If a GET NOTIFICATION wait is interrupted with e.g. the INTERRUPT AGENT statement (offered by sql92, the command line tool), the server returns an error message:

```
GET NOTIFICATION wait was interrupted, connection is OK
```

The server will enqueue notifications albeit with a limited queue length, so if a "consumer" is busy for a short period of time, i.e. not consuming notifications, it can quickly catch up without losing notifications. If a "consumer" stays in a "not consuming" state for a period of time long enough to cause the server to exceed its queue length limit, the server will note that the given "consumer" is behind. When the "consumer" issues a GET NOTIFICATION statement next, the server will return an error message:

```
GET NOTIFICATION wait failed, notification queue length was execeeded
```

The server will also reset the notification queue for the given "consumer". The "consumer" will consequently have to synchronize as necessary with its "producer" companion.