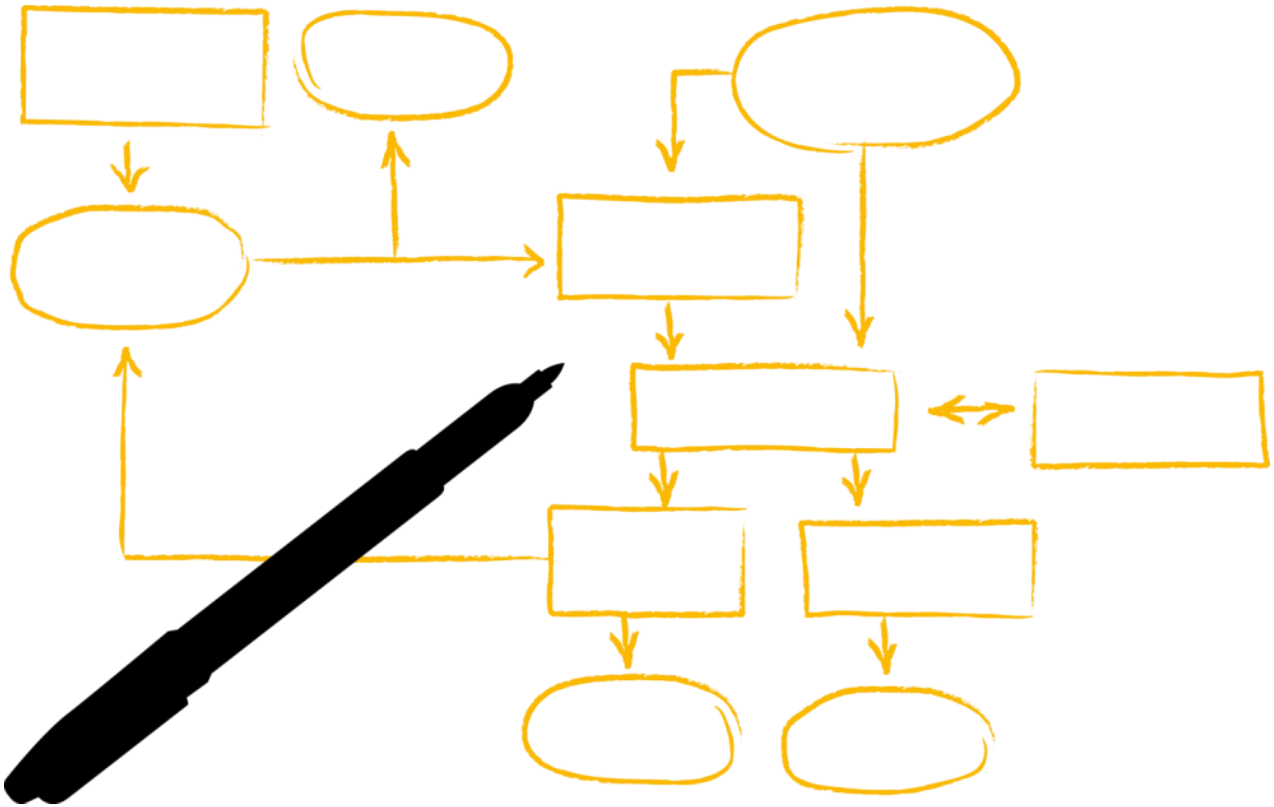
Public

LLVM in an in-memory database server

Markus Eble, SAP
March 28, 2017



Agenda

- **What are the challenges when working in an in-memory database?**
- **What are we doing with LLVM?**
- **How to meet the in-memory challenges with LLVM?**

What are the challenges for an in memory-database?

- **Never crash**
  - Survive out-of-memory
  - Prevent stack overflow
  - Long running operations must be interruptible
- **Massive parallelization**
  - Thread-safe programming
  - Avoid locks
- **SQL Semantics**
  - Arithmetic operations need overflow checks
  - Operands can have value NULL (NULL means undefined not ‘0’)
- **JIT Compile time**

What are we doing with LLVM?

We are using LLVM as compiler backend for

- **Stored procedures**
  - For operations that are hard to express in SQL
- **Query plans**
  - Generate a program to execute the query
  - Compile the program on-the-fly
  - More on query plan execution with LLVM: http://www.vldb.org/pvldb/vol4/p539-neumann.pdf

To simplify creation of LLVM-IR we use our own intermediate language “Llang”

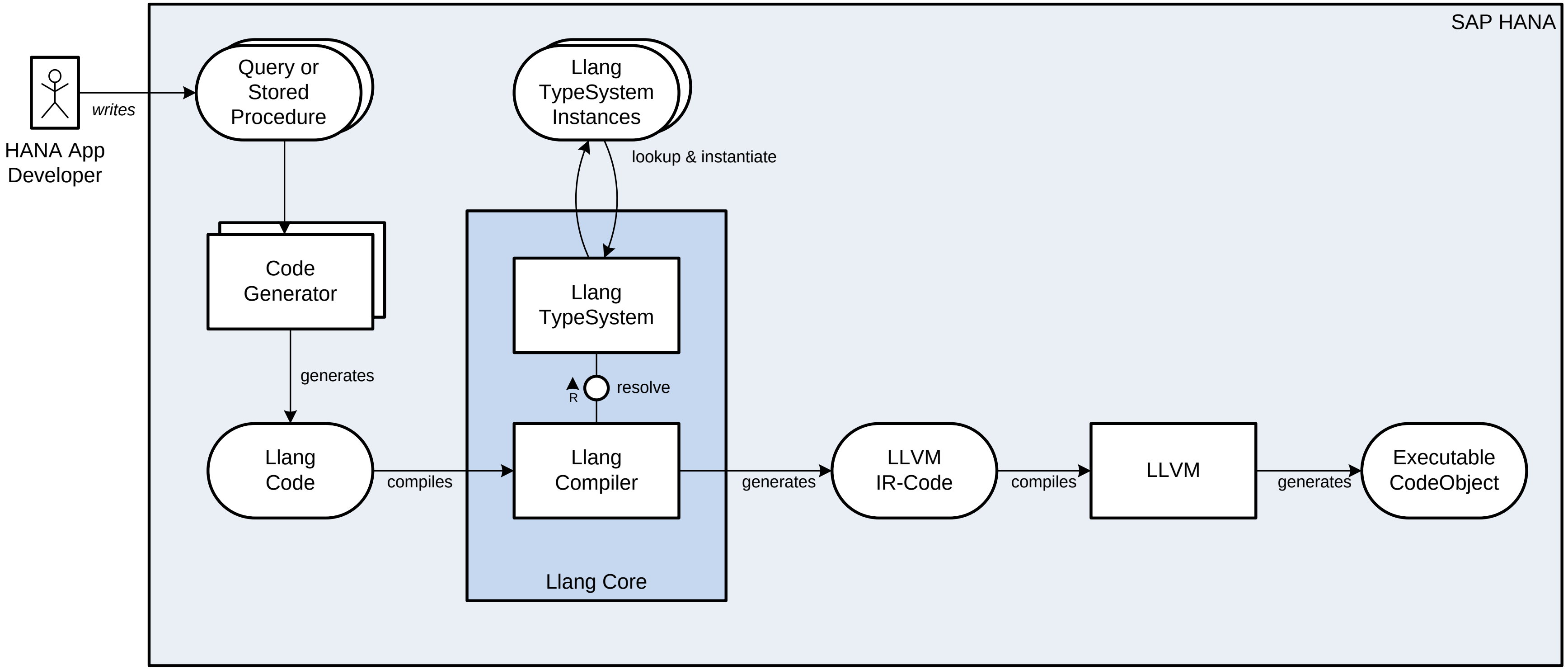
Sample Llang code and resulting LLVM-IR

```
// Llang Code
Int32 add(Int32 lhs, Int32 rhs) {
  return lhs + rhs;
}

; LLVM IR
; physical return is used to signal an exception
; logical return is first parameter of the function
; implicit parameter _ctxt containing runtime environment
define i64 @add(i32* %Result, {i1*}* %_ctxt, i32 %lhs, i32 %rhs){
add_first:
  ; arithmetic requires an overflow check
  %0 = call {i32,i1} @llvm.sadd.with.overflow.i32(i32 %lhs, i32 %rhs)
  %value.i = extractvalue {i32,i1} %0, 0
  %errorBit.i = extractvalue {i32,i1} %0, 1
  %extErrBit.i = zext i1 %errorBit.i to i64
  store i32 %value.i, i32* %Result, align 4
  br i1 %errorBit.i, label %add_rcc_unwind_top, label %add_return
```

Sample Llang code and resulting LLVM-IR

```
; function exit (with or without exception)
add_return:                        ; preds = %add_first, %add_rcc_unwind_top
  %RC.0 = phi i64 [%extErrBit.i,%add_rcc_unwind_top], [0,%add_first]
  ret i64 %RC.0

; unwinding and creation of error stack trace
add_rcc_unwind_top:                ; preds = %add_first
  %rc_l2_i2 = call i64 @"fn~_llangStackTracePush~Void"(
    {i1*}* %_ctxt,      ; stack trace is stored in the context
    i32 3,              ; line number of error
    i64 %extErrBit.i)   ; error code
  br label %add_return
}
```

Architecture of Llang-LLVM-Compiler



Our history using LLVM

2008	**Start developing an in-memory database by HPI+SAP with column based data layout**
2010	**First integration of LLVM using LLVM 2.7**
2010	**First productive delivery of SAP HANA**
2013	**Upgrade to LLVM 3.1 and mcjit**
2014	**Upgrade to LLVM 3.3**
2016	**Upgrade to LLVM 3.7**

Overall experience with LLVM

We are happy to use LLVM due to

- **Excellent quality**
- **No functional regressions when switching to a new release**
- **Easy to use API**
- **Supportability**
  Traces, debugger integration, profiler integration

How to meet the in-memory challenges with LLVM?

- **Never crash**
  - **Survive out-of-memory**
  - Prevent stack overflow
  - **Long running operations must be interruptible**
- **Massive parallelization**
  - Thread-safe programming
  - Avoid locks
- **SQL Semantics**
  - Arithmetic operations need overflow checks
  - Operands can have value NULL (NULL means undefined not ‘0’)
- **Compile time**

Long running operations must be interruptible

Add check for the transaction abort flag to each loop condition

```
while_head:
  ; regular loop condition
  %value_l4_c9 = load i1, i1* %cond_l3
  %1 = icmp ne i1 %value_l4_c9, false
  ; abort flag
  %2 = load volatile i1, i1* %doAbort
  %doContinue = xor i1 %2, true
  ; check loop condition and abort flag
  %enterLoop = and i1 %1, %doContinue
  br i1 %enterLoop, label %l4_while_body, label %l4_while_exit
```

Drawback: Many optimizer passes don’t like the volatile load

Survive out-of-memory

Follow rules for exception safe programming:

- **Resource allocation is initialization (Wikipedia)**
- **Use members to store allocated objects or unique_ptr/shared_ptr**
- **For each member: document if you own it**
- **Destructors have to be no throw**

Testing out-of-memory

Overload operator new and make it systematically fail at the n-th allocation:

```
void* operator new(size_t size) {
  static allocCounter = 0;
  allocCounter++;
  if (allocCounter < failingAllocCounter) {
    // regular allocation
    return std::malloc(size);
  } else {
    // fail when failingAllocCounter has been reached
    throw std::bad_alloc();
  }
}
```

What you find when testing out-of-memory

```
Use::~Use() {
       …
       m_count = 1;
}
```

Who can read a member of an object that has been set in the destructor?

What you find when testing out-of-memory

```
Use::operator delete(void* addr) {
        if (static_cast<Use*>(addr)->m_count == 1) {
                ...
        }
}
```

What can go wrong if the destructor communicates with operator delete?

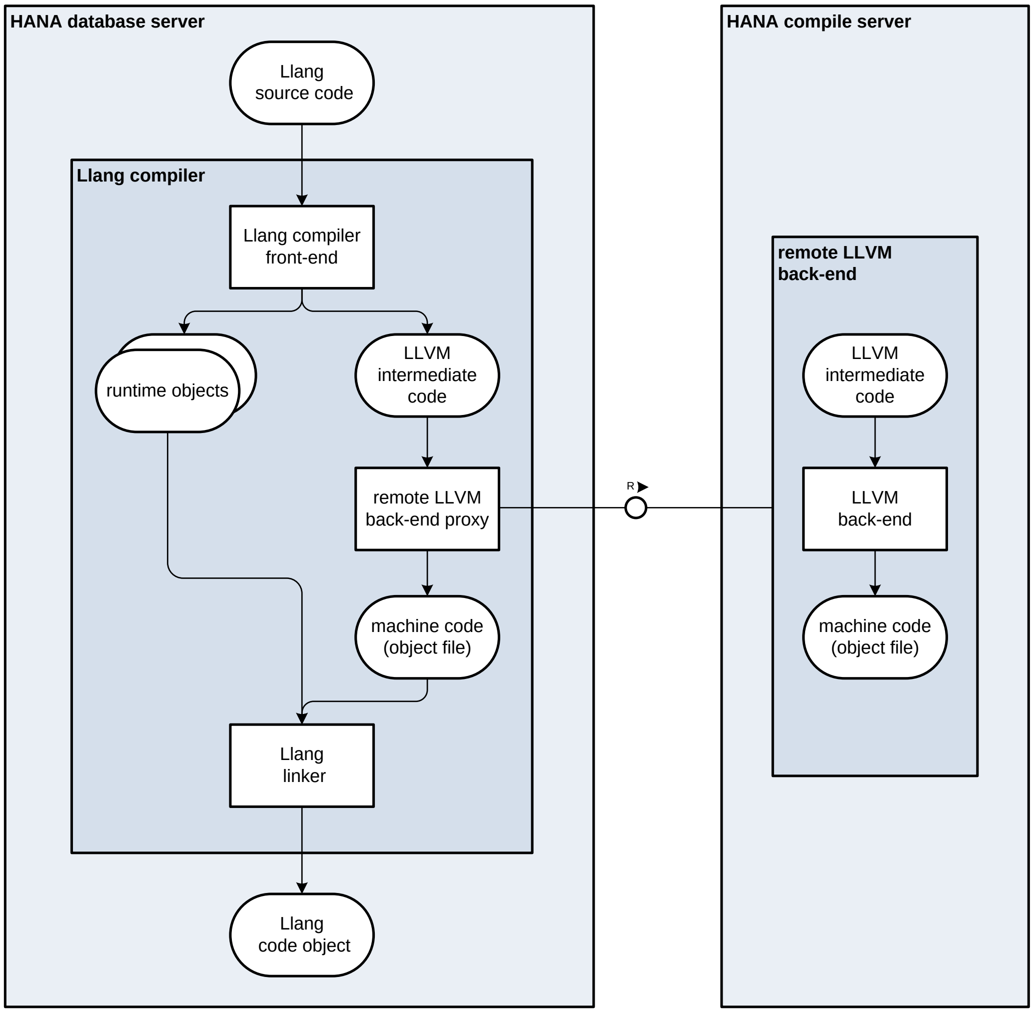
What you find when testing out-of-memory

Destructor is not called if the constructor exits with an exception

Conclusion:

- **Unwinding is only done for steps that have been completed successfully**
- **Work symmetric (operator new/operator delete, constructor/destructor)**

Surviving out-of-memory while using LLVM



Our approach:

- **Fix the frontend part (creation of LLVM-IR)**
- **Move the backend part (optimization and machine codegen) to a separate process**
- **Link the resulting machine code into the database process**
- **Abort and restart the backend in case of out-of-memory**

Compile Performance

Query Processing Time = Query Preparation Time + Query Execution Time

Compile Time matters!

Compile Performance – Large functions

Challenges

- Optimization and register allocation are complex algorithms.
- Code generated by code generators tends to contain large functions.
- This can lead to exploding compile times (up to hours).
- We tried to speed up register allocator without success.

Our solution:
Split large functions automatically into smaller LLVM functions

Sample function splitting

```
// original function
Int32 calc(Int32 op0, Int32 op1, Int32 op2, Int32 op3, Int32 op4) {
  Int32 result = op0;
  result = result + op1;
  result = result - op2;
  result = result * op3;
  result = result / op4;
  return result;
}

// function after splitting
Int32 calc(Int32 op0, Int32 op1, Int32 op2, Int32 op3, Int32 op4) {
  Int32 result = op0;
  calc_0(op1, op2, result);
  calc_1(op3, op4, result);
  return result;
}
```

Compile Performance – Small functions

Example Query Execution:

SELECT amount * 1.19 FROM sales;

Classic approach:

- select amount for all rows from sales
- use an expression interpreter to evaluate amount * 1.19

Code generation approach:

- create a program that selects the amount values and does the calculation
- compile the program
- execute the program

In order to beat the classic approach the savings by faster execution have to be larger than the additional compile costs.

Compile vs Interpret

	Compile	Interpret
Prepare	1 ms + 1 ms * LOC	250 µs + 20 µs * LOC
Execute	1 µs + 1 ns * LOC	2.5 µs + 150 ns * LOC

Currently our compiler approach beats the evaluator approach only if the number of iterations is >5,000

The faster the compile time the more often we can benefit from the compilation

Compile Performance – Small functions

Our tries to reduce compile time for small functions:

- **reduce optimization passes**
  keep fast optimization passes
  keep optimization passes that reduce effort for machine code generation
  remove loop optimization passes

- **Use fast instruction selector**
  didn’t improve compile time in most cases

Is there a way to get a fast machine code generation when you are willing to sacrifice execution performance?

Thank you

Contact information:

Markus Eble
markus.eble@sap.com

Interested in working on
compiler technology at SAP?
http://www.sap.com/germany/careers-thecore
thecore@sap.com