

Summary tables, aka materialized views using Flexviews



<http://Flexvie.ws/>

Improving database performance
using
materialized views

O'Reilly MySQL Conference 2011
Santa Clara, CA

Justin Swanhart

Introduction

Who am I?

What do I do?

Why did I write Flexviews?



Requirements



MySQL 5.1+, MySQL 5.5

Row based binary logging

PHP 5.2+

Requirements (cont)



SUPER privileges

READ-COMMITTED transaction isolation

log_slave_updates

What are views?



A view is a SQL statement that is saved in the database and executed on demand.

Views have a “first class” name and can be referenced as if they were tables themselves.

Views have many well known performance problems



Query is executed on every access

Because the query is executed each time, the query plan may change unexpectedly.

Temporary tables usually required

What are materialized views? (MV)



A view is “*materialized*” when **its contents are stored (*cached*) in a table.**

Materialized?

materialize, **materialise** [mə'tɪəriə,laɪz]vb

1. (*intr*) to become fact; actually happen our hopes never materialized
2. **to invest or become invested with a physical shape or form**
3. to cause (a spirit, as of a dead person) to appear in material form or (of a spirit) to appear in such form
4. (*intr*) to take shape; become tangible after hours of discussion, the project finally began to materialize
5. (Physics / General Physics) *Physics* to form (material particles) from energy, as in pair production

[Collins English Dictionary – Complete and Unabridged](#) ©

HarperCollins Publishers 1991, 1994, 1998, 2000, 2003

MV are real tables!



Real tables can be indexed, sorted, etc.

A point-in-time snapshot

Base tables are the underlying tables accessed by a view.

First class object

Why haven't I heard of them?



MySQL doesn't natively support them.

Microsoft SQL Server calls them indexed views.

DB2 calls them materialized query tables

Oracle calls them snapshots, or materialized views, depending on the version.

No support in MySQL?

Why am I here again?



Please sit back down

Why am I here again?

Because Flexviews adds feature rich materialized views to MySQL.



It is just a cache



When the base tables change, the view may lose synchronization with those tables.

To combat this, the MV must be updated.

It is important to update the MV efficiently.

When does it get updated?



Choice #1 –

Update the views 'on every commit'.

Each view is updated synchronously with changes to the base tables.

Cost incurred in every transaction

When does it get updated (cont)?



Choice #2

Update the views asynchronously.

MV may be out of date longer, but updates may be batched.

Very little impact on individual transactions

Flexviews takes approach #2



Flexviews uses stored procedures to create and manage materialized views

Can create views which can be *incrementally refreshed*.

Change data capture makes this possible

MySQL has the ingredients

CREATE TABLE ... AS SELECT ...;

Row based binary logging

Stored procedures

Flexviews



The stored procedures are documented online:

<http://flexviews.googlecode.com/svn/trunk/manual.html>

Supports two different refresh methods:

The INCREMENTAL method reads from change logs and updates views efficiently.

The COMPLETE refresh method rebuilds the MV from scratch each time.

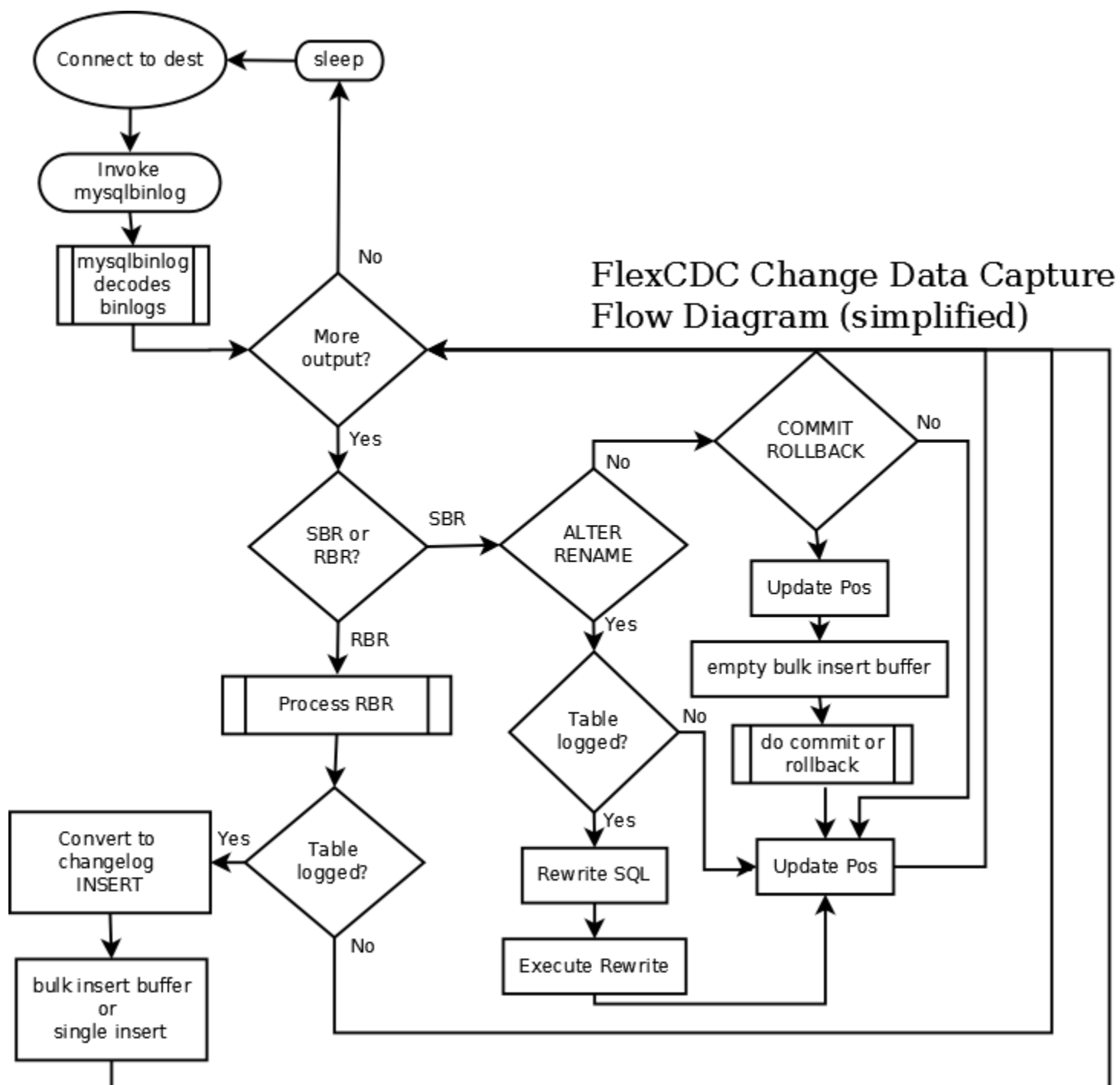
Why not use triggers?



Triggers can not capture transaction order.

The cost of every transaction is increased.

No remote processing/remote capture.



Change Data Capture (cont)

```
mysql> create database example;
Query OK, 1 row affected (0.00 sec)

mysql> create table example.ex1(
-> c1 int auto_increment primary key,
-> c2 int
-> ) engine=innodb;
Query OK, 0 rows affected (0.01 sec)

mysql> call
-> flexviews.create_mvlog('example','ex1');
Query OK, 1 row affected (0.02 sec)

mysql> insert into
-> example.ex1
-> values (NULL,1),
-> (3,2),
-> (NULL,3);
Query OK, 3 rows affected (0.01 sec)
Records: 3 Duplicates: 0 Warnings: 0
```

Create the log

```
mysql> select * from flexviews.example_ex1\G
***** 1. row *****
dml_type: 1
uow_id: 16402
fv$server_id: 999
c1: 1
c2: 1
***** 2. row *****
dml_type: 1
uow_id: 16402
fv$server_id: 999
c1: 3
c2: 2
***** 3. row *****
dml_type: 1
uow_id: 16402
fv$server_id: 999
c1: 4
c2: 3
3 rows in set (0.00 sec)
```

Make some changes

Changes are captured in a table changelog

Two types of refresh



Incremental Refresh

Uses table changelogs created by FlexCDC

Limited SQL syntax options

Built using stored procedures (the SQL_API)

Complete Refresh

Rebuilds the MV completely on each refresh

Supports all SQL syntax

Built directly from SQL statements

First, incremental refresh



Why use incremental refresh?

How do I create views?

How does it work?

Why use it?



Because it is faster!

Faster refresh means you use less resources.
This saves money over time.

Make your customers more happy by providing responsive dashboards or reports, even over large amounts of data.

Fast enough for “near real time”, if you want.

How do I create views?



Flexviews has an internal data dictionary
Stored procedures are used to manipulate
this dictionary.

The dictionary is used to construct the SQL that
computes the changes.

You can't create the view directly from SQL

Quick overview: SQL_API



```
flexviews.create($schema, $table, $method);  
flexviews.get_id($schema, $table);
```

```
flexviews.add_table($id, $schema, $table, $alias, $join_condition);  
flexviews.add_expr($id, $expr_type, $expr, $alias);  
flexviews.enable($id);
```

```
flexviews.refresh($id, $method, $to_trx_id);  
flexviews.disable($id);
```

The stored procedures are documented online:

<http://flexviews.googlecode.com/svn/trunk/manual.html>

“Simple” Example Part 1

DON'T PANIC – There is an easier way I'll show you

```
mysql> call
flexviews.create('example','incl','INCREMENTAL');
Query OK, 1 row affected (0.00 sec)

mysql> set @mvid :=
flexviews.get_id('example','incl');
Query OK, 0 rows affected (0.00 sec)

mysql> call flexviews.add_table(@mvid,
'demo','order_lines','ol',NULL);
Query OK, 1 row affected (0.00 sec)

mysql> call
flexviews.add_expr(@mvid,'COUNT','*','the_count');
Query OK, 1 row affected (0.01 sec)

mysql> call flexviews.enable(@mvid);
Query OK, 0 rows affected, 1 warning (19.61 sec)

mysql> select the_count from example.incl;
+-----+
| the_count |
+-----+
| 155186550 |
+-----+
1 row in set (0.00 sec)
```

← Reserve an ID for the view
Also note INCREMENTAL

← Save the ID

← Add the table

← Add a COUNT(*) expression

← Create the initial snapshot

← Check the contents

convert.php makes this easier

Input SQL statement(s)

```
echo "  
create table example.incl  
as select count(*) the_count  
from demo.order_lines ol;"|php convert.php example  
  
CALL flexviews.create('example', 'incl',  
'INCREMENTAL');  
  
SET @mvid := LAST_INSERT_ID();  
  
CALL flexviews.add_expr(@mvid, 'COUNT', '*',  
'the_count');  
  
CALL  
flexviews.add_table(@mvid, 'demo', 'order_lines', 'ol  
, NULL);  
  
CALL flexviews.enable(@mvid);
```

PHP script

SQL_API* output

These are the same commands as the last slide.

How the refresh works



The refresh algorithm executes in two stages:

Delta computation (COMPUTE phase)

Examines changes and builds *delta* records

Delta application (APPLY phase)

Applies the *delta* records to the MV

Delta tables



Every incrementally refreshable materialized view has an associated *delta* table.

The compute process for a view populates the *delta* table for that view.

Views based on a single table can compute deltas records *from the changelog directly*.

Why two phases?



The COMPUTE phase inserts rows into the delta tables.

Computing the changes frequently helps to ensure that the number of changes to be examined are small.

Computing changes is more expensive than applying deltas.

Delta computation

```
mysql> select * from example.incl;
+-----+-----+
| mview$pk | the_count |
+-----+-----+
|          1 | 155132000 |
+-----+-----+
1 row in set (0.00 sec)
```

← Before any changes

```
mysql> delete from demo.order_lines limit 30000;
Query OK, 30000 rows affected (1.04 sec)
```

```
mysql> delete from demo.order_lines limit 2000;
Query OK, 2000 rows affected (0.26 sec)
```

} Two transactions make changes

```
mysql> call flexviews.refresh(
-> flexviews.get_id('example','incl'),
-> 'COMPUTE',NULL);
Query OK, 0 rows affected (0.81 sec)
```

← Compute the changes

```
mysql> select * from example.incl_delta;
+-----+-----+-----+-----+
| dml_type | uow_id | mview$pk | the_count |
+-----+-----+-----+-----+
|          -1 | 16590 |          NULL |          -32000 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

← Delta table contains the delta record

Delta application

```
mysql> select * from example.incl1;
+-----+-----+
| mview$pk | the_count |
+-----+-----+
|          1 | 155132000 |
+-----+-----+
1 row in set (0.00 sec)
```

← Out of date now

```
mysql> call
flexviews.refresh(flexviews.get_id('example','inc
1'),'APPLY',NULL);
Query OK, 0 rows affected (0.01 sec)
```

← Apply the changes

```
mysql> select * from example.incl1_delta;
Empty set (0.00 sec)
```

← Delta table now empty

```
mysql> select * from example.incl1;
+-----+-----+
| mview$pk | the_count |
+-----+-----+
|          1 | 155100000 |
+-----+-----+
1 row in set (0.00 sec)
```

← View is now updated

Accessing base table still slow

```
mysql> select count(*) from demo.order_lines;
+-----+
| count(*) |
+-----+
| 155100000 |
+-----+
1 row in set (19.33 sec)
```

Complete refresh example

```
mysql> call flexviews.create(  
-> 'demo','dashboard_top_customers','COMPLETE');  
Query OK, 1 row affected (0.00 sec)
```

Reserve the name for the view

```
mysql> call flexviews.set_definition(  
->flexviews.get_id('demo','dashboard_top_customers'),  
-> 'select customer_id,  
-> sum(total_price) total_price,  
-> sum(total_lines) total_lines ,  
-> from demo.dashboard_customer_sales dsc  
-> group by customer_id  
-> order by total_price desc');  
Query OK, 1 row affected (0.00 sec)
```

flexviews.set_definition associates SQL with the view

Name is associated with an id

SQL that defines the view

```
mysql> call flexviews.enable(  
-> flexviews.get_id('demo','dashboard_top_customers'));  
Query OK, 0 rows affected (5.73 sec)
```

Make it available for querying

Complete refresh example

```
mysql> select mview$pk as rank,  
-> customer_id,  
-> total_price,  
-> total_lines  
-> from demo.dashboard_top_customers  
-> limit 5;
```

rank	customer_id	total_price	total_lines
1	689	770793	3811
2	6543	754138	3740
3	5337	742034	3674
4	5825	738420	3593
5	5803	733495	3670

```
5 rows in set (0.00 sec)
```

You get ranking for free when you use an ORDER BY in the SQL definition of the view.

Layering Materialized Views



Views that use the complete refresh method can not be refreshed incrementally.

This means that an expensive query will take a long time to refresh.

This affects the frequency at which you can refresh the view

Complete refresh (again)

```
mysql> call flexviews.create(  
-> 'demo','dashboard_top_customers','COMPLETE');  
Query OK, 1 row affected (0.00 sec)
```

```
mysql> call flexviews.set_definition(  
->flexviews.get_id('demo','dashboard_top_customers').  
-> 'select customer_id,  
-> sum(total_price) total_price,  
-> sum(total_lines) total_lines ,  
-> from demo.dashboard_customer_sales dsc  
-> group by customer_id  
-> order by total_price desc');  
Query OK, 1 row affected (0.00 sec)
```

```
mysql> call flexviews.enable(  
-> flexviews.get_id('demo','dashboard_top_customers'));  
Query OK, 0 rows affected (5.73 sec)
```

This is another MV

```
SELECT  
customer_id, ...  
-> FROM demo.orders as o  
-> JOIN demo.customers as c  
-> USING (customer_id)  
-> JOIN demo.order_lines as ol  
-> USING (order_id)  
-> GROUP BY (customer_id)  
-> ORDER BY total_price desc  
-> LIMIT 10;
```

```
...  
| 6019 | 718031 | 3475 |  
+-----+-----+-----+
```

10 rows in set (43 min 10.11 sec)

Very fast to build

A lot slower

Layering Views (cont)



Building complete refresh views on top of incremental ones allows you to get the best of both worlds

Use features like NOW() or ORDER BY which you can't use in incremental views

Keep multiple complete views in sync with each other by building them from the same incremental ones

Both types benefit from layering



Build an incrementally refreshable view aggregated to the day level.

Create a changelog on that view (*it is a table*)

Build a month level incrementally refreshable view from the day level view.

Functional Indexes



Some databases allow you to build an index on a function or an expression, or apply a filter on the values to index.

MV can be used to create similar structures

Simulate hash indexes

Simulated Hash Indexes



MySQL includes a CRC32 function

This can be used to create a hash index

A similar example would be an index on reverse()

To make LIKE '%string' more efficient

Simulated Hash Indexes

```
mysql> call flexviews.create('example','crc32_example','INCREMENTAL');  
Query OK, 1 row affected (0.10 sec)
```

```
mysql> set @mvid:=LAST_INSERT_ID();  
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> call flexviews.create_mvlog('example','url');  
Query OK, 1 row affected (0.05 sec)
```

```
mysql> call flexviews.add_table(@mvid,'example','url','url',NULL);  
Query OK, 1 row affected (0.07 sec)
```

```
mysql> call flexviews.add_expr(@mvid, 'COLUMN','url_id', 'url_id');  
Query OK, 1 row affected (0.52 sec)
```

```
mysql> call flexviews.add_expr(@mvid, 'COLUMN','crc32(url)',  
'url_hash');  
Query OK, 1 row affected (0.27 sec)
```

```
mysql> call flexviews.enable(@mvid);  
Query OK, 0 rows affected, 1 warning (2 min 25.05 sec)
```

PK



function



Simulated Hash Indexes (cont)

Smaller indexes are faster to update

Hash not useful for range scans, case sensitive searches, etc

Adding a b-tree index to the url table

```
mysql> alter table example.url add key(url);
```

← VARCHAR(1000)

Query OK, 0 rows affected, 0 warnings (31 min 8.81 sec)

Records: 0 Duplicates: 0 Warnings: 0

Adding a b-tree index to the hash table

```
mysql> alter table crc32_example add key(url_hash);
```

← INT UNSIGNED

Query OK, 0 rows affected (1 min 42.84 sec)

Records: 0 Duplicates: 0 Warnings: 0

Simulated Hash Indexes (cont)

```
SELECT url.*  
FROM demo.crc32_example crc  
JOIN demo.url url  
ON (  
  crc.url_hash=crc32('http://path/to/url')  
  AND crc.url_id = url.url_id  
  AND url.url = 'http://path/to/url'  
)
```

Could be an IN list

Need to handle hash collisions

Partial Indexes Too



An orders table contains order_status

Cardinality of order_status is usually very low

Distribution is usually very skewed

Most of the rows are in 'closed' for example

Use an MV to "index" rows where status != 'closed'

Improve caching with MV



Back to the concepts of cache

Memcache and other result caches usually work great

Miss path is as important as the hit path

Expensive joins and aggregation can make the miss expensive

Or use it with HandlerSocket



Pre-join and aggregate data into MVs

Access the MVs with HandlerSocket

Reduced round trips means that perhaps you don't need Memcached

Dashboards



Consider a typical dashboard

- Many totals/subtotals

- Frequent aggregation

- Long historical view

Executive level dashboards

- Low tolerance for data latency

- Low tolerance for high response times

Feed external systems



- **Sphinx** ← Inverted indexes (full text and GIS)
- **Lucene** ←
- **Fastbit** ← WAH compressed bitmap indexes
- **Gearman**
- **Etc**

Special Thanks To



Salem, K., Beyer, K., Lindsay, B., and Cochrane, R. 2000. How to roll a join: asynchronous incremental view maintenance. *SIGMOD Rec.* 29, 2 (Jun. 2000), 129-140. DOI= <http://doi.acm.org/10.1145/335191.335393>

Mumick, I. S., Quass, D., and Mumick, B. S. 1997. Maintenance of data cubes and summary tables in a warehouse. *SIGMOD Rec.* 26, 2 (Jun. 1997), 100-111. DOI= <http://doi.acm.org/10.1145/253262.253277>

Percona

AdBrite

Proven Scaling, LLC, and Jeremy Cole, in particular.

