JITing PostgreSQL using LLVM

Andres Freund
PostgreSQL Developer & Committer

Email: andres@anarazel.de
Email: andres.freund@enterprisedb.com
Twitter: @AndresFreundTec
anarazel.de/talks/fosdem-2018-02-03/jit.pdf
TPC-H Q01

SELECT
    l_returnflag,
    l_linestatus,
    sum(l_quantity) AS sum_qty,
    sum(l_extendedprice) AS sum_base_price,
    sum(l_extendedprice * (1 - l_discount)) AS sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) AS sum_charge,
    avg(l_quantity) AS avg_qty,
    avg(l_extendedprice) AS avg_price,
    avg(l_discount) AS avg_disc,
    count(*) AS count_order
FROM lineitem
WHERE l_shipdate <= date '1998-12-01' - interval '74 days'
GROUP BY l_returnflag, l_linestatus
ORDER BY l_returnflag, l_linestatus;
<table>
<thead>
<tr>
<th>Overhead</th>
<th>Command</th>
<th>Shared Object</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>- 35.96%</td>
<td>postgres</td>
<td>postgres</td>
<td>[] ExecInterpExpr</td>
</tr>
<tr>
<td></td>
<td>+ 72.86% ExecAgg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- 18.33%</td>
<td>tuplehash_insert</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LookupTupleHashEntry</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ExecAgg</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ExecSort</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+ 8.81% ExecScan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- 10.79%</td>
<td>postgres</td>
<td>postgres</td>
<td>[] slot_deform_tuple</td>
</tr>
<tr>
<td></td>
<td>slot_getsomeattrs</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- ExecInterpExpr</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+ 77.31% ExecScan</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+ 22.69% tuplehash_insert</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ 10.66%</td>
<td>postgres</td>
<td>postgres</td>
<td>[] slot_getsomeattrs</td>
</tr>
<tr>
<td></td>
<td>postgres</td>
<td>postgres</td>
<td>[] tuplehash_insert</td>
</tr>
<tr>
<td></td>
<td>postgres</td>
<td>postgres</td>
<td>[] float8_accum</td>
</tr>
<tr>
<td></td>
<td>postgres</td>
<td>postgres</td>
<td>[] float8pl</td>
</tr>
<tr>
<td></td>
<td>postgres</td>
<td>postgres</td>
<td>[] bpchareq</td>
</tr>
<tr>
<td></td>
<td>postgres</td>
<td>postgres</td>
<td>[] hashbpchar</td>
</tr>
</tbody>
</table>
What is “Just In Time” compilation

- Convert forms of “interpreted” code into native code
- Specialize code for specific constant arguments
- Achieve speedups via:
  - reduced total number of instructions
  - reduced number of branches
  - reduced number of indirect jumps / calls
- Well known from browsers for javascripts, java VMs and the like
Methods of JITing considered

- Emit C code, invoke compiler, generate shared object, dlopen()
  - requires a lot of forking
  - requires C compiler
  - doesn’t easily allow inlining

- Directly emit machine language, remap memory executable
  - fastest to emit
  - no optimization (including inlining)
  - lots of per-architecture work
  - very few people want / able to maintain
  - fun

- Use compiler / optimizer framework with JIT support
  - issues around licensing, portability, maturity
  - JIT often not most common user

=> LLVM
LLVM

- Compiler Framework
- Intermediate Representation
  - can be generated for C code using clang!
- Optimizations
- JIT Support
- https://llvm.org/
- Used among my others by
  - clang C, C++ compiler
  - swift
  - rust
  - ...

Postgres LLVM usage

- C vs. C++
- LLVM usage in shared library
  - can be installed separate from main postgres package
  - C++ usage encapsulated
- Error handling
- Emissions of JITed functions batched
- Type syncing
- Inlining Support
LLVM and errors

- LLVM is not exception safe (with some exceptions)
- Many errors returned to callers
- Out of memory is not reported to callers
- Postgres treats out of memory as a transient condition
- LLVM has OOM / error handler callbacks – which cannot abort in non-fatal manner
  - `llvm::install_fatal_error_handler`
  - `llvm::install_bad_alloc_error_handler`
- Lots of allocation errors outside above callbacks (via c++ NEW)
  - `std::set_new_handler`
- PostgreSQL API
  - `extern void llvm_enter_fatal_on_oom(void);`
  - `extern void llvm_leave_fatal_on_oom(void);`
  - `extern void llvm_assert_in_fatal_section(void);`
  - `extern void llvm_reset_fatal_on_oom(void);`
- IOW out of memory in LLVM results in a FATAL error (cancelled connection)
- Memory usage usually not that high, especially comparing to typical analytics queries
- Need better solution medium-long term
LLVM and Errors #2

- Most errors are non-FATAL
- generated functions need to be free’d
Emission of functions, batching & deallocation

- Emitting objects has near-constant overhead
- Objects need to be freed at query end
- API:
  
  ```c
  LLVMJitContext *llvm_create_context(int jitFlags);
  LLVMModuleRef llvm Mutable_module(LLVMJitContext*);
  void *llvm_get_function(ctx, const char *funcname);
  ```
- Emission of code delayed until `llvm_get_function`
- Context automatically deallocated via ResourceOwner mechanism / query end
v10+ Expression Evaluation Engine

- WHERE a.col < 10 AND a.another = 3
  - EEOP_SCAN_FETCHSOME (deform necessary cols)
  - EEOP_SCAN_VAR (a.col)
  - EEOP_CONST (10)
  - EEOP_FUNCEXPR STRICT (int4lt)
  - EEOP_BOOL_AND_STEP_FIRST
  - EEOP_SCAN_VAR (a.another)
  - EEOP_CONST (3)
  - EEOP_FUNCEXPR STRICT (int4eq)
  - EEOP_BOOL_AND_STEP_LAST (AND)

- direct threaded
- lots of indirect jumps
typedef struct FunctionCallInfoData {
    FmgrInfo *flinfo;    /* ptr to lookup info used for this call */
    fmNodePtr context;   /* pass info about context of call */
    fmNodePtr resultinfo; /* pass or return extra info about result */
    Oid fncollation;     /* collation for function to use */
    bool isnull;         /* function must set true if result is NULL */
    short nargs;         /* # arguments actually passed */
    Datum arg[FUNCTION_MAX_ARGS]; /* Arguments passed to function */
    bool argnull[FUNCTION_MAX_ARGS]; /* T if arg[i] is actually NULL */
} FunctionCallInfoData;

#define FunctionCallInvoke(fcinfo) 
   (*((fcinfo)->flinfo->fn_addr) (fcinfo))
EEO_CASE(EEOP_FUNCEXPR STRICT)
{
    FunctionCallInfo fcinfo = op->d.func.fcinfo_data;
    bool *argnull = fcinfo->argnull;
    int argno;
    Datum d;

    /* strict function, so check for NULL args */
    for (argno = 0; argno < op->d.func.nargs; argno++) // unnecessary
    {
        if (argnull[argno])
        {
            *op->resnull = true;
            goto strictfail;
        }
    }

    fcinfo->isnull = false; // optimized away
    d = op->d.func.fn_addr(fcinfo); // indirect
    *op->resvalue = d; // moved to register
    *op->resnull = fcinfo->isnull;

    strictfail:
        EEO_NEXT(); // indirect
}
JITed expressions

- directly emit LLVM IR for common opcodes
- emit calls to functions implementing less common opcodes
  - can be inlined
- indirect opcode → opcode jumps become direct
- indirect funcexpr calls become direct
  - can be inlined
- TPCH Q01 non-jitted vs jitted:
  - 28759 ms vs 22309 ms
  - branch misses: 0.38% vs 0.07%
  - iTLB load misses: 58,903,279 vs 48,986 (yes, really)
block.op.2.start: ; preds = %block.op.1.start
    %v_argnonnull = getelementptr inbounds %struct.FunctionCallInfoData,
    %struct.FunctionCallInfoData* %v_fcinfo, i32 0, i32 7
    store i8 1, i8* %v_argnonnull
    br label %check-null-arg

check-null-arg: ; preds = %block.op.2.start
    %25 = getelementptr inbounds [100 x i8], [100 x i8]* %v_argnonnull, i32 0, i32 0
    %26 = load i8, i8* %25
    %27 = icmp eq i8 %26, 1
    br i1 %27, label %block.op.3.start, label %check-null-arg1

check-null-arg1: ; preds = %check-null-arg
    %28 = getelementptr inbounds [100 x i8], [100 x i8]* %v_argnonnull, i32 0, i32 1
    %29 = load i8, i8* %28
    %30 = icmp eq i8 %29, 1
    br i1 %30, label %block.op.3.start, label %no-null-args

no-null-args: ; preds = %check-null-arg1
    %v_fcinfo_isnull = getelementptr inbounds %struct.FunctionCallInfoData,
    %struct.FunctionCallInfoData* %v_fcinfo, i32 0, i32 4
    store i8 0, i8* %v_fcinfo_isnull
    %funccall = call i64 @date_le_timestamp(%struct.FunctionCallInfoData* %v_fcinfo) #13
    %31 = load i8, i8* %v_fcinfo_isnull
    store i64 %funccall, i64* %resvaluep
    store i8 %31, i8* %resnullp
    br label %block.op.3.start
Type Synchronization

- Types like `%struct.FunctionCallInfoData` need to be available to IR / LLVM
- Manual syncing possible – but work intensive, failure prone, unmaintainable
- `llvmjit_types.c`:
  ```c
  ...  
  ExprState StructExprState;  
  FunctionCallInfoData StructFunctionCallInfoData;  
  HeapTupleData StructHeapTupleData;  
  ...
  ```
- `clang` converts `llvmjit_types.c` to `llvmjit_types.bc` at build time
- `llvmjit` infrastructure loads all known types from `llvmjit_types.bc` by name
- LLVM's IR doesn't have field names:
  ```c
  typedef struct FunctionCallInfoData
  {
    ...
    #define FIELDNO_FUNCTIONCALLINFODATA_ISNULL 4
    bool isnull; /* function must set true if result is NULL */
    ...
  } FunctionCallInfoData;
  ```
LLVMTypeRef members[8];

members[0] = LLVMPointerType(StructFmgrInfo, 0); /* flinfo */
members[1] = LLVMPointerType(StructPGFinfoRecord, 0); /* context */
members[2] = LLVMPointerType(StructPGFinfoRecord, 0); /* resultinfo */
members[3] = LLVMInt32Type(); /* fncollation */
members[4] = LLVMInt8Type(); /* isnull */
members[5] = LLVMInt16Type(); /* nargs */
members[6] = LLVMArrayType(TypeSizeT, FUNC_MAX_ARGS);
members[7] = LLVMArrayType(LLVMInt8Type(), FUNC_MAX_ARGS);

StructFunctionCallInfoData = LLVMStructCreateNamed(
    LLVMGetGlobalContext(),
    "struct.FunctionCallInfoData");
LLVMStructSetBody(StructFunctionCallInfoData, members, lengthof(members), false);
Tuple Deforming

• Often most significant bottleneck
• TupleDesc ("tuple format") can be made known at JIT time in many cases
• Optimizable:
  – Number of columns to deform - constant
  – Number of columns in tuple – if to-deform below last NOT NULL
  – column type - constant
  – column width – known for fixed width types
  – Variable alignment requirements – known for fixed width (depending on NULLness)
  – NULL bitmap – no need to check if NOT NULL
• Resulting code often very pipelineable, previously lots of stalls
• Access to tuple's t_hoff / HeapTupleHeaderGetNatts() still major source of stalls
• TPC-H Q01: unjitted deform vs jitted
  – time: 22277 ms vs 19580 ms
  – branches: 1396.318 M/sec vs 1161.628M/sec (despite higher throughput)
Inlining

- All operators in postgres are functions! Lots of external function calls
- Postgres function calls are expensive, lots of memory indirection
- Convert sourcecode to bitcode at buildtime, install into
  - $pkglibdir/bitcode/<module>.index.bc
  - $pkglibdir/bitcode/<module>/path/to/file.bc
- LLVM’s cross-module inlining not suitable
  - requires exporting of symbols at compile time, unknown which needed
- Postgres specific inlining logic:
  - build combined summary (via LLVM’s LTO infrastructure)
  - inlining safety check (no mutable static variables referenced)
  - cost analysis
  - inline function, referenced static functions, referenced constant static variables (mainly strings)
  - use llvm::IRMover to move relevant globals
  - can’t cache modules in memory, cloning expensive and incomplete
- Allows need to implement direct JIT emission for lots of semi critical code
- Function call interface significantly limits benefits
Planner

- Naive!
- Perform JIT if \( \text{query\_cost} > \text{jit\_above\_cost} \)
- Optimize if \( \text{query\_cost} > \text{jit\_optimize\_above\_cost} \)
- Optimize if \( \text{query\_cost} > \text{jit\_above\_cost} \)
- Whole query decision
- *NOT* a tracing JIT:
  - costing makes tracing somewhat superfluous
  - tracing decreases overall gains
Profiling

- Requires patches to LLVM
- To-be-submitted upstream
- `perf record -k 1 -p $pid`
- `perf inject --jit -i /tmp/perf.data -o /tmp/perf.jit.data`
- Issues:
  - function names not great
  - requires session to end to flush profiling data
- 24.75% postgres jitted-6402-7.so  
  99.99% evalexpr.1.7
  ExecAgg
- 20.28% postgres jitted-6402-4.so  
  evalexpr.1.0
  ExecScan
  ExecAgg
- 12.73% postgres jitted-6402-6.so  
  98.16% evalexpr.1.4
    tuplehash_insert
    LookupTupleHashEntry
    lookup_hash_entries
    ExecAgg
+  7.53% postgres postgres
+  4.93% postgres postgres
+  3.28% postgres postgres
+  2.93% postgres postgres

  [. evalexpr.1.7
  [. evalexpr.1.0
  [. evalexpr.1.4
  [. tuplehash_insert
  [. heap_getnext
  [. lookup_hash_entries
  [. hash_any
Faster Execution: JIT Compilation

TPCH Q01 timing
scale 100, fully cached
JIT Issues – Code Generation

- Expressions refer to permanently allocated memory
  - generated code references memory locations
  - optimizer can’t optimize away memory lots of memory references
  - FIX: separate permanent and per eval memory

- Function Call Interface requires persistence
  - lots of superfluous memory reads/writes for arguments, optimizer can’t eliminate in most cases
    - massively reduces benefits of inlining
  - FIX: pass FunctionCallInfoData and FmgrInfo separately to functions
    - remove FunctionCallInfoData->flinfo
    - move context, resultinfo, fncollation to FmgrInfo
    - move isnull field to separate argument? Return struct?

- Expression step results refer to persistent memory
  - move to temporary memory
JIT Issues - Caching

- Optimizer overhead significant
  - TPCH Q01: unopt, noinline: time to optimize: 0.002s, emit: 0.036s
  - TPCH Q01: time to inline: 0.080s, optimize: 0.163s, emit 0.082s
- References to memory locations prevent caching (prev slide)
- Introduce per-backend LRU cache of functions keyed by hash of emitted LRU (plus comparator)
  - relatively easy task
- Allow expressions to be generated at plan time, and tied to a prepared statement
  - medium – hard
JIT Issues – Planning

• Whole Query decision too coarse
  – use estimates about total number of each function evaluation?
• Some expressions guaranteed to only be evaluated once
  – VALUES()
  – SQL functions
Future things to JIT

- Aggregate & Hashjoin hash computation
  - easy
- entire in-memory tuplesort
  - easy
- on-disk tuplesort comparator
  - easy
- COPY input
  - medium
- Whole of Executor
  - wheeee
Future JIT Infrastructure

- Perform JIT without optimization in foreground
- Have background worker perform incrementally better optimization in background
- Replace JITed function once finished
- Relocations still need to be performed in backend
- Better error handling
- EXPLAIN (ANALYZE, JIT)?
LLVM Issues

- Error Handling
- C-API isn’t large enough, C++ API changes
- Medium-High level API documentation bad to nonexistent
- Some optimization passes (primarily dead store elimination) not aggressive enough
- Parts of API pointlessly complicated (welcome Error.h)
- My notebook’s battery doesn’t like it
JITing PostgreSQL using LLVM

Andres Freund
PostgreSQL Developer & Committer

Email: andres@anarazel.de
Email: andres.freund@enterprisedb.com
Twitter: @AndresFreundTec
anarazel.de/talks/fosdem-2018-02-03/jit.pdf