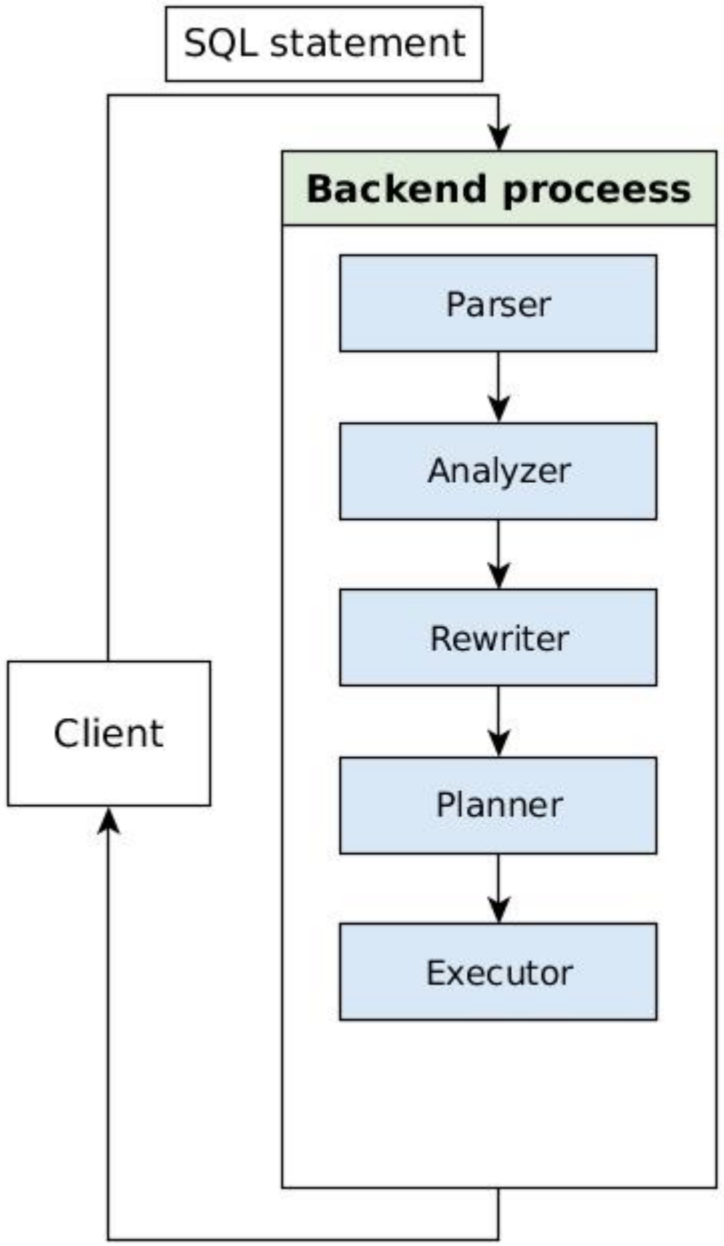
Machine Code Caching in PostgreSQL Query JIT-compiler

Ruben Buchatskiy, Mikhail Pantilimonov, Roman Zhuykov, Eugene Sharygin, Dmitry Melnik
Ivannikov Institute for System Programming of the RAS

Questions:
ruben@ispras.ru

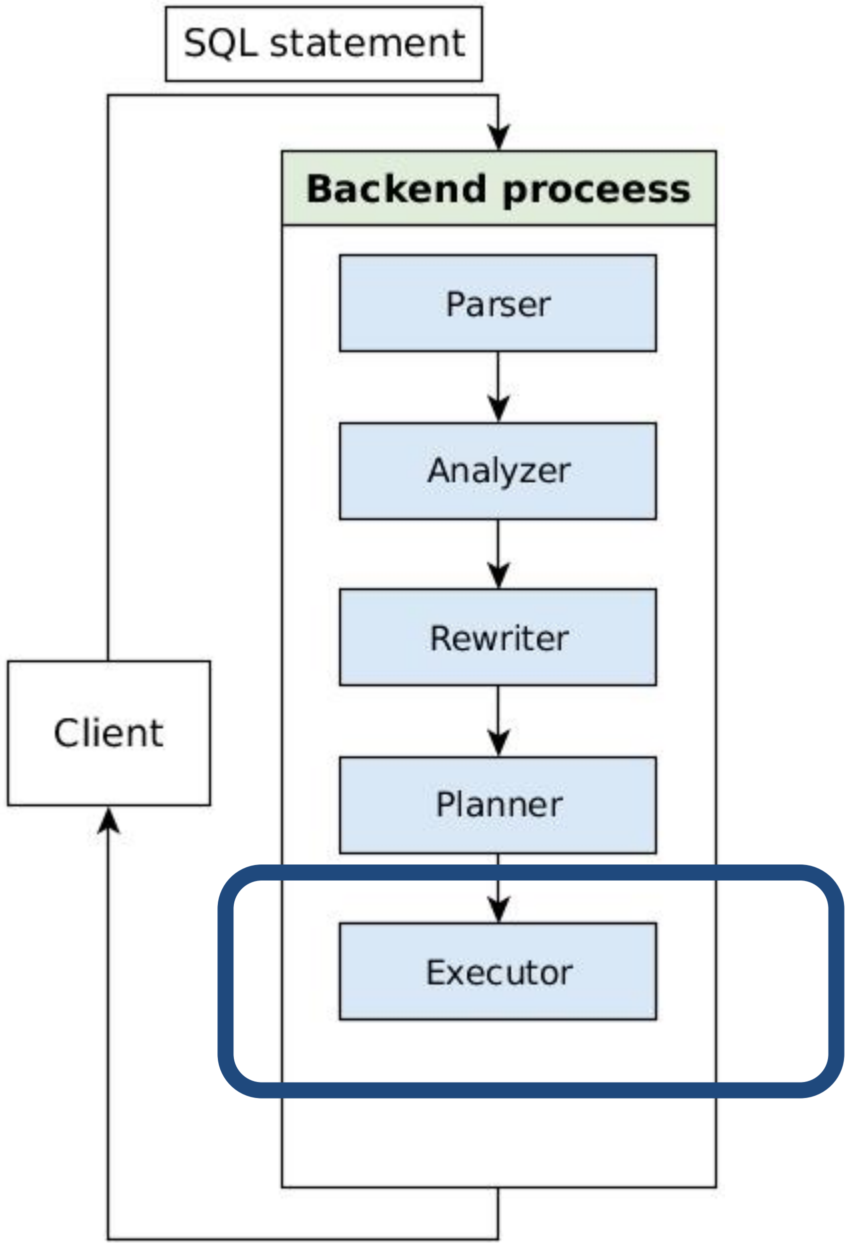
Ivannikov Memorial Workshop,
Veliky Novgorod, 2019

SQL Query Processing Pipeline



- Parser(Query_string) $\Rightarrow$ Parse_tree;
- Analyzer(Parse_tree) $\Rightarrow$ Query_tree;
- Rewriter(Query_tree, Rules) $\Rightarrow$ Query_tree;
- Planner(Query_tree, Costs) $\Rightarrow$ Plan_tree;
- Executor(Plan_tree) $\Rightarrow$ Query_results.

SQL Query Processing Pipeline: Execution



Execution models:
- Iterator
- Materialization
- Vectorized

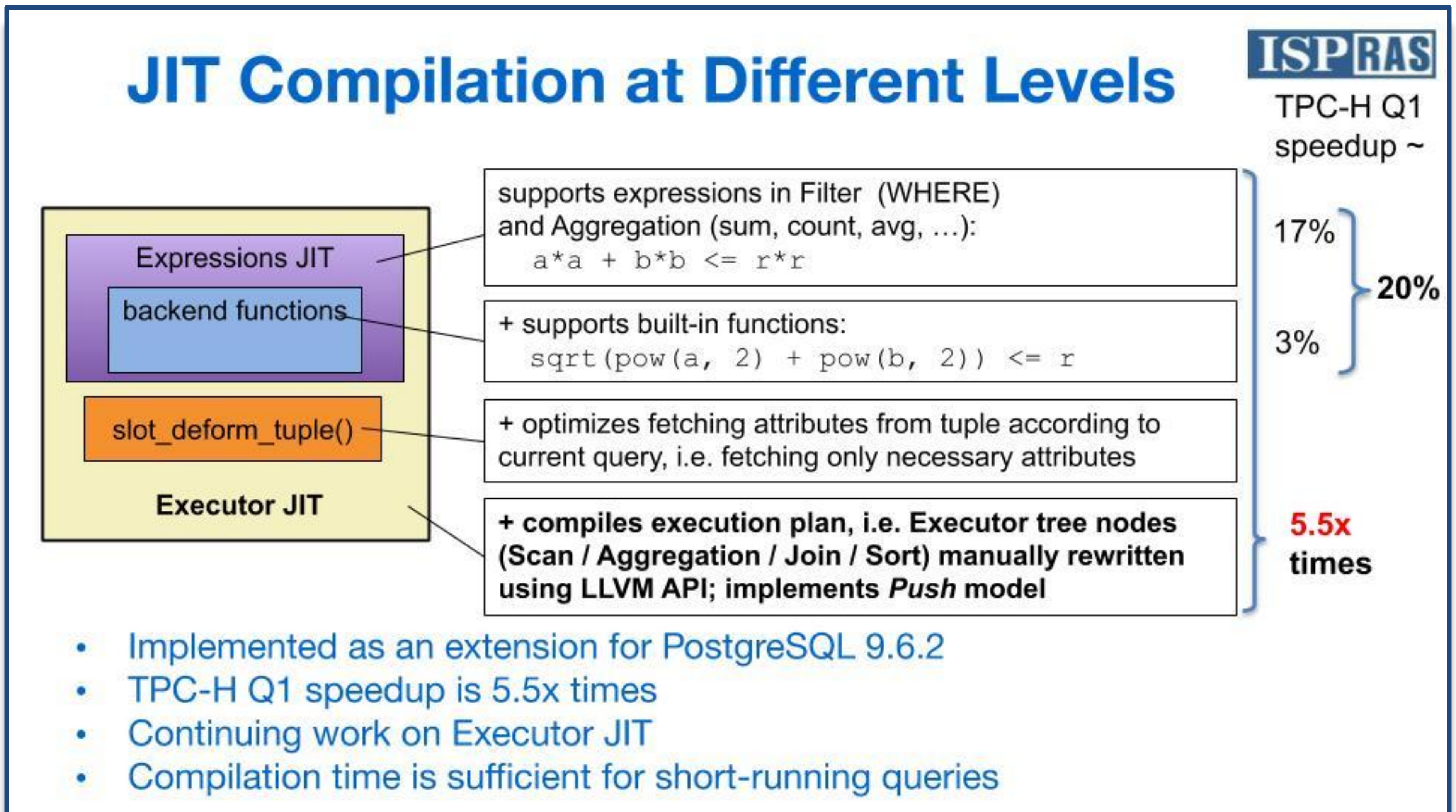
Execution direction:
- Top-Down
- Bottom-Up

Different approaches with pros and cons...

But there is interpretation overhead for all of them (especially in PostgreSQL combination):
- indirect function calls
- branch mispredictions
- bad code locality
- excessive run-time checks
- etc.

Our previous work on Dynamic Query Compilation in PostgreSQL

One approach to resolve high interpretation overhead is dynamic query plan compilation.



https://www.pgcon.org/2017/schedule/events/1092.en.html

Query Compilation Cost

Q1 from TPC-H benchmark

```
explain(timing off, analyze) select
    l_returnflag,
    l_linestatus,
    sum(l_quantity) as sum_qty,
    +7 more aggregate functions
from
    lineitem
where
    l_shipdate <= date '1998-12-01' - interval '68 days'
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus;
```

Average results on SCALE=2 database ~5.4GB
Query processes ~12 millions tuples

PostgreSQL 9.6.3 interpreter

```
                     QUERY PLAN
------------------------------------------------------------------------------
...
 Planning time:       0.275 ms
 Execution time: 9404.509 ms
```

PostgreSQL 9.6.3 + LLVM JIT

```
...
INFO: LLVM timer:   419.692 ms - optimization (llvm_opt_level = 3)
INFO: LLVM timer:   324.801 ms - machine code generation (compilation)
INFO: LLVM timer: 1584.526 ms - execution

                     QUERY PLAN
------------------------------------------------------------------------------
....
 Planning time:       0.278 ms
 Execution time: 2401.507 ms
```

JIT is reasonable on OLAP queries.

Query Compilation Cost

Q1 from TPC-H benchmark

```
explain(timing off, analyze) select
        l_returnflag,
        l_linestatus,
        sum(l_quantity) as sum_qty,
        +7 more aggregate functions
from
        lineitem
where
        l_shipdate <= date '1998-12-01' - interval '68 days'
        and l_partkey between 1 and 200  -- index access method
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus;
```

But what about OLTP queries that process small amount of data?

Dynamic compilation of this modified Q1
+ machine code execution
≈ 79x slower than interpretation

Average results on SCALE=2 database ~5.4GB

PostgreSQL 9.6.3 interpreter

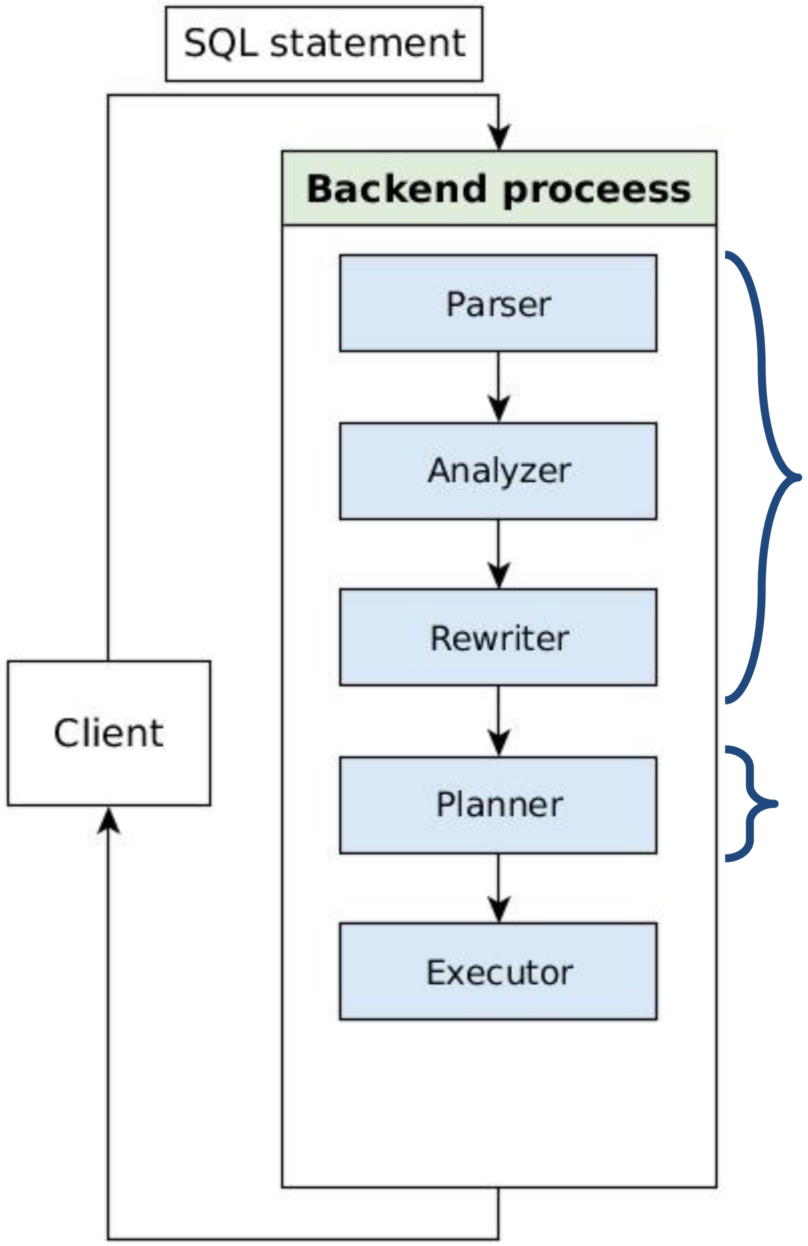
```
                          QUERY PLAN
---------------------------------------------------------------------------
...
 Planning time:     0.355 ms
 Execution time: 14.130 ms
```

PostgreSQL 9.6.3 + LLVM JIT

```
...
INFO: LLVM timer: 574.893 ms - optimization (llvm_opt_level = 3)
INFO: LLVM timer: 463.759 ms - machine code generation (compilation)
INFO: LLVM timer:     6.334 ms - execution

                          QUERY PLAN
 ---------------------------------------------------------------------------
....
 Planning time:        0.366 ms
 Execution time: 1119.507 ms
```

Query Plan Caching



One of the possible solutions — to cache generated machine code alongside with a query plan and reuse it in subsequent executions. Therefore, compilation cost pays off if the query executes many times.

PostgreSQL doesn't automatically cache queries, but offers a manual "**PREPARE**" mechanism that levels the overhead of first 3 stages. Prepared statements can use generic plans rather than re-planning with each set of supplied EXECUTE values.

GENERIC query plan after **5** executions that is used for the remaining lifetime of prepared statement.

```
PREPARE q1(int, date, date) as
 select * from orders where o_custkey = $1 and o_orderdate between $2 and $3;

                                QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using i_o_custkey on orders
  Index Cond: (o_custkey = $1)
  Filter: ((o_orderdate >= $2) AND (o_orderdate <= $3))

EXECUTE q1(...);
```

Query Plan Caching

With minor modifications to PostgreSQL query plan data structures we can save a pointer to generated machine code and reuse it together with prepared GENERIC plan.

But this is not enough as **absolute addresses** of run-time data structures that we use during code generation change on every query execution. At the run-time the execution control flows from JIT to PostgreSQL functions and vice versa.

=> Need to update obsolete addresses in generated machine code before it can be used again.

Machine Code Patching Tools

The LLVM Infrastructure provides tools for controlling and modifying the machine code generated by JIT component:

- `llvm.experimental.stackmap` – records the location of specified values in the Stack Map without generating any code.
- **`llvm.experimental.patchpoint`** – creates a function call to the specified <target> and records the location of specified values in the Stack Map.

Called object can be modified later using information from the Stack Map structure.

LLVM IR

```
...
entry:
  %pp_ret = call i64 (i64, i32, i8*, i32, ...)
    @llvm.experimental.patchpoint.i64(
      i64 0, i32 13, i8* inttoptr (i64 1311768467294899695 to i8*), i32 0)
  %pp_ret_pointer = inttoptr i64 %pp_ret to i32*
  store i32 0, i32* %pp_ret_pointer
...
```

x86_64 machine code

```
<main+1966>: movabs $0x1234567890abcdef, %r11
<main+1976>: callq  *%r11
<main+1979>: movl   $0x0, (%rax)
...
```

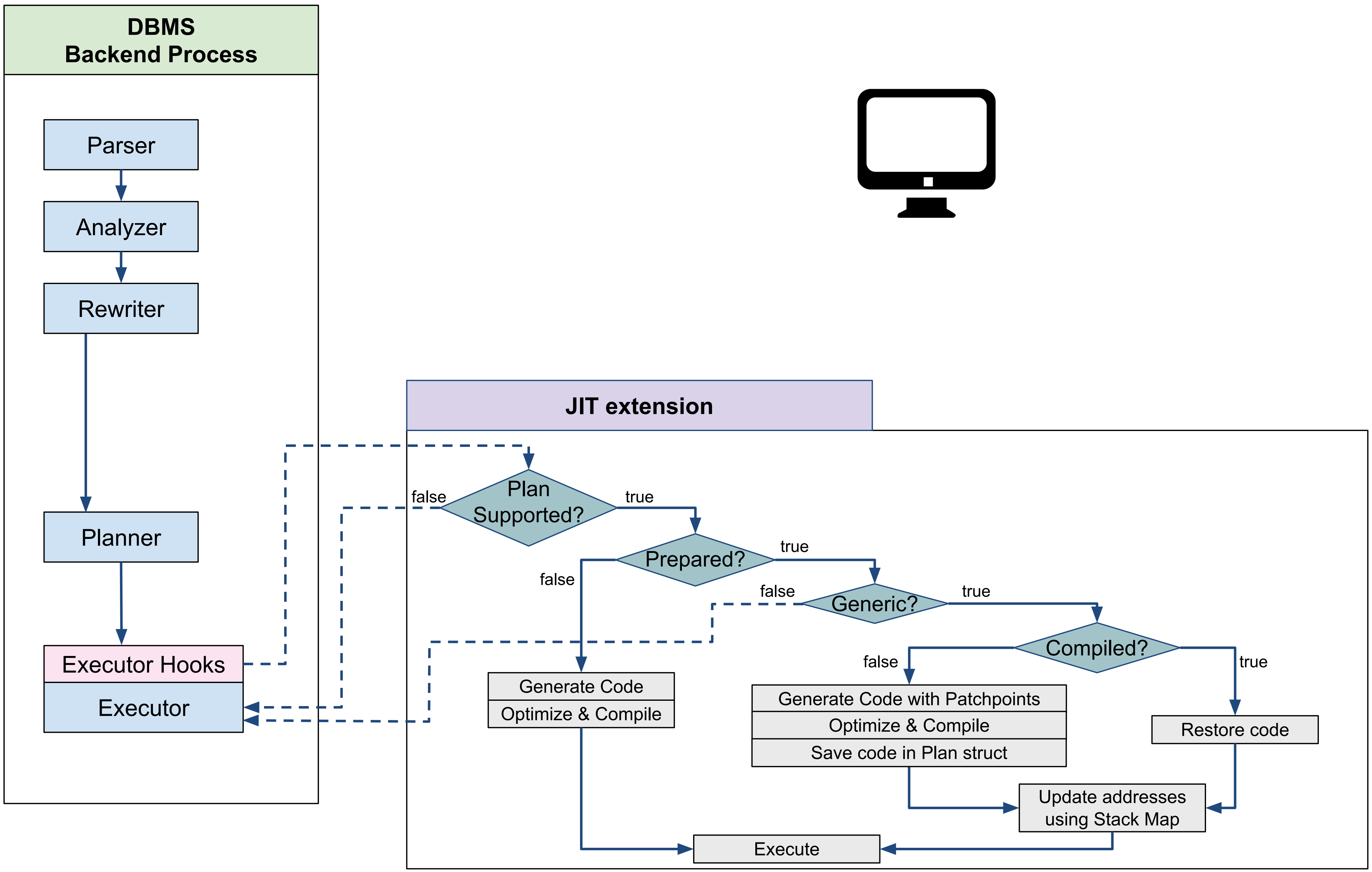
x86_64 **patched** machine code

```
<main+1966>: movabs $0x55b5b3195148, %rax
<main+1976>: data16 xchg %ax,%ax
<main+1979>: movl   $0x0, (%rax)
...
```

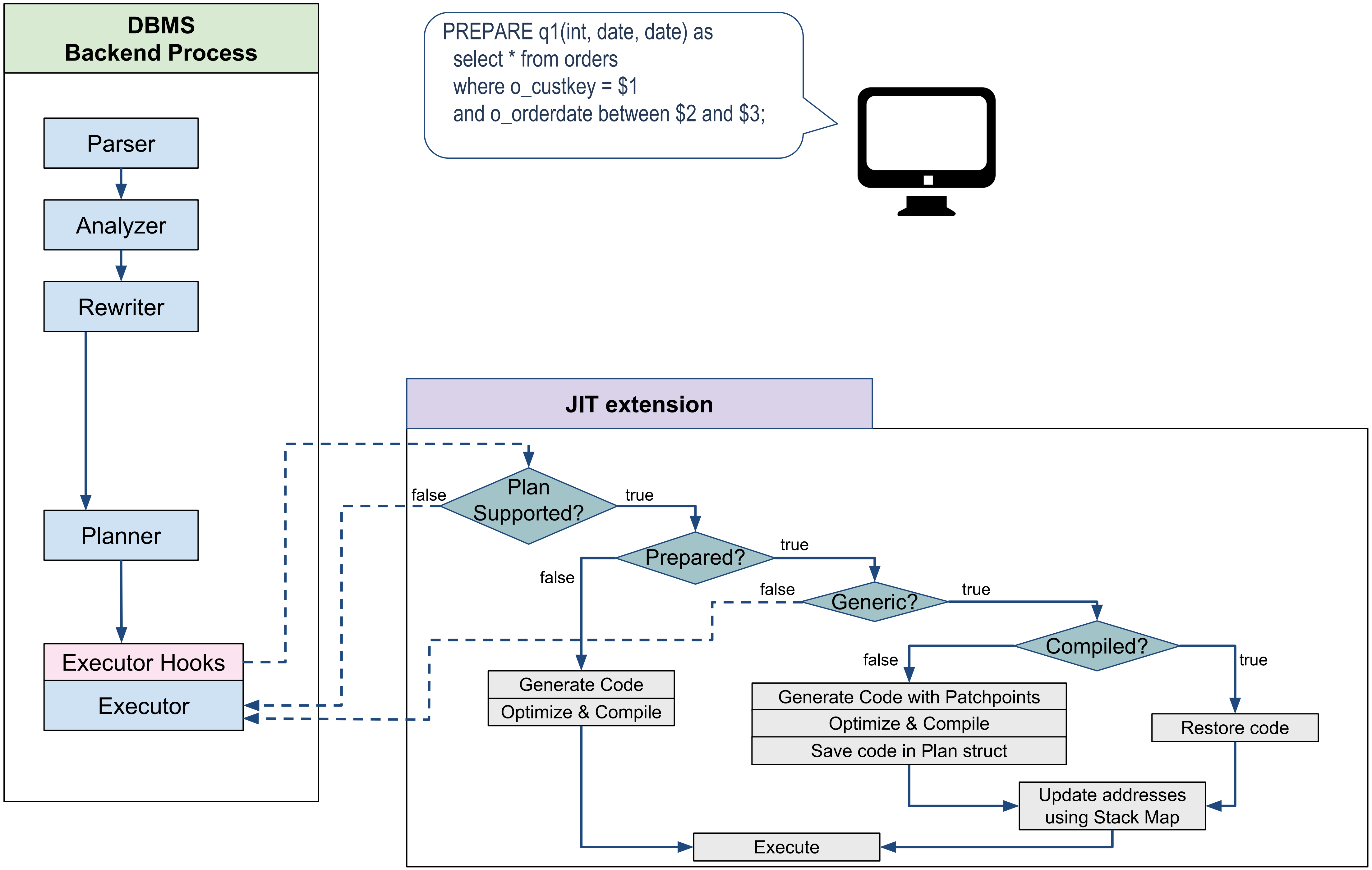
The **Stack Map** record includes an ID and the offset within the code from the beginning of the enclosing function.

A **patch point** is an instruction address at which space is reserved for patching a new instruction sequence at run time.

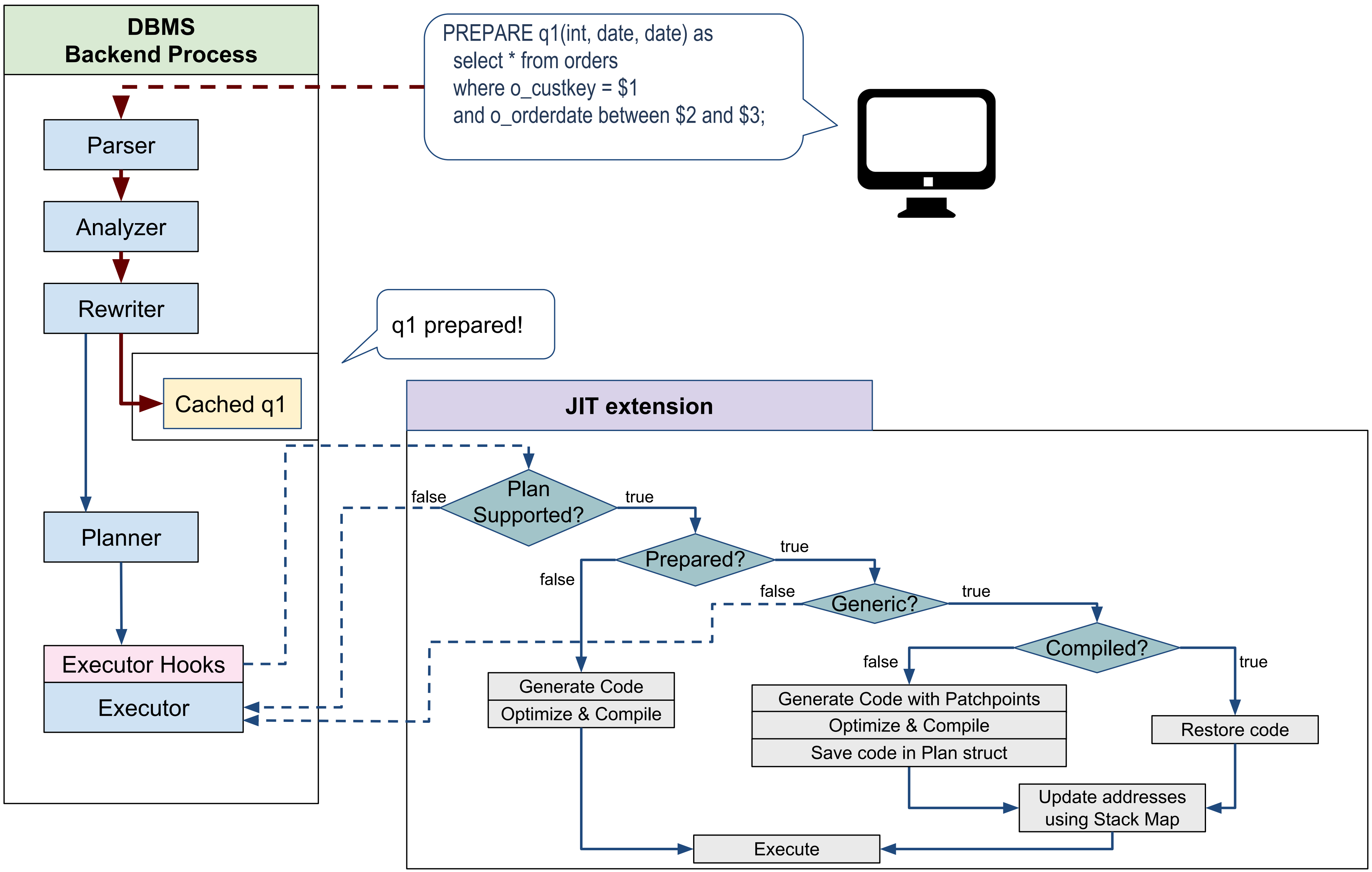
Machine Code Caching: High Level Scheme



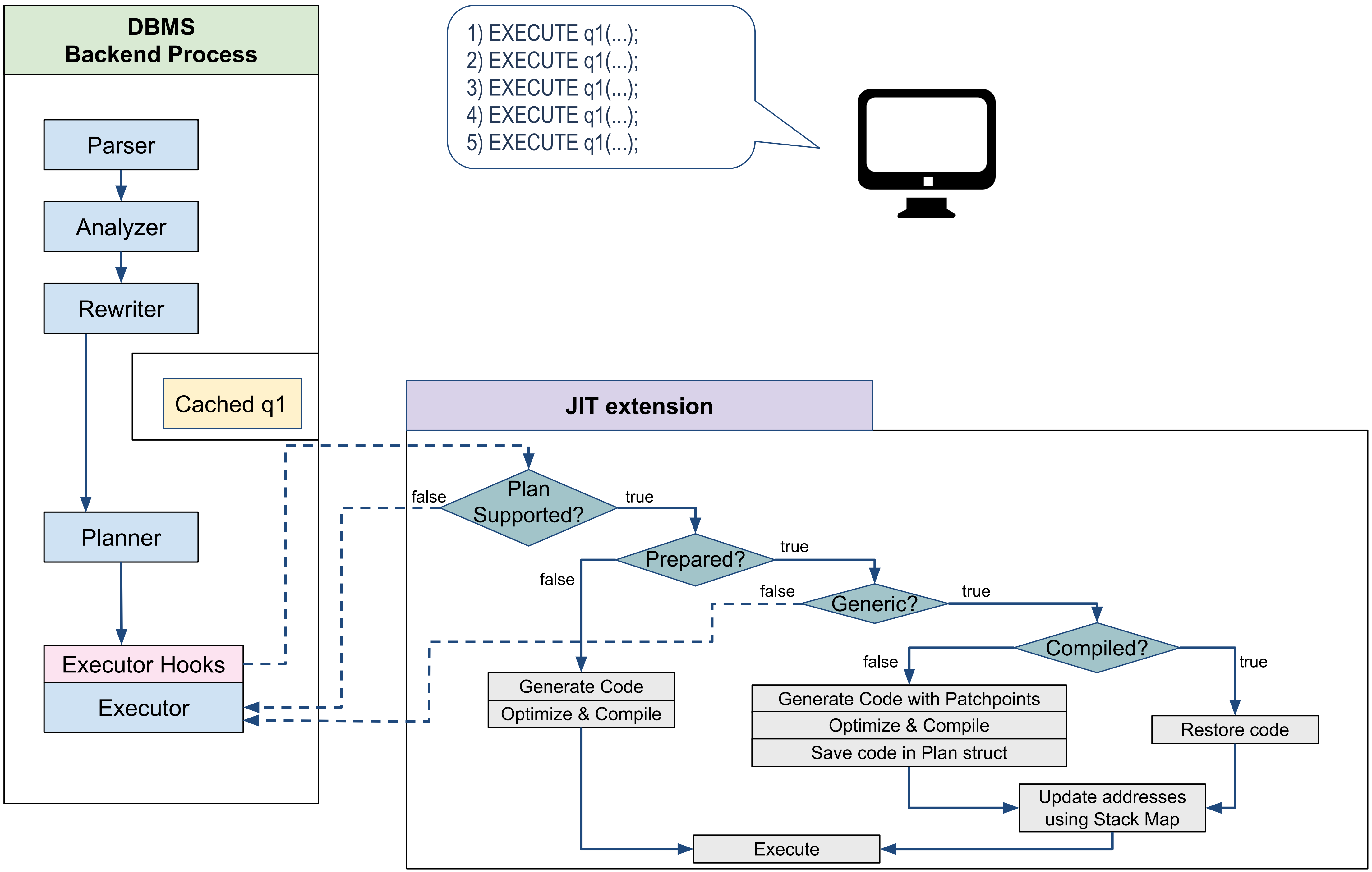
Machine Code Caching: High Level Scheme



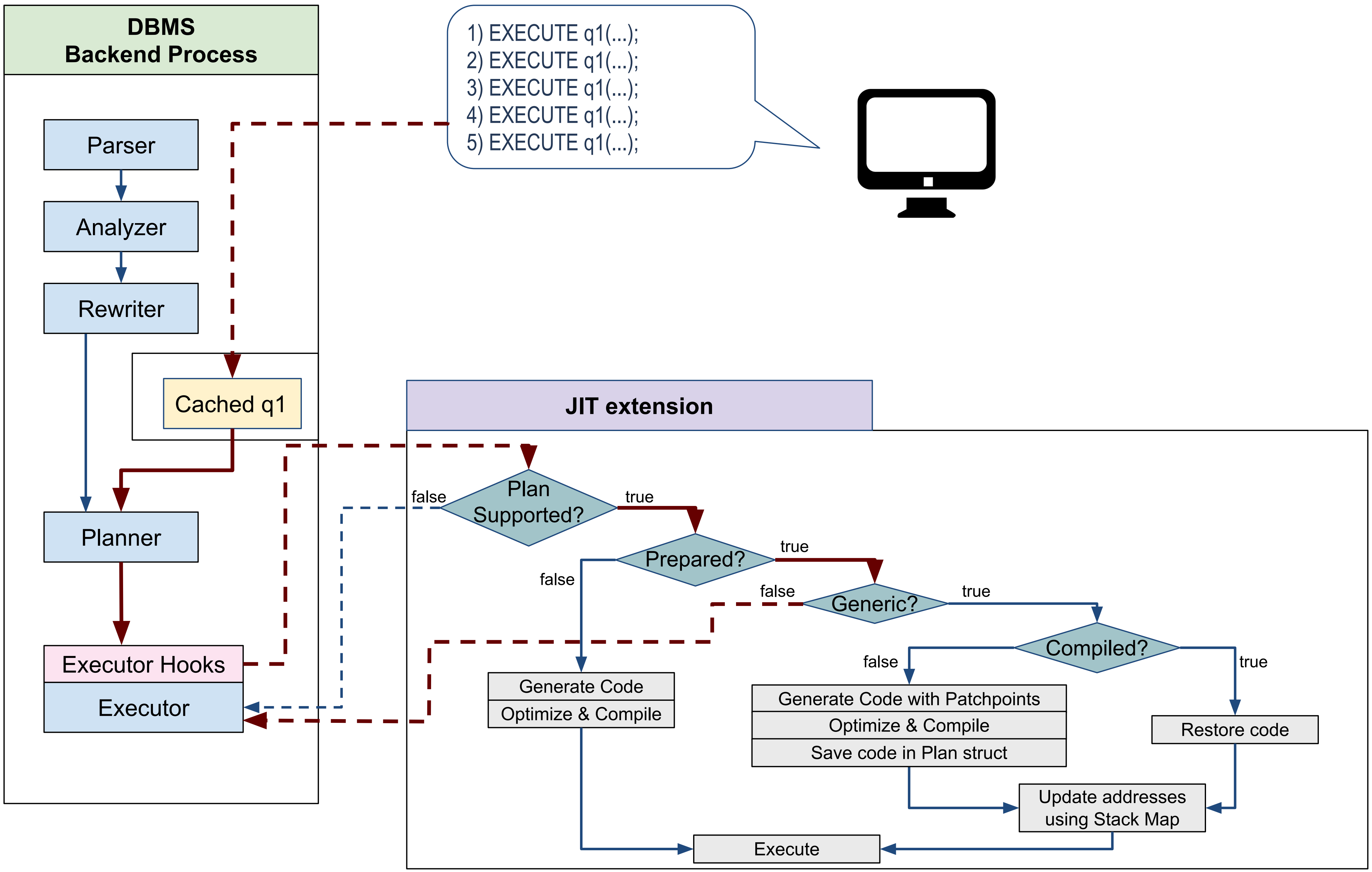
Machine Code Caching: High Level Scheme



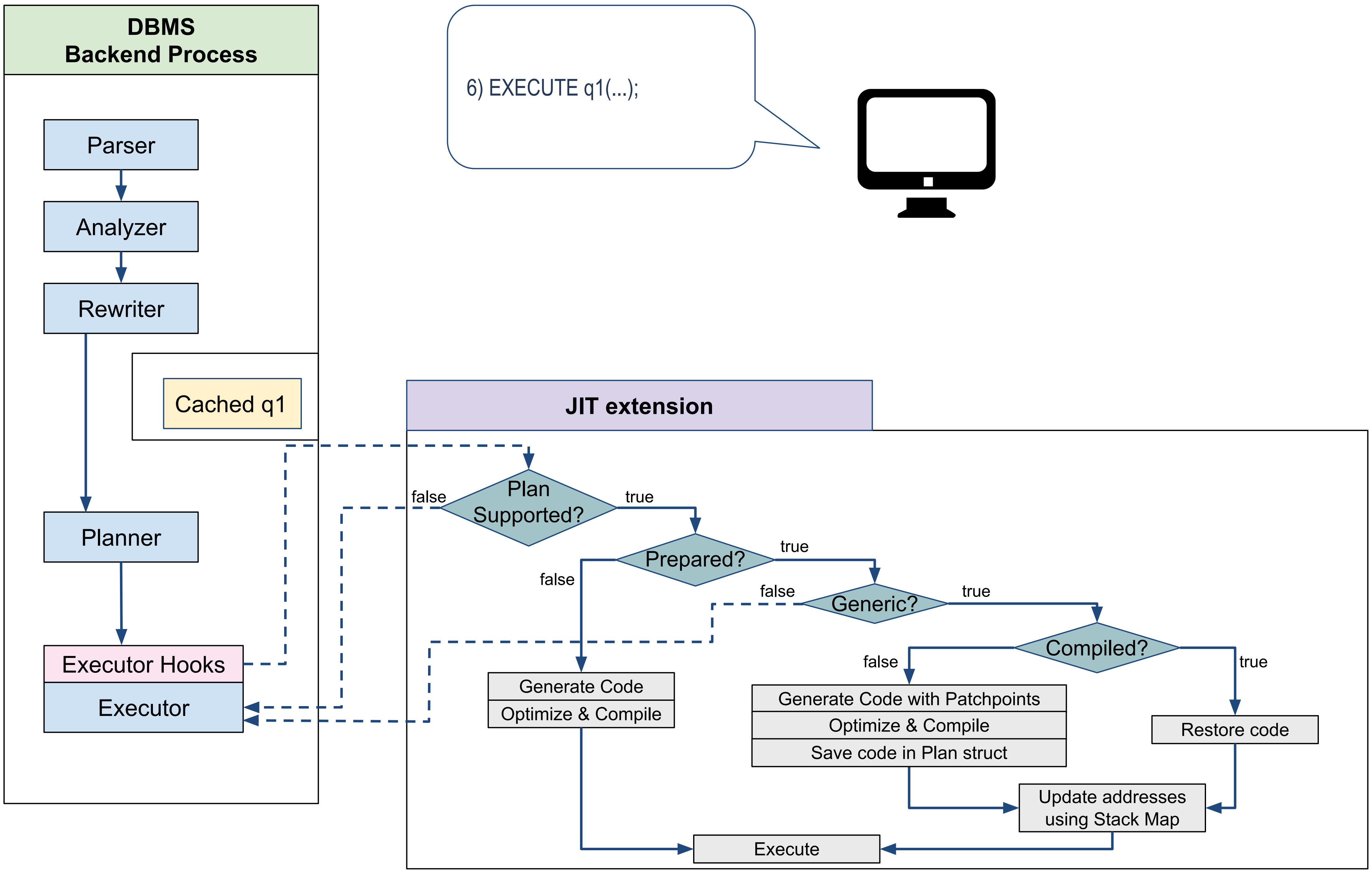
Machine Code Caching: High Level Scheme



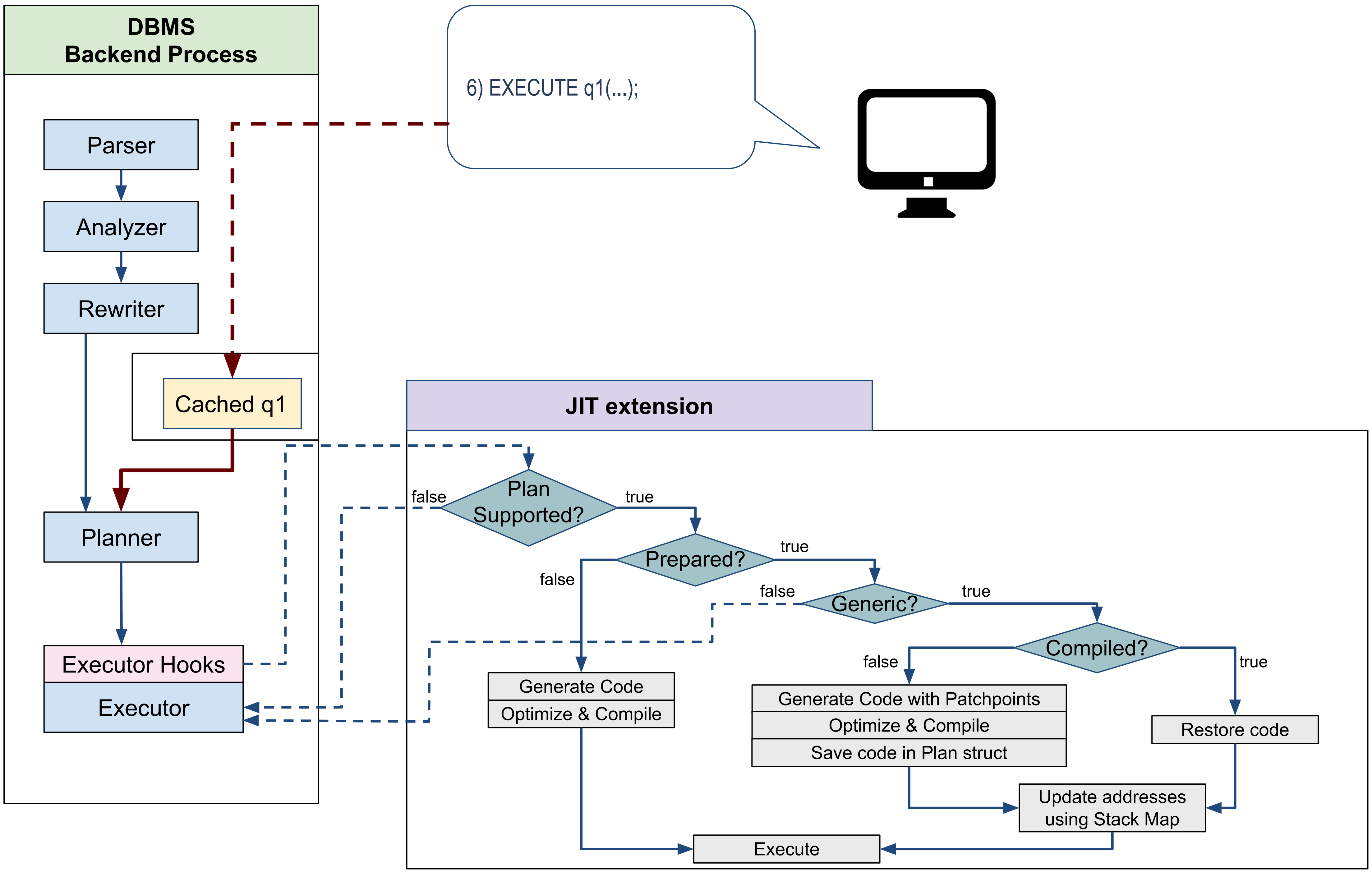
Machine Code Caching: High Level Scheme



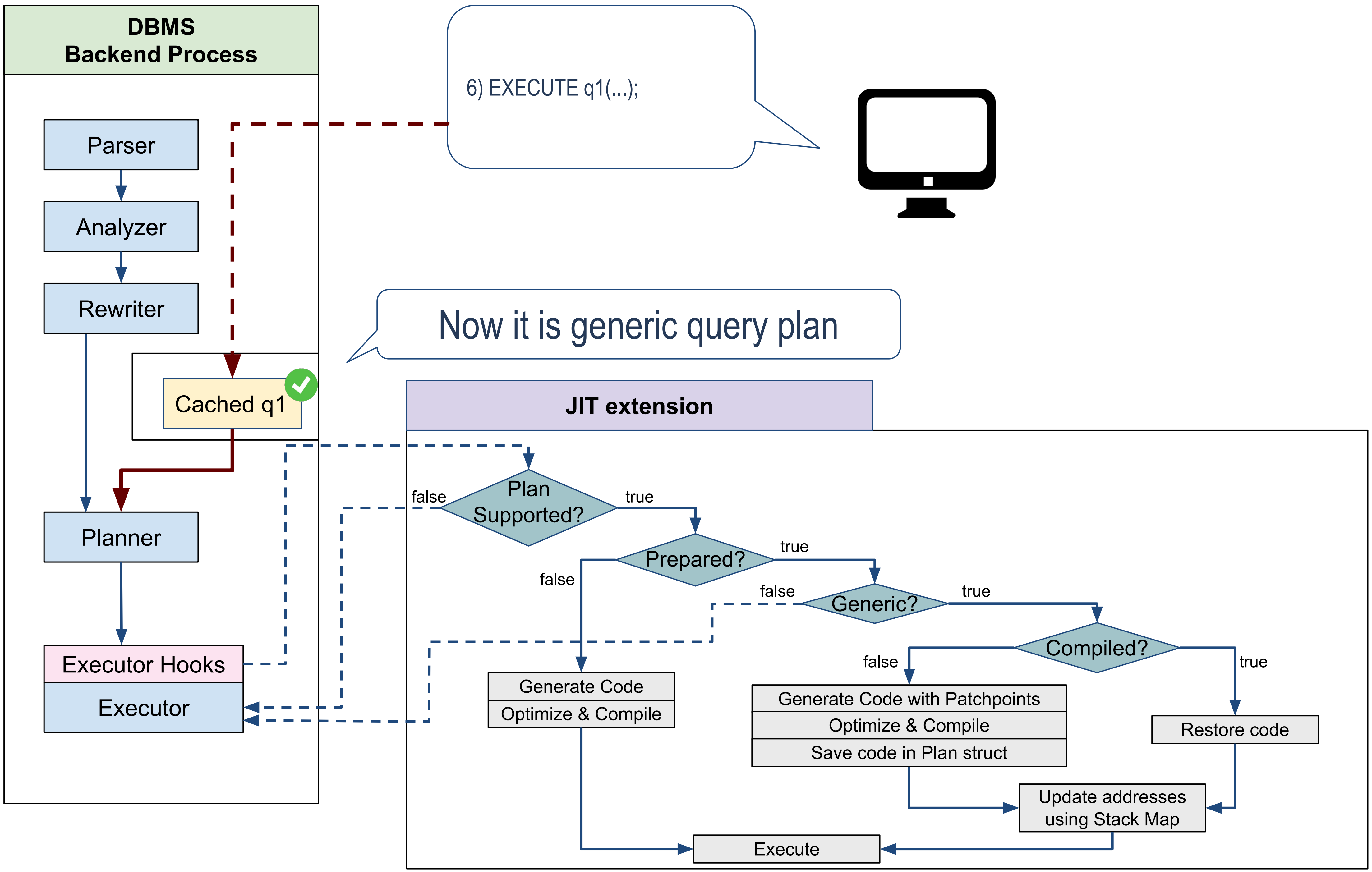
Machine Code Caching: High Level Scheme



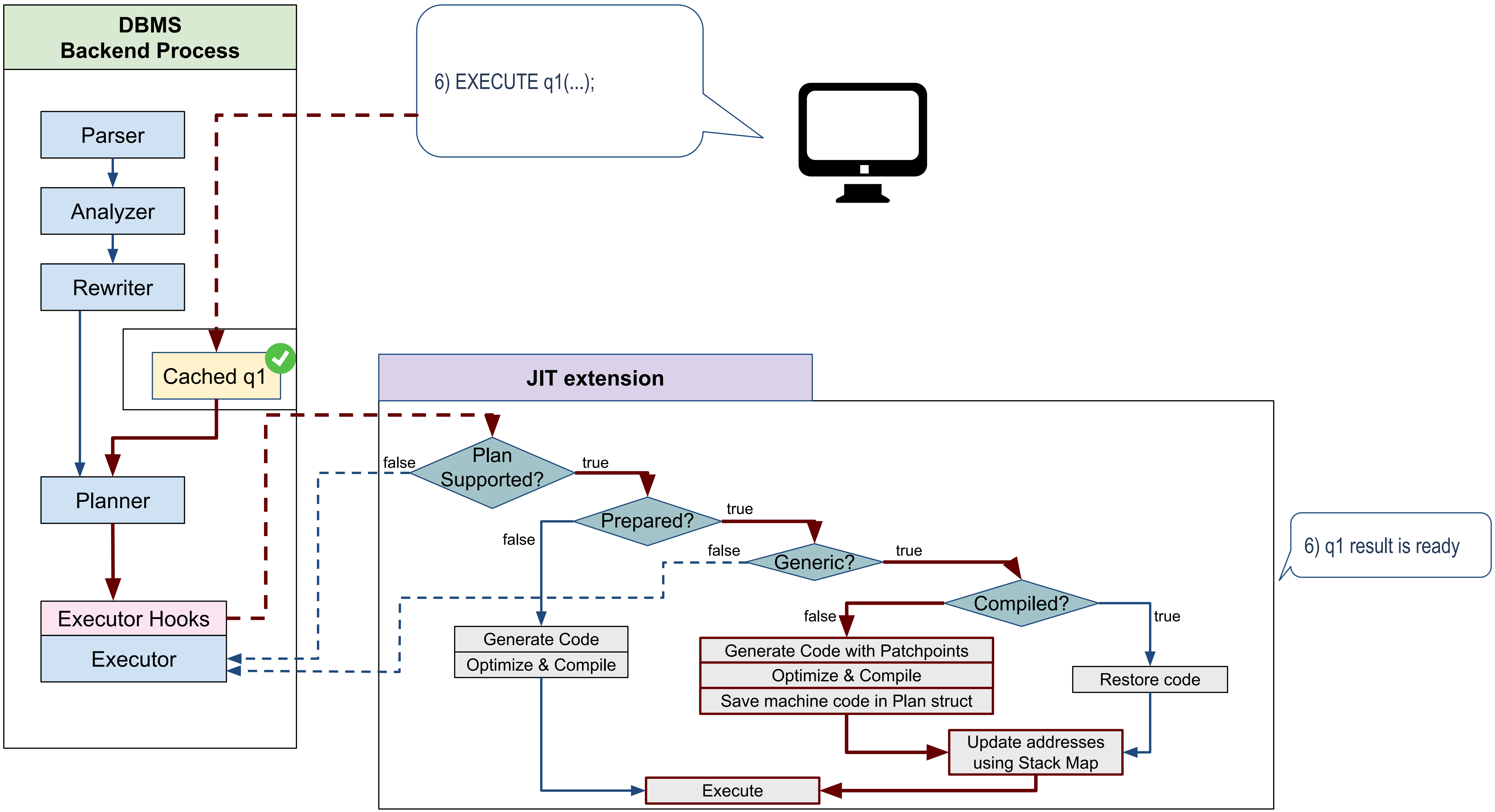
Machine Code Caching: High Level Scheme



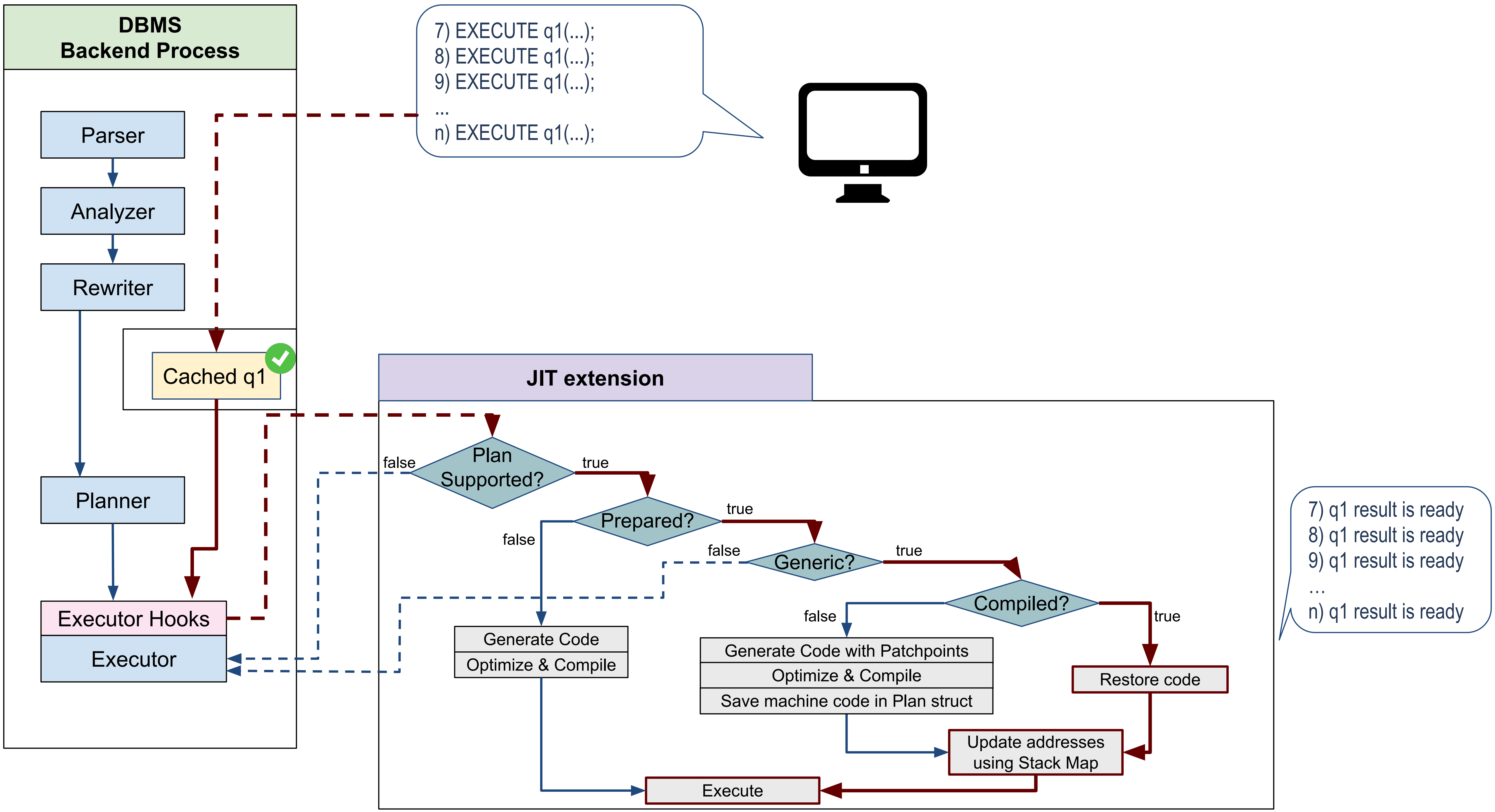
Machine Code Caching: High Level Scheme



Machine Code Caching: High Level Scheme



Machine Code Caching: High Level Scheme



Results: Code Quality Impact

Using of patchpoints restricts LLVM optimizations
=> considerably affects the quality of the machine code and the performance degrades.

The following is benchmark of TPC-H Q1:

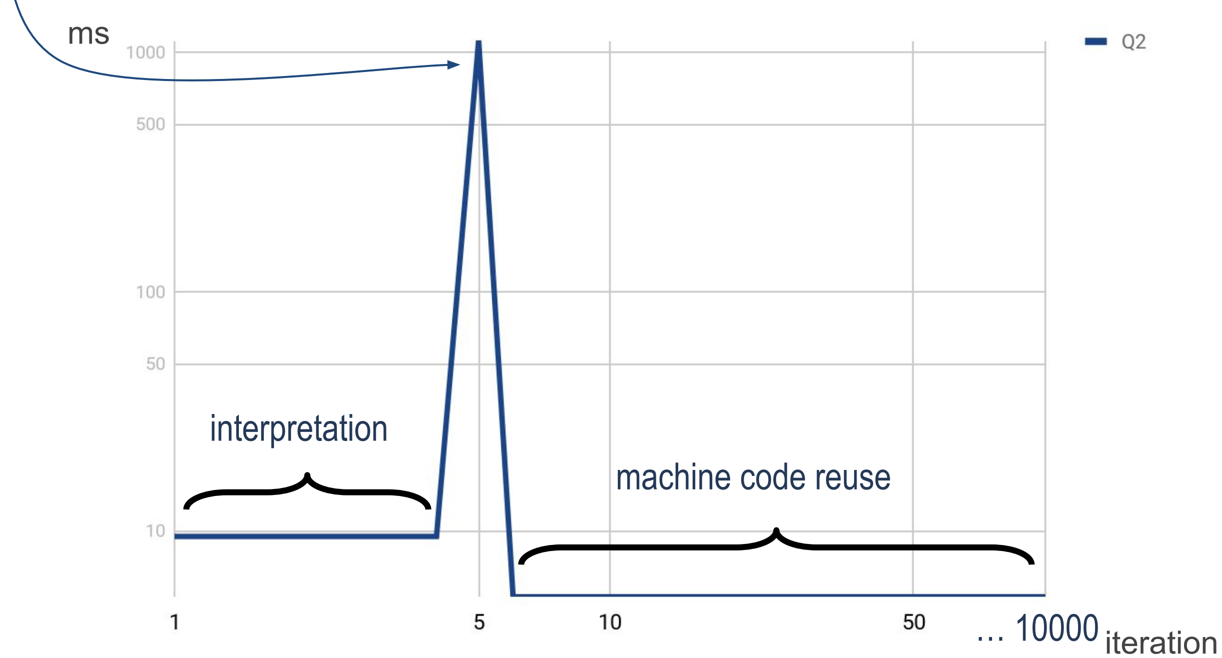
PG	JIT, one-time code generation			JIT with machine code caching (patchpoints)		
	compilation + optimization	execution	sum	compilation + optimization	execution	sum
10 s	(370 + 450) ms	**1,73 s**	2,65 s	(380 + 560) ms	2,4 s	3,4 s
				0,140 ms	**2,4 s**	2,4 s

PREPARE iteration

- Can save ~940ms on compilation and optimization, but the query runs ~670ms (38%) slower.
  Still the resulting code (which is slower than without patchpoints) is faster than the interpreter.
- To store query plan and machine code it takes 3-6 times more memory.

Results: TPC-H based OLTP like queries

Database: SCALE=2	Q1		Q2		Q3	
	PG	JIT	PG	JIT	PG	JIT
Avg TPS (more — better)	70,67	72,38	105,25	183,71	145,37	199,83
Generic plan compilation on 6th iteration + execution, ms	-	1342,5	-	1118,9	-	997,2
Avg execution time except first 6 iterations, ms	14,12	13,65	9,49	5,31	6,87	4,89
Avg improvement except first 6 iterations, X times	**1,03**		**1,78**		**1,40**	



Q1:
select c_custkey, c_name, c_phone, c_acctbal,
+17 more fields
from customer
join orders on c_custkey = o_custkey
+4 more joins
where c_custkey between :bid1 and :bid1 + 20
order by o_orderdate desc;

Q2:
select l_returnflag, l_linestatus,
+8 more aggregate functions
from lineitem
where l_shipdate <= date '1998-12-01' - interval '105 days'
and l_partkey between :bid1 and :bid1 + 200
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus;

Q3:
select l_returnflag, l_linestatus,
+3 more aggregate functions
from lineitem
where l_shipdate <= date '1998-12-01' - interval '105 days'
and l_partkey between :bid1 and :bid1 + 200
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus;

Questions?