MySQL 5.0 Views

MySQL 5.0 New Features Series – Part 3

A MySQL® Technical White Paper

Trudy Pelzer, Senior Software Architect, MySQL AB

March, 2005

Table of Contents

Introduction

This book is for the long-time MySQL user who wants to know "what's new" in version 5. The short answer is "stored procedures, triggers, views, and information schema". The long answer is the MySQL 5.0 New Features series, and this book is the third in that series.

What I'm hoping to do is make this look like a hands-on session where you, as if you're working it out yourself on your keyboard, can walk through the sample problems. To do this, I'll go through each little item, building up slowly. By the end, I'll be showing larger views that do something useful, as well as some things that you might have thought were tough.

Conventions and Styles

Whenever I want to show actual code, such as something that comes directly from the screen of my mysql client program, I switch to a Courier font, which looks different from the regular text font. For example:

```
mysql> DROP VIEW v CASCADE;
Query OK, 0 rows affected (0.00 sec)
```

When the example is large and I want to draw attention to a particular line or phrase, I highlight it with a double underline and a small arrow on the right of the page. For example:

```
mysql> CREATE VIEW v AS
    -> SELECT column1 AS c /* view col name is c */     <--
    -> FROM table1;
Query OK, 0 rows affected (0.01 sec)
```

Sometimes I will leave out the `mysql>` and `->` prompts, so that you can cut the examples and paste them into your copy of the mysql client program. (If you aren't reading the text of this book in a machine-readable form, try looking for the script on the mysql.com web site.)

All of the examples in this book were tested with the publicly-available alpha version of MySQL 5.0.3 on the SUSE Linux operating system (version 9.1). By the time you read this, the version number will be higher and the available operating systems will include Windows, Sparc, and HP-UX. So I'm confident that you'll be able to run every example on your own computer. But if not, well, as an experienced MySQL user you know that help and support is always available.

A Definition and an Example

A table is any collection of one or more columns and zero or more rows:

```
  +---------------+---------------+-------+
  | column name   | column name   | ...   |
  +---------------+---------------+-------+
  | column value  | column value  | ...   |
  | ...           | ...           | ...   |
  +---------------+---------------+-------+
```

With MySQL and the MyISAM storage engine, a table's contents (the column values) are in a pair of files (the .MYD and .MYI data and index files), stored on disk. This type of table is a base table, called "base" because it is basic, foundational, the basement of the structure. On a level above the base tables we find derived tables, whose column values come from base tables, from literals, or from environmental variables like CURRENT_TIME. You produce a derived table with SQL whenever you issue commands like "FROM table1, table2" or "GROUP BY x" or just "SELECT ..." – all of these are operations that, given a table, produce another table. The table that a SELECT produces is a result set. It has no name. But if you could give a result set a name and store that name (along with other data related to the definition, i.e., the metadata) you would have a viewed table, usually called – for short – a view. Thus:

A view is a named, derived table whose definition is a persistent part of the database.

To make a view, you say CREATE VIEW, plus the view name, plus the SELECT that defines the view. Here's an example:

```
mysql> CREATE VIEW v AS SELECT column1 FROM t;
Query OK, 0 rows affected (0.01 sec)
```

You can always SELECT from a view. Some views are updatable -- that is, you can perform UPDATE and DELETE operations on them. And some updatable views are also "insertable-into" -- that is, you can perform INSERT operations on them. For example:

```
mysql> INSERT INTO v VALUES (1);
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM v;
+---------+
| column1 |
+---------+
|       1 |
+---------+
1 row in set (0.00 sec)
```

Catching Up

Throughout MySQL's history there have been critics pointing out that MySQL lacks support for views, and that such support is compulsory for compliance with standard SQL (Core-level SQL:1999 to be specific), therefore MySQL fails to comply with standard SQL. Since other SQL DBMSs have support for views, there have been difficulties migrating to MySQL. And besides, views are useful.

Oleksandr Byelkin of Lugansk (Ukraine) was first assigned the task of implementing support for views in MySQL on October 9, 2003; his initial code appeared in the source downloads available via BitKeeper on July 15, 2004. Matthias Leich of Berlin (Germany) put the code through quality-assurance tests, including the NIST (National Institute of Software Testing) tests for compliance of a DBMS's support of views with standard SQL requirements. In the period to December 30, 2004 MySQL users downloaded the version 5.0 package approximately 1,194,834 times, and many of them contacted MySQL with comments about view features or bugs. On December 30, 2004, MySQL 5.0 was released as a binary alpha, meaning "all coding is done but the product is not yet robust" and "you can run it on any officially-supported platform without having to compile it yourself". The fact is, though, that at the time of writing, views are more stable than stored procedures, which Peter Gulutzan described in Book 1 of the "New Features in MySQL 5.0" series.

Views are one of the six "flagship features" of MySQL 5.0. The other five are stored procedures, triggers, server-side cursors, precision math, and a standard SQL INFORMATION_SCHEMA. When MySQL calls a feature "flagship" we are saying it's sine qua non -- if it doesn't come, then neither does the fleet. Arguably, views are even more important than the other flagship features because they require no programming skills and because they have always been part of fundamental SQL. Our marketing department sometimes prefers to talk about other things, but views are the big deal of the new version.

What are Views For?

1. Views can be effective copies of base tables

```
CREATE VIEW v AS SELECT * FROM t;
SELECT * FROM v;
```

When you use a view, for example by selecting from it, MySQL will find a way to execute the SELECT statement that you used to define the view. So for this simple example, the effect of "SELECT * FROM v" is precisely the same as the effect of "SELECT * FROM t".

A view that contains nothing but a "SELECT *" from a single table is virtually a synonym of that table.

2. Views can have column names and expressions

```
CREATE VIEW v AS
 SELECT UPPER(`table1`.`column1`), 'The rain in Spain' || 'q'
 FROM `table1`, table2;
```

You don't have to use asterisks in your list of columns to include in the view, though. Within a view's SELECT definition, you may include pretty well anything that you can use in an ordinary SELECT statement's select list -- including column names, functions, and arithmetic or string expressions.

3. You can use any clauses in views

```
CREATE TABLE t (col1 INT, col2 INT);
CREATE VIEW v AS SELECT col1 FROM t WHERE col1 > 4;
SELECT * FROM v WHERE col1 < 6;
```

has same effect as:

```
SELECT * FROM t WHERE col1 > 4 AND col1 < 6;
```

You can use WHERE clauses, ON clauses, GROUP BY clauses, HAVING clauses, or ORDER BY clauses in the SELECT definition of a view. Later, when you're selecting from the view, you can also use any of these clauses to formulate your queries. MySQL will figure out how to merge the clauses together to provide you with the correct result.

4. *Views can be used in INSERT/UPDATE/DELETE*

```
CREATE TABLE t (col1 INT, col2 INT);
CREATE VIEW v AS SELECT * FROM t;
INSERT INTO v VALUES (1,2);
UPDATE v SET col1 = col1 + 10;
DELETE FROM v WHERE col1 < 5;
```

Not every view is updatable, but you can make many views that target of INSERT, UPDATE, and DELETE statements. The effect will be that, by changing the view of the table, you are changing the table itself. Since this can't be done with every kind of view, I'll discuss data changes via views in more detail later.

5. *Views can contain expressions in the select list*

```
CREATE VIEW v AS SELECT AVG(col1) FROM t;
CREATE VIEW v AS SELECT col1 + 17 AS `Arn?s` FROM t;
```

You can make views that contain expressions in the select list. You can also use AS clauses in the select list to specify a name for the columns that result from these expressions.

6. *Views can be views of views*

```
CREATE TABLE t (s1 INT);
CREATE VIEW v1 AS SELECT * FROM t;
CREATE VIEW v2 AS SELECT * FROM v1;
CREATE VIEW v3 AS SELECT * FROM v2;
```

It's legal to have a view within a view. In this example, I started by making a view of a base table. Then I make a view of that view. Then I make a view of that view, in turn.

Views Need MySQL 5.0

To get views to work, you'll need to upgrade to MySQL version 5.0 (or higher).

The mysql_fix_privilege_tables script does the following:

```
UPDATE user SET
  Create_view_priv = Create_priv,
  Show_view_priv = Create_priv
WHERE user <> '' AND "Already Had Create_view_priv" = FALSE;
```

If you are upgrading from MySQL version 4.x, you should run mysql_fix_privilege_tables.
This script tries to set appropriate new privileges for views.
It works by setting the new CREATE VIEW privilege to have the same value as the existing CREATE privilege.

Or, if you are root, you can just edit the mysql user table directly. That's what I had to do, prior to version 5.0.2.

CREATE VIEW Syntax

Here's the full syntax supported by MySQL for the CREATE VIEW statement:

```
CREATE [OR REPLACE]
[ALGORITHM = {MERGE | TEMPTABLE | UNDEFINED}]
VIEW view-name
[(column-list)]
AS select-statement
[WITH [CASCADED | LOCAL] CHECK OPTION]
```

The keywords OR REPLACE are non-standard; we threw them in because Oracle allows them.

The ALGORITHM clause is also non-standard; I'll discuss it later.

To create a view, you'll most often start with the keywords CREATE VIEW, followed by a name for the view. The view name must be a valid identifier -- thus, it may be up to 64 characters long, must begin with a letter, and may be delimited with back ticks (``) or with double quotes ("") if your sql_mode settings include "ansi". The view name may also be qualified with the database name, that is, [database-name.]view-name is legal but is unnecessary if database-name is the default database.

The column-list is standard; to be discussed later.

The WITH CHECK OPTION clause is also standard; to be discussed later.

The main point of the CREATE VIEW statement is the SELECT statement that defined the view.

Oracle calls the SELECT statement within a CREATE VIEW a "subquery". The SQL Standard document calls it a "query expression" (in standard SQL, there are other statements that produce result sets, but MySQL supports only one kind of query expression, SELECT). I will use neither the Oracle nor the standard SQL term. Instead, I will call the SELECT statement within a CREATE VIEW the view's SELECT definition.

Terminology: "Underlying Table"

```
CREATE VIEW v1 AS SELECT * FROM t1;
/* t1 is the "underlying table" for view v1 */

CREATE VIEW v2 AS SELECT 'a';
/* there is no underlying table for view v2 */

CREATE VIEW v3 AS SELECT * FROM v1, v2;
/* there are three underlying tables for view v3 */
```

In terms of a view definition, an underlying table refers to any table upon which a view depends. So for the first example shown above, we can say that view v1 is based on table t1, and that table t1 underlies view v1.

Sometimes there is no underlying table. For example, a MySQL extension to standard SQL makes it legal to create a view just by saying CREATE VIEW v AS SELECT, followed by a literal value, as shown in the second example above. Since MySQL allows one to formulate a SELECT statement without a FROM clause, it follows that defining a view with CREATE VIEW v2 AS SELECT 'a' should also work. Since such a view contains no column references, it needs no special attention.

In the third example, there are three underlying tables for view v3. The view's directly underlying tables are v1 and v2. The view's indirectly underlying table is table t1, because view v1 is based on t1.

Illegal CREATE VIEW Statements

1. No temporary views

```
mysql> CREATE TEMPORARY TABLE tt (col1 INT);
Query OK, 0 rows affected (0.27 sec)

mysql> CREATE VIEW v_tt AS SELECT * FROM tt;
ERROR 1352 (HY000): View's SELECT contains a temporary table 'tt'

mysql> CREATE TEMPORARY VIEW vvv AS SELECT 'a';
ERROR 1064 (42000): You have an error in your SQL syntax; ...
```

You can't create a view for a temporary table. This is a standard restriction.

While I'm on the subject of views of temporaries, I'll mention that you can't create a temporary view either. Microsoft allows CREATE TEMPORARY VIEW statements. MySQL does not.

2. No duplicate names

```
mysql> CREATE TABLE t (s1 INT);
Query OK, 0 rows affected (0.30 sec)

mysql> CREATE VIEW t AS SELECT 'a';
ERROR 1050 (42S01): Table 't' already exists
```

You can't create a view that has the same name as a base table or any other view, or vice versa. That's probably as you'd expect. The view names and the base table names are in the same namespace. Just as with other object identifiers, MySQL checks for duplication by performing a case-insensitive search for the identifier, within the same database.

3. Missing table

```
mysql> CREATE VIEW v AS SELECT * FROM no_such_table;
ERROR 1146 (42S02): Table 'db5.no_such_table' doesn't exist
```

You can't create a view unless the tables to that you use in the view's SELECT definition actually exist in the specified database. In other words, you must make the underlying tables first. This is a reasonable and standard restriction. Oracle has an option called FORCE that lets you evade the restriction, but MySQL stays with the standard here.

4. No privilege

```
mysql> CREATE VIEW v AS SELECT 'a' FROM t;
ERROR 1142 (42000): create view command denied to user
'bob'@'localhost' for table 'v'
```

You can't create a view unless you have both the CREATE VIEW as well as privileges on the underlying tables. The whole subject of privilege checking is complex. We'll come back to it.

5. No variables or parameters

```
mysql> CREATE VIEW v AS SELECT * FROM t
->     WHERE s1 = @variable AND s2 = ?;
ERROR 1351 (HY000): View's SELECT contains a variable or
parameter

mysql> CREATE PROCEDURE p (param INT)
->     CREATE VIEW v AS SELECT param FROM t//
Query OK, 0 rows affected (0.01 sec)

mysql> CALL p(5)//
ERROR 1351 (HY000): View's SELECT contains a variable or
parameter
```

You can't create a view if there are any references to variables or parameters. For the first example of that, I've used a SELECT that refers to both a MySQL variable and a placeholder. For the second example, I've used a parameter in a stored procedure.

Either way, the CREATE VIEW is illegal, and that's standard SQL.

6. No subquery in the FROM clause

```
mysql> CREATE VIEW v3 AS SELECT * FROM
->     (SELECT s1 FROM t1) AS x;
ERROR 1349 (HY000): View's SELECT contains a subquery in the FROM
clause
```

You can't create a view if the view's SELECT definition has a FROM clause that contains a subquery. This restriction is a flaw.

7. No duplicate column names

```
mysql> CREATE TABLE t1 (col1 INT);
Query OK, 0 rows affected (0.29 sec)

mysql> SELECT col1, col1 FROM t1;
Empty set (0.00 sec)

mysql> CREATE VIEW vt1 AS SELECT col1, col1 FROM t1;
ERROR 1060 (42S21): Duplicate column name 'col1'
```

You can't create a view if the definition would result in duplicate column names. In this example, there are two columns named col1. That's legal in an ordinary SELECT. But it's not legal in a CREATE VIEW because the view must have unique column names, just as we must have unique column names in base tables.

8. *Wrong column count in column list*

```
mysql> CREATE VIEW v (s1) AS SELECT s1, s2 FROM t;
ERROR 1353 (HY000): View's SELECT and view's field list have different
column counts
```

You can't create a view using a column list unless the number of columns in the column list is the same as the number of columns in the view's SELECT definition select list.

Legal CREATE VIEW Statements

By this point, you may be wondering what sorts of useful constructs you *can* use to create a view. Here are some examples.

1. *Niladic function*

```
mysql> CREATE VIEW v AS SELECT CURRENT_TIME FROM t;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM v;
+--------------+
| CURRENT_TIME |
+--------------+
| 10:04:38     |
+--------------+
1 row in set (0.00 sec)
```

You can use any niladic function -- e.g. CURRENT_USER, CURRENT_TIME, NOW -- to create a view. MySQL stores the function name as part of the view definition; it does not store the result of the function. So the view will always return the "current" value from the function.

That's good. It means that if you create the view at nine o'clock, and then you select from the view at twelve o'clock, the result will be twelve o'clock, the value at the time you select from the view -- not the value at the time you created the view.

Most syntax and privilege checks (but unfortunately not all) happen when you create the view. Most evaluations of function expressions happen at the time you use the view.

2. *Underlying tables in other databases*

```
CREATE VIEW view1 AS
SELECT Create_view_priv
FROM mysql.user;
```

It is legal to refer to a table in a different database. I think this can be a bad idea, because whenever we tie two databases together, it's difficult to take one of the databases offline without affecting the other. My belief is that more isolation means easier maintenance. However, I'm not the DBA. MySQL says, let the DBA decide.

CREATE VIEW: [(column list)]

```
CREATE [OR REPLACE]
[ALGORITHM = {MERGE | TEMPTABLE | UNDEFINED}]
VIEW view-name
[(column-list)]                                          <--
AS select-statement
[WITH [CASCADED | LOCAL] CHECK OPTION]
```

Now let's get back to the "discuss later" parts of the CREATE VIEW syntax. I'll discuss the optional column list clause first.

The optional column list provides a list of names for the view's columns and comes right after the view name. If the list is omitted, then the view's columns take their names from the columns of the underlying tables. If the column list is present in CREATE VIEW, then the names of the view columns are the names that are in the column list. There must be at least one name, and there may be no duplicate names.

```
mysql> CREATE TABLE t1 (col1 INT);

...

mysql> CREATE VIEW v1 AS
->     SELECT col1 FROM t1;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM v1;
+------+
| col1 |
+------+
|    1 |
+------+
1 row in set (0.01 sec)
```

Unless you specify otherwise, the columns of a view get the same names as the columns upon which they're based, as shown in this example. That is, ordinarily MySQL uses the column name you provide in the view's SELECT definition select list, enclosing it in back ticks. (Enclosing the name in back ticks is a MySQL extension.)

If you'd prefer to give your view columns a different name, you can do so by adding an [AS name] clause in the select list, as shown in the following example:

```
mysql> CREATE VIEW v2 AS
->     SELECT col1 AS select_list_name FROM t1;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM v2;
+------------------+
| select_list_name |
+------------------+
|                1 |
+------------------+
1 row in set (0.00 sec)
```

Or, you can use the [(column list)] option in CREATE VIEW to specify names for the view's columns. The column list name takes precedence, as you can see from this example:

```
mysql> CREATE VIEW v3 (column_list_name) AS
->     SELECT col1 AS select_list_name FROM t1;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM v3;
+------------------+
| column_list_name |
+------------------+
|                1 |
+------------------+
1 row in set (0.00 sec)
```

View Columns Are Set at CREATE VIEW Time

```
mysql> CREATE TABLE t (s1 INT); ...

mysql> CREATE VIEW v AS SELECT * FROM t;

mysql> ALTER TABLE t ADD s2 INT; ...

mysql> SELECT * FROM v;
+------+
| s1   |
+------+
| 1    |
+------+
1 row in set (0.00 sec)
```

Here's an example that shows that "SELECT *" is not a macro. I create a table with one column. I make a view of that table, using "SELECT *". Then I add a new column to the table. When I select from the view, we see that the view still has only one column -- the only column that existed in table t at the time the view was created. Any alterations, to the definition of an underlying table, that occur after a view is created, do not affect the view's definition.

```
mysql> USE database1 ... CREATE TABLE t (s1 INT);

mysql> INSERT INTO t VALUES (1);

mysql> CREATE VIEW v AS SELECT * FROM t;

mysql> USE database2 ... CREATE TABLE t (sx INT);

mysql> SELECT * FROM database1.v;
+------+
| s1   |
+------+
| 1    |
+------+
1 row in set (0.00 sec)
```

Here's an example that shows the same principle. In this case, I have made a view of a table in database1. Then I switch databases before selecting from the view. The point is that MySQL searches for the view's underlying table in database1, not the current database. So the fact that there is a table with the same name in database2 doesn't matter -- the result is still, correctly, the view of table database1.t.

Environment Changes

```
mysql> CREATE TABLE t (c CHAR(5));

mysql> INSERT INTO t VALUES ('tom'),('dick'),('harry');

mysql> CREATE VIEW v AS SELECT group_concat(c) FROM t;

mysql> SELECT * FROM v;
+-----------------+
| group_concat(c) |
+-----------------+
| tom,dick,harry  |
+-----------------+

mysql> SET group_concat_max_len = 4;

mysql> SELECT * FROM v;
+----------------+
| group_concat() |
+----------------+
| tom,           |
+----------------+
```

Although MySQL tries to keep consistency with canonical form, it does not store all the environmental settings that are in effect, say, with the system variables that can be changed online. Here is an example. If I change the setting for the group_concat_max_len variable, then the value in a view can change. This might seem like a minor point, since surely I changed group_concat_max_len for a reason, and therefore it's desirable behaviour. But I have to note it, because it means that a change in the environment can change what's in a view. If you think about it, you'll realize that you can't change the data in a base table by changing a MySQL variable setting. Therefore, this example shows that a view doesn't always behave exactly like a base table.

Privileges

In order to allow you to specify how users may create and operate on views, MySQL has added some new view-specific privileges for use with the GRANT and REVOKE statements.

```
mysql> GRANT SHOW VIEW ON *.* TO peter;

mysql> GRANT CREATE VIEW ON *.* TO peter;

mysql> REVOKE CREATE VIEW ON *.* FROM peter;

mysql> REVOKE SHOW VIEW ON *.* FROM peter;
```

There are two new privileges that can be granted or revoked. One is CREATE VIEW; the other is SHOW VIEW.

The clear intent of the CREATE VIEW privilege is that you can't create a view unless you have the privilege.

The intent behind the SHOW VIEW privilege is that you can't SHOW anything about a view, or execute an EXPLAIN statement for a query on a view, when the view has underlying tables on which you have no privileges -- unless you have the SHOW VIEW privilege on that view.

Privileges, mysql.user, mysql.table_priv

```
Create_view_priv ... for CREATE VIEW
Show_view_priv ... for SHOW CREATE VIEW, EXPLAIN, and so on
```

MySQL stores privilege information in tables in the mysql database. Since there are new privileges associated with views, there are also new columns for these privileges in the mysql tables. The new columns in the mysql.user table are:

- Create_view_priv, which you will need if you want to create a view, and
- Show_view_priv, which you will need if you want to see any information about the view.

As with any other privilege, the possible values are either 'Y' or 'N'. Initially the values are set to 'N'. As mentioned earlier, to update the relevant privileges so that views can be created in your database, you can either run the mysql_fix_privilege_tables script or you can update the mysql tables directly.

CREATE VIEW Privilege

```
GRANT CREATE VIEW ON database1.table1 TO joe;
GRANT CREATE VIEW ON database1.* TO joe;
GRANT CREATE VIEW ON *.* TO joe;
```

In order to create a view, you need a CREATE VIEW privilege. This privilege is new effective with MySQL version 5.0. It is a non-standard privilege, but Oracle also has a privilege named CREATE VIEW.

In the first example shown here, the CREATE VIEW privilege is being granted for a specific view. Granting the privilege in this manner means that the user (joe, in this case) may create a view called table1 in database1 -- provided he also has privileges on the underlying tables. It does not mean that joe can create any other view.

The second example shows how to grant the CREATE VIEW privilege for any view in the database. Granting the privilege in this manner means that joe may create any view in the database, again, provided he has privileges on the underlying tables.

In the third example, the GRANT statement grants joe the right to create any view based on tables for which he has privileges, in any database.

You'll have noticed the common thread by now: The CREATE VIEW privilege is not sufficient, in itself, if you want to create a view. If you want to create a view on table x, you also need privileges on x.

Other Privileges Needed to CREATE VIEW

What other privileges are needed to create a view? The standard SQL requirement is that the user must have the SELECT privilege on every column that is referred to in the CREATE VIEW statement. MySQL's requirement is a bit looser: as long as the user has any privilege on every column that is referred to the view's SELECT definition (in addition to the CREATE VIEW privilege), the CREATE VIEW statement will succeed. Thus, for the following example, the user must have SELECT, UPDATE, or any other column-related privilege on columns c1, c2, and c3 of table1 and on column c3 of table3, as well as the CREATE VIEW privilege:

```
SELECT c1, c2 FROM table1
WHERE c3 IN (SELECT c3 FROM table3);
```

Of course, the fact that you can create a view doesn't automatically mean that you can do anything with the view; at the time of writing, there are some problems in the area of privileges and views.

Privileges Needed to Use a View

As for what privileges you need to use a view, they're not complex, because views are tables. Therefore, if you need a privilege to, for example, update or insert or delete from a base table, you'd need the same privilege to perform the same operation on a view. There are some rules to take note of:

- General rule: View privileges are like base table privileges.
- Special rule: If you have a privilege to perform an operation on a view, you don't need the same privilege on the underlying tables. That is, if you have the UPDATE privilege on view v, which is based on table t, you can update v -- assuming that v is updatable. There is no requirement that you have the UPDATE privilege on t as well.

CREATE VIEW: [ALGORITHM = ...]

```
CREATE [OR REPLACE]
[ALGORITHM = {MERGE | TEMPTABLE | UNDEFINED}]           <--
VIEW view-name
[(column-list)]
AS select-statement
[WITH [CASCADED | LOCAL] CHECK OPTION]
```

Now we're going to look at the non-standard CREATE VIEW algorithm clause.

MySQL uses one of two possible algorithms when processing a view: a merge (MERGE) algorithm or a temporary table (TEMPTABLE) algorithm. Setting the ALGORITHM to UNDEFINED just means "let MySQL decide which algorithm to use"; this is the default option for this clause.

Using the merge algorithm means that this query on a view:

```
SELECT * FROM v;
```

becomes this equivalent query on the underlying table:

```
SELECT * FROM t;
```

Using the temporary table algorithm means that this query on a view:

```
SELECT * FROM v;
```

becomes this equivalent temporary table definition and query on the temporary table:

```
CREATE TEMPORARY TABLE temporary_table AS SELECT * FROM t;
SELECT * FROM temporary_table;
```

As shown, there are two ways to process a view. The first method is rather like a macro, and we call it the merge algorithm because we're merging the expressions of the view into the expressions of the selection from the view. The second method is called the temporary table algorithm because MySQL invisibly makes a temporary table from the view definition and the results that you see when you SELECT from the view are actually the results from the temporary table, rather than from the original, underlying table.

Letting MySQL Choose the Right Algorithm

```
CREATE TABLE t (col1 INT);
CREATE VIEW v1 AS SELECT col1 FROM t;             <-- MERGE
CREATE VIEW v2 AS SELECT AVG(col1) FROM t;    <-- TEMPTABLE
```

Ordinarily, MySQL will choose the algorithm to use when processing a view. Sometimes the server has a choice: it prefers the merge algorithm when this is the case. But as a general observation, it's impossible to use the merge algorithm unless there is a one-to-one relationship between the rows in the underlying table and the rows in the view. So MySQL would choose to use the temporary table algorithm to process a view if the view definition contains an aggregate function, as in the second CREATE VIEW statement shown above. MySQL would also use the temporary table algorithm when the view's SELECT definition contains a DISTINCT, a UNION, a GROUP BY clause, a HAVING clause, or nothing but literal values.

Where possible, MySQL will choose the merge algorithm because it's usually a bit faster, and because views are not updatable with the temporary table algorithm.

Overriding MySQL by Choosing the Algorithm

```
mysql> CREATE ALGORITHM=TEMPTABLE VIEW v2 AS
->     SELECT * FROM t;
Query OK, 0 rows affected (0.00 sec)
```

As noted earlier in this section, you can force MySQL to use the temporary table algorithm even though it would ordinarily choose the merge algorithm, or vice versa. You do this using the ALGORITHM clause, which is a MySQL SQL-extension.

Why would you want to force MySQL to use temporary table, when it's probably faster to use merge? One possibility is that you might be worried about locking. If you use a temporary table, then it's possible that with some storage engine and some isolation level, locking will be reduced or will occur at a different time.

Illegal CREATE VIEW ALGORITHM = MERGE

```
mysql> CREATE ALGORITHM=MERGE VIEW v2 AS
->     SELECT AVG(col1) FROM t;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> SHOW WARNINGS;
+---------+------+-------------------------------+
| Level   | Code | Message                       |
+---------+------+-------------------------------+
| Warning | 1354 | View merge algorithm can't    |
|         |      | be used here for now          |
|         |      | (assumed undefined algorithm) |
+---------+------+-------------------------------+
```

Here is an example of a case where trying to force MySQL to use the merge algorithm when processing a view fails. If MySQL determines that the only algorithm it can use is temporary table, then we can't force the server to use merge.

I think the warning generated is somewhat cryptic. Some suggestions for changing the algorithm clause handling exist, and we'll see what develops.

Updatability

Many views have definitions that allow them to be operated on by data change statements. But not every view can accept such operations. In this section, I will discuss what makes a view updatable.

An updatable view is a view on which you can execute an UPDATE statement or a DELETE statement; the effect will be the same as if you updated or deleted using the underlying table. Not all views are updatable, and not all updatable views can be inserted into, for reasons that we'll see in a moment.

The MySQL Reference Manual used to have a remark that said "many DBMSs don't support updatable views" but actually they all do, except PostgreSQL and SQLite. It is in fact a standard SQL requirement that at least some views must be updatable.

What Makes a View Not Updatable

In the following listing, you see the features that MySQL looks for when deciding whether a view is updatable; we'll check each one in the following pages:

1. UNION ALL
2. UNION
3. DISTINCT or DISTINCTROW
4. Subquery in the select list
5. Non-updatable view in the FROM clause
6. SELECT ... FROM t WHERE x = (SELECT ... FROM t)
7. ALGORITHM = TEMPTABLE

MySQL does not meet Dr. E. F. Codd's requirement, that all theoretically-updatable views are actually updatable. But we come reasonably close.

What's not updatable: 1. UNION ALL

```
CREATE VIEW v AS
SELECT * FROM t1
UNION ALL
SELECT * FROM t2;
```

Our first non-updatable view is one that has UNION ALL in its SELECT definition. If you think about it, you will realize that such views are theoretically updatable. That is, there is no logical reason that updating this view is impossible. But there is an implementation reason: MySQL uses temporary tables to process UNION ALL.

What's not updatable: 2. UNION

```
CREATE VIEW v AS
SELECT * FROM t1
UNION               /* or EXCEPT or INTERSECT */
SELECT * FROM t2;
```

On the other hand, a UNION without an ALL, or a UNION DISTINCT, is not updatable. The same considerations will apply for two set operators that we will implement in the future, INTERSECT and EXCEPT. This is in accordance with standard SQL and Codd's rules; such views are not theoretically updatable.

What's not updatable: 3. DISTINCT or DISTINCTROW

```
CREATE VIEW v AS
SELECT DISTINCT col FROM t1;
```

A view is also not updatable if its SELECT definition contains DISTINCT or DISTINCTROW.

In addition, it's not updatable if there is a set function or a GROUP BY or a HAVING, as we discussed earlier.

What's not updatable: 4. Subquery in the select list

```
CREATE VIEW v AS
SELECT t1.*,(SELECT s1 FROM v2) FROM t1;
```

Also, a view is not updatable if there is a subquery in the select list of the SELECT statement that defines it. Notice I am saying subquery in the select list; I am not saying subquery in the FROM clause or subquery in the WHERE clause.

We are hopeful about removing this restriction in a later version of MySQL.

What's not updatable: 5. Non-updatable view in the FROM clause

```
CREATE VIEW v1 AS SELECT DISTINCT * FROM t1;
CREATE VIEW v2 AS SELECT * FROM v1;
```

A view is not updatable if it's based on a table that is not updatable. In the example here, we see that view v1 is not updatable because its SELECT definition includes the word DISTINCT. It follows that view v2, which is based on v1, is also not updatable, even though it doesn't directly contain DISTINCT.

What's not updatable: 6. SELECT ... FROM t WHERE x = (SELECT ... FROM t)

```
CREATE VIEW v AS
SELECT * FROM t
WHERE col1 IN (SELECT MAX(col1) FROM t);
```

A view is not updatable if its SELECT definition contains a subquery that refers to the same table as the view's underlying table. In this example, table t is both an underlying table of view v and a table that is referenced in the subquery in the WHERE clause -- so view v will not be updatable.

Put simply, the rule is: If the FROM clause and any subquery in the WHERE clause contain the same table name, the view is non-updatable. The real problem with this type of view is that we wouldn't be able to use a WITH CHECK OPTION clause with it, so it shouldn't be updatable.

What's not updatable: 7. ALGORITHM = TEMPTABLE

```
CREATE ALGORITHM=TEMPTABLE VIEW v AS
SELECT * FROM t;
```

Finally, a view is not updatable if you explicitly tell MySQL that it should use the temporary table algorithm.

"Deletable" Views

```
CREATE TABLE t (col1 INT, col2 INT);
CREATE VIEW v AS SELECT col1, col2 + 5 FROM t;
DELETE FROM v;
```

When we say that a view is updatable, we mean that DELETE operations are also supported: MySQL allows deletion rows from updatable views.

A situation arises with deletions that will probably cause a bit of controversy: MySQL allows DELETE from a view, even if that view does not contain all the columns of the underlying base table. That may be a bad thing.

But it's what other DBMSs allow. And you won't automatically have DELETE privilege on the view unless you had DELETE privilege on the underlying base table.

The View Updatability Flag

MySQL sets a flag, called the view updatability flag, at CREATE VIEW time. You don't see this flag, but it means that the server always knows whether a view is updatable. And if it's not, then statements like UPDATE and DELETE are illegal and will be rejected.

The view updatability flag is:
- Set at CREATE VIEW time.
- TRUE if UPDATE, DELETE and similar operations are legal for the view; otherwise FALSE.

MySQL can determine whether a view is updatable by scanning the tokens during parsing. The proposal is: do so once at CREATE VIEW time, and set the view updatability flag to either TRUE or FALSE.

You shouldn't be able to inherit or GRANT any UPDATE or DELETE privileges on the view if it's not updatable. Unfortunately, MySQL does allow

```
GRANT UPDATE ON v TO pierre;
GRANT DELETE ON v TO pierre;
```

even if UPDATE and DELETE operations are impossible on the view.

UPDATABLE_VIEWS_WITH_LIMIT

I recommend that you skip this section; it has been included for completeness, but you will never need to know the subject except in rare circumstances.

```
SET [GLOBAL | SESSION]
   UPDATABLE_VIEWS_WITH_LIMIT = {'YES' | 'NO'}
```

MySQL 5.0 includes a new server variable that relates to views, called updatable_views_with_limit. The variable may be set dynamically, but changing its value does not trigger any action. The variable name is meant to suggest the meaning: "Is a view that is the target of a data change statement that includes a LIMIT clause updatable?" The possible settings are:

- 'YES' -- which means, "Yes, a view that is the target of a data change statement that includes a LIMIT clause is an updatable view, provided that the other conditions for updatability (no GROUP BY, no DISTINCT, and so on) are met and provided that some other conditions are also met (see below)". However, issue a warning.
- 'NO' -- which means, "No, a view that is the target of a data change statement that includes a LIMIT clause is not an updatable view".

The default setting is 'YES'.

UPDATABLE_VIEWS_WITH_LIMIT: the conditions

This is the definition of what updatable-with-limit means. Ultimately, the view must have a single underlying base table, and either of the following must be true:

1. The underlying base table has a PRIMARY KEY or a UNIQUE key comprised only of columns defined as NOT NULL -- and the view's select list directly refers to all columns of that key.
2. The view's select list directly refers to every column of the underlying table.

That is: The key of a table is either the PRIMARY KEY, or the NOT NULL columns that make up any UNIQUE key, or every column in the table -- and the SELECT statement that defines the view has a select list that directly references every column of the key.

For the first possible condition:

- Suppose the underlying base table has a PRIMARY KEY, or has a UNIQUE key in which all the columns are defined as NOT NULL.
- Suppose that the view's select list mentions all the columns of that key directly. When I say "directly" I mean that the column name is specified alone, and not within an expression or function.

In that case, the updatable-with-limit condition is TRUE.

For the second possible condition:

- Suppose that the view's select list mentions every column of the base table directly.

In that case, the updatable-with-limit condition is TRUE.

UPDATABLE_VIEWS_WITH_LIMIT: some examples

Here are some examples of the updatable-with-limit conditions.

```
CREATE TABLE t1 (
 col1 INT, col2 INT, col3 INT,
 PRIMARY KEY (col, col2));

1. CREATE VIEW v1 AS SELECT col1, col2 FROM t1;
2. CREATE VIEW v2 AS SELECT * FROM t1;
3. CREATE VIEW v3 AS SELECT col2, col3 FROM t1;
```

In the first example, an updatable-with-limit condition is TRUE: the view's select list directly mentions all columns of the underlying table's PRIMARY KEY.

In the second example, an updatable-with-limit condition is also TRUE: the view's select list directly mentions every column of the underlying base table.

In the third example, no updatable-with-limit condition is TRUE.

UPDATABLE_VIEWS_WITH_LIMIT: setting it to 'NO'

Here is an example of what happens if you set UPDATABLE_VIEWS_WITH_LIMIT to 'NO'.

```
mysql> SET UPDATABLE_VIEWS_WITH_LIMIT = 'NO';

mysql> CREATE TABLE t1 (
->     col1 INT, col2 INT, col3 INT,
->     PRIMARY KEY (col1, col2));
Query OK, 0 rows affected (0.27 sec)

mysql> CREATE VIEW v3 AS SELECT col2, col3 FROM t1;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE v3 SET col3 = col3 + 1 LIMIT 1;
ERROR 1288 (HY000): The target table v3 of the UPDATE is not updatable
```

UPDATABLE_VIEWS_WITH_LIMIT: leaving it as 'YES'

Now here's what happens if you leave UPDATABLE_VIEWS_WITH_LIMIT set to the default of 'YES'. There's a warning, but no error.

```
mysql> SET UPDATABLE_VIEWS_WITH_LIMIT = 'YES';

mysql> UPDATE v3 SET col3 = col3 + 1 LIMIT 1;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> SHOW WARNINGS;
+-------+------------------------------------------+
| Level | Code | Message                           |
+-------+------+-----------------------------------+
| Note  | 1355 | View being updated does not have       |
|       |      | complete key of underlying table in it |
+-------+------+-----------------------------------+
```

"Insertable Into" Criteria

Even if a view is updatable, it might not be possible to INSERT into it. If you want to be able to INSERT into a view, there are three additional criteria that the view definition must meet:

1. The view must contain every column in the underlying base table which does not have a default value.
2. The view must not contain a derived column, i.e. its SELECT definition's select list must consist entirely of simple column references; it may not contain any expressions or function results.
3. The view must not contain a duplicated column, e.g. the following definition will create a view that won't allow INSERT operations:
   SELECT a AS a1, a AS a2 FROM t;

For each of these cases, if you try to INSERT into the view, you'll get an error message that says the view is not updatable. That's not so -- the view is updatable, so the error message is misleading. But it is true that one cannot insert into such views, and that's what the error message is trying to indicate.

"Insertable Into": 1. Contain all columns ...

Insertable criterion: The view must contain every column in the underlying base table which does not have a default value.

```
CREATE TABLE t (s1 INT, s2 INT NOT NULL);
CREATE VIEW v AS SELECT s1, s2 FROM t;
UPDATE v SET s1 = 5;                  /* succeeds */
INSERT INTO v (s1) VALUES (1);        /* should fail? */
```

The first additional insertable-into criterion doesn't really affect us at MySQL yet. For an ordinary DBMS, it would be improper to depend on a column having a default value, if no default value was defined for that column. But with the default sql_mode setting, MySQL assigns a default value of zero (0) to a NOT NULL numeric column, so the result of this INSERT is a row with the values {1, 0}. But if you have sql_mode set to 'traditional', MySQL should return an error for the INSERT.

"Insertable Into": 2. No derived column ...

Insertable criterion: The view must not contain a derived column, i.e. its SELECT definition's select list must consist entirely of simple column references; it may not contain any expressions or function results.

```
CREATE TABLE t (s1 INT, s2 CHAR);
CREATE VIEW v AS SELECT s1 + 1, UPPER(s2), s2 FROM t;
UPDATE v SET s2 = 5;                    /* succeeds */
INSERT INTO t (s2) VALUES (5);          /* fails */
```

The second additional insertable-into criterion is that a view can't contain any derived columns in its SELECT definition's select list, which we have interpreted as meaning: no expressions, please.

The view in this example is updatable provided that you don't try to update the derived columns themselves. If you do, you get something like this:

```
ERROR 1348 (HY000): Column 's1 + 1' is not updatable
```

"Insertable Into": 3. No duplicated column ...

Insertable criterion: The view must not contain a duplicated column, e.g. the following definition will create a view that won't allow INSERT operations:
SELECT a AS a1, a AS a2 FROM t;

```
CREATE TABLE t (a int);
CREATE VIEW v AS SELECT a AS a1, a AS a2 FROM t;
UPDATE v SET a1 = 1;                /* succeeds */
INSERT INTO v (a1) VALUES (1);      /* fails */
```

The final additional insertable-into criterion is that a view can't contain two columns which, ultimately, refer to the same base table column. This makes sense when you think about the conflict that would be encountered by an INSERT like:

```
INSERT INTO v (a1, a2) VALUES (10, 20);
```

So that's all the insertable-into examples. The important point to recall here is that inserting and updating/deleting rules are different for views. Usually, though, it's enough to know that a view is updatable.

Updatable Joined Views

A joined view -- that is, a view with a SELECT definition that has multiple tables in the FROM clause -- may be updatable, effective with MySQL 5.0.3.

```
mysql> CREATE VIEW v AS SELECT * FROM t1, t2;
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO v (t1_s1) VALUES (1);
Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO v (t2_s1) VALUES (1);
Query OK, 1 row affected (0.00 sec)

mysql> UPDATE v SET t1_s1 = 2;
Query OK, 2 rows affected (0.00 sec)
Rows matched: 2  Changed: 2  Warnings: 0

mysql> DELETE FROM v;
ERROR 1395 (HY000): Can not delete from join view 'db6.v'
```

To make updates on a joined view possible, the UPDATE statement (UPDATE v ...) must be translatable to the multi-table MySQL-extension UPDATE statement (UPDATE t1, t2 ...). This is not an easy feature to implement, and as you'd probably expect, there are limitations.

When you try to use WITH CHECK OPTION on a left-join view, you'll find it fails:

```
mysql> CREATE VIEW v2 AS
->     SELECT t1_s1,t2_s1
->     FROM t1 LEFT JOIN t2 on t1_s1 = t2_s1
->     WITH CHECK OPTION;
ERROR 1368 (HY000): CHECK OPTION on non-updatable view 'db6.v2'
```

When you INSERT into a joined view, the INSERT statement must contain a column list, and the column names in the list must be for only one (1) of the joined tables. If you skip the column list, or if the column list has column names from multiple tables, you'll get an error. For example:

```
mysql> INSERT INTO v VALUES (1,1);
ERROR 1394 (HY000): Can not insert into join view 'db6.v' without
fields list

mysql> INSERT INTO v (t1_s1,t2_s1) VALUES (1,1);
ERROR 1393 (HY000): Can not modify more than one base table through
a join view 'db6.v'
```

Comparison with other DBMSs

	MySQL	IBM DB2	Oracle	SQL Server
Basic	YES	YES	YES	YES
UNION ALL	NO	YES	YES	YES
Joins	YES	YES	YES	YES
INSTEAD OF	NO	YES	YES	YES
UPDATEABLE_KEY	YES	NO	NO	NO

To finish off the discussion of updatability, I'll quickly note how MySQL compares to other DBMSs. The fact that MySQL can do updatable views at all is a huge step, so it's almost even. But the Big Three DBMSs (DB2, Oracle, and SQL Server) can all handle updatable UNION ALL views with restrictions, and perhaps there are a few more joined-view situations that they can handle. And, of course, none of the other DBMSs have the UPDATABLE_VIEWS_WITH_LIMIT flag.

And finally: DB2, Oracle, and SQL Server support INSTEAD OF triggers (also known as rules), so a user can make a view updatable by specifying what the server should do during a data-change operation. The bottom line is, MySQL will not be able to immediately support as many updatable views as DB2, Oracle, and SQL Server.

WITH CHECK OPTION

```
CREATE [OR REPLACE]
[ALGORITHM = {MERGE | TEMPTABLE | UNDEFINED}]
VIEW view-name
[(column-list)]
AS select-statement
[WITH [CASCADED | LOCAL] CHECK OPTION]                  <--
```

Now we'll look at the optional with CHECK OPTION clause. Support for this option became available only with MySQL 5.0.3. WITH CHECK OPTION is a new type of constraint that applies only if you are changing the contents of a view.

The error "CHECK OPTION failed" does not depend on the new strict sql_mode setting. So it does not matter whether you say:

```
SET SQL_MODE='traditional'
```

WITH CHECK OPTION has the following effect: If the WHERE condition stated in a view's SELECT definition is that the view should not contain a data row with 'Z', and you try to insert a 'Z' or update to a 'Z', you'll get an error.

```
mysql> CREATE TABLE t (s1 CHAR(5));
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE VIEW v AS SELECT s1 FROM t WHERE s1 <> 'Z'
->     WITH CHECK OPTION;
Query OK, 0 rows affected (0.10 sec)

mysql> INSERT INTO v VALUES ('X'),('Y'),('Z');
ERROR 1369 (HY000): CHECK OPTION failed 'db6.v'
```

Since this example operates on a MyISAM table, only the third value is rejected; the first two inserts got through:

```
mysql> SELECT * FROM t;
+------+
| s1   |
+------+
| X    |
| Y    |
+------+
2 rows in set (0.01 sec)
```

If you're operating on an InnoDB table, though, a statement rollback ensures that all three values are not inserted:

CASCADED and LOCAL

The CASCADED and LOCAL options only affect situations where a view is based on another view. One example should be enough to make the difference clear.

CASCADED is the default, that is, if you don't explicitly say LOCAL, MySQL assumes you mean CASCADED. When a check option is cascaded, the cascade goes upwards (it goes from the "underlying" view up to the view that is based upon it). In other words: If you have a view v2 (with a check option) based on a view v1 (without a check option), the WHERE condition of view v1 must also be checked to determine whether a data-change operation is legal.

Example: WITH CASCADED CHECK OPTION

Here's how WITH CASCADED CHECK OPTION works: Under the following conditions, the WHERE clause of view v1 will be checked to ensure the INSERT does not violate its conditions; the check occurs even if view v2 has no WHERE clause:

- Assume we want to insert into a view v2
- View v2's definition includes WITH CASCADED CHECK OPTION
- View v2 is based on an underlying view v1
- View v1's definition includes WITH LOCAL CHECK OPTION

```
mysql> CREATE TABLE ti (s1 TINYINT);
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> CREATE VIEW vi1 AS
->     SELECT * FROM ti
->     WHERE s1 <> 0
->     WITH LOCAL CHECK OPTION;
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE VIEW vi2 AS
->     SELECT * FROM vi1
->     WITH CASCADED CHECK OPTION;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO vi2 VALUES (0);
ERROR 1369 (HY000): CHECK OPTION failed 'db6.vi2'
```

But a WITH LOCAL CHECK OPTION view based on view v1 will not cause a check on v1's condition (WHERE S1 <> 0):

```
mysql> CREATE VIEW vi3 AS
->     SELECT * FROM vi1
->     WITH LOCAL CHECK OPTION;
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO vi3 VALUES (0);
Query OK, 1 row affected (0.01 sec)
```

CHECK TABLE

According to the SQL Standard, if you create a view based on a specific table and then drop the table, then either (a) the DROP should fail (this is the DROP ... RESTRICT alternative) or (b) the view should be dropped as well (this is the DROP ... CASCADE alternative). This requires a search of all views, to see whether any of them contains a reference to the table being dropped. But MySQL doesn't really support RESTRICT or CASCADE; the server accepts all syntax variants, but does nothing with them. The result is that the table gets dropped, leaving a dangling view, to use a common, but informal, term.

There are other DBMSs which don't drop the view automatically in such cases, such as Oracle8i and SQL Server 2000. (To be specific, SQL Server will not cascade, but it will restrict if you create the view using a non-standard WITH SCHEMABINDING clause.) So we don't worry about dangling views much. But we do provide a way to check for them.

```
CHECK TABLE[S] view_name [,view_name...];
```

The CHECK TABLE or CHECK TABLES statement is useful if you worry that a view might have become invalid due to a change in an underlying table. Such a change can happen with DROP or ALTER of a table or view or database, or with REVOKE. To check the database integrity, you must list all of your views. For example:

```
mysql> CREATE TABLE t (column1 INT);
Query OK, 0 rows affected (0.01 sec)

mysql> CREATE VIEW v AS SELECT * FROM t;
Query OK, 0 rows affected (0.01 sec)

mysql> DROP TABLE t;
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> CHECK TABLE v;
+-------+-------+----------+------------------------+
| Table | Op    | Msg_type | Msg_text               |
+-------+-------+----------+------------------------+
| db6.v | check | error    | View 'db6.v' references |
|       |       |          | invalid table(s) or    |
|       |       |          | column(s) or function(s) |
+-------+-------+----------+------------------------+
```

In this example, I created a view v that depends on a table t. Then I dropped table t, without getting a warning or error. But a CHECK TABLE v statement tells me that v is no longer a useful view. Simply selecting from the view would cause the same error.

ALTER TABLE

```
CREATE TABLE t (s1 INT, s2 INT);
CREATE VIEW v AS SELECT * FROM t;
ALTER TABLE t ADD [COLUMN] ....
ALTER TABLE t DROP [COLUMN] ... [ RESTRICT | CASCADE ]
ALTER TABLE t ... MODIFY [COLUMN] ...
ALTER TABLE t ... CHANGE [COLUMN] ...
```

MySQL has a close relationship with SAP AG, the world's largest provider of total solutions for enterprises. We have to ensure that the MySQL server works well with their basic product, SAP R/3. So we follow advice from dedicated SAP liaison staff. One of the items which we solicited their opinions on is what should be done with a view if an underlying table's definition is altered, that is, if a new column is added or if a column that is part of the view is dropped or modified.

Here's what we came up with:

- For ALTER TABLE ADD: MySQL works in accordance with standard SQL and Oracle -- the view is unchanged. Adding and dropping columns would only affect "SELECT *" anyway.
- For ALTER TABLE DROP: The statement should fail if one uses RESTRICT, or the statement should cause the view to be dropped if one uses CASCADE. Right now, this type of change is caught only via the CHECK TABLE(S) statement, since MySQL doesn't support RESTRICT and CASCADE.
- For ALTER TABLE CHANGE/MODIFY: If the column is used in a view, there should be an error if the change makes the current data invalid for the new column definition. There isn't. But as with dangling views, there is an error message later.

```
mysql> CREATE TABLE ta (`The Column` INTEGER);
Query OK, 0 rows affected (0.01 sec)

mysql> CREATE VIEW va AS SELECT `The Column` FROM ta;
Query OK, 0 rows affected (0.01 sec)

mysql> ALTER TABLE ta CHANGE `The Column` c VARCHAR(1000);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM va;
ERROR 1356 (HY000): View 'db6.va' references invalid table(s) or
column(s) or function(s)
```

Other SQL Statements

To support views properly, it's sufficient if it is legal to use views in SELECT, INSERT, UPDATE, DELETE, SHOW, and DESCRIBE statement. But MySQL goes further.

Some of the statements that I mention here don't work properly with views at the time of writing. But they all eventually will. The idea, in most cases, is that the effect of the statement will happen on the underlying table.

1. HANDLER <View_name> does not work.

2. LOAD DATA ... INTO TABLE <View_name> works.
Example script:

```
CREATE TABLE tq (s1 INT, PRIMARY KEY (s1)) ENGINE=INNODB;
INSERT INTO tq VALUES (7), (3), (15);
CREATE VIEW vq AS SELECT * FROM tq;
SELECT s1 + 100 INTO OUTFILE '/tmp/data.txt' FROM tq;
LOAD DATA INFILE '/tmp/data.txt' INTO TABLE vq;
```

3. LOCK TABLES <View_name> works.
Example script:

```
CREATE TABLE tb1 (tb1_col INT);
CREATE TABLE tb2 (tb2_col INT);
CREATE VIEW vball AS SELECT * FROM tb1, tb2;
LOCK TABLES vball WRITE;
```

4. RENAME TABLE <View_name> does not work.

5. REPLACE ... <View_name> works.
Example script:

```
CREATE TABLE pk (s1 INT, s2 INT, UNIQUE (s1));
INSERT INTO pk VALUES (3, NULL), (5, 0);
CREATE VIEW vk AS SELECT s1, s2 FROM pk WHERE s1 + s2 > 0;
REPLACE vk VALUES (5, 777), (6, 777);
```

6. TRUNCATE TABLE <View_name> does not work.

7. CREATE INDEX ... <View_name> does not work.

For the statements that don't yet work, the most common error message looks like this:

```
ERROR 1346 (HY000): 'db5.v' is not BASE TABLE
```

EXPLAIN

One of the decisions that had to be made for implementing views was how much information on the underlying tables should be shown when an EXPLAIN statement is executed for a view.

There are two problems with providing information on underlying tables:
1. If, as is often the case, the user is not aware that a table is a view, it can be confusing to be presented with an EXPLAIN output that refers to another table.
2. If the user has no privileges on the underlying table, then it's a security flaw to show information on the underlying table.

So we decided that MySQL should return an error if a user without privileges on any underlying table tried to execute an EXPLAIN on a view. But this brought up another problem: a user should sometimes be able to execute EXPLAIN on a view, even without privileges on the underlying tables. That's one of the reasons that we introduced the SHOW VIEW privilege.

Here is an example of an EXPLAIN output on a view called v. As you can see, the EXPLAIN refers to the underlying table t, and to the index on t.

```
mysql> CREATE TABLE t (s1 INT, PRIMARY KEY (s1));

mysql> INSERT INTO t VALUES (1),(2),(3),(4),(5);

mysql> CREATE VIEW v AS SELECT * FROM t;

mysql> EXPLAIN SELECT * FROM v;
+----+------------+-------+-------+ ...
| id | select_type | table | type  | ...
+----+------------+-------+-------+ ...
|1   | PRIMARY     | t     | index | ...
+----+------------+-------+-------+ ...
```

A user with no privileges on table t will not be able to execute the EXPLAIN statement unless he/she has been granted the SHOW VIEW privilege on view v -- the assumption is that the user won't be granted the SHOW VIEW privilege unless it is legitimate that he/she sees information on table t.

Views and ORDER BY

Ordinarily, a DBMS does not allow ORDER BY in a view query; it is not a standard SQL feature, it is not necessary, and it can cause confusion. But we're going to allow it anyway. This is a rare feature, but you do see it in other DBMSs, for example in Oracle.

Example (1)

```
mysql> CREATE TABLE t (s1 CHAR(4);

mysql> INSERT INTO t VALUES ('Uez'),('Uda'),('Ufa');

mysql> CREATE VIEW v AS SELECT s1 FROM t ORDER BY s1;

mysql> SELECT * FROM v;
+------+
| s1   |
+------+
| Uda  |
| Uez  |
| Ufa  |
+------+
```

Here, although we entered the rows into table t in random order, they come out in sorted order because the view has an ORDER BY clause in it.

Example (2)

```
mysql> SELECT * FROM v ORDER BY s1 DESC;
+------+
| s1   |
+------+
| Ufa  |
| Uez  |
| Uda  |
+------+
3 rows in set (0.00 sec)
```

In this example, I'm selecting from the ordered view, but I'm putting my own ORDER BY clause in the SELECT statement. Note that the ORDER BY specified in the SELECT statement takes precedence over the ORDER BY specified in the view definition.

Switches in the View

MySQL supports additional (usually non-standard) SELECT syntax that gives hints to the optimizer, asks for extra information, and so on. I've listed the switches and hints here.

- ALL
- DISTINCT
- DISTINCTROW
- FOR UPDATE
- HIGH_PRIORITY
- INTO clause
- LIMIT
- LOCK IN SHARE MODE
- PROCEDURE
- SQL_BIG_RESULT
- SQL_BUFFER_RESULT
- SQL_CACHE
- SQL_CALC_FOUND_ROWS
- SQL_NO_CACHE
- SQL_SMALL_RESULT
- STRAIGHT_JOIN

The main rule is that switches in the view definition will be added to the switches of a query on the view. If there's a contradiction between two switches, they still get merged into the main query and the results are complex. In the case of SQL_CACHE and SQL_NO_CACHE, for instance, SQL_NO_CACHE takes precedence. But that's hard to remember, so it's best to avoid conflicts yourself. Here's an example:

```
mysql> CREATE TABLE t7 (s1 INT);
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> INSERT INTO t7 VALUES (5), (NULL), (15);
Query OK, 3 rows affected (0.00 sec)

mysql> CREATE VIEW v7 AS SELECT ALL s1 FROM t7 LIMIT 1;
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT DISTINCT s1 FROM v7 LIMIT 3;
+------+
| s1   |
+------+
|    5 |
+------+
2 rows in set (0.00 sec)
```

Switches in the Environment

Now let's look at environmental switches in more detail. Consider this example:

```
mysql> CREATE TABLE t (s1 CHAR(10));

mysql> INSERT INTO t VALUES ('aaaa1'),('aaaa7'),('aaaa4');

mysql> CREATE VIEW v AS SELECT * FROM t ORDER BY 1;

mysql> SET max_sort_length = 4;

mysql> SELECT * FROM v;
+-------+
| s1    |
+-------+
| aaaa1 |
| aaaa7 |
| aaaa4 |
+-------+
```

By "environmental switch" I mean any user-settable variable which affects the way that MySQL understands an SQL statement. For example, by setting max_sort_length, I affect the outcome of a sort. As you can see, the result set here is out of order, even though it would have been in order if MySQL applied the same switches as were in effect at the time of the CREATE VIEW. Frankly, I think it's just too hard to keep track of the tremendous variety of switches in MySQL. So although MySQL can handle changes to the major switches, like the sql_mode setting and the default character set, it's important to remember that changing the environment after CREATE VIEW may affect the data that you'll be able to see through the view. To avoid any problems that this may cause, it's best to decide upon the setting of any environment switches that may affect views before creating any views, and stick to those settings.

Metadata

The SHOW statements that work for base tables will show you information on views too. These statements include SHOW COLUMNS / DESCRIBE, SHOW TABLE STATUS, SHOW TABLES, SHOW GRANTS FOR, and SHOW CREATE TABLE. But I think there is a better way.

SHOW COLUMNS / DESCRIBE

```
mysql> CREATE TABLE t (s1 INT, s2 INT,
PRIMARY KEY (s1)) ENGINE=INNODB;

mysql> CREATE VIEW v AS SELECT * FROM t;

mysql> DESCRIBE v;
+-------+---------+------+-----+---------+-------+
| Field | Type    | Null | Key | Default | Extra |
+-------+---------+------+-----+---------+-------+
| s1    | int(11) |      |     | 0       |       |
| s2    | int(11) | YES  |     | NULL    |       |
+-------+---------+------+-----+---------+-------+
```

SHOW COLUMNS and DESCRIBE work for views, and the syntax is the same as for base tables. The result is the same except that the view description doesn't show index information, e.g., in this case, that column s1 is a PRIMARY KEY.

SHOW TABLE STATUS

```
mysql> CREATE VIEW v AS SELECT 'a';
Query OK, 0 rows affected (0.00 sec)

mysql> SHOW TABLE STATUS;
+------+--------+---------+...+---------+
| Name | Engine | Version |...| Comment |
+------+--------+---------+...+---------+
| v    | NULL   | NULL    |...| view    |
+------+--------+---------+...+---------+
```

SHOW TABLE STATUS shows the status for views as well as base tables. It has no great value, since all columns show NULL except for the name and the comment, which is always the word 'view'. So the only information it provides is the fact that a table is a view, and not a base table or temporary table.

SHOW [FULL] TABLES

```
mysql> CREATE TABLE t (s1 INT);
Query OK, 0 rows affected (0.28 sec)

mysql> CREATE VIEW v AS SELECT * FROM t;
Query OK, 0 rows affected (0.00 sec)

mysql> SHOW FULL TABLES;
+---------------+------------+
| Tables_in_db5 | table_type |
+---------------+------------+
| t             | BASE TABLE |
| v             | VIEW       |
+---------------+------------+
```

SHOW [FULL] TABLES shows views as well as base tables. If, and only if, you add the optional keyword FULL, you will also see a table_type field, which tells you whether the table is a base table or a view.

SHOW CREATE VIEW / SHOW CREATE TABLE

```
mysql> CREATE TABLE t (s1 INT);

mysql> CREATE VIEW v AS SELECT * FROM t;

mysql> SHOW CREATE VIEW v;
+-------+-----------------------------------------------+
| Table | Create Table                                  |
+-------+-----------------------------------------------+
| v     | CREATE VIEW `db5`.`v` AS select `db5`.`t`.`s1`|
|       | AS `s1` from `db5`.`t`                        |
+-------+-----------------------------------------------+
1 row in set (0.00 sec)
```

There is a new statement -- SHOW CREATE VIEW -- which shows the same information as the old SHOW CREATE TABLE statement, for views only. SHOW CREATE TABLE shows the same information; for both base tables and views. The noticeable fact about the display is that the description is in canonical form. Explanation of "canonical form" comes soon.

INFORMATION_SCHEMA

At the beginning of this section, I said that there was a better way to get metadata information than using MySQL's non-standard SHOW statements. I prefer to get information about views by selecting from the INFORMATION_SCHEMA "database"". In fact, although I've tried to give the SHOW statements a fair chance by talking about them first, I frankly recommend that you use INFORMATION_SCHEMA for all your metadata needs: the INFORMATION_SCHEMA views are standard SQL, are reasonably compatible with what other DBMSs do, conform to what Dr Codd recommended that a relational DBMS should do, and offer more options than the SHOW statements.

The most important INFORMATION_SCHEMA view, for our purpose, is -- surprise -- a view called INFORMATION_SCHEMA.VIEWS. Here is how to get its definition:

```
mysql> SELECT COLUMN_NAME, COLUMN_TYPE
->     FROM INFORMATION_SCHEMA.COLUMNS
->     WHERE TABLE_SCHEMA = 'INFORMATION_SCHEMA'
->     AND TABLE_NAME = 'VIEWS'
->     ORDER BY ORDINAL_POSITION;
+-----------------+-------------+
| COLUMN_NAME     | COLUMN_TYPE |
+-----------------+-------------+
| TABLE_CATALOG   | longtext    |
| TABLE_SCHEMA    | varchar(64) |
| TABLE_NAME      | varchar(64) |
| VIEW_DEFINITION | longtext    |
| CHECK_OPTION    | varchar(8)  |
| IS_UPDATABLE    | varchar(3)  |
+-----------------+-------------+
```

And here’s how to interpret the output:

- The TABLE_CATALOG column will always contain NULL, as long as MySQL doesn’t support the concept of a separate database catalog.
- The TABLE_SCHEMA column will contain the database identifier, that is, the name of the database in which a view resides.
- The TABLE_NAME column will contain the name of the view.
- The VIEW_DEFINITION column will contain the view's SELECT definition and is thus a subset of the "Create View" field that you get with SHOW CREATE VIEW.
- The CHECK OPTION column will contain 'NONE' or 'CASCADED' or 'LOCAL'.
- The IS_UPDATABLE column will contain 'YES' or 'NO'.

Here's an example:

```
mysql> CREATE TABLE tv (s1 INT);
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE VIEW vv AS SELECT s1 FROM tv;
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT * FROM INFORMATION_SCHEMA.VIEWS
->     WHERE TABLE_NAME = 'vv';
+---------------+--------------+------------+
| TABLE_CATALOG | TABLE_SCHEMA | TABLE_NAME |
+---------------+--------------+------------+
| NULL          | db6          | vv         |
+---------------+--------------+------------+
...
+-------------------------+--------------+--------------+
| VIEW_DEFINITION         | CHECK_OPTION | IS_UPDATABLE |
+-------------------------+--------------+--------------+
| select `db6`.`tv`.`s1`  | NONE         | YES          |
| AS `s1` from `db6`.`tv` |              |              |
+-------------------------+--------------+--------------+

mysql> SELECT COLUMN_NAME, DATA_TYPE
->     FROM INFORMATION_SCHEMA.COLUMNS
->     WHERE TABLE_NAME = 'vv';
+-------------+-----------+
| COLUMN_NAME | DATA_TYPE |
+-------------+-----------+
| s1          | int       |
+-------------+-----------+
1 row in set (0.03 sec)
```

Other INFORMATION_SCHEMA tables with view-related data include: TABLES, TABLE_CONSTRAINTS and COLUMNS. Eventually MySQL should also support the additional standard tables INFORMATION_SCHEMA.VIEW_TABLE_USAGE and INFORMATION_SCHEMA.VIEW_COLUMN_USAGE.

Canonical Form

MySQL lets you use different sql_mode settings; each setting tells the server the type of SQL syntax to support and the type of data validation checks it should perform. For example, you might want to use the "ansi" mode so that you can use the standard SQL concatenation operator, the double bar, in your queries.

If you have created a view that specifies concatenation in its definition, you might worry that changing the sql_mode setting from "ansi" could cause the view to become invalid. But this is not the case -- no matter how you write out a view definition, MySQL always stores it the same way. For example, the server changes a double bar concatenation operator to a CONCAT function, and strips comments, as shown here:

```
mysql> SET SQL_MODE = 'ANSI';

mysql> CREATE VIEW v AS SELECT 'a' || 'b';

mysql> SHOW CREATE VIEW v;
+-------+-----------------------------------------------+
| Table | Create Table                                  |
+-------+-----------------------------------------------+
| v     | CREATE VIEW "db5"."v" AS select               |
|       | concat(_latin1'a',_latin1'b') AS `'a' || 'b'` |
+-------+-----------------------------------------------+
1 row in set (0.00 sec)
```

This is called storage in a canonical form and is a huge advantage. It means that if you switch later to using a different sql_mode setting, the view will still work as expected. If, instead, the view definition were stored using a double bar, then queries from the view would fail in other sql_mode settings because the double bar would no longer be interpreted as concatenation.

Another thing that you'll notice in the example is that the character literals are all stored with introducers, such as latin1 in this case (because that was the default character set at the time that the view was created). Again, this is an advantage. It means that even if you change the default character set to (e.g.) latin2 later, MySQL will still return the same view as the one in effect when you made the view.

Notice also that MySQL gave a name to the column that results from the unnamed expression 'a'||'b'. To create the name, the server took the expression and put it within back ticks (``). Anything within back ticks can be an identifier.

In other words, MySQL tries to produce consistent results for view selections regardless of the changes that you make to the environment. That's what the canonical form is for, and it's a good thing.

Style

There are not many style issues that refer specifically to views. I pass on the items that we think are relevant. These are just opinions, but they are common opinions.

• Always give view columns a name, instead of relying on MySQL's automatic name generation.

• Use AS clauses rather than [(column-list)]
In early versions of SQL, there was no AS clause, so the [(column-list)] was necessary in order to give names to view columns. Now the AS clause is standard, normal, and better because there's one less thing to learn (anybody who knows SELECT should know the AS clause anyway). Therefore, it's better to do this:

CREATE VIEW v AS SELECT 1 AS a, 2 AS b, 3 AS c;
rather than this:
CREATE VIEW v (a,b,c) AS SELECT 1, 2, 3;

• It is all very well for me to name my views "v" for examples, but you should use meaningful names when you have some serious data in your database. There are pundits who suggest a convention, using a prefix to indicate that the object is a view, such as "v_employees" or "ViewEmployees". I disagree: A view should look in all respects like a base table, unless you are searching the metadata. So "Employees" is a better name, especially if you're using views to help augment database security. If "Employees" is already used to name the base table upon which the view is based, and the view's SELECT definition restricts the data in some way -- for example, as in CREATE VIEW ... AS SELECT name FROM Employees WHERE country = 'UK'; -- then let the name be "Employee_names" or "Employees_uk".

• Don't put ORDER BY in a view definition. Sort view results as you do table results, in queries.

• If a view is updatable, note what the CHECK OPTION clause should be. If you're going to use cascading, specify WITH CASCADED CHECK OPTION, rather than letting MySQL assume that CASCADED is the default.

Some Examples of Views in Use

In this section, I'll show you some ways in which you can make use of views.

Example: Synonym for too-long table name

I prefer to use INFORMATION_SCHEMA rather than SHOW, for reasons I've already stated. But typing out SELECT ... FROM INFORMATION_SCHEMA.COLUMNS ... is tediously long. And besides, I usually want to see only the columns in my own database. So I make a view that gives me the information I want and has a shorter name, thus:

```
mysql> CREATE VIEW cols AS
->     SELECT * FROM INFORMATION_SCHEMA.COLUMNS
->     WHERE TABLE_SCHEMA = DATABASE();
Query OK, 0 rows affected (0.00 sec)
```

Then later I can say SELECT ... FROM cols .... This is one time when it makes sense to use SELECT * in the view's SELECT definition.

There are, of course, other ways to accomplish the same thing:

• CREATE SYNONYM -- MySQL doesn't support this
statement, and I'm not suggesting that support will be added. But a synonym for a table, which is simply another name for the table, is slightly handier because its use requires no privileges. So you might want to switch to using synonyms in some future MySQL version.

• CREATE PROCEDURE -- I could have done this:
```
CREATE PROCEDURE cols ()
SELECT * FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_SCHEMA = DATABASE();
```

Then, later, CALL cols() would get me what I want. But format and order would be fixed. I wouldn't be able to say something like CALL cols() ORDER BY result-column. So I like views better for this purpose.

Example: Constraint checks

MySQL supports the NOT NULL column constraint, e.g.:

```
CREATE TABLE t (
  column1 INT NOT NULL,
  column2 INT NOT NULL);
```

But that's all. I can't, for example, specify that "either column can be null but not both", I can't say that "column1 must be between 1 and 5" nor can I require that "column1 can be NULL on special occasions". But I restrict data in all of these ways if I use a constrained view rather than a base table. Here's how:

```
CREATE TABLE t (column1 INT, column2 INT);

CREATE VIEW v AS
SELECT column1, column2 FROM t
WHERE (column1 is NOT NULL OR column2 IS NOT NULL)
  AND column1 BETWEEN 1 AND 5
WITH CHECK OPTION;
```

Then I do all my inserts and updates via view v rather than base table t, except on special occasions when I want to avoid the constraint.

From time to time, I might want to know if t has rows that violate the constraints on v, but for some reason (probably lack of privileges) I don't know what the constraints are. To do this, I can say:

```
SELECT (SELECT COUNT(*) FROM t) - (SELECT COUNT(*) FROM v);
```

MySQL's support for "views with check option" thus partially compensates for MySQL's lack of support for the standard SQL requirement "CHECK clause [on base table]". (A CHECK clause on a base table, for example CREATE TABLE t (column1 INT, CHECK (column1 BETWEEN 1 AND 5); makes inserts or updates on t impossible if the condition is untrue.)

This technique is good for inserts and updates, but bad for queries. If I now say

```
SELECT column1 FROM v;
```

MySQL will translate the query to:

```
SELECT column1 FROM t
WHERE column1 IS NOT NULL OR column2 IS NOT NULL;
```

If I had made sure that the view's check option was always checked, that is if I had always updated the view and never the base table, then the WHERE clause is a waste of time.

Example: Hiding a complex WHERE clause

Suppose there is some long and tricky predicate in a WHERE clause:

```
SELECT column1
FROM t
WHERE UPPER(column1) = 'WHO';
```

As a result of tedious checking and testing, you know that (a) you have to say UPPER because column1 is case sensitive and (b) you know that UPPER(column1)='WHO' is faster than 'WHO'=UPPER(column1) because MySQL caches statements if (and only if) they look the same. (I'm not saying this is true, but let's suppose it is for the sake of the example.)

You decide it would be better if everybody re-used your library of tested queries, so you create a view for others to use, like this:

```
CREATE VIEW t_column1_equal_who AS
SELECT column1
FROM t
WHERE UPPER(column1) = 'WHO';
```

Your view has to have a good, explanatory name, because you can't use comments. MySQL doesn't store comments in the view definition and there is no "COMMENT = 'string'" clause in CREATE VIEW. Therefore your view must be easy for others to look up.

How can you grant everyone access to your views, without having to say GRANT SELECT ON VIEW view_name ... every time you create a new view? Answer: By putting the view in a database that's made for the purpose, and executing this statement, just once:

```
GRANT SELECT ON made_for_purpose.* ...
```

Next question: How can you grant access to everybody, but only if they (now and in the future) are privileged to SELECT from a view's underlying base tables as well? I confess that I can't think of a way to do that with views, so I'll answer: When security is a concern, use stored procedures instead. For example,

```
CREATE PROCEDURE t_column1_equal_who ()
SQL SECURITY INVOKER
SELECT column1
FROM t
WHERE UPPER(column1) = 'WHO';

CALL t_column1_equal_who(); /* equivalent to SELECT view */
```

Example: Instead of a subquery

Beginning with MySQL 4.1, you can put a subquery in a FROM clause. For example:

```
SELECT *
FROM (SELECT * FROM t2 UNION SELECT * FROM t3) AS t2_t3;
```

However, this might be better:

```
CREATE VIEW t2_t3 AS
SELECT * FROM t2 UNION SELECT * FROM t3;

SELECT * FROM t2_t3;
```

Querying the view is better when you would otherwise have a large number of (say, 55) SELECT statements that contain the same subquery and your database definition is subject to change. That way, if somebody renames table t2, you only have to re-do the view definition. You don't have to change all 55 SELECT statements.

Another subquery example: In MySQL 5.0.3, you can't use AVG(DISTINCT ...), that is, this statement is not legal syntax:

```
SELECT AVG(DISTINCT column1) FROM t;
```

But the next set of statements is legal and returns the average of the distinct values rather than just the average:

```
CREATE VIEW v AS SELECT DISTINCT column1 FROM t;

SELECT AVG(column1) FROM v;
```

Example: Joins

Views are not always the answer. Suppose you have two tables and an ordinary equijoin view like this:

```
CREATE TABLE t1 (s1 int);

CREATE TABLE t2 (s1 int);

CREATE VIEW v AS
SELECT t1.s1 AS t1_s1, t2.s1 AS t2_s1
FROM t1, t2 WHERE t1.s1=t2.s1;
```

Statements like the following might cause trouble:

```
SELECT DISTINCT t1_s1 FROM v WHERE t1_s1 = 1;
```

It's probably faster to query the tables directly with a subquery, e.g.:

```
SELECT DISTINCT s1 FROM t1
WHERE s1 = 1 AND s1 IN
(SELECT s1 FROM t2);
```

This is because using this type of query will guarantee that the optimizer has a chance to do some filtering on one of the tables before having to match the relevant rows to the other. Joined views are notoriously difficult to optimize when there is a filter in the main query.

Another reason not to prefer the view is that statements like the following will fail:

```
UPDATE v SET s1 = 5;
```

Although MySQL can do multi-table updates via a view, the required syntax is complex. So at the moment you'll still be better off doing updates directly on the base tables.

Is there some sort of general lesson that we can draw from these examples? It's not something that you can rely on as a law, but I'll tentatively suggest that:

- Views are designed to hide (irrelevant) information, but speeding up a query requires the presence of maximal information.

• A view's operations tend to be performed at an early stage especially if the view gets materialized, and that might be undesirable when the main query is very specific.
• There are still some restrictions that apply to views but not to base tables.

Examples: Privilege granting

Let's make a table of employees, their names and their wages, with notes:

```
CREATE TABLE Employees (
  name CHAR(20),
  wage DECIMAL(5,2),
  note CHAR(20));

INSERT INTO Employees VALUES ('pierre',3.25,'reliable');

INSERT INTO Employees VALUES ('wanda',4.00,'worthy');
```

Now, assume I want Pierre to have access to the name and wage columns of this table, but not to the notes column. So I grant these privileges:

```
GRANT SELECT (name, wage) ON Employees TO pierre;
```

When Pierre tries:

```
SELECT * FROM Employees;
```

He gets this error message:

```
select command denied to user 'pierre'@'localhost' for column 'note' ...
```

Not a good result! Pierre should be able to execute a "SELECT *" query without receiving errors -- and without being told the name of a column to which he shouldn't have access. Granting privileges on the base table doesn't have the result I intended. So instead, I do this:

```
CREATE VIEW Employees_name_and_wage AS
SELECT name, wage FROM Employees;

GRANT SELECT ON Employees_name_and_wage TO pierre;
```

Ah, but Pierre should not know Wanda's salary. How can we ensure that users see only their own information? Like this:

```
CREATE VIEW Employees_name_and_wage_2 AS
SELECT name, wage FROM Employees
WHERE CAST(LEFT(CURRENT_USER,
      POSITION('@' IN CAST(CURRENT_USER
      AS CHAR CHARACTER SET latin1))-1)
      AS CHAR CHARACTER SET latin1)
      = name;

GRANT SELECT ON Employees_name_and_wage_2 TO pierre;
```

The casting has to be done because my default character set is latin1 and the CURRENT_USER function returns a utf8 string. It wasn't necessary, but I decided to do this illustration the hard way.

Now let's assume that Pierre should know the total number of employees ... so let him see a view that says SELECT COUNT(*). Again, that is a security provision that is impossible to do by saying GRANT (column-list) ON base-table-name. But it's straightforward with views.

Bugs and Feature Requests

MySQL 5.0 was not a production release version at the time this book was written, so some aspects of the views implementation were incomplete. This section provides some information on bugs and features outstanding when we went to press.

Bugs

To get current details on outstanding view bugs, go to the following web page and search for "view" or "views":
http://bugs.mysql.com

As you'd expect with any new large feature, there were numerous outstanding view bugs when the code was originally released in the summer of 2004. At the time of writing, a great many had already been fixed.

Feature Requests

The following small feature requests were outstanding at the time of writing:

- TRUNCATE, RENAME, etc. should work with views.
- INFORMATION_SCHEMA.VIEW_TABLE_USAGE should be implemented.
- CHECK TABLE[S] should not require a list of view names.
- Comments in views should be preserved.

There was also one big feature request:

- Triggers should work with views.

MySQL AB doesn't promise to look at these feature requests for version 5.0 or even for version 5.1. All we promise is: We will listen to our users. So go to http://bugs.mysql.com, find the request for your favorite feature, click "Add Comment", and say "I need this too [because ...]".

Resources

For my last act I'll show you where you can go for further information.

Source Code

sql/sql_view.cc
mysql-test/t/view.test

As always with MySQL, you can download our source code and see how Oleksandr did it. The main program for views is called sql_view.cc. Another interesting file is view.test, which will give you an idea of how thorough our test suite is.

Literature

There's a lot that you can read on the subject of views. I recommend the writings of C.J. Date most heartily. Try his Guide to the SQL Standard.

Peter Gulutzan wrote two articles that are useful backgrounders:

- SQL Views Transformed
  http://www.dbazine.com/gulutzan9.shtml
- Views work in MySQL 5.0
  http://databasejournal.com/features/mysql/article.php/3399581

And of course there's a chapter on views in the book that we wrote together: SQL-99 Complete, Really.

Conclusion

You've come to the end of the book. I don't bother with a review or an index, since I'm sure you've had no trouble memorizing it all.

If you enjoyed this book, you should look for others in the "MySQL 5.0 New Features" series. The previous books talk about "Stored Procedures" and "Triggers". The next book will be "INFORMATION_SCHEMA: The MySQL Data Dictionary".

Thank you very much for your attention. If you have any comments about this book, please send them to one of the MySQL forums at:
http://forums.mysql.com

About MySQL

MySQL AB develops and supports a family of high performance, affordable database servers and tools. The company's flagship product is MySQL, the world's most popular open source database, with more than six million active installations. Many of the world's largest organizations, including Yahoo!, Sabre Holdings, The Associated Press, Suzuki and NASA are realizing significant cost savings by using MySQL to power high-volume Web sites, business-critical enterprise applications and packaged software.

With headquarters in Sweden and the United States – and operations around the world – MySQL AB supports both open source values and corporate customers' needs in a profitable, sustainable business. For more information about MySQL, please visit www.mysql.com.