PostgreSQL System Architecture

Heikki Linnakangas / VMware Ltd

July 4th, 2014

What is Software Architecture?

`the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution. [IEEE 1471]`

Software Architecture document

- Not a single diagram
- Multiple diagrams presenting different viewpoints
- Process View
- Development View
- Functional View

Architecture vs. Design

Architecture = Design

just at a higher level

- The Software Architect decides which decisions are part of architecture, and which can be left to Design or Implementation.
- This presentation is based on my opinions.

Questions? Feel free to interrupt.

Part 0: Non-functional requirements

- Data integrity
- Performance
- Scalability
- Reliability
- Interoperability
- Portability
- Extensibility
- Maintainability (code readability)

Example: Extensibility

PostgreSQL has a highly extensible type system.

- Behavior of all datatypes is defined by operators and operator classes:

  ```
  CREATE OPERATOR name (
      PROCEDURE = function_name
      [, LEFTARG = left_type ] [, RIGHTARG = right_type ]
      [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
      [, RESTRICT = res_proc ] [, JOIN = join_proc ]
      [, HASHES ] [, MERGES ]
  )
  ```

- Including all built-in types

Example: Extensibility vs Performance

Built-in int4 $<$ operator:

```
Datum
int4lt(PG_FUNCTION_ARGS)
{
    int32       arg1 = PG_GETARG_INT32(0);
    int32       arg2 = PG_GETARG_INT32(1);

    PG_RETURN_BOOL(arg1 < arg2);
}
```

Example: Extensibility vs Performance

Extensibility makes e.g PostGIS possible.

It comes at a price:

- Cannot assume anything about how an operator behaves, except what's specified in the CREATE OPERATOR statement
- No exceptions for built-in types.
- No special case e.g. for sorting integers

Part 1: Process View

PostgreSQL consists of multiple process that communicate via shared memory:

- Postmaster
- One backend process for each connection
- Auxiliary processes
  - autovacuum launcher
  - autovacuum workers
  - background writer
  - startup process (= WAL recovery)
  - WAL writer
  - checkpointer
  - stats collector
  - logging collector
  - WAL archiver
  - custom background workers

Part 1: Process View

```
~$ ps ax | grep postgres
29794 pts/7   S    0:00 ~/pgsql.master/bin/postgres -D data
29799 ?       Ss   0:00 postgres: checkpointer process
29800 ?       Ss   0:00 postgres: writer process
29801 ?       Ss   0:00 postgres: wal writer process
29802 ?       Ss   0:00 postgres: autovacuum launcher proce
29803 ?       Ss   0:00 postgres: stats collector process
29826 ?       Ss   0:00 postgres: heikki postgres [local] i
```

Postmaster

- Listens on the TCP socket (port 5432) for connections
- forks new backends as needed
- Parent of all other processes
- listens for unexpected death of child processes
- initiates start / stop of the system

Postmaster

Reliability is very important:

- Does not touch shared memory
  - except a little bit
- No locking between postmaster and other processes
- Never blocks waiting on the client

Backends

One regular backend per connection. Communicates with other server processes via shared memory

1. Postmaster launches a new backend process for each incoming client connection.
2. Backend authenticates the client
3. Attaches to the specified database
4. Execute queries.

... when the client disconnects:

5. Detaches from shared memory
6. exit

Startup process

Launched once at system startup

- Reads the pg_control file and determines if recovery is needed
- Performs WAL recovery

Auxiliary processes

Background writer:

- Scans the buffer pool and writes out dirty buffers

WAL writer:

- Writes out dirty WAL buffers

The system will function OK without these.

More auxiliary processes

These are not attached to shared memory:

- stats collector
- logging collector
- WAL archiver

Part 2: Shared Memory

PostgreSQL processes use shared memory to communicate.

- Fixed size, allocated at startup.
- Divided into multiple small areas for different purposes.
- $>$99% of the shared memory is used by the shared buffer pool (*shared_buffers*).
- Most shared memory areas are protected by LWLocks

Shared Memory / Buffer Pool

- Consists of a number of 8k buffers.
  - sized by shared_buffers
  - e.g shared_buffers=1GB –> 131072 buffers
- Each buffer can hold one page
- Buffer can be dirty
- Buffer must be "pinned" when accessed, so that it's not evicted
- Buffer can be locked in read or read/write mode.

Buffer replacement uses Clock algorithm

Shared Memory / Buffer Pool

```
typedef struct sbufdesc
{
    BufferTag   tag;            /* ID of page contained in
    BufFlags    flags;          /* see bit definitions abov
    uint16      usage_count;    /* usage counter for clock
    unsigned    refcount;       /* # of backends holding pi
    int         wait_backend_pid;       /* backend PID of p

    slock_t     buf_hdr_lock;   /* protects the above field

    int         buf_id;         /* buffer's index number (f
    int         freeNext;       /* link in freelist chain *

    LWLock     *io_in_progress_lock;    /* to wait for I/O
    LWLock     *content_lock;   /* to lock access to buffer
} BufferDesc;
```

Shared Memory / Proc Array

One entry for each backend (PGPROC struct, $< 1KB$). Sized by max_connections.

- database ID connected to
- Process ID
- Current XID
- stuff needed to wait on locks

Acquiring an MVCC snapshot scans the array, collecting the XIDs of all processes.

Deadlock checker scans the lock information to form a locking graph.

Shared Memory / Lock Manager

Lock manager handles mostly relation-level locks:

- prevents a table from begin DROPped while it's being used
- hash table, sized by max_locks_per_transaction * max_connections
- deadlock detection
- use "SELECT * FROM pg_locks" to view

Aka known as heavy-weight locks. Data structures in shared memory are protected by lightweight locks and spinlocks.

Shared Memory / Other stuff

Communication between backends and aux processes:

- AutoVacuum Data
- Checkpointer Data
- Background Worker Data
- Wal Receiver Ctl
- Wal Sender Ctl

pgstat

- Backend Status Array
- Backend Application Name Buffer
- Backend Client Host Name Buffer
- Backend Activity Buffer

Shared Memory / Other stuff

Other caches (aka. SLRUs):

- pg_xlog
- pg_clog
- pg_subtrans
- pg_multixact
- pg_notify

Misc:

- Prepared Transaction Table
- Sync Scan Locations List
- BTree Vacuum State
- Serializable transactions stuff

Shared Memory / Other stuff

Communication with postmaster:

- PMSignalState

Shared Cache Invalidation:

- shmInvalBuffer

Locking

1. Lock manager (heavy-weight locks)

- deadlock detection
- many different lock levels
- relation-level
- *pg_locks* system view

2. LWLocks (lightweight locks)

- shared/exclusive
- protects shared memory structures like buffers, proc array
- no deadlock detection

3. Spinlocks

- Platform-specific assembler code
- typically single special CPU instruction
- busy-waiting
- used to implement higher level locks

Shared Memory / What's *not* in shared memory

- Catalog caches
- Plan cache
- work_mem

Data structures in shared memory are simple, which is good for robustness.

Part 3: Backend-private memory / Caches

Relcache:

- information about tables or indexes

Catalog caches, e.g:

- operators
- functions
- data types

Plan cache:

- plans for prepared statements
- queries in PL/pgSQL code

When a table/operator/etc. is dropped or altered, a "shared cache invalidation" event is broadcast to all backends. Upon receiving the event, the cache entry for the altered object is invalidated.

Backend-private memory

All memory is allocated in MemoryContexts. MemoryContexts form a hierarchy:

- TopMemoryContext (like malloc())
- per-transaction context (reset at commit/rollback)
- per-query context (reset at end of query)
- per-expression context (reset ~ between every function call)

Most of the time, you don't need to free small allocations individually.

Backend-private memory

```
TopMemoryContext: 86368 total in 12 blocks; 16392 free (37
  TopTransactionContext: 8192 total in 1 blocks; 7256 free
  TableSpace cache: 8192 total in 1 blocks; 3176 free (0 ch
  Type information cache: 24248 total in 2 blocks; 3712 fre
  Operator lookup cache: 24576 total in 2 blocks; 11848 fre
  MessageContext: 32768 total in 3 blocks; 13512 free (0 ch
  Operator class cache: 8192 total in 1 blocks; 1640 free (
  smgr relation table: 24576 total in 2 blocks; 9752 free (
  TransactionAbortContext: 32768 total in 1 blocks; 32736 f
  Portal hash: 8192 total in 1 blocks; 1640 free (0 chunks)
  PortalMemory: 8192 total in 1 blocks; 7880 free (0 chunks
    PortalHeapMemory: 1024 total in 1 blocks; 784 free (0 c
      ExecutorState: 8192 total in 1 blocks; 784 free (0 ch
        printtup: 8192 total in 1 blocks; 8160 free (0 chun
        ExprContext: 0 total in 0 blocks; 0 free (0 chunks)
  Relcache by OID: 24576 total in 2 blocks; 13800 free (2 c
  CacheMemoryContext: 516096 total in 6 blocks; 82480 free
    pg_class_tblspc_relfilenode_index: 3072 total in 2 bloc
```

Part 3: Error handling / ereport()

```
ereport(ERROR,
        errmsg("relation \"%s\" in %s clause not found in F
           thisrel->relname,
               LCS_asString(lc->strength)),
               parser_errposition(pstate, thisrel->locatio
```

- Jumps out of the code being executed, like a C++ exception. (uses longjmp())
- Sends the error to the client
- Prints the error to the log
- Error handler aborts the (sub)transaction:
  - per-transaction memory context is reset
  - locks are released

Part 3: Error handling / FATAL

Typically for serious, unexpected, internal errors:

```
if (setitimer(ITIMER_REAL, &timeval, NULL) != 0)
    elog(FATAL, "could not disable SIGALRM timer: %m");
```

- Also when the client disconnects unexpectedly.
- Like ERROR, jumps out of the code being executed, and sends the message to client and log
- Releases locks, detaches from shared memory, and terminates the process.
- The rest of the system continues running.

Part 3: Error handling / PANIC

Serious events that require a restart:

```
ereport(PANIC,
          (errcode_for_file_access(),
           errmsg("could not open transaction log file \"%s\"
```

- Prints the error to the log
- Terminates the process immediately with non-zero exit status.
- Postmaster sees the unexpected death of the child process, and sends SIGQUIT signal to all remaining child processes, to terminate them.
- After all the children have exited, postmaster restarts the system (crash recovery)

Part 3: Backend programming

- Hierarchical memory contexts
- Error handling

->

Robustness

Thank you

- PostgreSQL is easily to extend
- PostgreSQL source code is easy to read

Start hacking! Questions?