Just-in-time Compilation in SQLite using LibJIT

FADHIL ABUBAKER

1 INTRODUCTION & OBJECTIVES

Just-in-time (JIT) compilation is a technique that involves compilation of code dynamically after a program has started execution. This is opposed to ahead-of-time (AOT) compilation that compiles code statically prior to run-time [2]. JIT compilation is used for improving the performance of interpreters, where run-time information about the executing program is used to apply further optimizations. Notable interpreted run-times that perform JIT compilation include the Java Virtual Machine (JVM), the V8 JavaScript engine and the LuaJIT run-time for the Lua programming language.

Recent work has looked at adapting JIT compilation techniques to the query execution engine of a database management system (DBMS) [8, 11, 15]. A DBMS can be considered as an interpreter for SQL, since it must process generic queries against generic schemas. However at run-time, the exact nature of the schema and query are known. JIT compilation in a DBMS involves using this run-time information to generate efficient machine code that processes a particular query against a particular schema.

Most DBMSes that perform JIT compilation use the LLVM [10] compiler framework for run-time code generation. LLVM's statically typed intermediate representation and numerous optimization passes make it ideal for implementing JIT compilation. By bundling LLVM and using its On-Request-Compilation (ORC) APIs, DBMSes can generate native code for query execution.

However, there is a class of lightweight DBMSes that cannot afford to bundle heavyweight dependencies such as LLVM: the embedded DBMS. An embedded DBMS is integrated directly into the application and runs in the same process. This is opposed to a client-server DBMS that runs in a separate process (and possibly on a separate machine) than the application. Embedded DBMSes are built to have a small footprint since they have to be embedded inside applications. An example of an embedded DBMS is SQLite [17], which has a library size of 750KB. Comparatively, LLVM has a library size of 65MB, making it unsuitable for driving JIT compilation in a DBMS like SQLite.

An alternative to LLVM for use in embedded DBMSes are purpose-built JIT frameworks. Since LLVM is mainly a static AOT compiler, it comes with a lot of bloat that is unnecessary for JIT compilation. As a result, purpose-built JIT frameworks tend to be lighter since they only contain modules for JIT compilation. An example of such a JIT framework is LibJIT, which has a library size of around 600KB.

In this project, we explore JIT compilation in the SQLite embedded DBMS using the LibJIT library. The objective of the project is to implement interpreter loop unrolling and arithmetic operation inlining in SQLite using LibJIT. Benchmarks are then run to see how much performance is gained. Due to time constraints, only simple arithmetic queries, such as `SELECT 1+2*3-4`, are JIT compiled. Evaluation results show JIT compiling such queries in SQLite offers good performance, however, these are offset by the cost of compile times.

```
sqlite> EXPLAIN SELECT l_extendedprice + 1 FROM lineitem;
addr  opcode         p1    p2    p3    p4    p5
----  -------------  ----  ----  ----  ----  ----
0     Init           0     8     0           0
1     OpenRead       0     9     0     6     0
2     Rewind         0     7     0           0
3       Column       0     5     2           0
4       Add          3     2     1           0
5       ResultRow    1     1     0           0
6     Next           0     3     0           1
7     Halt           0     0     0           0
8     Transaction    0     0     8     0     1
9     Integer        1     3     0           0
10    Goto           0     1     0           0
```

Fig. 1. VDBE program for a simple analytical query.

2 BACKGROUND

2.1 Just-in-time Compilation in Database Management Systems

Traditional DBMS architectures are designed to overcome disk I/O as the main bottleneck. However, as RAM became cheaper, it is now possible to fit the entire working set of a database in memory. As a result, this shifts the bottleneck from disk I/O to CPU. To overcome the CPU bottleneck, DBMSes now have to execute fewer instructions during query processing. In modern database literature, JIT compilation is one technique for speeding up queries by executing fewer instructions.

We now give a quick history of JIT compilation in DBMSes. In 2006, Rao et al. first explored compiling SQL queries to Java bytecode using the JVM [15]. The authors of [8] take this further by compiling relational operators to C++ code using operator templates. Finally, in [11], the authors demonstrate HyPer, a DBMS that JIT compiles query plans using LLVM. HyPer introduced techniques that would influence future work, namely, a query execution architecture amenable to JIT compilation as well as using LLVM for code generation.

Production-grade DBMSes also use JIT compilation to speed up query processing. The PostgreSQL DBMS uses LLVM to accelerate expression evaluation and for transforming on-disk tuples into their in-memory representation [13]. QuestDB, an open-source time-series DBMS uses the asmjit [1] library for speeding up filter evaluation in queries with `WHERE` clauses [14]. Finally, the Hekaton engine in Microsoft SQL Server translates a query plan into succinct C/C++ code, which is then compiled into native code for execution [4].

2.2 SQLite

Most DBMSes typically process a SQL query by first parsing it and then generating a logical query plan with relational operators as

```
sqlite> EXPLAIN SELECT 1*3-4/2+7;
addr  opcode         p1    p2    p3    p4    p5
----  -------------  ----  ----  ----  ----  --
0     Init           0     4     0           0
1     Add            3     2     1           0
2     ResultRow      1     1     0           0
3     Halt           0     0     0           0
4     Integer        1     5     0           0
5     Integer        3     6     0           0
6     Multiply       6     5     4           0
7     Integer        4     5     0           0
8     Integer        2     7     0           0
9     Divide         7     5     6           0
10    Subtract       6     4     2           0
11    Integer        7     3     0           0
12    Goto           0     1     0           0
```

Fig. 2. VDBE program for a simple arithmetic query.

nodes [3]. The logical query plan is then translated into a physical execution plan and executed in an interpreted manner [7].

SQLite deviates from this architecture and has a unique query execution style: rather than producing physical query execution plans, SQLite generates bytecode that is then executed by a register-based virtual machine called the Virtual Database Engine (VDBE). The bytecode consists of low-level opcodes that are the building blocks of data processing in SQLite. Fig. 1 shows a VDBE program consisting of opcodes for a simple analytical SQL query. For instance, the `Column` opcode extracts a particular column from a row and stores it in a virtual register. Opcode dispatch is implemented as large switch statement that cases on each one of the 186 opcodes. A full list of the opcodes can be found in [16].

2.3 LibJIT

LibJIT is a purpose-built JIT framework, originally developed for use in the now discontinued DotGNU project [9]. LibJIT aims to provide language-agnostic JIT functionality, and provides APIs that are independent of any particular bytecode, language or runtime. It has its own intermediate representation based on a three-address code. LibJIT comes with its own register allocator and supports only basic optimizations such as dead code elimination and copy propagation. While LibJIT is not an industrial-grade compiler framework like LLVM, its small library footprint of 600KB makes it appropriate for implementing JIT compilation in SQLite.

3 IMPLEMENTATION

This section expands on the details of the JIT implementation. The code is contained in `src/vdbe-jit.c`, with modifications made to VDBE execution in `vdbeapi.c:740-758`.

3.1 Translating SQLite structs to LibJIT IR

SQLite uses various internal struct definitions for executing VDBE operations. These have to be translated into LibJIT IR for use in LibJIT's APIs. There are two definitions that are crucial for the JIT implementation. The first is `Vdbe`, which is used to hold the state of a VDBE program such as the opcodes making up the program and their operands. The second is `Mem`, which represents a single virtual register used by the virtual machine.

The `Vdbe` struct contains many other struct definitions embedded within it, and so translating it into LibJIT IR requires translating each of these other struct definitions. Translating `Mem` is a simpler task, as it is simply a union of five primitive types: a double, a long, an int and two void pointers. The LibJIT translation of `Mem` can be found in `src/vdbe-jit.c:32`. Finally, since we are only interested in arithmetic queries in this project, it is possible to JIT compile such queries without using the `Vdbe` struct.

3.2 Embedding constant pointers and values into JIT code

The opcodes of a VDBE program process a query by reading and writing to virtual registers. These registers are allocated prior to execution. However, it is necessary to embed the memory addresses allocated by the host program for use in the JIT code. This is done by treating the memory addresses as constant pointers in the JIT code; see `src/vdbe-jit.c:46`.

Certain VDBE opcodes write constant values and their types into specific registers. For example, the `Integer` opcode writes the constant represented by the `p1` operand into the register indicated by the `p2` operand. In fig. 2, the 4th opcode is an `Integer` opcode that writes the constant 1 into the 5th register. In the JIT implementation, these constants are directly embedded into the virtual register without having to read from their respective operand during execution time.

3.3 Inlining arithmetic operations

SQLite has five polymorphic arithmetic opcodes: `Add`, `Subtract`, `Multiply`, `Divide`, `Remainder`. These opcodes perform their operations on the values stored in the registers indicated by the `p1` and `p2` operands and stores the result in the register indicated by the `p3` operand. As an example, in fig. 2, the 1st opcode is the `Add` opcode, which reads values from the 3rd and 2nd registers, adds them and stores the result in the 1st register.

These opcodes are polymorphic in that they work on both integer and real types in SQLite. These opcodes perform type checks for the values in the registers at run-time and calls the appropriate function for that operation. For example, adding two integers is done by the `sqlite3AddInt64` function.

During JIT compile time, the appropriate arithmetic operation can be hoisted based on compile time type checks and can also be inlined. In the example above, after specifying typing, adding two integers is done using the `jit_insn_add` method in LibJIT.

3.4 Unrolling VDBE interpreter loop

The VDBE interpreter uses switch-based opcode dispatch to execute VDBE programs. Switch-based opcode dispatch is simple to implement, however is inefficient due to poor code locality and increased branch mispredictions [5]. In the case of SQLite, the VDBE program representing a query and the VDBE interpreter are separate. It is possible to combine the VDBE program and its execution into a single code block, yielding better code locality and branch prediction. The JIT implementation does this by performing a pass over the

```
function 0x55653832E340() : void
.L:
        store_relative_long(93893222899968, 1, 0)
        store_relative_long(93893222900024, 3, 0)
        l12 = load_relative_long(93893222900024, 0)
        l13 = load_relative_long(93893222899968, 0)
        l14 = l12 * l13
        store_relative_long(93893222899912, l14, 0)
        store_relative_long(93893222899968, 4, 0)
        store_relative_long(93893222900080, 2, 0)
        l17 = load_relative_long(93893222900080, 0)
        l18 = load_relative_long(93893222899968, 0)
        l19 = l18 / l17
        store_relative_long(93893222900024, l19, 0)
        l20 = load_relative_long(93893222900024, 0)
        l21 = load_relative_long(93893222899912, 0)
        l22 = l21 - l20
        store_relative_long(93893222899800, l22, 0)
        store_relative_long(93893222899856, 7, 0)
        l24 = load_relative_long(93893222899856, 0)
        l25 = load_relative_long(93893222899800, 0)
        l26 = l24 + l25
        store_relative_long(93893222899744, l26, 0)
.L:
end
```

Fig. 3. LibJIT IR for VDBE program in fig. 2

VDBE program and emitting the appropriate JIT instructions for opcodes. Fig. 3 shows such an example of a code block for the query in fig. 2 in LibJIT IR.

3.5 Executing JITed code

There are two opcodes that pause the execution of a VDBE program and return control to the user: `ResultRow` and `Halt`. The `ResultRow` opcode pauses VDBE execution to return a generated result to the end user. The `Halt` opcode stops VDBE execution and cleans up any intermediate state. In this project, we do not JIT the `Halt` opcode since it uses the `Vdbe` struct extensively. As a result, the execution of the `Halt` opcode is handled by the native VDBE interpreter. This means that in the JIT implementation, the initial execution of a query is done by JIT code (up until `ResultRow`) and the subsequent execution is handled by the interpreter (`Halt`).

This logic is implemented in `src/vdbeapi.c:740-759`. Depending on the program counter, either the JIT implementation (`sqlite3VdbeJITExec`) is executed, or the native interpreter is executed (`sqlite3VdbeExec`).

4 BUILDING & RUNNING

The provided build script is `build-842-project.sh`. Executing `build-842-project.sh` creates a directory called `842-build` outside of the current working directory. This directory contains two SQLite binaries, `sqlite3-jit` and `sqlite3-no-jit`. Note that the LibJIT library is already bundled as an archive file for `x86-64` (see `libjit.a`) and does not need to be separately installed.

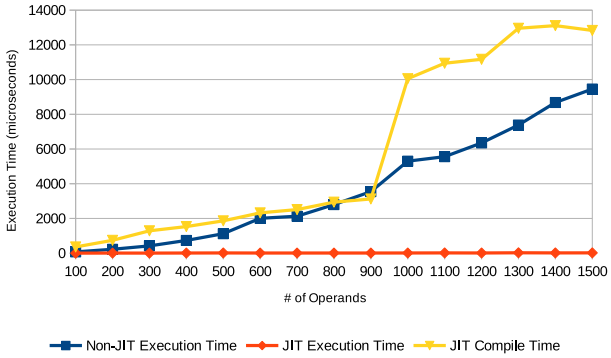

Fig. 4. Non-JIT vs JIT execution times with JIT compile times.

Either sqlite shells can be started by running `./sqlite3-jit` or `./sqlite3-no-jit`. Note that `sqlite3-jit` only supports arithmetic queries such as `SELECT 3*5+1`.

The provided benchmarking script is `run-842-benchmarks.py`. This script generates random arithmetic SQL queries, runs them through both SQLite implementations and returns benchmark numbers. The script takes as a single argument the number of operands in the generated SQL query. Note that the script requires at least `python 3.5` to work.

To see the LibJIT IR or assembler for the JIT compiled code, uncomment lines 270 and 285 in `src/vdbe-jit.c` and rebuild the project. Subsequent execution of queries will print the IR/assembler for that query.

See `README.md` for example usages of the scripts.

5 EVALUATION

The JIT implementation was evaluated on a machine with an Intel i7-7700HQ CPU @ 2.80GHz with 32GB of RAM. The provided benchmarking script was run while increasing the number of operands. Execution times were averaged over 5 runs. Table 1 shows the results of the evaluation, while fig. 4 plots the results.

The evaluation shows that JIT execution time significantly outperforms non-JIT execution as the number of operands increase. In fact, JIT execution remains relatively constant while non-JIT execution increases linearly with the number of operands. However, this speedup comes at the cost of compile times that also increase linearly with the number of operands.

Interestingly, JIT compile times were on par with non-JIT execution times up until 900 operands. There is a sudden spike in compile times for 1000 operands after which it resumes the linear trend. The spike is suspected to be a performance issue within LibJIT itself, however, the exact reason for the spike could not be determined due to time constraints.

6 DISCUSSION

From the evaluation we see that JIT execution times outpaces non-JIT execution times; however once compile times are factored in, JIT execution tends to be slower than non-JIT execution. This is because JIT compilation times are directly proportional to the number of opcodes in the VDBE program. Within the scope of this project, where

# of Operands	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500
Non-JIT Execution Time (μs)	74	223	418	733	1121	2008	2129	2797	3543	5301	5560	6350	7381	8679	9455
JIT Execution Time (μs)	2	5	4	7	12	9	8	8	8	10	14	11	21	16	18
JIT Compile Time (μs)	358	739	1287	1532	1862	2320	2510	2938	3121	10050	10941	11169	12956	13107	12831

Table 1. Non-JIT vs JIT execution times with JIT compile times.

workloads are restricted to simple arithmetic queries, JIT compiling SQL queries seems ineffective at improving query execution performance once compilation times are factored in.

However, queries that read from database tables typically contain hot loops which can be more effectively JIT compiled. For example, the query in fig. 1 reads the `l_extendedprice` column from each row in the `lineitem` table and adds 1 to it. The VDBE program for this query contains a hot loop consisting of the `Column`, `Add` and `ResultRow` opcodes. This loop is executed for each row in the `lineitem` table. A JIT implementation would only pay a one-time cost to compile these opcodes and reuse the compiled code for each iteration in the hot loop. This would result in JIT execution potentially outperforming non-JIT execution, even after factoring in compilation times. Implementing JIT compilation for such read-only queries requires translating more SQLite structs into LibJIT IR, and hence is out-of-scope for this project.

7 CONCLUSION

SQLite query execution does see a speedup from JIT compilation, however, these performance gains are largely eliminated due to excessive compilation times. One solution to overcome compilation times is to cache the compiled query and reuse it, eliminating compilation costs on subsequent queries. The generated code can be built to take arguments, which means parametrized SQL queries such as `SELECT * FROM table where column = ?` can be compiled once and executed multiple times.

Future work can look at compiling hot loops in queries that read from database tables. In [6], the authors identify that the `Column` and `Filter` opcodes consume the most CPU cycles when running the Star Schema Benchmark [12] on SQLite. JIT compiling only these opcodes can potentially accelerate query execution performance, while keeping compile times small. While this project looks at JIT compiling arithmetic queries, further investigation is needed to effectively implement JIT compilation for generic queries in SQLite.

REFERENCES

[1] asmjit. 2022. AsmJit — asmjit.com. https://asmjit.com/. [Accessed 14-Dec-2022].
[2] John Aycock. 2003. A Brief History of Just-in-Time. *ACM Comput. Surv.* 35, 2 (jun 2003), 97–113. https://doi.org/10.1145/857076.857077
[3] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (jun 1970), 377–387. https://doi.org/10.1145/362384.362685
[4] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1243âĂŞ1254. https://doi.org/10.1145/2463676.2463710
[5] M. Anton Ertl and David Gregg. 2001. The Behavior of Efficient Virtual Machine Interpreters on Modern Architectures. In *Euro-Par 2001 Parallel Processing*, Rizos Sakellariou, John Gurd, Len Freeman, and John Keane (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 403–413.
[6] Kevin P. Gaffney, Martin Prammer, Larry Brasfield, D. Richard Hipp, Dan Kennedy, and Jignesh M. Patel. 2022. SQLite: Past, Present, and Future. *Proc. VLDB Endow.* 15, 12 (sep 2022), 3535–3547. https://doi.org/10.14778/3554821.3554842
[7] G. Graefe. 1994. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.* 6, 1 (feb 1994), 120–135. https://doi.org/10.1109/69.273032
[8] Konstantinos Krikellas, Stratis D. Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. 613–624. https://doi.org/10.1109/ICDE.2010.5447892
[9] LibJIT. 2022. LibJIT - GNU Project - Free Software Foundation. https://www.gnu.org/software/libjit/
[10] LLVM. 2022. The LLVM Compiler Infrastructure Project. https://llvm.org/
[11] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (jun 2011), 539–550. https://doi.org/10.14778/2002938.2002940
[12] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. *The Star Schema Benchmark and Augmented Fact Table Indexing*. Springer-Verlag, Berlin, Heidelberg, 237–252. https://doi.org/10.1007/978-3-642-10424-4_17
[13] PostgreSQL. 2022. 32.1. What Is JIT compilation? — postgresql.org. https://www.postgresql.org/docs/current/jit-reason.html. [Accessed 14-Dec-2022].
[14] QuestDB. 2022. How we built a SIMD JIT compiler for SQL in QuestDB | QuestDB: the database for time series — questdb.io. https://questdb.io/blog/2022/01/12/jit-sql-compiler/. [Accessed 14-Dec-2022].
[15] Jun Rao, H. Pirahesh, C. Mohan, and G. Lohman. 2006. Compiled Query Execution Engine using JVM. In *22nd International Conference on Data Engineering (ICDE'06)*. 23–23. https://doi.org/10.1109/ICDE.2006.40
[16] SQLite. 2022. The SQLite Bytecode Engine. https://www.sqlite.org/opcode.html
[17] sqlite. 2022. Sqlite Home Page. https://www.sqlite.org/