Database Development

Dr. Dobb's Journal

June 2013

Profiling SQL Activity in MySQL 5.6

Next

ALSO INSIDE

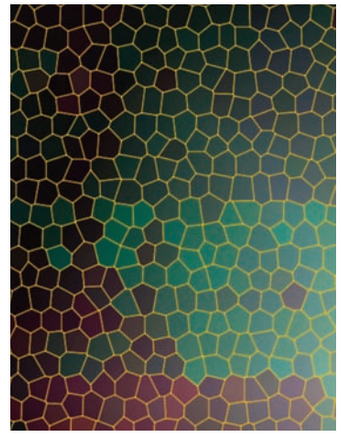
UBM Tech

www.drdobbs.com

Previous Next

Dr. Dobb's Journal

CONTENTS



June 2013

COVER ARTICLE

EDITORIAL

FEATURES

More on DrDobbs.com

Introduction to Hadoop: Real-World Hadoop Clusters and Applications

The Hadoop ecosystem relies on composability — the ability to use output from one tool as input to the next — to efficiently process data at scale, from simple projects, to processing streams of real-time data, to building data warehouses.

http://www.drdobbs.com/240153375

Working with TypeScript in Visual Studio 2012

Using Visual Studio makes working with TypeScript easy and helps reveal subtleties of the language implementation and code generation.

http://www.drdobbs.com/240154792

Supercharging ASP.NET Web Form Apps

With the introduction of the ASP.NET Web API in ASP.NET 4.5, it is now possible to add advanced client-side functionality to existing sites.

http://www.drdobbs.com/240153012

Keccak: The New SHA-3 Encryption Standard

After years of testing and analysis, the U.S. government selected the Keccak algorithm to be the new SHA-3 encryption standard. Here is how it works.

http://www.drdobbs.com/240154037

NoSQL Options Compared

As NoSQL takes root in the enterprise, it's becoming clear that developers will need to gain greater familiarity with both the concepts behind it, as well as the various products that support it.

http://www.drdobbs.com/240151198

Orthodoxy vs. Pragmatism, or How I Became a Better Developer

Few indeed are the practices that can be recommended universally. Wisdom suggests the humility of knowing this and applying it liberally.

By Andrew Binstock

It is a distressing thing to see how much of programming today is still discussed in terms of all-or-nothing propositions. Whether it's the insistence that one language, or type system, or model of development is inherently and universally superior, I find the lack of nuance troubling. Life has taught me well that most of the benefit of techniques and technologies lies not in their absolute or universal application, but rather in using them with discrimination.

The dogmatic insistence on using one exclusive approach rings of inexperience or, if proclaimed by an experienced hand, a lack of thoughtful consideration. When young I was and green, I had all kinds of ideas on how code should be written. Since I was inexperienced — and didn't appreciate how inexperienced — I would passionately argue for techniques and tools that, in retrospect, I only partially understood. Mostly, they appealed to me for some aspect whose limitations were a sealed book, but whose benefits were completely obvious — if only I could get others to recognize them, too. I was full of wonderful ideas and remedies. I was living the life of an insufferable fellow, but enjoying it because, dang!, I just knew so much. And where I lacked specific knowledge, I relied on people I admired, who used this or that tool or some special technique. If it was good enough for them, well, it was good enough for me, let me tell you. (Today's more common equivalent is, "My professor says that…")

Maturity brought wisdom and I began to realize that few indeed were the tools and practices that could be recommended universally. I also started to see that even for proponents of various techniques, there was always wiggle room. Exceptions to blanket rules were not to them evidence of imperfection in their beliefs, but validation that the real world could not be entirely reified to comply with a single edict.

For example, let's take a practice that seems to elicit more religion than many others: test-driven development (TDD). The steps in TDD are all well-known, formulaic, and perfectly set up for doing all things exactly one way. A key principle of TDD is that you write no code without first writing a failing unit test. But in fact, if you talk to the principal

advocates of TDD (such as Kent Beck, who popularized the technique, and Bob Martin [http://is.gd/u6pN9C], who has taught it to thousands of developers), you find that both of them write some code without writing tests first. They do not — I should emphasize this — view these moments as lapses of faith, but rather as the necessary pragmatism of the intelligent developer.

It is tempting to view programming in moral terms because we so often refer to code as good or bad, but programming lacks the defining overarching moral imperative towards which good things tend. To wit, the frequent example of the three divergent poles in programming: good, fast, and cheap. Different projects work within the constraints imposed by individual blends of these properties, which is why a CRUD app has more tolerance for errors than a pacemaker — and why holding that some practice is the right way in all circumstances is bound to be wrong. Such a view, in my opinion, is a meta-level code smell.

Even if we limit ourselves to a smaller segment of software — for example, pacemakers — the way forward is certainly one of discipline and rigor (within the constraints of quality-performance-cost), but not orthodoxy. Where exactly is the line drawn? It is, in my opinion, wherever activity that is valid in one area is extended beyond the area of its benefit. Let's choose a simple example: commenting. If the rule is to comment all code, then orthodoxy has prevailed over pragmatism. Such excessive commenting is likely to be costly, slow, and bug-ridden. That is, all three vectors point the wrong way. 100% test coverage of code is a more subtle example. It's the extension of a good idea to every aspect of an activity — almost always a sign of orthodoxy. I could go on with other instances, such as language selection, agile and DevOps practices, testing, and nearly every other aspect of development.

The best programmers I've met are not only intensely pragmatic, they actively pursue pragmatism. They look at new technologies, so as to avoid choices made inadvertently orthodox by ignorance of alternatives. They use tools in a balanced way that ruthlessly seeks delivery of intended results within the established project constraints. The more I work like them, the better programmer I am.

— Andrew Binstock is Editor in Chief of **Dr. Dobb's.** *Contact him at alb@drdobbs.com.*

Comment

News Briefs

By Adrian Bridgwater

Mozilla and Otoy Announce ORBX.js JavaScript Library

Graphics-focused cloud services firm Otoy has worked with Mozilla to release the ORBX.js JavaScript library. This new software works on Windows, Linux, or Mac OS X apps to "cloud virtualize" them so that they can be streamed to HTML5-enabled browsers, including those on mobile devices.

http://www.drdobbs.com/240154247

Open Source Usage Up As Controls and Processes Fail

Sonatype has gone public with the findings of its annual Open Source Development Survey. The study claims to be the "largest of its kind" surveying (as it does) more than 3,500 developers, architects and IT managers currently using open source. Key findings suggest that much of software today is now assembled from open source components and frameworks downloaded from repositories (at least 80% of the app). But the investigation also proposes that few organizations have the controls or processes to identify which components are in use, to govern their usage, or to eradicate flawed components from production applications.

http://www.drdobbs.com/240153975

Multi-Device True Native App Development

Embarcadero has launched its RAD Studio XE4 app development suite for building true native apps for PCs, tablets, and smartphone with one codebase.

http://www.drdobbs.com/240153793

450 Million Lines Of Scanned Software Code Can't Be Wrong

Coverity's analysis found an average defect density of .69 for open source software projects that leverage the firm's own scan service. It also found an average defect density of .68 for proprietary code developed by the firm's own enterprise customers.

http://www.drdobbs.com/240154424

Building Apps With CDK In Cloudera Apache Hadoop

Apache Hadoop software, support, and services company Cloudera has released the Cloudera Developer Kit (CDK) at its first developer kit created for the firm's own distribution of Hadoop (CDH).

http://www.drdobbs.com/240154745

Crowdsourced Scripts For MSPs

GFI Software has launched a new social community as a forum for technical staff among managed services providers (MSPs) to obtain crowdsourced scripts that solve IT problems. The FixIT Scripts platform bids to unite software application developers within the MSP community by connecting those looking to source a script with those who can provide scripts that solve a specific IT problem.

http://www.drdobbs.com/240154713

NuoDB Releases 1.1 For Microsoft .NET And Azure

Starlings Release 1.1 is an advanced architecture for performing real-time operational analytics.

http://www.drdobbs.com/240154806

Detailed Profiling of SQL Activity in MySQL 5.6

MySQL's latest update to the Performance Schema brings the ability to profile a statement's activity, low-level wait events, and I/O impact. It is the easiest and most detailed way to identify what statements to tune and how.

By Mark Leith

The new MySQL 5.6 release adds instrumentation that enables developers and DBAs to better understand what is happening at all layers of the MySQL server. One of the major improvements is the addition of statement statistics, in either a raw (per-statement) or aggregate form, with the ability to drill down into the stages of execution that each statement goes through. It also enables further drill down into the low-level wait events that occur during each stage of execution.

Performance Schema Introduction

Introduced in MySQL 5.5, the Performance Schema (http://is.gd/pdrUNt) is a new default database schema (called `performance_schema`), focused primarily on low-level performance metrics of the MySQL server runtime. This feature is designed to complement the `INFORMATION_SCHEMA`, which is focused on database metadata.

The tables in the `performance_schema` database use a new storage engine (called `PERFORMANCE_SCHEMA`), which is specifically designed for tracking the low-level information required to profile the server. All tracked performance metrics are stored in fixed-size ring buffers, which are allocated when the MySQL database is started. (It's designed this way for performance, so no costly memory allocation is needed while recording performance statistics).

Like `INFORMATION_SCHEMA`, all data within `performance_schema` is non-persistent — performance metrics within it are reset when the MySQL database is restarted. The core job of `performance_schema` is to track events. With MySQL 5.6, there are now three classes of events:

- Statement Events: Per-statement statistics, such as runtime, lock time, etc.
- Stage Events: These record the stages of statement execution, such as optimizing, sending data, etc.
- Wait Events: These track low-level wait periods, such as file I/O, network I/O, mutex contention points, etc.

Each of these events is classified by Instruments. Each defined instrument records runtime statistics on a specific aspect of the MySQL database, with the naming convention for each wait event instrument following a hierarchical taxonomy: `Class/Order/Family/Genus/Species`, while stages and statements only go as far as the `Family` level. For instance, file I/O against an InnoDB data file is called `wait/io/file/innodb/innodb_data_file`, while a wait on an InnoDB Buffer Pool mutex is called `wait/synch/mutex/innodb/buf_pool_mutex`.

Within the current version of MySQL 5.6, there are 541 individual instrumentation points that Performance Schema can track, across 11 major types of instrumentation:

- `statement/sql/`: Normal SQL statement executions (DML or DDL, etc.)
- `statement/com/`: Protocol level commands
- `stage/sql/`: The stages within each statement execution
- `wait/io/file/`: File I/O to data/index files or logs
- `wait/io/socket/`: Network I/O between the clients and server
- `wait/io/table/`: I/O between the storage engines and the SQL layer of the server
- `wait/lock/table/`: Table locks (at the SQL layer, not including InnoDB row-level locks)
- `wait/synch/mutex/`: Attempts to grab a mutex lock internally within the server
- `wait/synch/rwlock/`: Attempts to grab a read/write lock internally within the server
- `wait/synch/cond/`: Waiting on a condition to be raised within the server
- `idle`: Records the time that a connection sits idle

For each event, there are two core performance metrics tracked: the number of times that the event happened, and the latency of each event down to picosecond (thousandth of a nanosecond) precision. On top of this, there may be various other statistics recorded, which differ based upon the type of instrumentation point. For instance, a file I/O event will record the file that was being accessed, the type of file access (open, read, write, sync, close, and so on), and how many bytes were read or written.

When enabled, each instrument is tracked with `Consumers`, which are individual tables within Performance Schema that track the events in various ways. The major consumers are:

```
mysql> SHOW TABLES FROM performance_schema
-> WHERE Tables_in_performance_schema LIKE '%current'
-> OR Tables_in_performance_schema LIKE '%history%';

+----------------------------------+
| Tables_in_performance_schema     |
+----------------------------------+
| events_stages_current            |
| events_stages_history            |
| events_stages_history_long       |
| events_statements_current        |
| events_statements_history        |
| events_statements_history_long   |
| events_waits_current             |
| events_waits_history             |
| events_waits_history_long        |
+----------------------------------+
```

Each type of table has these characteristics:

- `*_current` tracks the currently executing or last executed event per thread
- `*_history` tracks the last 10 events executed per thread
- `*_history_long` tracks the last 10,000 events executed globally

The data from these tables is then further aggregated over several different dimensions. More information is available in the Performance Schema documentation (http://is.gd/kUpOeY).

Configuring Performance Schema for Profiling in Development

As of MySQL 5.6, Performance Schema is enabled by default. However, not all instrumentation has been enabled by default, as this can impact overall performance of the MySQL database when used in production. With production deployments in mind, only the higher latency types of events are enabled automatically: statements, file I/O, table I/O, table locks, and idle events.

When profiling your application activity in development, it's beneficial to have much more information available to you at the cost of a little runtime overhead to get at those performance metrics. Instruments can be enabled in two ways: either at server start, using the `performance_schema_instrument` system variable; or via the `performance_schema.setup_instruments` table at runtime, which has this structure:

```
mysql> SELECT * FROM performance_schema.setup_instruments;
+-----------------------------------------+---------+-------+
| NAME                                    | ENABLED | TIMED |
+-----------------------------------------+---------+-------+
| wait/synch/mutex/sql/PAGE::lock         | NO      | NO    |
| wait/synch/mutex/sql/TC_LOG_MMAP::LOCK_sync | NO  | NO    |
...
| wait/io/file/innodb/innodb_data_file    | YES     | YES   |
| wait/io/file/innodb/innodb_log_file     | YES     | YES   |
...
+-----------------------------------------+---------+-------+
```

Any of the tables within the Performance Schema whose names are prefixed with `setup_` can be used to update the configuration dynamically using a regular `UPDATE` statement. For example, use the following statement to enable all instruments dynamically:

```
UPDATE performance_schema.setup_instruments
   SET enabled = 'YES', timed = 'YES';
```

Once this is set, you can use the following statement to disable all mutex instruments:

```
UPDATE performance_schema.setup_instruments
   SET enabled = 'NO', timed = 'NO'
 WHERE name LIKE 'wait/synch/mutex/%';
```

The `performance_schema.setup_consumers` table can be modified in the same way for consumers. A good base configuration to use in your options file during development is to enable everything on server start:

```
[mysqld]

performance_schema
performance_schema_instrument = '%=on'
performance_schema_consumer_events_stages_current = ON
performance_schema_consumer_events_stages_history = ON
performance_schema_consumer_events_stages_history_long = ON
performance_schema_consumer_events_statements_history = ON
performance_schema_consumer_events_statements_history_long = ON
performance_schema_consumer_events_waits_current = ON
performance_schema_consumer_events_waits_history = ON
performance_schema_consumer_events_waits_history_long = ON
```

Finding High-Overhead Statements

With a development instance set up to trace everything, you can run tests against your applications to acquire performance metrics and focus tuning efforts. Generally, this starts with trying to pinpoint the statements with the most overhead.

In the past, this task was done by turning on the Slow Query Log, then post-processing this log file with a tool such as mysqldumpslow to aggregate the statistics. As part of the new statement instrumentation within MySQL 5.6, a handy aggregate table has been implemented: `performance_schema.events_statements_summary_by_digest`.

With the aggregate table, you don't have to access the file system to look at the log and you can analyze the statistics easily with simple SQL.

This table has the following structure:

```
mysql> DESC events_statements_summary_by_digest;
+-----------------------------+---------------------+------+-----+---------------------+-------+
| Field                       | Type                | Null | Key | Default             | Extra |
+-----------------------------+---------------------+------+-----+---------------------+-------+
| SCHEMA_NAME                 | varchar(64)         | YES  |     | NULL                |       |
| DIGEST                      | varchar(32)         | YES  |     | NULL                |       |
| DIGEST_TEXT                 | longtext            | YES  |     | NULL                |       |
| COUNT_STAR                  | bigint(20) unsigned | NO   |     | NULL                |       |
| SUM_TIMER_WAIT              | bigint(20) unsigned | NO   |     | NULL                |       |
| MIN_TIMER_WAIT              | bigint(20) unsigned | NO   |     | NULL                |       |
| AVG_TIMER_WAIT              | bigint(20) unsigned | NO   |     | NULL                |       |
| MAX_TIMER_WAIT              | bigint(20) unsigned | NO   |     | NULL                |       |
| SUM_LOCK_TIME               | bigint(20) unsigned | NO   |     | NULL                |       |
| SUM_ERRORS                  | bigint(20) unsigned | NO   |     | NULL                |       |
| SUM_WARNINGS                | bigint(20) unsigned | NO   |     | NULL                |       |
| SUM_ROWS_AFFECTED           | bigint(20) unsigned | NO   |     | NULL                |       |
| SUM_ROWS_SENT               | bigint(20) unsigned | NO   |     | NULL                |       |
| SUM_ROWS_EXAMINED           | bigint(20) unsigned | NO   |     | NULL                |       |
| SUM_CREATED_TMP_DISK_TABLES | bigint(20) unsigned | NO   |     | NULL                |       |
| SUM_CREATED_TMP_TABLES      | bigint(20) unsigned | NO   |     | NULL                |       |
| SUM_SELECT_FULL_JOIN        | bigint(20) unsigned | NO   |     | NULL                |       |
| SUM_SELECT_FULL_RANGE_JOIN  | bigint(20) unsigned | NO   |     | NULL                |       |
| SUM_SELECT_RANGE            | bigint(20) unsigned | NO   |     | NULL                |       |
| SUM_SELECT_RANGE_CHECK      | bigint(20) unsigned | NO   |     | NULL                |       |
| SUM_SELECT_SCAN             | bigint(20) unsigned | NO   |     | NULL                |       |
| SUM_SORT_MERGE_PASSES       | bigint(20) unsigned | NO   |     | NULL                |       |
| SUM_SORT_RANGE              | bigint(20) unsigned | NO   |     | NULL                |       |
| SUM_SORT_ROWS               | bigint(20) unsigned | NO   |     | NULL                |       |
| SUM_SORT_SCAN               | bigint(20) unsigned | NO   |     | NULL                |       |
| SUM_NO_INDEX_USED           | bigint(20) unsigned | NO   |     | NULL                |       |
| SUM_NO_GOOD_INDEX_USED      | bigint(20) unsigned | NO   |     | NULL                |       |
| FIRST_SEEN                  | timestamp           | NO   |     | 0000-00-00 00:00:00 |       |
| LAST_SEEN                   | timestamp           | NO   |     | 0000-00-00 00:00:00 |       |
+-----------------------------+---------------------+------+-----+---------------------+-------+
```

Within this table, Performance Schema tracks and aggregates statistics by normalizing the statements, and assigning that normalized statement a "Digest," which is an MD5 hash of the normalized form of the statement. These digests are available in the `DIGEST` and `DIGEST_TEXT` columns. The table is aggregated by both the `SCHEMA_NAME` and `DIGEST` columns, so that you can also drill into which schemas are generating the most overhead, even if you have an instance that has multiple schemas executing the same kinds of statements (such as a shared instance running multiple Wordpress installations, for example).

A normalized statement may have any of the following actions performed upon it:

- Stripping whitespace
- Stripping comments:
  Replacing literals (integer and string inputs) with a "?" placeholder. For example, `SELECT foo, bar FROM foobar WHERE foo = 100` becomes `SELECT foo, bar FROM foobar WHERE foo = ?`
- Lists of values are folded:
  - Folding lists of `IN` values. For example, `SELECT foo FROM foobar WHERE bar IN (1, 2, 3)` becomes `SELECT foo FROM foobar WHERE bar IN (...)`
  - Folding multirow `INSERT`s. For example, `INSERT INTO t1 VALUES (1, 2, 3), (4, 5, 6)` becomes `INSERT INTO t1 VALUES (...) /* , ... */`
  - Listing of values in the `SELECT` list. For example, `SELECT 1, 2, 3, foo, bar FROM foobar` becomes `SELECT ?, ... , foo, bar FROM foobar`
  - Long normalized statements are truncated, with "..." added at the end. "Long" is currently defined as 1024 bytes; however, note that this is on the normalized statement, so large string values are not a concern

Finding the statements that have the highest latency in the server is simple: just specify a criteria against one of the `*_TIMER_WAIT` columns. For instance, to find the top five statements that have the highest latency overall (and are probably prime candidates for tuning):

```
SELECT *
FROM performance_schema.events_statements_summary_by_digest
ORDER BY SUM_TIMER_WAIT DESC LIMIT 5\G

******************** 1. row ***********************
SCHEMA_NAME: mysql
     DIGEST: a3efbb2ddf943115f3cb50b810910160
     DIGEST_TEXT: SELECT * FROM ( SELECT username AS `user`
...
COUNT_STAR: 27757
SUM_TIMER_WAIT: 3753097139331384
MIN_TIMER_WAIT: 30471498720
AVG_TIMER_WAIT: 135212475084
MAX_TIMER_WAIT: 18453317809272
SUM_LOCK_TIME: 33214892000000
SUM_ERRORS: 0
SUM_WARNINGS: 0
SUM_ROWS_AFFECTED: 0
SUM_ROWS_SENT: 5987687
SUM_ROWS_EXAMINED: 263772439
SUM_CREATED_TMP_DISK_TABLES: 83271
SUM_CREATED_TMP_TABLES: 194299
SUM_SELECT_FULL_JOIN: 83271
SUM_SELECT_FULL_RANGE_JOIN: 0
SUM_SELECT_RANGE: 0
SUM_SELECT_RANGE_CHECK: 0
SUM_SELECT_SCAN: 166542
SUM_SORT_MERGE_PASSES: 0
SUM_SORT_RANGE: 0
SUM_SORT_ROWS: 685834
SUM_SORT_SCAN: 25134
SUM_NO_INDEX_USED: 27757
SUM_NO_GOOD_INDEX_USED: 0
         FIRST_SEEN: 2012-12-06 21:19:50
         LAST_SEEN: 2012-12-31 11:20:28
```

(The `DIGEST_TEXT` column has been truncated for brevity.) There are several things of interest in this raw data:

- The statement is scanning many more rows than are actually sent back to the client (`SUM_ROWS_SENT` is much less than `SUM_ROWS_EXAMINED`, on average the statement is scanning some 9,500 rows to return around 215 rows per execution). This is likely due to the fact that no index is ever being used to satisfy the query (`SUM_NO_INDEX_USED == COUNT_STAR`).
- The statement is creating a lot of temporary tables (an average of seven per execution), and more than 40% of those are on disk averaging three per execution (`SUM_CREATED_TMP_TABLES` vs. `SUM_CREATED_TMP_DISK_TABLES`)
- Interestingly, we seem to be doing joining against the temporary tables that are created on disk, and that's not using an index (`SUM_SELECT_FULL_JOIN == SUM_CREATED_TMP_DISK_TABLES`)

This statement is certainly worth tuning. First, try and find the right combination of indexes within your schema to satisfy the query, and also to see if something can be done about the excessive temporary table use — first by seeing if it can be created in memory only (assuming it has no long text columns and so forth). If not, try to make it perform better, perhaps by creating the temporary table up front with an index defined.

There are items that can be identified with this table as well:

- Finding statements that cause errors or warnings: `SUM_ERRORS > 0 OR SUM_WARNINGS > 0`
- Finding statements that always create temporary tables on disk: `SUM_CREATED_TMP_TABLES = SUM_CREATED_TMP_DISK_TABLES`
- Finding new statements that have turned up in your applications: `FIRST_SEEN > (NOW() - INTERVAL 2 DAY)`

Locating Where Statements Have High Latency

Once you have identified a statement to tune, you can infer where problems might lie; however, a more scientific approach would be to find out exactly where the time is being spent within the statement's execution window. This is what the new Stages instrumentation is for. Stages record the time that a connection spends in the various thread states (http://is.gd/RYC2S1) that you see within the output of something like `SHOW PROCESSLIST` (http://is.gd/IGPnUd).

In the past, you might have used `SHOW PROFILE` (http://is.gd/jP7CCj) to get this kind of data. If so, this approach should be fairly familiar to you. The new Stages instrumentation within Performance Schema in MySQL 5.6 is intended to eventually replace `SHOW PROFILE` (which has been deprecated), and it has several advantages:

- You can now get stage/state information for all connections, not just your current connection
- Stages are linked to statements, and linked further to underlying waits (more on this shortly)
- Stages are exposed in regular tables, giving you a SQL interface to look at the data
- Stage instruments are aggregated in multiple ways: globally, by thread, by user, by host, by account (user@host)

When profiling a specific statement, you are not interested in the aggregated information. Rather, you need to drill down into the raw data that is linked to a specific statement. On a busy system, this data can be particularly volatile, however, so you need to take this into account and set up your environment appropriately.

One of the easy ways to focus your profiling efforts is to turn off instrumentation for all threads apart from the one that you are examining. This will allow you to look at the history tables and see only data that is relevant to your tuning efforts, while also helping to ensure that no other activity on the instance ages out the data that you are trying to analyze.

This is where the `performance_schema.threads` table can help (http://is.gd/70NTVK). This table exposes the same data as the output of `SHOW PROCESSLIST`, and also provides information on all background threads:

```
SELECT * FROM performance_schema.threads\G
       ...
       *************************** 19. row ***************************
               THREAD_ID: 20
                    NAME: thread/sql/one_connection
                    TYPE: FOREGROUND
          PROCESSLIST_ID: 1
        PROCESSLIST_USER: root
        PROCESSLIST_HOST: localhost
          PROCESSLIST_DB: performance_schema
     PROCESSLIST_COMMAND: Query
        PROCESSLIST_TIME: 0
       PROCESSLIST_STATE: Sending data
        PROCESSLIST_INFO: SELECT * FROM performance_schema.threads
        PARENT_THREAD_ID: 1
                    ROLE: NULL
            INSTRUMENTED: YES
```

However, the table also has an `INSTRUMENTED` column that, like the `setup_*` tables, can be updated at runtime to turn on and off instrumentation by thread. To disable all threads before running your tests:

```
UPDATE performance_schema.threads SET instrumented = 'NO';
```

New connections will still be instrumented (unless you also update the `setup_actors` table appropriately (http://is.gd/YEC2tY), by removing the default row). So, make sure you're not running any other application tests against your development instance, then make a new connection to run the specific statement(s) that you want to tune.

Stages do not have a `DIGEST` column from which to infer a statement, but they are linked to statements using event nesting. Each of the history tables has an `EVENT_ID` column for the current event, and a `NESTING_EVENT_ID` column, which is the event ID of its parent event.

With these you can `JOIN` the `events_statements_history_long` table (http://is.gd/I6LWrH) to the `events_stages_history_long` (http://is.gd/f5INkx). If you have just executed one statement to profile, you can do this easily with a derived table:

```
SELECT event_name, timer_wait/1000000000 wait_ms
      FROM events_stages_history_long AS stages
            JOIN (SELECT event_id
                        FROM events_statements_history_long
                        ORDER BY event_id DESC limit 1) AS statements
        ON stages.nesting_event_id = statements.event_id
            ORDER BY stages.event_id;

+--------------------------------+---------+
| event_name                     | wait_ms |
+--------------------------------+---------+
| stage/sql/init                 |  0.0755 |
| stage/sql/checking permissions |  0.0020 |
| stage/sql/checking permissions |  0.0066 |
| stage/sql/Opening tables       |  0.1513 |
| stage/sql/init                 |  0.0119 |
| stage/sql/System lock          |  0.0036 |
| stage/sql/optimizing           |  0.0013 |
| stage/sql/optimizing           |  0.0060 |
| stage/sql/statistics           |  0.0103 |
| stage/sql/preparing            |  0.0079 |
| stage/sql/Sorting result       |  0.0086 |
| stage/sql/executing            |  0.0007 |
| stage/sql/Sending data         |  0.0010 |
| stage/sql/Creating sort index  |  0.5062 |
| stage/sql/statistics           |  0.0079 |
| stage/sql/preparing            |  0.0030 |
| stage/sql/Sorting result       |  0.0007 |
| stage/sql/executing            |  0.0003 |
| stage/sql/Sending data         |  0.0040 |
| stage/sql/Creating sort index  |  0.0987 |
| stage/sql/end                  |  0.0013 |
| stage/sql/query end            |  0.0013 |
| stage/sql/closing tables       |  0.0003 |
| stage/sql/removing tmp table   |  0.0036 |
| stage/sql/freeing items        |  0.0374 |
| stage/sql/cleaning up          |  0.0083 |
+--------------------------------+---------+
```

Here, the longest period of waiting is during the "Creating sort index" stage, which is again related to creating a temporary table to satisfy a `SELECT` query. You can order this in any way you like (the example is chronologically), such as `ORDER BY timer_wait DESC`, for example, to make it more obvious which stages are the highest consumers of time.

"You can drill further into each of the wait events during each stage. This can be particularly telling if you have to do a lot of I/O"

If you have executed many queries, then you will want to look at these on a per-statement basis, finding each `event_id` from `events_statements_history_long`, and then selecting from the `events_stages_history_long` table using that ID to filter on the `NESTING_EVENT_ID` column.

But it doesn't stop here; you can drill further into each of the wait events during each stage. This can be particularly telling if you have to do a lot of I/O, for example.

To look at the entire history of the thread, you can use a statement like the following example. It will dump the entire history available in order. This may start out with just statements, and no relating

```
SELECT thread_id,
       CONCAT( CASE WHEN event_name LIKE 'stage%' THEN
                      CONCAT('  ', event_name)
                 |    WHEN event_name LIKE 'wait%' AND nesting_event_id IS NOT NULL THEN
                      CONCAT('    ', event_name)
                    ELSE IF(digest_text IS NOT NULL, SUBSTR(digest_text, 1, 64), event_name)
               END,
               ' (',ROUND(timer_wait/1000000, 2),'µ) ') event
  FROM (
       (SELECT thread_id, event_id, event_name,
               timer_wait, timer_start, nesting_event_id, digest_text
          FROM events_statements_history_long)
       UNION
       (SELECT thread_id, event_id, event_name,
               timer_wait, timer_start, nesting_event_id, NULL
          FROM events_stages_history_long)
       UNION
       (SELECT thread_id, event_id, event_name,
               timer_wait, timer_start, nesting_event_id, NULL
          FROM events_waits_history_long)
       ) events
 ORDER BY thread_id, event_id;
+-----------+-------------------------------------------------------------------------------+
| thread_id | event                                                                         |
+-----------+-------------------------------------------------------------------------------+
|   2782380 | DELETE FROM `env_inventory` . `Network_v4Addresses` WHERE `Netw (71.18µ)      |
|   2782380 | DELETE FROM `env_inventory` . `Network_v6Addresses` WHERE `Netw (69.19µ)      |
|   2782380 | COMMIT  (612.14µ)                                                             |
...
|   2782380 | INSERT INTO `env_inventory` . `Network_v4Addresses` ( `Network` (77.47µ)      |
|   2782380 |   stage/sql/end (0.66µ)                                                       |
|   2782380 |   stage/sql/query end (0.99µ)                                                 |
|   2782380 |   stage/sql/closing tables (1.99µ)                                            |
|   2782380 |   stage/sql/freeing items (21.85µ)                                            |
|   2782380 |   stage/sql/cleaning up (0.66µ)                                               |
|   2782380 | COMMIT  (903.82µ)                                                             |
|   2782380 |   stage/sql/init (860.78µ)                                                    |
|   2782380 |   stage/sql/query end (1.32µ)                                                 |
|   2782380 |   stage/sql/closing tables (0.99µ)                                            |
|   2782380 |   stage/sql/freeing items (39.40µ)                                            |
|   2782380 |   stage/sql/cleaning up (0.66µ)                                               |
...
|   2782380 | idle (40.72µ)                                                                 |
|   2782380 | INSERT INTO `env_inventory` . `Network_v4Addresses` ( `Network` (77.47µ)      |
|   2782380 |   stage/sql/init (15.89µ)                                                     |
|   2782380 |     wait/io/socket/sql/client_connection (1.41µ)                              |
|   2782380 |     wait/io/socket/sql/client_connection (1.00µ)                              |
|   2782380 |     wait/synch/mutex/sql/THD::LOCK_thd_data (0.04µ)                           |
|   2782380 |     wait/synch/mutex/sql/THD::LOCK_thd_data (0.03µ)                           |
|   2782380 |   stage/sql/checking permissions (1.32µ)                                      |
|   2782380 |     wait/synch/rwlock/sql/LOCK_grant (0.13µ)                                  |
|   2782380 |   stage/sql/Opening tables (4.30µ)                                            |
|   2782380 |     wait/synch/rwlock/sql/MDL_lock::rwlock (0.06µ)                            |
|   2782380 |     wait/synch/mutex/sql/LOCK_table_cache (0.06µ)                             |
|   2782380 |     wait/synch/mutex/sql/THD::LOCK_thd_data (0.03µ)                           |
|   2782380 |   stage/sql/init (3.97µ)                                                      |
|   2782380 |   stage/sql/System lock (1.32µ)                                               |
|   2782380 |     wait/lock/table/sql/handler (0.33µ)                                       |
|   2782380 |     wait/lock/table/sql/handler (0.29µ)                                       |
|   2782380 |     wait/synch/mutex/mysys/THR_LOCK::mutex (0.06µ)                            |
|   2782380 |   stage/sql/update (23.84µ)                                                   |
|   2782380 |     wait/io/table/sql/handler (21.77µ)                                        |
|   2782380 |     wait/synch/rwlock/innodb/dict_operation_lock (0.09µ)                      |
|   2782380 |     wait/synch/rwlock/innodb/btr_search_latch (0.06µ)                         |
|   2782380 |     wait/synch/mutex/innodb/lock_mutex (0.08µ)                                |
...
```

stage/wait events, depending on how many statements you have executed on the thread. It will give progressively more information as you track the thread history.

Take a look at the example to the left.

As you can imagine, there are many more options that can provide useful data for tuning the performance of MySQL. Many of those options are explained in greater detail in the links throughout this article. In all cases, if your work requires improving MySQL performance, you'll find the new Performance Schema to be a boon to your efforts.

— Mark Leith is a Senior Software Development Manager, MySQL Enterprise Tools Group.

Comment

Acquiring Big Data Using Apache Flume

Data analysis is only half the battle; getting the data into a Hadoop cluster is the first step in any Big Data deployment. Apache Flume uses an elegant design to make data loading easy and efficient.

By Dan McClary

No technology is more synonymous with Big Data than Apache Hadoop (http://is.gd/O6VZQE). Hadoop's distributed filesystem and compute framework make possible cost-effective, linearly scalable processing of petabytes of data. Unfortunately, there are few tutorials devoted to how to get big data into Hadoop in the first place.

Some data destined for Hadoop clusters surely comes from sporadic bulk loading processes, such as database and mainframe offloads and batched data dumps from legacy systems. But what has made data really big in recent years is that most new data is contained in high-throughput streams. Application logs, GPS tracking, social media updates, and digital sensors all constitute fast-moving streams begging for storage in the Hadoop Distributed File System (HDFS). As you might expect, several technologies have been developed to address the need for collection and transport of these high-throughput streams. Facebook's Scribe (https://github.com/facebook/scribe) and Apache/LinkedIn's Kafka (http://kafka.apache.org/) both offer solutions to the problem, but Apache Flume (http://flume.apache.org/) is rapidly becoming a de facto standard for directing data streams into Hadoop.

This article describes the basics of Apache Flume and illustrates how to quickly set up Flume agents for collecting fast-moving data streams and pushing the data into Hadoop's filesystem. By the time we're finished, you should be able to configure and launch a Flume agent and understand how multi-hop and fan-out flows are easily constructed from multiple agents.

Anatomy of a Flume Agent

Flume deploys as one or more agents, each contained within its own instance of the Java Virtual Machine (JVM). Agents consist of three pluggable components: sources, sinks, and channels. An agent must have

at least one of each in order to run. Sources collect incoming data as events. Sinks write events out, and channels provide a queue to connect the source and sink. (Figure 1.)

Sources

Put simply, Flume sources listen for and consume events. Events can range from newline-terminated strings in `stdout` to HTTP `POST`s and RPC calls — it all depends on what sources the agent is configured to use. Flume agents may have more than one source, but must have at least one. Sources require a name and a type; the type then dictates additional configuration parameters.

On consuming an event, Flume sources write the event to a channel. Importantly, sources write to their channels as transactions. By dealing in events and transactions, Flume agents maintain end-to-end flow reliability. Events are not dropped inside a Flume agent unless the channel is explicitly allowed to discard them due to a full queue.

Channels

Channels are the mechanism by which Flume agents transfer events from their sources to their sinks. Events written to the channel by a source are not removed from the channel until a sink removes that event in a transaction. This allows Flume sinks to retry writes in the event of a failure in the external repository (such as HDFS or an outgoing network connection). For example, if the network between a Flume agent and a Hadoop cluster goes down, the channel will keep all events queued until the sink can correctly write to the cluster and close its transactions with the channel.

Channels are typically of two types: in-memory queues and durable disk-backed queues. In-memory channels provide high throughput but no recovery if an agent fails. File or database-backed channels, on the other hand, are durable. They support full recovery and event replay in the case of agent failure.

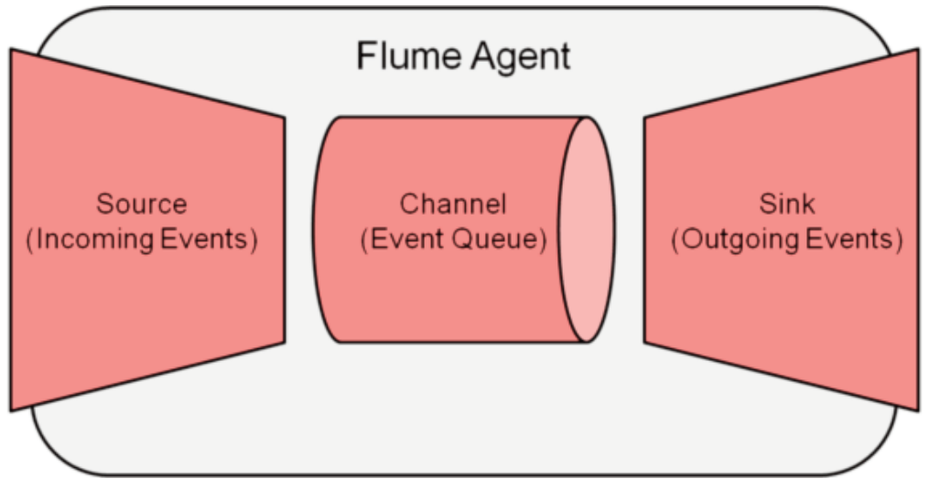


Figure 1: Flume Agents consist of sources, channels, and sinks.

Sinks

Sinks provide Flume agents pluggable output capability — if you need to write to a new type storage, just write a Java class that implements the necessary classes. Like sources, sinks correspond to a type of output: writes to HDFS or HBase, remote procedure calls to other agents, or any number of other external repositories. Sinks remove events from the channel in transactions and write them to output. Transactions close when the event is successfully written, ensuring that all events are committed to their final destination.

Setting up a Simple Agent for HDFS

A simple one-source, one-sink Flume agent can be configured with

just a single configuration file. In this example, I'll create a text file named `sample_agent.conf` — it looks a lot like a Java properties file. At the top of the file, I configure the agent's name and the names of its source, sink, and channel.

```
hdfs-agent.sources= netcat-collect
hdfs-agent.sinks = hdfs-write
hdfs-agent.channels= memory-channel
```

This defines an agent named `hdfs-agent` and the names of the sources, sinks, and channels; keep the name in mind, because we'll need it to start the agent. Multiple sources, sinks, and channels can be defined on these lines as a whitespace-delimited list of names. In this case, the source is named `netcat-collect`, the sink `hdfs-write`, and the channel is named `memory-channel`. The names are indicative of what I'm setting up: events collected via `netcat` will be written to HDFS and I will use a memory-only queue for transactions.

Next, I configure the source. I use a `netcat` source, as it provides a simple means of interactively testing the agent. A `netcat` source requires a type as well as an address and port to which it should bind. The `netcat` source will listen on localhost on port 11111; messages to `netcat` will be consumed by the source as events.

```
hdfs-agent.sources.netcat-collect.type = netcat
hdfs-agent.sources.netcat-collect.bind = 127.0.0.1
hdfs-agent.sources.netcat-collect.port = 11111
```

With the source defined, I'll configure the sink to write to HDFS. HDFS sinks support a number of options, but by default, the HDFS sink writes Hadoop `SequenceFiles`. In this example, I'll specify the sink write raw textfiles to HDFS so they can be easily inspected; I'll also set a roll interval, which forces Flume to commit writes to HDFS every 30 seconds. File rolls can be configured based on time, size, or a combination of the two. The file rolls are particularly important in environments for which HDFS does not support appending to files.

```
hdfs-agent.sinks.hdfs-write.type = hdfs
hdfs-agent.sinks.hdfs-write.hdfs.path =
     hdfs://namenode_address:8020/path/to/flume_test
hdfs-agent.sinks.hdfs-write.rollInterval = 30
hdfs-agent.sinks.hdfs-write.hdfs.writeFormat=Text
hdfs-agent.sinks.hdfs-write.hdfs.fileType=DataStream
```

Finally, I'll configure a memory-backed channel to transfer events from source to sink and connect them together. Keep in mind that if I exceed the channel capacity, Flume will drop events. If I need durability in a file or JDBC, channel should be used instead.

```
hdfs-agent.channels.memoryChannel.type = memory
hdfs-agent.channels.memoryChannel.capacity=10000
hdfs-agent.sources.netcat-collect.channels=memoryChannel
hdfs-agent.sinks.hdfs-write.channel=memoryChannel
```

With the configuration complete, I start the Flume agent from a terminal:

```
flume-ng agent -f /path/to/sample_agent.conf
               -n hdfs-agent
```

In the Flume agent's logs, I look for indication that the source, sink, and channel have successfully started. For example:

```
  INFO nodemanager.DefaultLogicalNodeManager:
    Starting Channel memoryChannel
  INFO instrumentation.MonitoredCounterGroup:
    Component type: CHANNEL, name: memoryChannel started
  INFO nodemanager.DefaultLogicalNodeManager:
    Starting Sink hdfs-write
  INFO nodemanager.DefaultLogicalNodeManager:
    Starting Source netcat-collect
INFO source.NetcatSource: Source starting
```

In a separate terminal, connect to the agent via `netcat` and enter a series of messages.

```
> nc localhost 11111
> testing
> 1
> 2
> 3
```

In the agent logs, an HDFS file will be created and committed every 30 seconds. If I print the contents of the files to standard `out` using HDFS `cat`, I'll find the messages from `netcat` are stored.

More Advanced Deployments

Regardless of source, direct writers to HDFS are too simple to be suitable for many deployments: Application servers may reside in the cloud while clusters are on-premise, many streams of data may need to be consolidated, or events may need to be filtered during transmission. Fortunately, Flume easily enables reliable multi-hop event transmission. Fan-in and fan-out patterns are readily supported via multiple sources and channel options. Additionally, Flume provides the notion of interceptors, which allow the decoration and filtering of events in flight.

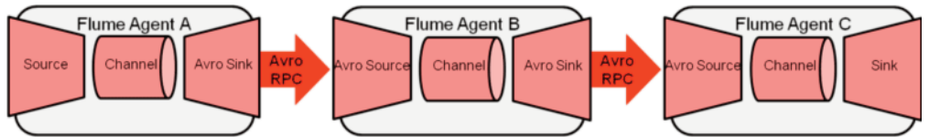


Figure 2: Multihop event flows are constructed using RPCs between Avro sources and sinks.

Multi-Hop Topologies

Flume provides multi-hop deployments via Apache Avro-serialized RPC calls. For a given hop, the sending agent implements an Avro sink directed to a host and port where the receiving agent is listening. The receiver implements an Avro source bound to the designated host-port combination. Reliability is ensured by Flume's transaction model. The sink on the sending agent does not close its transaction until receipt is acknowledged by the receiver. Similarly, the receiver does not acknowledge receipt until the incoming event has been committed to its channel.

```
#sender configuration
avro-agent.sinks= avro-sink
avro-agent.sinks.avro-sink.type=avro
avro-agent.sinks.avro-sink.host=remote.host.com
avro-agent.sinks.avro-sink.port=11111

#receiver configuration on remote.host.com
hdfs-agent.sources=avro-source
hdfs-agent.sources.avro-source.type=avro
hdfs-agent.sources.avro-source.bind=0.0.0.0
hdfs-agent.sources.avro-source.port=11111
hdfs-agent.sources.avro-source.channels=memoryChannel
```

Fan-In and Fan-Out

Fan-in is a common case for Flume agents. Agents may be run on many data collectors (such as application servers) in a large deployment, while only one or two writers to a remote Hadoop cluster are required to handle the total event throughput. In this case, the Flume topology is simple to configure. Each agent at a data collector implements the appropriate source and an Avro sink. All Avro sinks point to the host and port of the Flume agent charged with writing to the Hadoop cluster. The agent at the Hadoop cluster simply configures an Avro source on the designated host and port. Incoming events are consolidated automatically and are written to the configured sink.

Fan-out topologies are enabled via Flume's source selectors. Selectors can be replicating — sending all events to multiple channels — or multiplexing. Multiplexed sources can be partitioned by mappings defined on events via interceptors. For example, a replicating selector may be appropriate when events need to be sent to HDFS and to flat log files or a database. Multiplexed selectors are useful when different mappings should be directed to different writers; for example, data destined for partitioned Hive tables may be best handled via multiplexing.

```
hdfs-agent.channels=mchannel1 mchannel2
hdfs-agent.sources.netcat-collect.selector.type = replicating
hdfs-agent.sources.r1.channels = mchannel1 mchannel2
```

Interceptors

Flume provides a robust system of interceptors for in-flight modification of events. Some interceptors serve to decorate data with metadata useful in multiplexing the data or processing it after it has been written to the sink. Common decorations include timestamps, hostnames, and static headers. It's a great way to keep track of when your data arrived and from where it came.

More interestingly, interceptors can be used to selectively filter or decorate events. The Regex Filtering Interceptor allows events to be dropped if they match the provided regular expression. Similarly, the Regex Extractor Interceptor decorates event headers according to a regular expression. This is useful if incoming events require multiplexing, but static definitions are too inflexible.

```
hdfs-agent.sources.netcat-collect.interceptors = filt_int
hdfs-agent.sources.netcat-
  collect.interceptors.filt_int.type=regex_filter
hdfs-agent.sources.netcat-
  collect.interceptors.filt_int.regex=^echo.*
hdfs-agent.sources.netcat-
  collect.interceptors.filt_int.excludeEvents=true
```

Conclusion

There are lots of ways to acquire Big Data with which to fill up a Hadoop cluster, but many of those data sources arrive as fast-moving streams of data. Fortunately, the Hadoop ecosystem contains a component specifically designed for transporting and writing these streams: Apache Flume. Flume provides a robust, self-contained application which ensures reliable transportation of streaming data. Flume agents are easy to configure, requiring only a property file and an agent name. Moreover, Flume's simple source-channel-sink design allows us to build complicated flows using only a set of Flume agents. So, while we don't often address the process of acquiring Big Data for our Hadoop clusters, doing so is as easy and fun as taking a log ride.

— Dan McClary is the Principal Product Manager of Big Data at Oracle.

Comment

From the Vault

Kernels, Contexts, Threads, and Extensible Database Architecture

If you are building a new database management system or simply want to write better database applications, you should be aware of the extensible architecture of operating systems and database managers.

By **Ken North**

Servers with multiple processor cores, NewSQL, cloud computing, and advanced analytics have put a spotlight on extensible database architecture. It has long been possible to use a plugin to add functionality to a database, such as a new type, just as we can use add-ins to extend a Web browser. But today we see plugins gaining traction for a variety of purposes. If you are building a new database management system (DBMS) or simply want to write better database applications, you should be aware of the extensible architecture of operating systems and database managers.

To undertake a study of kernels, microkernels, and database architecture, a good departure point is found in some features of modern processors.

Processor Operating Modes

Processors have long supported multiple operating modes to segregate system services and application code at execution time. The purpose of these designs was to afford a level of protection for the routines that comprised the system software. The idea is that you don't want faulty application code to overwrite a part of memory that's used by the operating system (OS); overwriting memory or data can cause a crash.

The venerable IBM System 360 mainframe of the 1960s was a processor that offered a separation between System State and User State. The former was intended for code that executed to provide OS services when a program executed a supervisor call (SVC). The latter was how applications normally operated.

IBM OS/360 was designed during an era of expensive main memory, when machines with 512K of memory were the top of the line. To keep the operating system to a manageable size, IBM developed an extensible OS architecture. It supported the execution of Transient SVCs: small (2K) special-purpose modules that executed in a transient program area of memory.

The popular PDP 11 series minicomputer offered multiple processor modes (Supervisor, User, and Kernel mode); it also offered separate memory spaces for instructions and data. The Intel Xeon processor (in fact, all Intel processors since the 386) also has three processor modes. A program can operate in Real Mode, Protected Mode, or SMM Mode. These processor architectures enable us to write programs that change modes for critical operations, such as invoking an interrupt service routine that executes in kernel mode. A mode transition goes hand-in-hand with a context switch that preserves and restores the state of the CPU. Table 1 shows a breakdown of processor modes.

Memory-managed computers and multiple processor modes added the term "context switch" to the programming lexicon. In a protected memory environment, "context" refers to a virtual address space containing executable code. Various activities, such as invoking a kernel mode function or preempting a task, will initiate a context switch for the purpose of storing state information for the old executing task and retrieving state information for the new code.

The context switch has been integral to modern systems and is essential for multitasking operations. With today's processors and operating systems, the term takes on a broader meaning. You can have register, task, thread, or process context switches. Some processors now provide hardware support for context switching by providing a data segment for task state information.

Computer	Processor Modes (Operating States)
IBM System 360	System, User
Digital PDP 11	Supervisor, User, Kernel
Intel Xeon	Real, Protected, SMM

Table 1.

Evolution of Kernels

The kernel and microkernel form the code that's at the heart of an operating system. For processors with multiple operating modes or rings, the kernel or microkernel executes at the most privileged state (supervisor or kernel mode).

The microkernel was developed in part because monolithic OS kernels were becoming large and unwieldy, making them difficult to maintain. It seemed practical to implement support for new filesystems, protocol stacks, and device drivers in code that was outside the kernel and kernel address space. The microkernel enabled OS architects to develop privileged servers — such as networking servers, filesystem servers, and display servers — to deliver essential services without being built into the kernel. It's possible to start and stop these servers without rebooting, which makes it easier for development and

testing. If a server crashes, it does not corrupt data in the kernel address space.

There have been divergent schools of thought about which is the preferred architecture, the microkernel or monolithic kernel. Dr. Andrew S. Tanenbaum of MINIX fame (http://is.gd/iMiYfQ) favors the microkernel architecture, whereas Linus Torvalds stated his preference for the monolithic kernel architecture in a 2006 "Hybrid kernel, not NT" forum post (http://is.gd/GjlKp1).

"The dichotomy of opinion among OS developers also extends to the hypervisor and virtualization community."

Torvalds felt the separation of address spaces, making it impossible to share data structures, introduced too much complexity:

"Microkernels are much harder to write and maintain exactly because of this issue."

The dichotomy of opinion among OS developers also extends to the hypervisor and virtualization community. Xen and VMWare are examples of the microkernel architecture, whereas MokaFive BareMetal is based on the monolithic model.

SQL Database Servers

Prior to the famous Tanenbaum-Torvalds debate (http://is.gd/slP2Cs), the SQL database community had already been weighing monolithic database architecture against client-server database processing. The latter moved database processing off shared, centralized mainframes to distributed, dedicated servers. DBMS architects also came to favor a separation between the database kernel and services, plug-ins, or extenders.

This was noticeable over time as we saw SQL platforms evolve in a Swiss Army Knife manner. Instead of adding replication, for example, to the core database engine, it made more sense to write a replication server.

When you install a high-end SQL product, it uses a layered architecture that brings up a number of servers at startup time. The layers of services in a DBMS can include storage management, network access, lock management, replication, security, query, cache management, memory management, OLAP services, and so on.

For a product such as Oracle Database, the kernel is the core of the server process. Network communications map to a layer known as the Transparent Network Substrate (TNS), which was designed to support heterogeneous connectivity. An Oracle server runs a TNS listener for processing database requests using client-server protocols. Listeners are linked to end points (port numbers) for HTTP, FTP, and XML DB requests. The listener forwards requests to either a shared server or a dedicated server process. The kernel uses background processes, such as the Process Monitor, System Monitor, Log Writer, and Database Writer. It also uses slave processes working in the background, such as I/O Slave Processes and Parallel Query Slaves.

One benefit of the Oracle kernel architecture is being able to provide a top-level trace. For debugging and tuning queries, it helps you see what's happening during execution of a query, from the kernel down to other processes.

Threads and Processes

The Oracle Database architecture for threads and processes differs depending on whether Oracle is running on a UNIX or Windows platform. On UNIX machines, Oracle Database uses a process to implement each background task. Each connection to the database results in the operating system spawning a process for that session. Client processes, such as sqlplus, connect to the TNS listener (`tnslsnr`), which does a `fork()` and then `exec()` in the database kernel program in `$ORACLE_HOME/bin/oracle`. The newly created process is known as a server process or shadow process.

On a Windows machine, there is a corresponding process (.exe) for each Oracle Database instance or SID. Each background task is a thread inside a single process. All background, server, and client processes are threads of a master Oracle Database process and all threads share resources.

For Sybase Adaptive Server Everywhere (ASE), connections to the database equate to tasks that are each assigned an engine. Prior to version 15.7, Sybase ASE executed each engine in a separate process. Starting with ASE 15.7, Sybase has offered a threaded kernel with each engine being a thread of a process and the engines communicating via shared memory. Engines run only user tasks, not I/O. Sybase has always used the thread-based model for ASE running on Windows platforms.

Continual Evolution of Database Kernels

SQL platforms have gained a large share of the database market over the past three decades. The instinct for wanting to build a better mousetrap is endemic in the computing and software community. It's no surprise that we've seen the development of new architectures and new kernels for SQL, NoSQL, NewSQL and in-memory database platforms.

The original code base for MySQL (http://www.mysql.com/) dates back more than 15 years. Several factors (modern CPU architectures, the growth of cloud computing, and a wide following for Web applications) contributed to a desire to refactor MySQL as a lightweight SQL DBMS.

Drizzle (http://www.drizzle.org/) is a leaner version of MySQL built using a microkernel architecture and plugins. To build Drizzle, developers moved code from the MySQL kernel into plugins. MySQL found success with pluggable storage engines, such as InnoDB. Drizzle extends plugins to storage engines, authentication, replication, a function engine, string functions, catalog functions, schema dictionary, IPV6 functions, math functions, a multithread scheduler, and other features. When it starts up, Drizzle loads about four dozen default plugins.

Another interesting development is Monet (http://www.monetdb.org/), a main memory database kernel. Monet has roots in Troll, a relational kernel developed about three decades ago. MonetDB version 4 became an open source project in 2004. MonetDB is an implementation of a column-store database kernel, a departure for traditional SQL databases that are row oriented. Although column stores are traditionally associated with analytics and business intelligence, MonetDB is capable of supporting ACID transactions. MonetDB has an algebraic database kernel that makes it versatile enough to support multiple query languages, including SQL and XQuery. It's based on a layered architecture that provides multiple interfaces for plugging in extensions. Support for math functions, strings, and temporal data types is provided by

linked-in libraries. MonetDB is the subject of ongoing research and a recent interesting development is DataCell (http://is.gd/96gDc1), a project that provides sensor stream-processing capabilities built over MonetDB.

Another innovation has come from McObject, maker of the eXtremeDB embedded database product. McObject offers eXtremeDB-KM (http://www.mcobject.com/kernel_mode_database), a version that deploys in kernel mode to operate in the same address space as the operating system. This provides direct access of data structures to kernel processes, eliminating the overhead of context switches for applications requiring extreme performance.

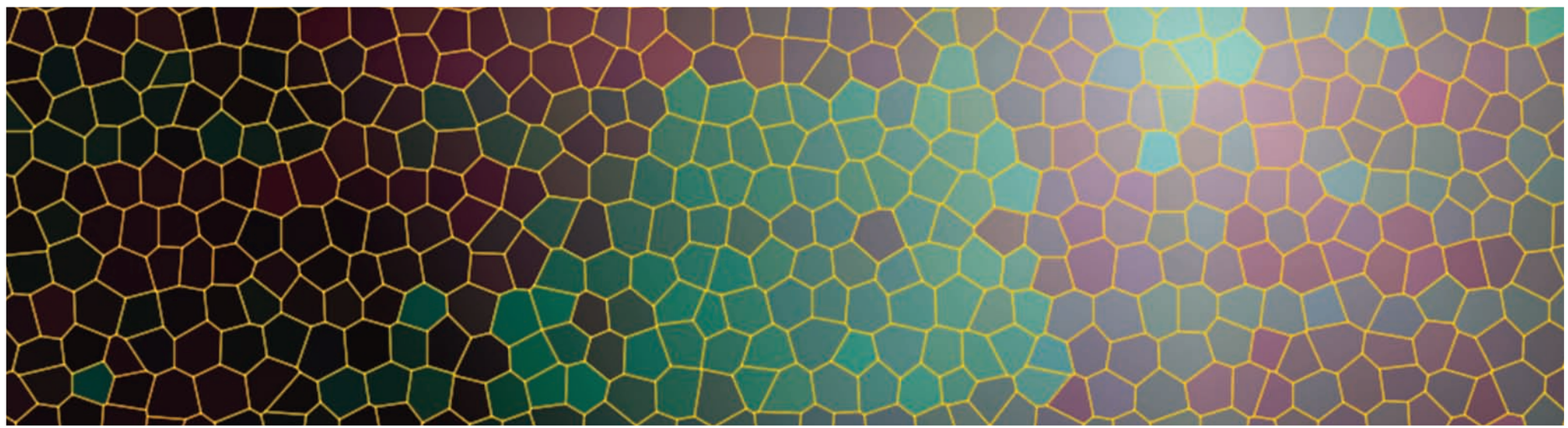
Database Plugins

The extensible database manager, with an architecture that supports plugins, gives developers the ability to adapt a database to an applications' requirements. We can write pluggable indexing and storage engines, for example. We can also write plugins that extend the SQL language, but that is a subject for a future article.

Aside from Drizzle, we've also seen several NewSQL platforms (http://newsql.sourceforge.net/) that represent a refactoring of the relational database. Plugins and refactoring provide a reminder that the database is not an example of static technology.

— Ken North is a well-known expert in database technologies. He regularly contributes to Dr. Dobb's *on the subjects of databases and software, including trends and techniques.*

Comment



This Month on DrDobbs.com

Items of special interest posted on www.drdobbs.com over the past month that you may have missed

THE PRAGMATIC SIDE OF USING BIG DATA

A successful big data project is far less about correlating disparate data structures and far more about articulating a testable hypothesis, designing that test, and evaluating the results.

http://www.drdobbs.com/240152350

THE CLOJURE PHILOSOPHY

Simplicity, freedom to focus, empowerment, consistency, and clarity: Nearly every element of the Clojure programming language is designed to promote these goals.

http://www.drdobbs.com/240150710

MICROSOFT TYPESCRIPT: THE LAY OF THE LAND

By creating a language that adds types to JavaScript, Microsoft made it easier to write complex Web apps, do compile-time syntax checking, and get better coding support in Visual Studio.

http://www.drdobbs.com/240153025

MAKING APIS ATTRACTIVE TO DEVELOPERS

Some simple, but important, steps make it easy for developers to choose your API for their apps.

http://www.drdobbs.com/240153548

SOMETIMES, MAKING A PROGRAM CLEARER MAKES IT FASTER

We can "optimize" our code by removing requests for operations that our data structures do not really need to support.

http://www.drdobbs.com/240154079

JAVA SE 8 SCHEDULE UPDATE

Yes, the release of Java SE 8 (JDK8) has slipped again. But it's for good reason: platform-wide security enhancements.

http://www.drdobbs.com/240153268

CAN GOOGLE'S DART HIT ITS TARGET?

How can the fast, boring language from Google be a viable solution?

http://www.drdobbs.com/240153181

IN THIS ISSUE

UBM TECH
Paul Miller, CEO
Kathy Astromoff, CEO, Electronics
Robert Faletra, CEO, Channel
Kelley Damore, Chief Community Officer
Marco Pardi, President, Business Technology Events
David Berlind, Chief Content Officer
John Dennehy, Chief Financial Officer
David Michael, Chief Information Officer
Martha Schwartz, Chief Sales Officer, Business Technology Media
Simon Carless, EVP, Game & App Development and Black Hat
Lenny Heymann, EVP, New Markets
Angela Scalpello, SVP, People & Culture

UNITED BUSINESS MEDIA LLC
Pat Nohilly Sr. VP, Strategic Development and Business Administration
Marie Myers Sr. VP, Manufacturing

INFORMATIONWEEK VIDEO
informationweek.com/tv
Fritz Nelson Executive Producer
fritz.nelson@ubm.com

INFORMATIONWEEK BUSINESS TECHNOLOGY NETWORK

DarkReading.com
Security
Tim Wilson, Site Editor
tim.wilson@ubm.com

IntelligentEnterprise.com
App Architecture
Doug Henschen, Editor in Chief
doug.henschen@ubm.com

NetworkComputing.com
Networking , Communications, and Storage
Andrew Conry-Murray, Editor
andrew.murray@ubm.com

PlugIntoTheCloud.com
Cloud Computing
Benjamin Tomkins, Site Editor
benjamin.tomkins@ubm.com

Byte.com
Larry Seltzer
Editorial Director
larry.seltzer@ubm.com

Dr. Dobb's
Good Stuff for Serious Developers
Andrew Binstock
Editor in Chief
andrew.binstock@ubm.com