Contents

amcheck/amcheck_next: functions for verifying PostgreSQL relation integrity

Current version: 1.4 (`amcheck_next` extension/SQL version: 2)

Author: Peter Geoghegan `<pg@bowt.ie>`

Supported versions: PostgreSQL 9.4 - PostgreSQL 10

Note that Microsoft Windows is supported, but only on point releases that have the necessary workaround for various restrictions on dynamic linking that only exist on that platform. The minimum supported point releases are 9.4.16, 9.5.11, 9.6.7, and 10.2.

Overview

The `amcheck` module provides functions that allow you to verify the logical consistency of the structure of PostgreSQL indexes. If the structure appears to be valid, no error is raised. Currently, only B-Tree indexes are supported, although since in practice the majority of PostgreSQL indexes are B-Tree indexes, `amcheck` is likely to be effective as a general corruption smoke-test in production PostgreSQL installations.

See Using amcheck effectively for information about the kinds of real-world problems `amcheck` is intended to detect.

Project background

`amcheck` is a contrib extension module that originally appeared in PostgreSQL 10. This externally maintained version of the extension, which is formally named `amcheck_next` to avoid conflicts with `contrib/amcheck`, provides the same functionality to earlier versions of PostgreSQL. `amcheck_next` also exists to provide additional verification checks that do not yet appear in stable PostgreSQL `contrib/amcheck` releases.

It is safe (though generally not useful) to install `amcheck_next` alongside `contrib/amcheck`.

Invariants

`amcheck` provides functions that specifically verify various *invariants* in the structure of the representation of particular indexes. The correctness of the access method functions behind index scans and other important operations relies on these invariants always holding. For example, certain functions verify, among other things, that all B-Tree pages have items in "logical", sorted order (e.g., for B-Tree indexes on text, index tuples should be in collated lexical order). If that particular invariant somehow fails to hold, we can expect binary searches on the affected page to incorrectly guide index scans, resulting in wrong answers to SQL queries.

Verification is performed using the same procedures as those used by index scans themselves, which may be user-defined operator class code. For example, B-Tree index verification relies on comparisons made with one or more B-Tree support function 1 routines, much like B-Tree index scans rely on the routines to guide the scan to a point in the underlying table; see the PostgreSQL documentation on Index Access Methods and Operator Classes for details of operator class support functions.

Installation

Prebuilt packages

It is recommended that `amcheck` be installed using prebuilt packages where available.

Debian/Ubuntu The most recent `amcheck` release is available from the PostgreSQL Community APT repository. Setup instructions can be found in the APT section of the PostgreSQL Wiki.

Once the Community APT repository is set up, and PostgreSQL has itself been installed from a community package, installation of `amcheck` is generally a simple matter of installing the package that matches your PostgreSQL version:

```
sudo aptitude install postgresql-10-amcheck
```

Redhat/CentOS/SLES The most recent `amcheck` release is available from the PostgreSQL Community yum repository. Setup can be performed by following the PostgreSQL yum Howto.

Once the Community yum repository is set up, and PostgreSQL has itself been installed from a community package, installation of `amcheck` is generally a simple matter of installing the package that matches your PostgreSQL version:

```
sudo yum install amcheck_next10
```

Building from source

Building using PGXS (generic) The module can be built using the standard PGXS infrastructure. For this to work, you will need to have the `pg_config` program available in your $PATH.

If you are using a packaged PostgreSQL build and have `pg_config` available (and in your OS user's $PATH), the procedure is as follows:

```
tar xvzf amcheck-1.3.tar.gz
cd amcheck-1.3
make
make install
```

Note that just because `pg_config` is located in one user's $PATH does not necessarily make it so for the root user.

Building Debian/Ubuntu packages from source The Makefile also provides a target for building Debian packages. The target has a dependency on `debhelper`, `devscripts`, `postgresql-server-dev-all`, and the PostgreSQL source package itself (e.g. `postgresql-server-dev-9.4`).

The packages can be created and installed from the amcheck directory as follows:

```
sudo aptitude install debhelper devscripts postgresql-server-dev-all
make deb
sudo dpkg -i ./build/postgresql-9.4-amcheck_*.deb
```

Setting up PostgreSQL

Once the module is built and/or installed, it may be created as a PostgreSQL extension:

```
mydb=# CREATE EXTENSION amcheck_next;
```

`amcheck` functions may be used only by superusers.

Interface

The `amcheck_next` extension has a simple interface. `amcheck_next` consists of just a few functions that can be used for verification of a named B-Tree index. Note that currently, no function inspects the structure of the underlying heap representation (table).

`regclass` function arguments are used by `amcheck` to identify particular index relations. This allows `amcheck` to accept arguments using various SQL calling conventions:

```
-- Use string literal regclass input:
SELECT bt_index_check('pg_database_oid_index', true);
-- Use oid regclass input (both perform equivalent verification):
SELECT bt_index_check(2672, false);
SELECT bt_index_check(oid, false) FROM pg_class
WHERE relname = 'pg_database_oid_index';
```

See the PostgreSQL documentation on Object identifier types for more information.

`bt_index_check`

```
bt_index_check(index regclass, heapallindexed boolean DEFAULT false)
returns void
```

`bt_index_check` tests that its target, a B-Tree index, respects a variety of invariants. Example usage:

```
SELECT bt_index_check(index => c.oid, heapallindexed => i.indisunique),
        c.relname,
        c.relpages
FROM pg_index i
JOIN pg_opclass op ON i.indclass[0] = op.oid
JOIN pg_am am ON op.opcmethod = am.oid
JOIN pg_class c ON i.indexrelid = c.oid
JOIN pg_namespace n ON c.relnamespace = n.oid
WHERE am.amname = 'btree' AND n.nspname = 'pg_catalog'
-- Don't check temp tables, which may be from another session:
AND c.relpersistence != 't'
-- Function may throw an error when this is omitted:
AND c.relkind = 'i' AND i.indisready AND i.indisvalid
ORDER BY c.relpages DESC LIMIT 10;
```

```
 bt_index_check |            relname             | relpages
----------------+--------------------------------+----------
                | pg_depend_reference_index      |       43
                | pg_depend_depender_index       |       40
                | pg_proc_proname_args_nsp_index |       31
                | pg_description_o_c_o_index     |       21
                | pg_attribute_relid_attnam_index|       14
                | pg_proc_oid_index              |       10
                | pg_attribute_relid_attnum_index|        9
                | pg_amproc_fam_proc_index       |        5
                | pg_amop_opr_fam_index          |        5
                | pg_amop_fam_strat_index        |        5
```

This example shows a session that performs verification of catalog indexes. Verification of the presence of heap tuples as index tuples is requested for unique indexes only. Since no error is raised, all indexes tested appear to be logically consistent. Naturally, this query could easily be changed to call `bt_index_check` for every index in the database where verification is supported. An `AccessShareLock` is acquired on the target index and heap relation by `bt_index_check`. This lock mode is the same lock mode acquired on relations by simple `SELECT` statements.

`bt_index_check` does not verify invariants that span child/parent relationships, but will verify the presence of all heap tuples as index tuples within the index when `heapallindexed` is `true`. When a routine, lightweight test for corruption is required in a live production environment, using `bt_index_check` often provides the best trade-off between thoroughness of verification and limiting the impact on application performance and availability.

`bt_index_parent_check`

```
bt_index_parent_check(index regclass, heapallindexed boolean DEFAULT false)
returns void
```

`bt_index_parent_check` tests that its target, a B-Tree index, respects a variety of invariants. Optionally, when the `heapallindexed` argument is `true`, the function verifies the presence of all heap tuples that should be found within the index, and that there are no missing downlinks in the index structure. The checks performed by `bt_index_parent_check` are a superset of the checks performed by `bt_index_check` when called with the same options. `bt_index_parent_check` can be thought of as a more thorough variant of `bt_index_check`: unlike `bt_index_check`, `bt_index_parent_check` also checks invariants that span parent/child relationships. `bt_index_parent_check` follows the general convention of raising an error if it finds a logical inconsistency or other problem.

A `ShareLock` is required on the target index by `bt_index_parent_check` (a `ShareLock` is also acquired on the heap relation). These locks prevent concurrent data modification from `INSERT`, `UPDATE`, and `DELETE` commands. The locks also prevent the underlying relation from being concurrently processed by `VACUUM` (and other utility commands). Note that the function holds locks for as short a duration as possible, so there is no advantage to verifying each index individually in a series of transactions, unless long running queries happen to be of particular concern.

`bt_index_parent_check`'s additional verification is more likely to detect various pathological cases. These cases may involve an incorrectly implemented B-Tree operator class used by the index that is checked, or, hypothetically, undiscovered bugs in the underlying B-Tree index access method code. Note that `bt_index_parent_check` cannot be called when Hot Standby is enabled (i.e., on read-only physical replicas), unlike `bt_index_check`.

Optional `heapallindexed` verification

When the `heapallindexed` argument to verification functions is `true`, an additional phase of verification is performed against the table associated with the target index relation. This consists of a "dummy" `CREATE INDEX` operation, which checks for the presence of all would-be new index tuples against a temporary, in-memory summarizing structure (this is built when needed during the first, standard phase). The summarizing structure "fingerprints" every tuple found within the target index. The high level principle behind `heapallindexed` verification is that a new index that is equivalent to the existing, target index must only have entries that can be found in the existing structure.

The additional `heapallindexed` phase adds significant overhead: verification will typically take several times longer than it would with only the standard consistency checking of the target index's structure. However, verification will still take significantly less time than an actual `CREATE INDEX`. There is no change to the relation-level locks acquired when `heapallindexed` verification is performed. The summarizing structure is bound in size by `maintenance_work_mem`. In order to ensure that there is no more than a 2% probability of failure to detect the absence of any particular index tuple, approximately 2 bytes of memory are needed per index tuple. As less memory is made available per index tuple, the probability of missing an inconsistency increases. This is considered an acceptable trade-off, since it limits the overhead of verification very significantly, while only slightly reducing the probability of detecting a problem, especially for installations where verification is treated as a routine maintenance task.

With many databases, even the default `maintenance_work_mem` setting of `64MB` is sufficient to have less than a 2% probability of overlooking any single absent or corrupt tuple. This will be the case when there are no indexes with more than about 30 million distinct index tuples, regardless of the overall size of any index, the total number of indexes, or anything else. False positive candidate tuple membership tests within the summarizing structure occur at random, and are very unlikely to be the same for repeat verification operations. Furthermore, within a single verification operation, each missing or malformed index tuple independently has the same chance of being detected. If there is any inconsistency at all, it isn't particularly likely to be limited to a single tuple. All of these factors favor accepting a limited per operation per tuple probability of missing corruption, in order to enable performing more thorough index to heap verification more frequently (practical concerns about the overhead of verification are likely to limit the frequency of verification). In aggregate, the probability of detecting a hardware fault or software defect actually *increases* significantly with this strategy in most real world cases. Moreover, frequent verification allows problems to be caught earlier on average, which helps to limit the overall impact of corruption, and often simplifies root cause analysis.

Using amcheck effectively

Causes of corruption

`amcheck` can be effective at detecting various types of failure modes that data page checksums will always fail to catch. These include:

- Structural inconsistencies caused by incorrect operator class implementations.

This includes issues caused by the comparison rules of operating system collations changing. Comparisons of datums of a collatable type like `text` must be immutable (just as all comparisons used for B-Tree index scans must be immutable), which implies that operating system collation rules must never change.

Though rare, updates to operating system collation rules can cause these issues. More commonly, an inconsistency in the collation order between a master server and a standby server is implicated, possibly because the *major* operating system version in use is inconsistent. Such inconsistencies will generally only arise on standby servers, and so can generally only be detected on standby servers.

If a problem like this arises, it may not affect each individual index that is ordered using an affected collation, simply because *indexed* values might happen to have the same absolute ordering regardless of the behavioral inconsistency.

- Structural inconsistencies between indexes and the heap relations that are indexed (when `heapallindexed` verification is performed).

There is no cross-checking of indexes against their heap relation during normal operation. Symptoms of heap corruption can be very subtle.

- Corruption caused by hypothetical undiscovered bugs in the underlying PostgreSQL access method code, sort code, or transaction management code.

Automatic verification of the structural integrity of indexes plays a role in the general testing of new or proposed PostgreSQL features that could plausibly allow a logical inconsistency to be introduced. Verification of table structure and associated visibility and transaction status information plays a similar role. One obvious testing strategy is to call `amcheck` functions continuously when running the standard regression tests.

- Filesystem or storage subsystem faults where checksums happen to simply not be enabled.

Note that `amcheck` examines a page as represented in some shared memory buffer at the time of verification if there is only a shared buffer hit when accessing the block. Consequently, `amcheck` does not necessarily examine data read from the filesystem at the time of verification. Note that when checksums are enabled, `amcheck` may raise an error due to a checksum failure when a corrupt block is read into a buffer.

- Corruption caused by faulty RAM, and the broader memory subsystem and operating system.

PostgreSQL does not protect against correctable memory errors and it is assumed you will operate using RAM that uses industry standard Error Correcting Codes (ECC) or better protection. However, ECC memory is typically only immune to single-bit errors, and should not be assumed to provide *absolute* protection against failures that result in memory corruption.

When `heapallindexed` verification is performed, there is generally a greatly increased chance of detecting single-bit errors, since strict binary equality is tested, and the indexed attributes within the heap are tested.

Overhead

The overhead of calling `bt_index_check` for every index on a live production system is roughly comparable to the overhead of vacuuming; like `VACUUM`, verification uses a "buffer access strategy", which limits its impact on which pages are cached within `shared_buffers`. A major design goal of `amcheck` is to support routine verification of all indexes on busy production systems. Note that `heapallindexed` verification *significantly* increases the runtime of verification.

No `amcheck` routine will ever modify data, and so no pages will ever be "dirtied", which is not the case with `VACUUM`. On the other hand, `amcheck` may be required to verify a large number of indexes all at once, which is typically not a behavior that autovacuum exhibits. `amcheck` exhaustively accesses every page in each index verified. This behavior is useful in part because verification may detect a checksum failure, which may have previously gone undetected only because no process needed data from the corrupt page in question, including even an autovacuum worker process.

Note also that `bt_index_check` and `bt_index_parent_check` access the contents of indexes in "logical" order, which, in the worst case, implies that all I/O operations are performed at random positions on the filesystem. In contrast, `VACUUM` always removes dead index tuples from B-Tree indexes while accessing the contents of B-Tree indexes in sequential order.

Acting on information about corruption

No error concerning corruption raised by `amcheck` should ever be a false positive. `amcheck` raises errors in the event of conditions that, by definition, should never happen. It seems unlikely that there could ever be a useful *general* remediation to problems it detects.

In general, an explanation for the root cause of an invariant violation should be sought. `contrib/pageinspect` can play a useful role in diagnosing corruption that `amcheck` highlights. A `REINDEX` may or may not be effective in repairing corruption, depending on the exact details of how the corruption originated.

In general, `amcheck` can only prove the presence of corruption; it cannot prove its absence.