Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Compiling SQLite queries to native code with LLVM

Author:
Thomas Kowalski

Supervisor:
Holger Pirk

Submitted in partial fulfillment of the requirements for the MSc degree in Computing of Imperial College London

Summer 2020

Abstract

Performance in data management systems has been a topic of interest for several decades. To this end, the latest state-of-the art query processing engines use just-in-time compilation and its benefits over traditional query plans to reduce processing time.
In this project, we introduce LLVMSQLite: a just-in-time compiler for SQLite, a widely-used embedded data management system. Using the LLVM/MLIR compiler infrastructure, we compile SQLite virtual machine bytecode programmes to native code at query execution time. We also propose a solution to generate more efficient code when prior knowledge on the column types is available by addressing the slowdowns incurred by the dynamic type system used in SQLite.
Our evaluation demonstrates that our just-in-time compiler can reduce query execution time by up to 40% on traditional OLAP benchmarking workloads when compared to the default SQLite interpreter. Still, we observe that the design of SQLite and its limitations, such as disk-residency and the limited versatility of its operator implementations prevent it from reaching the same performance as state-of-the-art databases.

Acknowledgements

I would like to express my gratitude to Holger Pirk, my project supervisor, for having this project idea in the first place and trusting me with it, but also for his feedback over the summer and more broadly for this whole year. It would not have been as interesting without his teaching methods and his involvement in my training and projects.

I also would like to thank my family Philippe, Christine and Léna for the support and the love they have given me throughout my life, and without who I would certainly not be writing these lines today.
I especially want to mention Mathilde, who has always been extremely supportive in my studies and has made quarantine a much happier time.

Finally, I would like to show appreciation to the LLVM Community as a whole for the invaluable support and tips they offered me during this project.
Especially, I must thank Lang Hames, LLVM contributor at Apple, who helped me a lot with performance issues in the LLVM JIT API.

Contents

Chapter 1

Introduction

Performance in data analysis software has been a topic of growing interest over the last decades. In half a century, the amount of information stored in data centres has grown exponentially – much faster than available processing power – and information is more present in our lives today than ever before.
From the birth of *big data* and its analytics to the latest datasets used by artificial intelligence companies, to the highly intricate processes developed by advertising companies to provide the best products to their most promising targets, processing large quantities of data efficiently is a prerequisite to many modern applications and businesses.

The end of the 20th century has seen the development of many commercial data management systems. These brought new guarantees – often known as ACID (Atomicity, Consistency, Isolation and Durability) – over the old file-based storage methods which, already at that time, could not handle the transaction rate required by emerging software. The features offered by modern data management systems have made these solutions increasingly complex and resource-hungry: they often rely on several processes to keep data consistent and to communicate with clients.
SQLite emerged in 2000 and took the opposite direction. It does not use a client-server protocol but rather runs as part of the software using it. It also stores all its data in a single *database file* instead of scattering it. SQLite offers features similar to more complex database products, but does so in a much lighter way and at the cost of a lower performance than the one offered by other systems.
The versatility of SQLite has made it extremely popular for many use cases: it is estimated that today, SQLite is the most used data management system in the world [49].

One of the most prominent solutions to run-time performance issues both in interpreted languages and data management systems is *Just-in-Time compilation* (or JIT compilation). Just-in-Time compilation consists in generating machine code at run-time, just before its execution. It is the opposite of *Ahead-of-Time compilation*, in which all machine code is produced when the software is built.
Just-in-Time compilation uses run-time knowledge to generate code specifically for one task, which is not possible using Ahead-of-Time compilation. It can thus draw on

the versatility of dynamic languages without suffering much from the performance costs they incur.

LLVM [29] is a set of tools designed to make state-of-the-art optimising compilers. It is the backbone of widely known projects such as Clang or Rust and has been used successfully to implement Just-in-Time compilers in the past [21, 36, 41].
The latest versions of LLVM include MLIR [30], an infrastructure and API that allows compiler developers to define intermediate representations for high-level languages, on which optimisations and transformations can be carried out from the LLVM/MLIR API.

The aim of this individual project is to create a just-in-time compiler for *SQLite prepared statements* using LLVM and MLIR. Behind this goal lies the question of whether the performance of SQLite can be on-par with state-of-the-art data management engines that use JIT compilation by default such as HyPer [36], when given similar means.

Chapter 2

Background

In this chapter, we cover the necessary background to design and implement a just-in-time compiler for SQLite. We present SQLite and its internals more in depth, we review several just-in-time compilers for interpreted languages and data management systems. We then introduce the LLVM compiler infrastructure and provide an overview of two standard analytical benchmarks for databases.

2.1 SQLite: an embedded data management system

2.1.1 Presentation and use cases

SQLite [19] is an embedded relational data management system. Whereas most data management systems communicate over the network using socket-based client-server protocols and require one or several separate daemon processes to execute queries issued by users, to manage information on disk and to keep data consistent, embedded databases can be thought of as a part of the software that uses them.

In an SQLite database, all the data is materialised and stored in a single *database file*. Database files are cross-platform, support several encoding schemes, can store any type of data – numeric, text or even raw bytes – allow to maintain indexes on relations and even support access by several concurrent processes – one of the advantages of databases over raw file storage – though with some limitations [55].

SQLite also offers full ACID support and, as a relational database, exposes a fully fledged API to prepare and execute SQL queries. The SQL dialect supported by SQLite is a subset of the Standard SQL-92 ([20]) [49]. Some features are omitted because they do not make sense in an embedded data management system – `GRANT` and `REVOKE` – some are because of the way SQLite stores its data and the cost they would have – `ALTER TABLE` only supports adding columns and changing their names for example [49].
The SQL dialect supported by SQLite also adds features that are not specified in the SQL standard, such as standard C date-time processing functions.

Finally, SQLite supports database encryption and a wide range of plug-ins and extensions that make it more suitable for certain workloads and use cases.

The versatility, ease of use, simplicity of deployment and loose license of SQLite have made it the storage system of choice for many use cases. Many Android applications have used it as their primary storage API and Room[1], a Google-made ORM for SQLite, is now the recommended method for storing data on newer Android applications.
SQLite is also used in several desktop applications, such as Mozilla Firefox – for both user preferences and plug-in data storage, for which a JavaScript API wrapper exists[2] – or Adobe Lightroom which uses it to store all user data (*Catalogs*) in a space-efficient way that also supports quick retrieval.
Constrained environments are another use case for which SQLite is a relevant storage solution: it is lightweight, does not require auxiliary processes nor even threading and provides fast access to data when used right. That is why SQLite is also chosen for many commercial embedded devices but also in Internet of Things research projects [16, 54].
The development of data science has popularised the SQLite database format for raw data transfers. As previously explained, it is adequate for storing large quantities of raw information and making it available to others, as it solves the common issue of incompatible encoding between platforms. It also is more space-efficient than text formats like CSV or JSON: SQLite uses binary representations for numerical types and compresses their on-disk representation, using only the narrowest possible types.
Furthermore, when working on large datasets, using SQLite as a database system efficient data preparation possible. SQLite is implemented in C and makes filtering entries, computing aggregates, or intermediate results more efficient than carrying out equivalent operations in a high-level scripting language, such as Python.
Many other SQLite use cases exist: small dynamic websites, temporary data storage and cache – thanks to its ability to hold small databases entirely in main memory, SQL databases education and training and even file archiving, for which SQLite has official support since version 3 [49, 28].

Today, it is estimated that SQLite is the most used data management system in the world [49]. Thanks to its permissive license agreement – SQLite is public domain – many vendors use it and deploy it along with their software products or SDKs, on an immeasurable number of devices. From laptops to smartphones, to smart-cars, to servers, to planes [53]. Every piece of software that needs to store any kind of data is a potential SQLite user.

[1] See `https://developer.android.com/training/data-storage/room`
[2] https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM/Storage

```
> CREATE TABLE Test(I INTEGER, F REAL, S TEXT);
> INSERT INTO Test(I, F, S) VALUES ("Integer", "3.14", 10);
> SELECT * FROM Test;
Integer|3.14|10
```

Figure 2.1: Example of manifest typing in SQLite

2.1.2 Data typing in SQLite

Many relational data management systems – if not all [49] – use *static rigid typing*. That means that the values inserted in a relation must match the type declared for their column in the relation schema. SQLite uses an alternative type system called *manifest typing*.

SQLite supports five basic data types: `INTEGER`, `REAL`, `TEXT` (for strings of arbitrary length), `BLOB` (for arbitrary-length byte arrays) and `NULL`. As opposed to many other database products, it does not differentiate integers of different types or signedness (`INTEGER`, `SMALLINT`, `BIGINT UNSIGNED`), nor floats and doubles. It also makes all strings necessarily text (no `VARCHAR`-like type is available). Finally, SQLite does not implement a boolean type nor a date-time type – the official documentation recommends using a UNIX timestamp, the Julian day representation or a serialised string representation for storing date and times [49]. The reason to this is manifest typing.

In SQLite, although columns are typed, their types are only a hint – called an *affinity* – for the database engine, never a constraint. In other words, it is correct in SQLite to insert a string in a column typed `INTEGER`. Depending on the text, it may be converted to the equivalent integer value or simply kept as a string. In short, the philosophy of manifest typing is that the type of the values in a tuple does not depend on the types defined in the relation but primarily on the data itself [18].

The column type affinity, though, is still useful. First, when inserting values, if the type of the provided value does not match the column type, SQLite will try and convert it into the expected type. That is why inserting `"3.14"` in a `REAL` column converts it into a float. Similarly, inserting `3` in a `REAL` column will yield `3.0`. In the event where it is not possible to store the value without losing precision (like storing `3.1415` as an `INTEGER` or `"Foo"` as a `REAL`), SQLite will store the value as-is.

Fig. 2.1 illustrates these principles. Although the type of column `I` is `INTEGER`, inserting the string `"Integer"` into it does not cause an error, as a result of manifest typing. Similarly, inserting the string `"3.14"` into `F` (declared `REAL`) does not cause an error as SQLite uses the type hint and successfully converts it to a float.

At run-time, SQLite sometimes needs to convert Values to other types. This is done by calling the `applyAffinity` function. `applyAffinity`, as its name suggests, applies an *affinity* to the type *i.e.* it *attempts* to make it match the target type: the operation

```
{
  u: 3.1415
  flags: Real | Str
  z: "3.1415\0"
  n: 6
  // Other properties
}
```

(a) A Real value

```
{
  u: (unspecified)
  flags: Null
  z: nullptr
  n: (unspecified)
  // Other properties
}
```

(b) A NULL value

```
{
  u: (unspecified)
  flags: Str
  z: "GERMANY\0"
  n: 7
  // Other properties
}
```

(c) A String value

Figure 2.2: Example in-memory representations of SQLite Values

is not guaranteed to succeed (for instance, converting `"monday"` to a numerical value will not).

Manifest typing offers a lot of flexibility, particularly in applications that are not data-centric. In desktop applications that use an SQLite database for storing user preferences, it is desirable to store entries of differing types in the same column, and to proceed to type-checks and sanitation directly in the application.

In practice, to allow tuples to hold values of any type, SQLite uses a single data structure, the *SQLite Value* (`sqlite3_value` in the source code and sometimes simply *Value* in the remainder of this report).
An SQLite Value is a C structure that holds a 64 bits wide union – used to hold the numeric value when there is one, a C-string – the text representation of the numeric value or the string held, a type flag integer – which indicates what type(s) the Value currently holds – for example an integer and its string representation – and metadata (encoding, length of string buffer, *etc*.). Fig. 2.2 displays examples of SQLite Values in memory, Appendix 6.2 implements a new `Print` VDBE operation, which relies on the Value data structure to pick the right format.
Manifest typing has a high memory footprint that is inherent to the in-memory representation of Values. Indeed, in a statically typed system, storing a 32 bits integer takes four bytes of memory. In SQLite, integers are necessarily 64 bits wide, and the SQLite Value data structure itself occupies 56 bytes of memory. In some cases, a text representation of the number is also stored, which leads to an even higher memory footprint.

2.1.3 Data access in SQLite

Data storage and page caching

As mentioned earlier, all the data in SQLite is stored in a single database file. The file holds user data (in a 'row-store' fashion, *i.e.* tuples are serialised and stored in contiguous blocks, rather than first split into their column components and stored in various places like a 'column-store' or decomposed storage) but also system data such as indexes or database schema [18]. Data is decomposed into pages on disk, which are read and written by the Pager module.

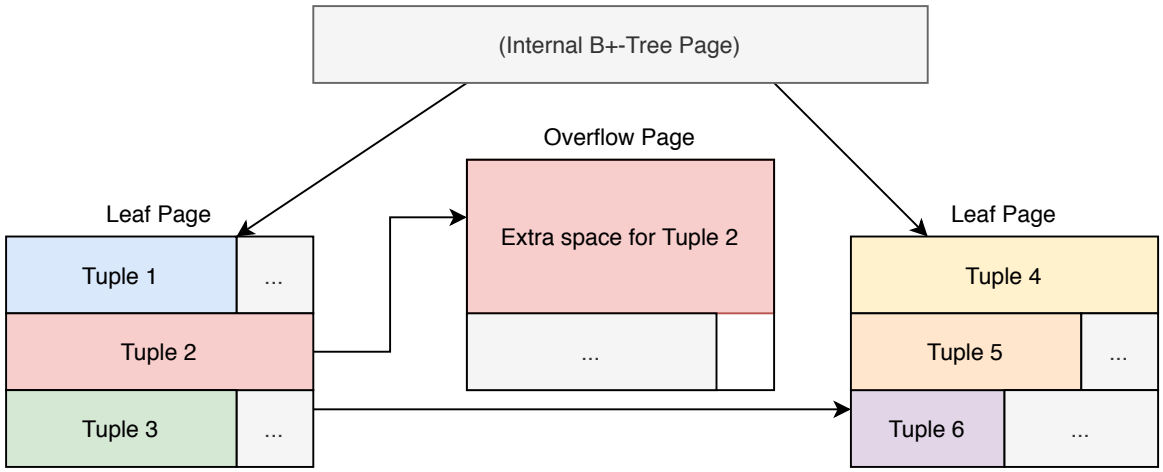


Figure 2.3: Schematic view of various kinds of B^+-Tree node pages

Four page types exist: *navigation pages* (for internal B^+-Tree nodes), *leaf pages* (for nodes that store data), *overflow pages* and *free pages*.

Leaf pages hold a certain number of tuples, reserving some space for each. All tuples are stored at least partly in a leaf page.

Overflow pages are used to store additional data for a tuple. If a tuple is too large to fit in its reserved leaf page space, the remaining data *spills* to an overflow page [18]. If needed, overflow pages may refer to another overflow page in a linked-list fashion.

Free pages are pages that have been used in the past to store data which has since then been deleted. When a tuple is deleted in SQLite, the pager does not return the corresponding pages to the operating system but rather keeps them in the *free list* as a reserve for later. Indeed, writing tuples to an free/unused page already in the file is much less costly than triggering a system call to expand the file on disk.

Fig. 2.3 shows a partial SQLite B^+-Tree. The top internal B^+-Tree page points to two leaf pages. The left leaf page contains three tuples, among which one spills to the middle overflow page. The right leaf page contains three tuples that all fit in their allocated buffer.

The Pager also holds certain pages in cache. When a page is read from memory, the Pager first stores it into a in-memory cache before providing it to the B^+-Tree module. This cache reduces the number of disk reads – supposedly much slower than RAM reads – for repeated accesses to the same pages. The cache uses an LRU eviction rule and is based on a hash table.
Still, although it supports it, SQLite is not designed to be an in-memory database. Even though the cache can theoretically be expanded as much as memory allows it, the engine is not designed to use RAM as its primary storage and as a result does

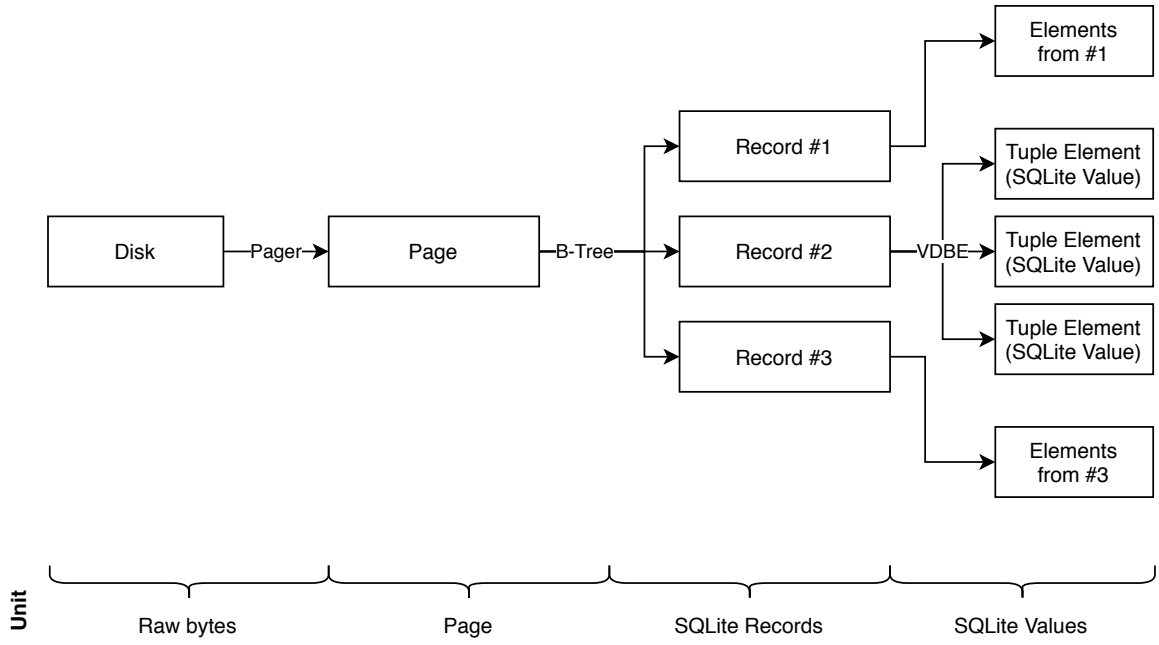


Figure 2.4: Data parsing in SQLite from disk data to SQLite Values

not use techniques in-memory databases use nor draw from RAM-only features from which the performance can benefit greatly.

Page parsing

Query and data processing in SQLite are split between various layers. Each layer has a limited visibility and shows the different levels of abstraction. Fig. 2.4 illustrates the parsing process.
When SQLite scans a table, the Pager module reads the data from the disk and updates the pages present in cache. The Pager then provides the raw page (in the form of a byte array) to the B^+-Tree module.
The B^+-Tree module 'knows' how to interpret a part of these sequences, their header. The header provides the necessary information to locate the start of each *record* in a page and to keep the invariants of the tree. The remainder of the payload is handed over to the next layer up, the Virtual Database Engine (VDBE), a domain specific virtual machine which is used to process queries (See 2.1.4).
It is the VDBE that parses records from their raw byte-sequence form and transforms them into Values. This requires some bookkeeping: all SQLite Values are allocated on the heap as well as their variable-length fields (`TEXT`- and `BLOB`-typed Values) whose contents need to be copied from the page cache to a non-ephemeral memory location.
As a result, the SQLite Value data structure implementation holds additional fields that describe allocated memory, the attached database using them (to define when to release the resources), what function to use to free memory, *etc.*.
Note all SQLite Values hold space for these values, but not all use them. To reduce

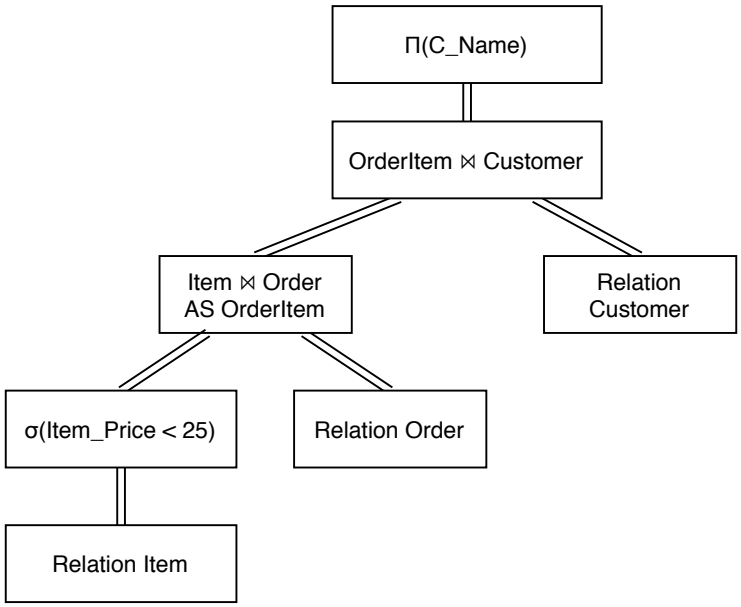


Figure 2.5: Example query plan for query `SELECT C_Name FROM (Item NATURAL JOIN Order AS ItemOrder) NATURAL JOIN Customer WHERE Item.Price < 25`

the memory footprint and speed up execution, SQLite Values can be shallow-copied. In this case, memory management properties are not replicated – allocated buffers should only be freed when the original Value is released.

2.1.4 Query execution in SQLite

Building a just-in-time compiler for SQLite requires understanding how it executes queries. Although it shows certain similarities with traditional query processing models, the way SQLite executes query – using a virtual machine – is uncommon.

In the following paragraphs, we present traditional query processing and execution models. We then introduce the SQLite approach and analyse how it relates or differs from others.

Traditional query execution models

Most relational data management systems execute SQL statements by translating them into a *physical relational algebra query plan*, which describes the work and computations that need to be carried out to execute an update or query data as requested by the user. Query plans can take different forms, but the general concept stays the same.

Different *operators* (selection, join, cross-product, *etc.*) are instantiated and can *produce* and *consume* tuples. For instance, a selection from relation A would *consume* relation A by scanning it and *produce* all tuples that match the given predicate. A join

between A and B, using a hash join[3], would *consume* all tuples from A to build the hash index and then *consume* tuples from B one by one while looking up their join key value in the hash index, *producing* all joined tuples that match the join predicate. Fig. 2.5 displays a query plan using a selection, two joins and a projection. The Item relation is the only input to the Selection operator. The Selection and the Order Relation are the two inputs to the `OrderItem` join operator. This join operator and the Customer relation are the inputs for the second join operator, which is the input for the Projection operator on `C_Name`.

Processing query plans can be done in different ways. *Volcano-style processing* [17] was one of the first. It uses a set of simple operators which can have one or two inputs (relations or other operators). Each operator has a `next` method that retrieves the next unseen tuple from the operator (for instance, the next tuple from $A \times B$). The *next* function of an operator in turn calls its input's `next` functions and processes the data internally before returning the next tuple. Appendix 6.3 provides a pseudo-code implementation of a hash join operator in the Volcano-style processing model. When its `next` method is called, it first builds the internal hash table if it has not yet been built. It then probes
This model is simple, easy to implement, allows *pipelining* in some operators (that is to say, all the data does not need to be read to start producing tuples, which helps keeping memory usage low).

Though, Volcano dates from an era where the dominating costs were operations on storage media. Its approach is very expensive on modern hardware, as it causes many instruction cache stalls and incurs serious overhead caused by function calls.
That is why other processing models, such as *batch processing* (or *bulk processing*), were created. Batch processing was pioneered by the X100 query engine that was used in MonetDB [8].
In batch processing, the number of function calls is minimised by not producing single tuples but rather sets (or *vectors*) of tuples. As the cost of a function call stays the same, batch processing effectively highly reduces the cost they incur, the trade off is that memory usage is higher.

Most commercial data management systems use query plan optimisers to make query processing faster, by using either pattern-based optimisations (for instance, selecting from A before joining on A and B to reduce the number of candidates) or *a priori* knowledge on the data to yield more efficient query plans.

[3] In query processing engines, hash joins are join algorithms based on hash tables. The simplest form of hash join consists in building a hash table with join key values for each tuple in the relation of lower cardinality n and the probing this hash table for each tuple in the second relation of higher cardinality m, leading to a complexity of $O(n + m)$.

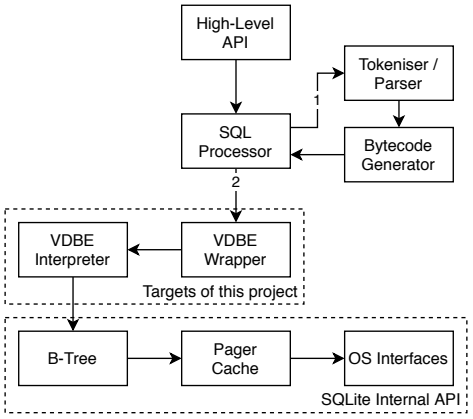


Figure 2.6: Schematic architecture of SQLite

Query processing in SQLite

SQLite does not use query plans to process SQL statements, but a domain-specific register-based virtual machine language called **V**irtual **D**ata**b**ase **E**ngine (VDBE)[4] [18, 49].
The VDBE bytecode language has conventional operations (such as `Add`, `Goto` or `Integer`, to store a constant in a register), but also some that are specific to query processing: `OpenRead` to open a B^+-Tree cursor, `Next` to advance a cursor, `Column` to extract a column from the tuple pointed by a cursor... These operation codes allow to process any SQL statement supported by SQLite by executing a VDBE bytecode programme.

Fig. 2.6 shows the architecture of SQLite. Before running an SQL statement, the SQLite API requires the user to call an `sqlite_prepare` API function. The statement is then first passed to the *SQL Processor* and parsed. From there, the *Code Generator* generates a VDBE[5] and returns in to the *SQL Processor*. The VDBE stores the bytecode for the statement, along with several book-keeping fields as well as a pointer to an array of registers allocated for its execution.
The *VDBE* interface is then used to execute the SQL query. The bytecode programme is executed by an interpreter which contains the code for some operations and that delegates others to internal APIs for specific actions (like interacting with a B^+-Tree).

Operation codes take up to five parameters. The first three (`P1`, `P2` and `P3`) are general-purpose 32-bit integer arguments that can represent databases, cursors, integer literals... `P4` is 64 bits wide and can be a pointer to a string, binary data or internal objects (C-string, pointer to a value to copy, function definitions...). `P5` is 16

[4] Although the VDBE bytecode language does not follow a specification and is subject to change from a version to the other, a complete description of the operation codes for the latest version of SQLite is available at https://sqlite.org/opcode.html

[5] Note that in SQLite, several terms can be used to designate the same thing. Each bytecode programme is stored in a VDBE instance – a C structure of type `Vdbe`. Executing a VDBE or the programme it holds is thus equivalent. VDBE instances are also sometimes called *virtual machines* or *prepared statements*.

bits wide and used as a set of flags for the operation [18].

Some operation codes are polymorphic: depending on the value of `P5`, an integer parameter can be interpreted as its value, the cursor designated by its value, the attached database whose index is its value, *etc.* The remainder of the time, `P5` can be hints for the VDBE interpreter – to pick an algorithm rather than another – or even hints for the bytecode printer – to indent loops correctly.

Finally, some operations alter the bytecode at run-time [49]. For instance, the VDBE language has a `Once` operation that makes the interpreter jump to a certain programme counter, but only the first time the operation is seen. To achieve this, the implementation of `Once` compares the value of its `P1` parameter with the value of the `P1` parameter of the first operation in the bytecode (which is always `Init`). If they differ, `Once` rewrites its `P1` parameter to match the value of `Init`'s `P1` and jumps to the given counter.

The interpreter function adopts a relatively simple implementation. It loops over the VDBE operation array and uses a large `switch-case` statement, where each `case` label contains the implementation of at least one operation code. Error handling in the interpreter is done by always checking internal API function return codes and jumping to error-handling labels when they hold unexpected values.

This approach, although it makes the implementation easier, causes many instruction cache stalls and makes the code rather hard to predict [23]. Furthermore, some operations that could benefit from loop unrolling in a compiled statement (such as `Copy` – that copies n registers from $P1$ to $P2...P2+n$ – need to be implemented using a C loop, as the number of copied registers is not known at compile-time.

Tuple production in SQLite

SQLite does not use batch processing: tuples are emitted one by one by the VDBE. If the current operation code is `ResultRow`, the virtual machine engine copies some registers to output registers (which are also SQLite Values). It then exits the VDBE interpreter loop by returning an error code that indicates that a new tuple has been produced (`SQLITE_ROW`).
Once the tuple has been read and used by the host software, VDBE execution can be resumed in the state where it was interrupted (the next step is often to advance the cursor to process the next tuple).
In essence, a VDBE programme works like a coroutine: it pauses its execution to yield tuples and can be resumed to continue query processing in its previous state when called back.

In SQLite, executing a statement is thus done by *stepping* through the VDBE programme, as illustrated by Fig. 2.7. In other words, it is a matter of iteratively calling the `sqlite3_step` function, a wrapper around the interpreter. While the stepping function returns `SQLITE_ROW`, query execution can be resumed. Once `SQLITE_OK` has

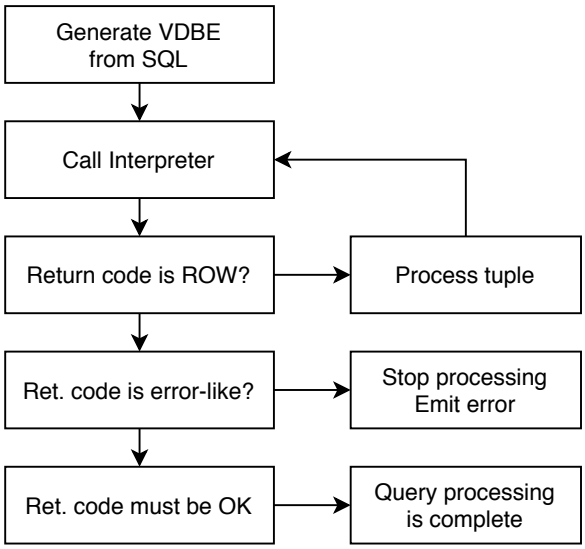


Figure 2.7: Execution workflow in SQLite

been returned, the VDBE programme has been halted and query processing is complete. In certain cases, the interpreter may return an error code – in case of a memory allocation failure, if a corruption is found, or if any inconsistency in the database is detected. When that happens, higher layers should propagate or handle that error.

The VDBE tuple emission behaviour has been pointed as a possible performance bottleneck as it causes many jumps and function calls [23].

Figure 2.8 shows an example of a VDBE bytecode programme. The `Init` operation code allows to setup query execution by jumping to programme counter 9. In that case, it begins the transaction, stores the integer 25 into register 2 and returns to programme counter 1. A cursor is then opened for reading on table 2, then rewound to start processing the first tuple. The *hot loop* starts at programme counter 3: the `Column` operation extracts column `i` from the tuple currently pointed at by the pointer into register 1. If the value of `i` is less than register 2 (the constant integer 25), the virtual machine jumps to 7 (skips that tuple). Otherwise, column `i` is re-extracted into register 3 (as comparison may have changed its type in register 1). `ResultRow` then emits the tuple by exiting the interpreter with return code `ROW`. This pauses the coroutine/VDBE, and allows the host application to process the emitted tuple. Once the tuple has been used, the host application can resume the coroutine/VDBE by calling the interpreter back. When execution is resumed – or after having skipped the current tuple – , the `Next` operation advances the cursor. If it is not at end, control flow jumps back to programme counter 3. Otherwise, it continues execution to `Halt`, which returns from the interpreter with an `OK` return code.

```
   PC       OpCode       P1    P2    P3     P4     P5        Comment
  ----  -------------  ----  ----  ----  ------  --  --------------------
-> 0     Init           0     9     0             00  Start at 9
   1     OpenRead       0     2     0     1       00  root=2 iDb=0; rel
   2     Rewind         0     8     0             00
   3       Column         0     0     1           00  r[1]=rel.i
   4       Le             2     7     1     (BIN) 54  if r[1]<=r[2] goto 7
   5       Column         0     0     3           00  r[3]=rel.i
<- 6       ResultRow      3     1     0           00  output=r[3]
-> 7     Next           0     3     0             01
   8     Halt           0     0     0             00
   9     Transaction    0     0     1     0       01  usesStmtJournal=0
   10    Integer        25    2     0             00  r[2]=25
   11    Goto           0     1     0             00
```

Figure 2.8: VDBE Bytecode Programme Example for SQL query `SELECT [Col 0 "i"] FROM [Table 2 "rel"] WHERE [Column 0 "i"] > 25.` `->` and `<-` respectively designates points at which execution can be resumed and interrupted.

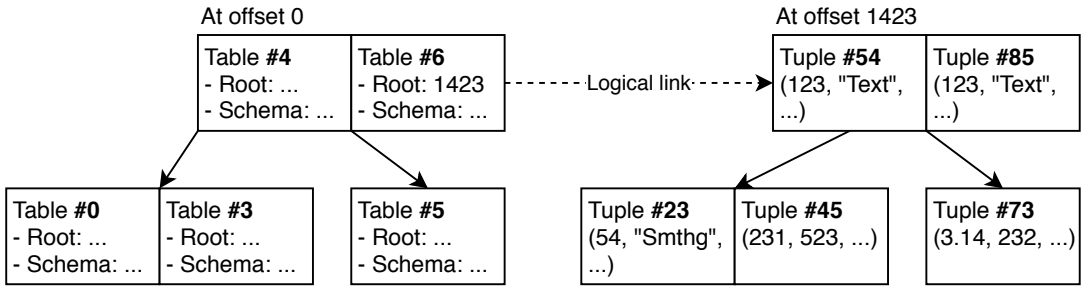


Figure 2.9: Schematic representation of an SQLite database file

2.1.5 Data management in SQLite

In an SQLite database, all objects are represented and indexed in B^+-Trees. Consequently, tables can be referred to using their names or by their root page identifier. The database schema itself is stored in a table, `sqlite_master`, whose root page is at offset zero in the SQLite database file. That way, it is possible for SQLite to store the database schema, table/B^+-Tree offsets and table data (tuples) in a single file.
Fig. 2.9 displays a simplified database file. The page at offset 0 is the root of the `sqlite_master` table, which contains information about user-created tables. This table references the offset for the root page of every other table. This allows to lookup table data efficiently and removes the need to store any other table/data offset information.
Any data access or modification is done by the virtual machine by using the B^+-Tree interface [1].

Despite the use of B^+-Trees to store all the data, tables do not necessarily have an indexed column. If no index is created by the user upon relation creation, SQLite

uses an implicit *Row ID* column to index the data and store it in a B^+-Tree.

Before being able to carry out any operation on a user table, SQLite generates a VDBE programme to fetch schema data from the `sqlite_master` table and determine what tables exist as well as what their columns are.

Using B^+-Trees as the only data structure for storage results in a lot of uniformity. Whatever the target table is, probing it is always a matter of opening a read cursor to a certain B^+-Tree and walking it using the B^+-Tree API.
In a similar way, tuple sorting is done using an empty, ephemeral B^+-Tree filled by SQLite with all the tuples to be sorted – which effectively leads to a sorted data structure, though the build costs are rather high.
Although different kinds of cursors exist in the internal SQLite API, they offer the same operators and that leads to a rather contained – if not small – bytecode language.
On the other hand, this comes at the cost of a lower versatility, particularly in performance matters. For instance, whereas some database systems provide many different implementations for each operation, picking the best candidate based on the query plan and sometimes even on the data, SQLite most of the time sticks to the *simple* solution. Optimisations are done only at query level and many algorithms that perform better than naive implementations are omitted (hash joins do not exist, sort-merge joins are only used in rare occasions, only one sorting algorithm exists) [49, 18].

2.2 Just-in-Time Compilation

2.2.1 Introduction

Just-in-Time (JIT) Compilation designates the compilation of code during the execution of a programme rather than before its execution (which is known as *Ahead-of-Time (AoT) Compilation*).
It is often used by languages which are usually interpreted to provide better performance by translating a higher-level language (such as bytecode) to a lower-level language (typically, native code) [15].

The typical reasoning behind JIT compilation is that interpreters are often implemented using a `for` loop that reads through the bytecode and a `switch-case` block that holds the implementation of each instruction. Historically, branch prediction did not work well in these configurations and the overhead induced by control hazards was seen as a prominent caveat that compiling bytecode just-in-time could defeat (by 'unrolling' the interpreter loop). Although this consideration was proven to be less significant on modern hardware by recent studies [44], branch prediction is still cited as a cause to that inefficiency.

However, JIT compilation can also be seen as a way to use the capabilities of the target platform better [4]. By emitting a cross-platform bytecode (such as Java

bytecode) and providing the toolchain to compile it to native code at run time, the user can benefit from platform-dependent features, such as data-level parallelism (SIMD), but also to adapt some parameters to their most adequate values on the target computer. For instance, programmes can choose the size of certain buffers based on the size of caches on the target CPU.

Another benefit brought by JIT compilation is its ability to leverage run-time knowledge. For instance, at machine code level, multiplying a number by a power of two can be done by doing a multiplication or by using a bit shift. However, a general *multiplication* function would need to retain the implementation that works for all numbers, preventing a significant optimisation for the power-of-two case. If the same function is compiled just-in-time and the value of the multiplier is known, then the compiler can optimise certain cases by emitting a bit shift.
This idea has been used in [14], where authors use *value specialisation* in functions to reduce execution time in client-side JavaScript libraries. The run-time knowledge problem is also well illustrated by the *C++ Should Support Just-in-Time Compilation*[6] proposal to the C++ standard. In C++ performance-sensitive programmes, the use of template specialisation is widespread in the industry.
By using templates to generate as many functions as there are special cases, execution time can be significantly reduced. Unfortunately, this method has a high cost on compilation times, as the number of functions to optimise and compile is much higher. In this case, just-in-time compilation is seen as an agile middle ground. Compiling a different implementation based on the value of parameters allows for improved performance, but does not cause a compile-time overhead. More importantly, it adds delays in execution time only when the specialisations are used: otherwise, they are not compiled.

Another kind of run-time knowledge that can be useful is type information. This is more relevant in dynamically typed languages, such as JavaScript. Chang et al. [12] use type inference at execution time to determine the type of expressions and uses that knowledge to 'specialize the code for the most common dynamic types of receiver arguments' [12], which greatly improves performance on various JavaScript benchmarks.

Finally, in more advanced cases, it is possible to use JIT compilation to integrate profile-driven optimisation[7]. By collecting application usage data and detecting hot paths and most used code sections, a JIT compiler can make the best decisions when it comes to loop-unrolling or the ordering of cases in a if–else-if-chain [29].

In practice, JIT compilation is often done in two steps. First, the compiler takes the input code (in assembly or high-level languages), potentially optimises it and creates a dynamic object in a memory buffer. Then, it dynamically loads the object

[6] The complete proposal can be seen at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1609r3.html`

[7] Profile-driven optimisation consists in deploying instrumented software and profiling it on the field, then using that additional knowledge to carry out better optimisations.

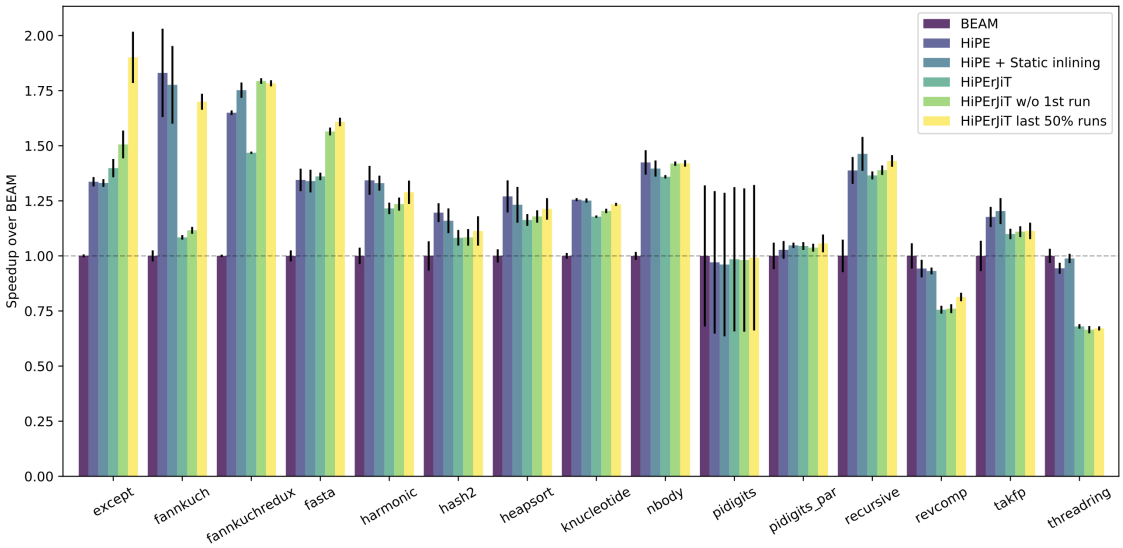


Figure 2.10: Experimental performance evaluation results for HiPerJIT [21]

from memory, links it with the host process and returns the address of the just-in-time compiled function.

Note that when a JIT compiler is used, it is assumed that the overhead induced by translation and compilation is outweighed by the gains in performance that executing native code offers over interpreting bytecode. When that condition is not met, it is often assumed that the cost can be amortised by subsequent executions of the programme – whose native representation can be cached and reused.

2.2.2 Just-in-Time compilation in interpreted languages

In SQLite, virtual machine programmes are executed by an interpreter: the VDBE module. JIT Compilation has been used as a solution to run-time performance issues for more than twenty years [4]. The first notable example was given by Cramer et al. in 1997 [15] who were the first of many to implement a JIT compiler for Java. It has been since used successfully to implement several languages, including the high performance computing language FORTRAN. To this end, reviewing JIT compilers for interpreted languages may help implementing one for SQLite.

PyPy [42] is an example of a JIT compiler for an interpreted language: Python. It is a drop-in replacement for CPython – the default, most commonly used, Python implementation – implemented using RPython [2].
The automated performance reviews[8] show that the geometric average of the speed-up (in comparison with CPython) is $4.3\times$, but varies greatly depending on the workload.

[8] speed.pypy.org shows a detailed performance comparison between RPython and CPython on a variety of workloads.

Erlang [3] is a dynamically typed language designed for applications that need real-time performance, run in distributed environments and need high levels of availability (continuous execution). The default implementation of Erlang runs on the BEAM virtual machine, which interprets programmes essentially like the SQLite VDBE interpreter does – reading through the bytecode and picking the implementation using a `switch-case`.
HiPerJIT [21] is a profile-guided JIT compiler for the BEAM virtual machine. In order to compile code just-in-time, it reads through the bytecode and sequentially inserts the implementation of each operation.
Its authors provide empirical evaluations and show that just-in-time compilation provides a speed-up factor of up to $2.5\times$ on benchmarks, and only rarely causes a slowdown (Fig. 2.10).

2.2.3 Just-in-Time compilation in data management systems

Though, just-in-time compilation has not only been used to improve the performance of bytecode and interpreter-based programming languages. Several database engines have started supporting JIT compilation of queries, aiming to to optimise query execution time.
Namely, Lee et al. [31] note that although performance of CPUs and main memory has got much better over time, the still widely-used Volcano processing model does not allow to use these new technologies to the best of their abilities and propose. They thus propose a unified optimisation system that includes the traditional high-level query optimisation but also just-in-time compilation [31].

HyPer [36] is an example of a JIT-oriented database whose architecture resembles the one proposed in [31]. Its inception point is that even with modern processing methods (such as batch-processing and column-based storage rather than Volcano-style), hand-written execution plans outperform the ones generated by the query processor.
In HyPer, queries are represented as an algebraic query plan, like in most data management systems, but they are translated to a sequence of native instructions instead of being translated to a physical plan [36].
The programme is generated on-the-fly by the query processor using rules and code templates. Whereas classical query plans *pull* tuples from their inputs (in what is called *iterator-style processing*), the just-in-time compiled code in HyPer *pushes* tuples from each *pipeline breaker* (an operator for which the data cannot be held in CPU registers) to the next. This significantly reduces query processing time by improving instruction prediction and code locality [36].
The just-in-time compiled code is not C nor C++, but LLVM Intermediate Representation (LLVM IR), a typed assembly-like language. This gives full control over the behaviour of the operators but also cuts the compilation costs. Indeed, an optimising C/C++ compiler is much slower than an LLVM IR compiler [36].
This does not mean, though, that all the database code is translated to assembly, as this would be cumbersome and error-prone: a hybrid approach is chosen. The

	Q1	Q2	Q3	Q4	Q5
HyPer + C++ [ms]	142	374	141	203	1416
compile time [ms]	1556	2367	1976	2214	2592
HyPer + LLVM	35	125	80	117	1105
compile time [ms]	16	41	30	16	34
VectorWise [ms]	98	-	257	436	1107
MonetDB [ms]	72	218	112	8168	12028
DB X [ms]	4221	6555	16410	3830	15212

Figure 2.11: Performance Comparison between HyPer and non-JIT data management systems on OLAP Workloads [36]

```
indirect call ExecEvalFunc() {
  P = indirect call ExecEvalFunc()
    X = indirect call ExecEvalVar()
    Y = indirect call ExecEvalVar()
    return int8pl(X, Y)
  C = indirect call ExecEvalConst(1)
  return int8lt(P, C)
}
```

(a) Default virtual call-based implementation

```
define i1 @ExecQual() {
  %x = load &X.attr
  %y = load &Y.attr
  %pl = add %x, %y
  %lt = icmp lt %pl, 1

  ret %lt
}
```

(b) Equivalent LLVM IR code

Figure 2.12: Comparison of two possible implementations for the same $X + Y < 1$ predicate [34]

hot path of query processing (*i.e.* the operators and tuple transmission between operators) is executed by the just-in-time compiled code. On the other hand, non performance-critical components – ”complex data structure management or spilling to disk” [36] – are pre-compiled C++ functions that can be called from LLVM IR when needed.

Figure 2.11 displays a performance comparison between HyPer (with LLVM IR) and competing database systems on various OLAP workloads. Results show that even when accounting for the compilation cost incurred by JIT compilation in HyPer + LLVM IR, the performance is far superior to all other presented databases.

Melnik et al. [34] propose a JIT compiler for expressions in PostgreSQL queries. In PostgreSQL, the default expression resolver uses many function calls: it computes the result of expressions in a way similar to how physical query plans are executed, by creating an abstract syntax tree of the mathematical or logical expression and linking operands with dynamically dispatched (virtual) function calls.
This approach, although it offers a certain simplicity of implementation, is far from optimal. As already mentioned, function calls can be far more costly than other costly operations (such as memory access) on modern hardware. As a result, in the default implementation of PostgreSQL, a mathematical expression on the hot path (for example, executed for each tuple) would incur a high overhead, and that even for rather simple computations.

Fig. 2.12 displays two possible implementations for a predicate $X+Y < C$ in an SQL

statement, where X and Y are tuple attributes and C is a constant set to 1. Whereas the PostgreSQL default implementation causes many costly virtual function calls, the LLVM Intermediate Representation version is much simpler and highly reduces overhead by using native CPU operations.
As explained by Melnik et al., their JIT compilation makes use of run-time information (the actual filter expression, the types of the operands...) to generate native code for the expression, and eliminates the need for cumbersome operation wrappers, which are necessary in the traditional query processing engine used by PostgreSQL.

Furthermore, Melnik et al. propose more transformations to reduce global function-call overhead. For instance, they pre-compile some back-end functions (`pow`, `sqrt`...) to IR and allow them to be inlined in the expression JIT-compiled code, further reducing the number of function calls [34].
This results in a significant performance gain: Melnik et al. claim a speedup of up to $2\times$ on computation-heavy queries. Their evaluation shows a reduction of the time spent computing expressions from 56% to 6%.

Although just-in-time compilation can bring a significant run-time performance improvement, it also adds an inevitable overhead: compilation itself.
Many just-in-time compilers use a low-level language (such as LLVM IR) whose compilation costs are fairly low (as compared to the ones of high level languages like C++ [36]). Still, compilation time can still be an important cost in query processing, particularly when querying small sets (where the compilation costs will represent a large portion of the total processing time) or complex queries (where compilation costs will be high) [38].

To limit this overhead, Pantilimonov et al. [38] propose a caching technique for just-in-time compiled code. By dumping optimised LLVM modules to disk, they improve performance by removing the costs incurred by function-level and module-level optimisations on LLVM-IR.
Caching compiled objects could be done in theory, though in this case the use of integer-to-pointer instructions in LLVM-IR (for hard-coded run-time pointers) makes editing modules for substituting these values compulsory before re-using a prepared module [38].

2.3 LLVM: A compiler infrastructure for multi-stage optimisation

2.3.1 LLVM

LLVM [29] is a compiler infrastructure proposed by Lattner designed for continuous optimisation of software. Observing a lack of efficiency of optimisation techniques in traditional compilers (built on the compile-link-execute model) and in their optimisation passes, Lattner presents LLVM, a new compiler infrastructure designed

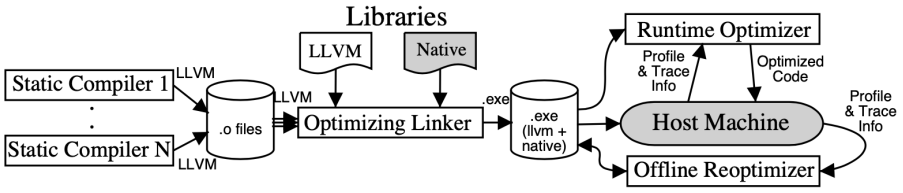


Figure 2.13: Diagram of the LLVM infrastructure [29]

for *multi-stage optimisation* and more precisely for allowing *interprocedural optimisations* and *profile-driven optimisation*.

Interprocedural optimisation consists in gathering as much information about the programme as possible and to analyse the code, reason on it and carry out optimisations on more than a single translation unit [29]. In C and C++, individual source files are generally compiled to independent object files which are then linked together to form a single executable. In traditional approaches, optimisations are only carried out at the translation-unit level (i.e. in a source file). This results in many missed optimisation opportunities. For instance, functions that may be inlined if they were defined in the same translation unit cannot be, because optimisers do not cross the source code file frontier.

Traditionally, interprocedural optimisation can be done in different ways, with various drawbacks. A native code (assembly) optimiser can be run on the resulting executable. Assembly otimisers can transform binaries produced by any compiler, but they often lack context and higher-level information – such as typing – which makes analysis and thus optimisation more difficult[9].
On the other hand, interprocedural optimisations carried out on high level representations (such as abstract syntactic trees) comes at the high cost of having to recompile everything after any change, as this approach makes most of the optimisation effort happen at link-time.

The LLVM (Low-Level Virtual Machine) compiler infrastructure aims at resolving these issues by different means (Fig. 2.13):

- It introduces *Low-Level Virtual Machine Intermediate Representation* (LLVM IR), a low-level language that retains typing information. The design of LLVM IR makes optimisation on it much more efficient than traditional approaches (low-level optimisations on non-typed assembly or high-level optimisations on the AST);
- In the LLVM compiler design, high level languages (C, C++, ...) can be *lowered* (compiled) to LLVM IR by LLVM *front-ends*. As a common representation

[9] 'Being typed enables a number of optimizations to be performed on the intermediate representation directly, without having to do extra analyses on the side before the transformation. A strong type system makes it easier to read the generated code and enables novel analyses and transformations that are not feasible to perform on normal three address code representations.' [33]

between languages, optimisations can be carried out on generated IR via *optimisation passes* (dead-code elimination, constant folding...). Once linked, interprocedural optimisations can be performed. All IR optimisations are shared among all high-level languages (as they all target the same IR language). Optimised IR is then translated to machine-code, which can be executed;

- IR also has the ability to retain debugging information. This is fundamental, as optimisation passes may transform IR heavily and debugging optimised code has been repeatedly pointed at as an issue [10];
- Instrumentation can be inserted in IR before compilation to native code. The programme can thus be profiled at run-time. This process is completely transparent to both the compiler user and the end-user. Binaries are shipped along with their LLVM IR 'bytecode' representation, as well as an embedded IR compiler. As a result, programmes can be re-optimised and re-compiled automatically at run-time based on profiling results. This can be done while the programme is being used or at idle time for most complex optimisations.

Once the high-level source code has been compiled to LLVM IR, which has been optimised, the last step is to transform it into native code. Once again, the LLVM design proves to be very powerful.

Whereas other compilers usually have AST-to-assembly lowering passes for each target architecture, using LLVM IR allows to decorrelate high-level representations and low-level compilation.
Allowing an LLVM-based compiler to support a new target platform simply takes writing an LLVM *back-end*, a compiler taking LLVM IR as input and producing native code. Making back-ends common to all LLVM-based compilers grants more uniformity, reduces the amount of duplicated work between high-level language compiler developers, makes porting languages to new architectures simpler and makes bug-fixing common among all compilers.

LLVM embeds several APIs that leverage its core features for just-in-time compilation purposes. Indeed, LLVM exposes an LLVM IR module building API, native code generation capabilities as well as powerful linking features. These traits make it an ideal framework for building JIT compilers. That is why LLVM has several specialised APIs (MCJIT and *On-Request Compilation* or ORC) designed for emitting intermediate representation, compiling it to native code and linking the result just-in-time in reasonable time [24].

The LLVM framework has, since its inception, been the host to several state-of-the-art optimising (ahead-of-time) compilers. The `clang` and `clang++` C/C++ compilers are based on LLVM, as are `rustc`, the official Rust compiler or `swiftc`, the official Swift compiler. Just-in-time compilers for several languages – Java, Python, Erlang... – have also been implemented using LLVM, mainly thanks to its ease of use and wide targetting capabilities.

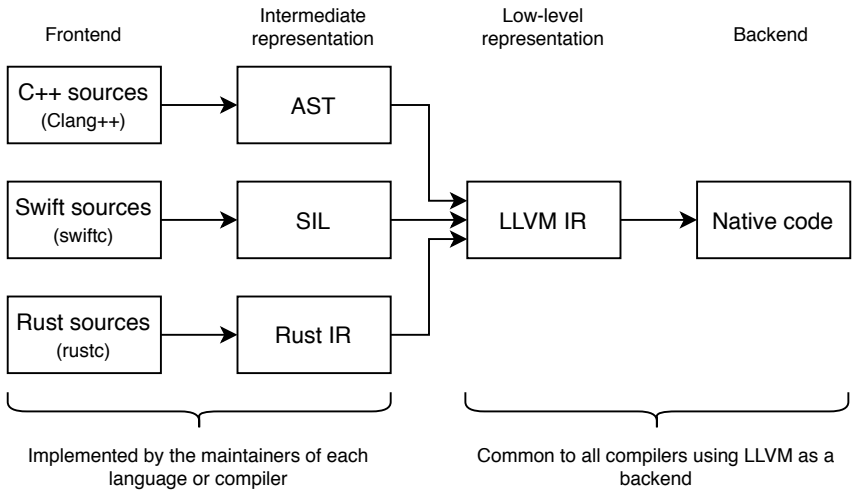


Figure 2.14: Compilation pipelines for various high-level languages [30]

2.3.2 MLIR: A framework for intermediate code representations

Although LLVM has had a lot of success with its low-level optimisation and native code generators, many modern programming languages still have their own intermediate – but higher than LLVM IR – representation. For instance, Rust uses two intermediate languages – HIR and MIR – and Swift uses SIR.
The need for other intermediate representations stems from the fact that these languages were designed to allow carrying out certain optimisations efficiently on high-level representations. These optimisations are difficult to reproduce once the source code has been translated to LLVM IR. This is somewhat similar to the issue LLVM was designed to address. While it made interprocedural optimisation more efficient thanks to its high-level assembly, this same assembly still lacks some even higher-level, non language-agnostic, information. That is why many compilers that use LLVM for native code generation and interprocedural optimisations still first generate their own intermediate representation before translating it to LLVM IR (Fig. 2.14). Although they hold very similar roles and operate in similar ways, SIL, MIR and Julia IR do not use the same code base or framework.

This comes at high costs. First, each language toolchain needs to maintain an IR compiler, IR optimisation passes and IR manipulation utilities [30]. Noticing that this approach is common among compiler developers, Lattner et al. propose MLIR [30], a compiler infrastructure integrated in LLVM that allows to define new Intermediate Representation Languages (or *Dialects*, in the MLIR jargon).

The aims of MLIR are manifold. First, it makes sharing IR optimisations between different languages easier. By compiling certain operations of an MLIR dialect code to another dialect which 'knows' more optimisations (instead of only compiling to LLVM IR), a source written in a certain language can benefit from more optimisations.

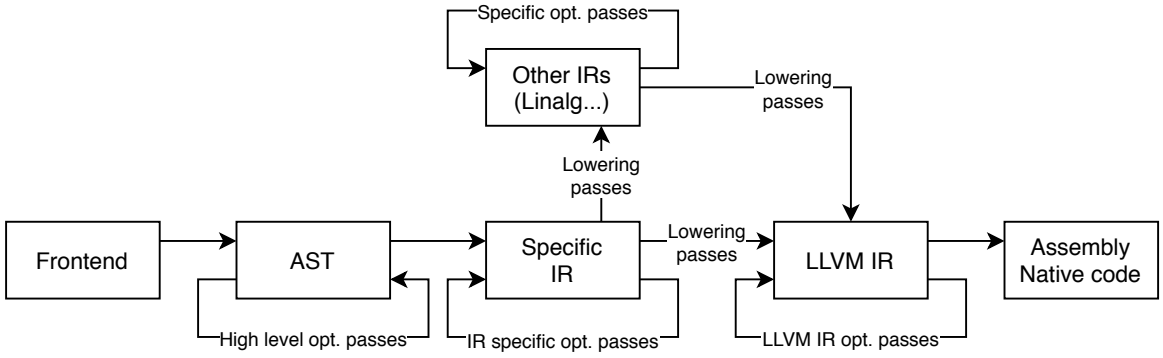


Figure 2.15: Workflow of a compiler based on MLIR for a high-level language

Second, making the medium representation language definition tools common, as well as making their compilers and optimisers respect a certain protocol reduces the risk of bugs and security faults. That reduces the burden for programming languages/compiler maintainers, who do not have to worry about maintaining a custom toolchain anymore but rather only the transforms that the toolchain proceeds to.
Using a common language definition also makes code location tracking (making possible to easily determine what optimisation or compiling process created an instruction) a first-class issue. Indeed, it was pointed out by Lattner et al. to be a recurring issue with custom IR generators and optimisers, whose generated code is often hard to debug because of the heavy transformations carried out that make debug information invalid.

Fig. 2.15 illustrates a possible design for an MLIR-based high-level language to native code compiler. Source code is first translated into an abstract syntactic tree, which is then lowered to a language-specific IR in the form of an MLIR dialect. Language-specific optimisation passes can be applied on that IR. The resulting code can also be lowered to other dialects to leverage performance gains brought by existing implementations and then to LLVM IR as a first step into low-level compilation. The remainder of the compilation process is carried out by the back-end, which converts LLVM IR to native code [24].
Making the language-specific IR compatible with other higher-level IRs (via a common framework and 'translation' passes) allows to interleave optimisations from different IR languages, and thus to reach a better result.

The MLIR framework allows compiler developers to define intermediate representations for new languages called *MLIR Dialects*. Then, using the *MLIR Module Builder* the compiler front-end developer can convert another programme representation (such as an AST) to an MLIR module by programmatically emitting MLIR operations.
These operations can be operations from the newly created dialect, LLVM IR instructions or even operations defined in other dialects.

By defining *lowering passes*, the developer specifies how an operation from their

```
def f(x) {
  y = x + 2
  print y
}
```

(a) Code in the *Custom* language

```
func f(%x: int) {
  %cst = constant 2
    : () -> int
  %y = add %x, %cst
    : (int, int) -> int
  print %y : (int) -> ()
}
```

(b) Code in the *Custom* MLIR Dialect

```
define @fmt = "%d\n"

define @f(i32 %x) {
  %cst = i32 2
  %y = add i32 %x, i32 %cst
  call @printf(@fmt, %y)
}
```

(c) *Custom* Dialect lowered to LLVM

Figure 2.16: Three compilation steps for the *Custom* language: high-level source code, MLIR representation and LLVM IR

dialect should be *lowered* (mapped) to its equivalent in other dialects (including LLVM IR). MLIR also allows to define *optimisation passes* that use rules and patterns to transform parts of an MLIR module into a target dialect.
These passes often rely on language-specific information and guarantees that allow some optimisations that could not be proven valid by an LLVM-IR (lower-level) optimiser.

Finally, once all operations have been lowered to the LLVM IR dialect, the MLIR framework has built-in features to transform it into a classic LLVM IR module, compile the programme to native code on-the-fly and to either run it immediately or create a binary.

Fig. 2.16 illustrates the progressive lowering to LLVM IR in a new *Custom* language. The *Custom* language works only with integers, allows to declare and define variables with the = operator and to add expressions using the + operator. Finally, *Custom* allows to output the result of an expression using the `print` statement.
An *Custom* compiler would first turn the high-level representation (*Custom* source code) and convert it into an MLIR module. The *Custom* MLIR dialect defines the `constant`, `add` and `print` MLIR operations, along with corresponding *lowering passes*. In this example, the module would have only one function, `f`.
The last step for an *Custom* compiler would be to *lower* the MLIR module to an LLVM module. In this case, it does so by emitting an integer literal for `%cst`, using the `add` LLVM instruction on `%cst` and `%y`. The `print` instruction is lowered to an LLVM `call` instruction: the printing effort is delegated to the standard `printf` function with the `%d` format specifier.

Another illustration of the MLIR compilation workflow is the new version of Flang[10], an MLIR based Fortran compiler. After applying pre-processing to FORTRAN source code, Flang emits FIR[11].
The produced FIR module is optimised using Flang-defined MLIR optimisation passes, then lowered to LLVM IR using Flang-defined lowering passes. The resulting LLVM

[10] https://github.com/llvm/llvm-project/tree/master/flang
[11] https://github.com/llvm/llvm-project/blob/master/flang/docs/FortranIR.md

module can be compiled to native code using LLVM backends.

MLIR is a young project, and few compilers already use it widely. Apart from Flang, TensorFlow, a machine-learning framework, uses MLIR for unifying just-in-time code generation in its various implementations and tools[12] .

As a part of the TensorFlow dialect implementation, the `linalg` (linear algebra) MLIR dialect has been created. As explained earlier, should other developers need to integrate linear algebra operations in their language, they can lower these operations to the `linalg` dialect instead of implementing them themselves.
This reduces the amount of duplicate work needed to implement similar features in different languages, and allows all programming languages to benefit from the same optimisations. Indeed, if a new technique for faster numerical linear algebra is discovered and implemented in the `linalg` dialect, all languages that lower their linear algebra operations to `linalg` will benefit from it without further work.

Another example of MLIR usage is the `affine` dialect [30]. The `affine` dialect is not an intermediate representation from a higher-level language, but rather allows higher-level language to integrate polyhedral optimisation easily and reliably into their native code [13, 9].
The rationale is the following. While it is a common construct in high-level languages, detecting `for` loops in LLVM IR is rather difficult. Using the MLIR framework, any high-level language can map its adequate `for` loops directly to `affine.loop` MLIR operations. These in turn will be lowered and optimised using the `affine` dialect's `loop` lowering passes. The goal of the `affine` dialect is similar to the one of `linalg`. If new loop optimisations are found and implemented in the `loop` dialect, every language that lowers to it will benefit from them.
Achieving the same result without MLIR would entail either making `for` loops easily detectable in the generated LLVM IR (which is a trade-off, as it can prevent other aggressive optimisations) or performing polyhedral optimisation in the custom IR optimiser (which is error-prone but also represents a needless effort as a valid implementation already exists in the MLIR source).

2.3.3 JIT-compilation features in LLVM

LLVM does not only expose a code-emission and native code generation API: it also allows the developer to execute the IR module they generate immediately.
Indeed, LLVM abstracts away the cumbersome part of just-in-time compilation by using its own object file generation features to compile and dynamically load objects at run-time.
It also makes symbol resolution simple by providing helper functions which allow to share code and data between the host process and the just-in-time compiled code.
As a result, executing IR just-in-time is straightforward. Calling the LLVM JIT API compiles IR to native code, loads the object and returns a function pointer that can

[12] https://www.tensorflow.org/mlir

be used directly from C / C++ programmes.

The following paragraphs introduce MCJIT, an LLVM API that allows to compile and execute IR just-in-time.
The main feature of MCJIT is the `ExecutionEngine` component, which can be used to wrap the necessary tools to compile and execute code just-in-time. When building an MCJIT instance, one or several LLVM IR modules can be passed and various options can be set (such as a custom memory manager for the session, for instance).

A *target triple* can also be passed to MCJIT. In LLVM, a target triple is a string of the form `<arch><sub>-<vendor>-<sys>-<abi>` which represents the target CPU, vendor and OS.
For example, setting this triple adequately enables CPU-specific hardware features, which back-ends can for example use to emit SIMD instructions when translating vector operations[13].

Once created, the MCJIT instance takes the form of an `ExecutionEngine` C++ object and holds a member of type `RuntimeDyld`. This component is at the core of the just-in-time compilation process: it allows to load a dynamically compiled object and to probe it for symbols.

At this point, no native code has yet been generated. Indeed, the user may still be adding information to the `ExecutionEngine` (by adding external LLVM modules, global definitions, low-level optimisation passes...).
Native code generation is triggered either manually or implicitly when the user calls a function from the `getPointerToFunction` family. That means the module and execution engine initialisations are complete and thus that the native code should be generated. The generated assembly is dumped into a memory-resident buffer.
Note that, as opposed to other JIT implementations (such as LLVM ORC JIT), all native code is generated and materialised at once. Even functions that are not requested by the user or that do not appear in the body of a requested function are compiled (which may cause unnecessary overhead). Extra care should be taken so that no unnecessary symbols are compiled.

Once native code has been generated, the `RuntimeDyld` implementation loads them from the buffer. It iteratively reads it and adds symbols to a table (a string to pointer map) that can be later probed by `getPointerToFunction`-like functions.
Finally, the memory manager is called for each external symbol – functions or global variables – to insert their address in the just-in-time compiled code. As the behaviour of the default memory manager depends on the host operating system, LLVM exposes several APIs to help communication.
For instance, global symbols can be added to a *global dynamic library* in which the default memory manager searches for symbols by using the `llvm::sys::Dynamic`

[13] More information about target triples can be read at `https://clang.llvm.org/docs/CrossCompilation.html`

`Library::AddSymbol` function.
This is required on some operating systems in order to have the just-in-time compiled module and the host process share code and data.

Once the previous steps have been taken, the address of the requested function can be obtained and returned to the caller. The calling convention of this function depends on the attributes defined in IR, but the default parameter allows any C programme using the System-V ABI[14] to directly invoke just-in-time compiled functions in the same way a function pointer would be called.

2.4 Classical database evaluation workloads

With a rich and wide market, assessing the performance of data management systems in real-life conditions to pick the best one based on a company's use cases and usage patterns is a hard task, and marketing does not work towards making comparisons easy.
That is why several workloads have been designed over the last decades to experimentally evaluate the performance of database products for different kinds of workloads (analytical, transactional, *map-reduce* systems, *etc.*). The Transaction Processing Performance Council (TPC) was created in 1988 as an attempt to 'civilize competition' between database vendors by creating 'good benchmarks' and 'good process for reviewing and monitoring those benchmarks' [47]. The Council has since then created many benchmarks, each with a database schema definition, constraints on the data it should contain and query set with well defined parameters.

In this report, we focus on two OLAP benchmarks for relational data management systems: TPC-H [52], a decisional workload designed by TPC and Star-Schema Benchmark (SSB) [37], a modified version of TPC-H that aims at addressing design flaws found later in TPC-H.

2.4.1 TPC-H

TPC-H [52] is a benchmark oriented towards decision support software. Its queries process the data of a generic international trade company (Fig. 2.17), whose data fits constraints on the relations size, for standardisation reasons.
The query set gives insight in:

- Pricing and promotion;
- Supply and demand;
- Profit and revenue;
- Customer satisfaction and market share;
- Shipping management [5].

[14] The System-V ABI is used by many systems, including Linux and BSD systems such as macOS or FreeBSD. More can be read at `https://wiki.osdev.org/System_V_ABI`.

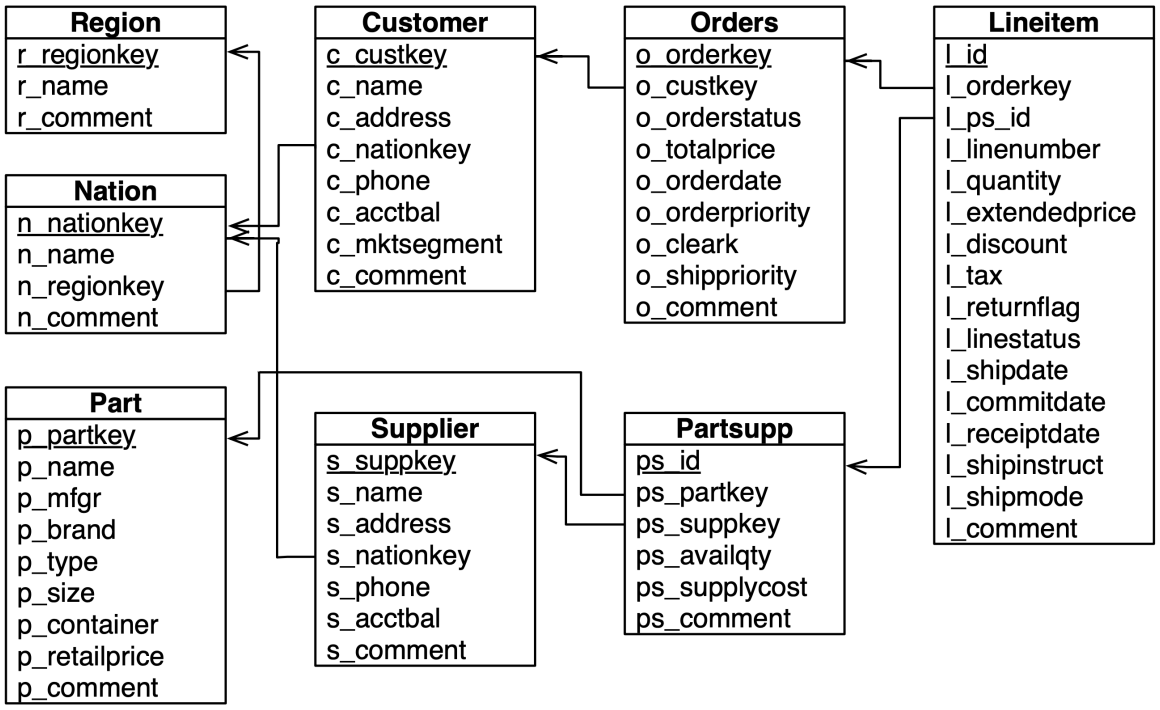


Figure 2.17: The database schema used in TPC-H [11, 52]

The strength of TPC-H as compared to generic microbenchmarks is that it answers business questions derived from real-world studies, but also executes query complex enough to provide a metric on the performance of the database engine under test [5].

In TPC-H, the business intelligence data is stored in eight relations:

- Regions of the world;
- Nations (countries) which belong to a region;
- Customers, that are in a Nation;
- Suppliers, which are in a Nation;
- Orders, that are made by a Customer;
- Parts that can be ordered;
- Part/Supplier couples which joins Parts and Suppliers;
- LineItems which describe what Parts an Order contains [52].

TPC-H is a standard benchmark and most commercial data management systems provide their results for it.

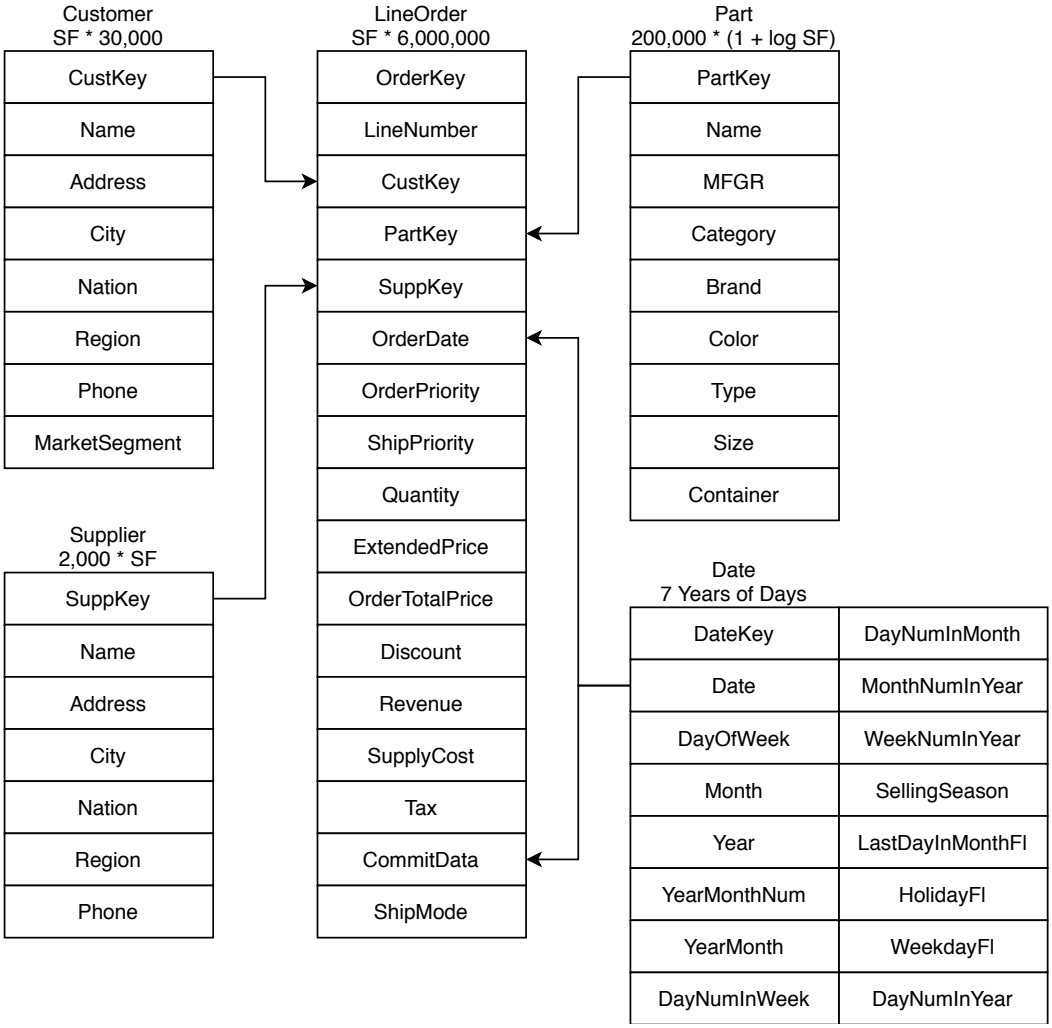


Figure 2.18: The database schema used in SSBM [37]

2.4.2 Star-Schema Benchmark (SSBM)

The need for SSBM stems from a divergence between data management as imagined (and used) by TPC-H and industry standards used in data warehouses [37] based on the works of Kimball and Ross [26]. Fig. 2.18 displays the schema, similar to the one of TPC-H but amended to take these considerations into account.

For instance, O'Neil et al. argue that long comment strings would not be stored along with actual business data in real-life conditions, that date data would usually be pre-computed and stored to reduce date-related processing in queries or that normalisation in the `LineItems` and `Orders` tables creates the need for many joins in queries that would not exist in actual data warehouses – where disk space would be sacrificed in exchange for better query processing times.

SSBM does not have the same approach to benchmarking as TPC-H. Whereas TPC-H have a specification of valid SQL queries to run, the SSBM specification has only four queries for each of which two to four instances are given. This approach is preferred

because it allows more control on the output data, for instance by picking a certain selectivity (this could not be done in TPC-H as it only specifies *valid* parameters but not how two parameter values should be related).
Furthermore, fortuitous parameter values in TPC-H may cause the database to access the same tuple several times, which would show an improvement in performance not as a result of intrinsic database features but rather by chance. This is unfair, and does not make benchmarks reproducible, which is one of the fundamental goals of TPC-H workloads [47].

SSBM has been reviewed and deemed better than TPC-H thanks to its simpler schema and query set, better fit to real-life data warehouses and stronger evaluation for query optimisers [45].

2.5 Legal and ethical considerations

The goal of this project is to write an embedded just-in-time compiler for prepared statements in SQLite using the LLVM/MLIR compiler infrastructure. To this end, the licenses of all tools and libraries used in the project should be examined.

The SQLite license is very permissive: SQLite is public domain[15].
This means anyone 'is free to copy, modify, publish, use, compile, sell, or distribute the original SQLite code, either in source code form or as a compiled binary, for any purpose, commercial or non-commercial, and by any means' [50]. That is also why contributions from external persons are not allowed – as they may put public-domain licensing at risk.
As a result, the implementation of a just-in-time compiler for SQLite is legal and can be done without further authorisations – even for commercial purposes.

LLVM uses the Apache License v2.0 with exceptions[16].
That means anyone is free to modify it, distribute it – even commercially – or make private use of it, as long as the original LLVM license is provided with it, that edited files carry a mark saying they have been modified, and that attribution notices from the LLVM source are kept in the modified work.
This means implementing a just-in-time compiled in SQLite using LLVM is legal and can be done without further limitations. For technical reasons, this project needs to modify certain parts of LLVM, they are thus duly annotated. All other source code and text is kept the same.

TPC provides an EULA for its TPC-H database generation tool[17] which states that

[15] More information on the SQLite license can be found at `https://www.sqlite.org/copyright.html`

[16] The complete LLVM license can be read at https://github.com/`llvm-project/llvm/blob/master/LICENSE.TXT`

[17] The TPC-H generation tool EULA can be read at http://www.tpc.org/tpc_documents_current_versions/source/tpc_eula.txt

'You may modify the Software' and that distribution is allowed assuming that (a) The software is distributed with the same EULA; (b) It is clearly stated that the software is available without charge from TPC; (c) No copyright should be attributed to the modifier; (d) No fee may be charged for distributing the software.
LLVMSQLite uses this tool for benchmarking purposes, without modifying it and redistributes it as-is, which is legal according to the EULA.

On the other hand, the SSBM database generation tool[18] , does not provide a license. Though, it is based on the TPC-H DB generation tool, which means its license must be the same according to TPC's EULA. As a result, the usage made by LLVMSQLite is legal. Though we need to modify the source to make it work on UNIX systems, this falls under the 'You may modify the Software' clause as all other conditions are respected.
As a result, the distribution of these various database generation tools as one package from the `LLVMSQLite_DBGen` repository is legal.

There exists, to our knowledge, no ethical considerations in this project. We do not believe it could be used to cause harm to human beings nor to systems.
Furthermore, should the performance improvements brought by these new just-in-time compilation techniques for SQLite have commercial outcomes, it would be up to the vendor to take ethical considerations about donating a one-time fee or a share of their profits to the organisations behind the open-source and free tools LLVMSQLite uses.

Finally, and as a result of these considerations, we made LLVMSQLite free *as-in-free-speech* and available publicly online at the end of 2020[19].

[18] The original repository can be found at `https://github.com/rxin/ssb-dbgen`, though this version does not compile on UNIX systems.
[19] The source code of LLVMSQLite and its dependencies are available on GitHub at `https://github.com/KowalskiThomas/LLVMSQLite`.

Chapter 3

Design and Implementation

In order to just-in-time compile SQLite virtual machine code, it is possible to 'unroll' the corresponding VDBE interpreter loop. To do this, we build an imperative programme, whose code consists of the implementation of each operation code in the bytecode, as defined in the interpreter.
Doing so improves code locality, mitigates branching costs – by specialising implementations – and reduces the number of function calls – by *inlining* functions on the *hot path*. It is the approach we take in that project.

Fig. 3.1 displays the process we propose for compiling SQLite prepared statements to native code:

- The SQLite Code Generator is used to obtain a VDBE for the statement;
- The VDBE programme is converted into a function in an MLIR module with operations from a novel MLIR dialect: VDBE-IR.
- The MLIR VDBE-IR module is lowered to LLVM IR Dialect via *MLIR lowering passes*. These can specialise the implementation depending on the context;
- The LLVM IR Dialect module is converted into a classic LLVM module via the existing conversion pass;
- The LLVM module is optimised using various optimisations techniques operating on IR, critical function calls are inlined;
- The optimised module is compiled to optimised native code;
- The just-in-time compiled coroutine is called from the C++ LLVMSQLite backend as the default `sqlite3VdbeExec` would be.

The following sections describe how this novel approach is integrated transparently into SQLite and how the steps towards native code are undertaken by LLVMSQLite.

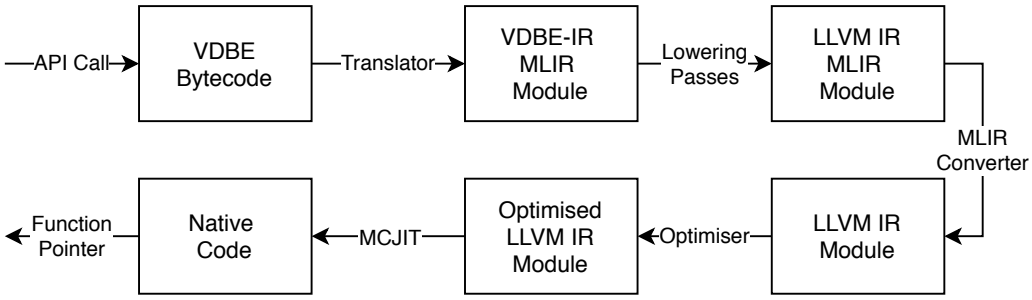


Figure 3.1: The steps in the VDBE JIT-compilation process

3.1 VDBE-IR: an MLIR dialect for VDBE bytecode

In this section, we introduce VDBE-IR, a novel MLIR dialect based on the SQLite VDBE bytecode language. We present its semantics, as well as the approach taken for replicating several behaviours used in SQLite that cannot be immediately translated when using native code instead of an interpreter.

3.1.1 The need for a VDBE bytecode intermediate representation

Obtaining an LLVM IR equivalent to the VDBE interpreter is not technically difficult. It is possible using `clang` to generate an IR module equivalent to the whole `vdbe.c` source, which contains `sqlite3VdbeExec`. As this function holds the implementation of every VDBE operation code, it would then be possible, to implement just-in-time compilation, to read through the VDBE bytecode and insert the implementation of each operation code using code templates derived from this IR file. Though, this approach is suboptimal in several ways.

First, in terms of run-time performance: the `clang`-generated IR is merely a translation of the C AST into a lower level. As a result, some idiomatic C statements are translated into low-quality code, even after optimisation by LLVM, which can result in poorer performance.
Fig. 3.2 shows an example of sub-optimised C code. In this case, the `func` function does not have side effects on `cur`. This means the two operations on `cur` could be merged into one. Though, LLVM has not been able to infer it from the code, and thus emits two memory changes (read, bitwise and, bitwise or and write): `%3, %4, %5, store` and `%6, %7, %8, store`. The code is thus sub-optimised.

Second, `clang` and LLVM lack an understanding of the *bigger picture*: they cannot see that some functions have implicit invariants that allow to simplify the code around call sites or that some values are constant throughout the VDBE execution. Although LLVM embeds efficient optimisers and alias analysers, some C idioms cannot be optimised well, especially since SQLite contains many manual optimisations, often on pointers, which do not play well with alias analysis.

```
            struct cursor {
              char type: 4;
              char cache: 2;
              // ...
            }
            /* ... */
            struct cursor* cur;
            /* ... */
            cur->type = CURTYPE_BTREE;
            func(&cur);
            cur->cache = CACHE_STALE;
%1 = load %struct.cursor%** CURSOR_ADDR
%2 = getelementptr inbounds %struct.cursor,
       %struct.cursor* %1, i64 0, i32 0
%3 = load i8, i8* %2, align 4
%4 = and i8 %3, -16
%5 = or i8 %4, 2
store i8 %5, i8* %2, align 4
call void @func(%struct.cursor* %1)
%6 = load i8, i8* %2, align 4
%7 = and i8 %6, -49
%8 = or i8 %7, 16
store i8 %8, i8* %2, align 4
```

Figure 3.2: A simplified excerpt from `sqlite3VdbeExec`, along with an optimised LLVM IR representation of it

Finally, the VDBE interpreter has been written for generality and thus does not assume anything in the code of polymorphic operations. For instance, the `Next` operation code – which advances a cursor to the next element – can be used on *B^+-Tree cursors* but also on *Sorter cursors*. This means that at run-time, even though the type of a cursor is constant throughout execution, both branches will be considered by the interpreter (like on Fig. 3.3.a).

Since `Next` is often used in the hot path (for processing each tuple in a table), this can lead to a degradation in performance. In a just-in-time compiler, this phenomenon can be avoided by *lifting* (or *hoisting*) the implementation choice at run-time, and only inserting the branch for the right cursor type.

Fig. 3.3 shows an instance of *lifting* applied at JIT-compilation time. In the first case, the general implementation is copied, both branches are kept. In the second case, the value of `P5` has been checked at compile-time by the JIT and only the relevant branch has been inserted.

This results in both a smaller code – which is important for instruction cache efficiency – and a more predictable code. Failing to seize that opportunity to optimise native code at a high-level would lead to degraded performance: the branching cost in the JIT-compiled statement would be approximately equivalent to the one in the interpreter.

```
// Operation 15
// P1 = Cursor index
if (P5 & CURTYPE_BTREE) {
    advance_btree(aCur[P1]);
} else {
    assert(P5 & CURTYPE_SORTER);
    advance_sorter(aCur[P1]);
}
```

(a) Cursor next implementation without *lifting* applied

```
// Operation 15
// P1 = Cursor index
// P5 has been checked at compile time.
// P1 is thus known as a B-Tree cursor
// Only the relevant implementation
// is inserted.
    advance_btree(aCur[P1]);
```

(b) Cursor next implementation without *lifting* applied

Figure 3.3: A simplified `Next` operation with and without *lifting*

On a more practical note, using the default implementation also raises the issue of traceability in debugging. Resolving issues in just-in-time compiled code is a difficult task: the generated native code is often very long and lacks both the context and the typing information provided by a higher-level representation.
MLIR has also been designed with ease-of-debugging in mind [30] and consequently, using the MLIR framework as an intermediate when generating native code theoretically guarantees a better debugging experience and an improved debugger integration.

Although generating LLVM IR programmatically can also be achieved with the classic LLVM API – without the need for the concepts of *Dialect* or *Lowering* – adding an intermediate step in the just-in-time compilation process in the form of an MLIR module is a step taken towards the future.
Indeed, though it is not the goal of this project, the MLIR code could be used for pattern-based rewriting and optimisation, as well as static analysis of the bytecode, from which optimisations could be derived.

In the following paragraphs, we present the novel VDBE-IR MLIR Dialect. It is designed to be as close as possible to the original VDBE bytecode language. It also aims at lifting run-time decisions which can be made at compile-time based on the operation codes themselves and their flags.

3.1.2 Using MLIR to model VDBE bytecode

In VDBE-IR, each VDBE operation code maps to at least one dialect operation. These operations have the same name as the original operation code.
Some VDBE operation codes share a common VDBE-IR Operation. For instance, `Add`, `Subtract`, `Multiply`, `Divide` and `Remainder` (the *arithmetic family*) map to the same `Arithmetic` operation. That is a direct consequence of the fact they share the same implementation (with different branches) in the interpreter. It is then up to the lowering pass to determine what the original operation code is by reading the VDBE bytecode listing and emitting the right LLVM IR instructions into the module.

```
case OP_Compare:
  int iCompare = compareVal(...);
  if (P5 & STORE_P2) {
    // Flags: store result mode
    regs[P2] = iCompare;
  } else if (iCompare) {
    // Flags: jump mode
    goto jump_to_p2;
  }
```

(a) Excerpt from the `Compare` implementation

```
case OP_OpenRead:
  // Determine nbr of fields
  // based on P4.
  if (op.p4type == P4_KEYINFO) {
    pKeyInfo = P4.pKeyInfo;
    nField = pKeyInfo->nAllField;
  } if (op.p4type == P4_INT32) {
    nField = pOp->p4.i;
  }
```

(b) Excerpt from the `OpenRead` implementation

Figure 3.4: Example of polymorphic operations

VDBE-IR operations have the same inputs as the VDBE operation codes (three integers `P1`, `P2` and `P3`, a pointer-sized value `P4` and a 16-bits wide `P5`), plus a 64-bits wide programme counter named `pc`). In the interpreter, the programme counter is implicit (it is the loop index), but this information cannot be used in VDBE-IR as it unrolls the interpreter loop (and thus removes the concept of loop index).
All inputs in VDBE-IR are MLIR *attributes* (constant MLIR values): in VDBE-IR, MLIR (non-constant) *values* are not needed. Indeed, most operation parameters are known at compile time and do not change during the execution of the prepared statement. Although some parameters vary (in self-altering operation codes), an implementation that does not use values is preferred (See 3.1.3). That is mainly because scoping would erase them when the control flow returns from the just-in-time compiled function.
VDBE programmes also use dynamic data (pointers to SQLite objects, blobs, strings, *etc.*). In this case, lowering passes do not *hard code* the pointer value in the module but rather emit code for gathering the parameter value from memory (by reading the `P4` value from the VDBE instance).

VDBE-IR does not register new MLIR types: it only relies on types from the LLVM dialect (`mlir::LLVMType`'s), which are direct translations from SQLite C-defined types. More information on the types VDBE-IR uses can be read in 3.3.4.
As a result, VDBE-IR is not type-safe: VDBE operations themselves are loosely typed by design. Depending on the context, the integer value `1` can be interpreted as 'the integer value `1`', 'the table whose index is `1`', 'the cursor whose index is `1`', *etc.*
How a value is interpreted is either determined by the operation code itself – the first parameter of `Next` is always a pointer index – or by the flags in `P5` for polymorphic operations – the fourth parameter `P4` of `OpenRead` can be either a pointer to a SQLite `KeyInfo` structure or an integer which designates a table index.
To illustrate that behaviour, consider Fig. 3.4. In the `Compare` case, `P5` is checked against the `STORE_P2` flag. If set, `P2` is interpreted as a register index, the comparison result is then stored in that register. Otherwise, `P2` is interpreted as a target programme counter, to which the VDBE interpreter jumps if the registers compared are not equal.

In the `OpenRead` case, the interpreter checks the `p4type` field of the current operation. If it is the `P4_KEYINFO` flag, `P4` is interpreted as a pointer to a `KeyInfo` structure from which the number of fields should be extracted. On the other hand, if it is the `P4_INT32` flag, `P4` is interpreted as an integer: the number of fields.
In certain cases, the values of parameters can be constant-folded and branches can be eliminated. In a general way, using `P5` allows for branch elimination as it often designates directly what 'version' of the operation code is to be executed.

Appendix 6.4 displays an example function defined in the VDBE-IR dialect, for the `SELECT * FROM Relation` statement.
The function does not only contain VDBE-IR operations, but also operations from the `standard` MLIR dialect (such as `constant`, used to assign constant integers to SSA values) and the LLVM dialect. Values that are useful to all lowering passes – for example, the in-memory location of the array of registers, or the location of the array of operations (the bytecode) – are computed at entry. This allows lowering passes to easily refer to them when generating LLVM IR.
The actual VDBE-IR operations come after (in the example, from block `^bb3` on to `^bb16`).

The just-in-time compiled function works in a way similar to the interpreter. As a result, the same mechanics used in the interpreter can be used during execution (for instance, on manifest typing). But there is more: this also allows to use SQLite built-in API functions.
Consequently, the VDBE Dialect does not need to define operations to instantiate VDBE registers nor output registers, as they all are already allocated and initialised by the SQLite code generator when the SQL statement is prepared and the VDBE instance created.

Note on manifest typing MLIR dialects, as well as LLVM IR, do not support dynamic typing. LLVM IR only allows integral types and compound types (See 3.3.4). MLIR allows some flexibility with template-like type definitions (for instance, one can define a general matrix type whose dimensions are only known to be integers, without a precise value).
This is not dynamic typing though: dynamic typing requires the type to be determined only at run-time, whereas MLIR flexible types require the full type to be known at JIT-compile time.
That is why, to implement manifest typing, LLVMSQLite uses the same mechanisms as the interpreter. It extracts type information from SQLite Value structures and then interprets the data based on that knowledge and using *bitcasts* (casting an IR value to another unrelated type, for instance `void*` to `char*`). This decision also allows to have no overhead between SQLite internal APIs and LLVMSQLite, as no conversion from and to Value types needs to happen.
In practice, profiling shows that these mechanisms and more broadly manifest typing cause overhead. To mitigate it, we propose a partial statically typed VDBE-IR implementation in 3.4.

```
OP_AggStep:
  // (code to initialise
  //  the FuncDef struct)

  // Update op code
  vdbe->aOp[pc].opcode
    = OP_AggStep1
  // Fallthrough

OP_AggStep1:
  // Compute aggregate value
  // ...
  break;
```

(a) Interpreter implementation of `AggStep`

```
; OPCODE_ADDR = &vdbe->aOp[pc].opcode
%AggStepJmp:
  %opcode = load i32* %OPCODE_ADDR
  %opcodeIsAggStep1 =
    icmp eq i32 %opcode, i32 OP_AggStep
  condBranch
    opcodeIsAggStep1 %AggStepComp, %AggStepInit
%AggStepInit:
  ; Setup function definition
  store i32 OP_AggStep1, i32* %OPCODE_ADDR
  branch %AggStepComp
%AggStepComp:
  ; Compute aggregate
```

(b) JIT implementation of `AggStep`

Figure 3.5: Simplified C implementation of `OP_AggStep`/`OP_AggStep1` and simplified IR that allows self-altering operation codes

3.1.3 Self-altering operations

Some SQLite operation codes are self-altering (their implementation modifies the VDBE bytecode at run time). An example of this is the `AggStep` operation code. The first time the `AggStep` operation is executed, it needs to setup a `FuncDef` – a structure that keeps track of the aggregate function in use. This only needs to be done once, as the same `FuncDef` can be used for subsequent iterations. That is why, after defining the aggregate function, the default implementation modifies the operation code value to `AggStep1`, which allows to jump over the initialisation routine.

Reproducing this behaviour in our implementation is necessary. Not only does it lead to much better performance – as some of the one-time-only operations are expensive in terms of memory allocation – but it is also used for tracking processes such as transaction start, which should only be done once per VDBE execution.

Unfortunately, doing that natively is impossible in MLIR. Changing the value of parameters is possible – by using a global variable initialised to the initial parameter value – but changing the operation code value does not make sense, as the concept of VDBE operation code does not exist in the generated LLVM module nor in native code.

In LLVMSQLite, this behaviour is thus implemented by modifying the data in the VDBE instance, like the default implementation does it. Though the compiled prepared statement does not interpret the bytecode, it uses it and the VDBE registers for storage purposes.
In self altering operations, the JIT-issued implementation reads the bytecode from

the VDBE (operation codes and parameters), and executes different branches depending on it.
To illustrate this, consider Fig. 3.5. In the interpreter, the `AggStep` operation code is implemented using a first `case`, `AggStep`, that sets up the aggregate then continues to a second `case`, `AggStep1`, which actually computes the aggregate step. In `AggStep`, once the aggregate has been initialised, the VDBE operation code for the current programme counter is modified to `AggStep1`. That way, subsequent executions at that programme counter will skip aggregate initialisation, and only compute it.
To replicate that behaviour in the JIT compiler, we split the `AggStep` implementation in three blocks. The first block (`AggStepJmp`) reads the operation code from the VDBE. If the operation code is `AggStep`, execution continues at block `AggStepInit`. Otherwise, execution continues at block `AggStepComp`. As well as setting up the aggregate, the IR in `AggStepInit` modifies the bytecode in the VDBE and changes the operation code to `AggStep1`. That way, the next time the code section is executed, the code in `AggStepJmp` will jump to `AggStepComp` instead of `AggStepInit`, effectively leading to the same result as in the interpreter.

As a result and for practical reasons, the VDBE instance for the session is accessible throughout the execution of the MLIR JIT-compiled code. Any state variable is read from and stored in the VDBE. The JIT-compiled code only allocates local and temporary variables.
This is for two reasons. First, SQLite emits tuples by exiting VDBE execution. That means any local, stack-allocated state variable in the just-in-time compiled function would be lost when a tuple is emitted. This in turn would prevent some VDBE behaviours to be implemented in a lightweight way.
Second, using the VDBE instance to store dynamic data allows the default interpreter to use it too. This is required when using the interpreter along with the just-in-time compiler, which is useful and discussed later (See 4.1).

All VDBE Dialect operations have a lowering pass that translates them to LLVM IR using the MLIR rewriting API. Some MLIR dialects, such as `loop` or `linalg`, have high performance implementations for complex mathematical operations, loop vectorisation or mathematical operations on vectorised data.
This project, though, does not aim at using these MLIR dialects, as they are meant to be used to lower-level computational-heavy workloads or tight loops, which is not the use case in VDBE-IR.

```
; ... Operations 1 to 5 ...
^Ins6:
  vdbe.IfPos [^Ins2,    ^Ins7] { Register = 8 }
  ;            ^^^^Jump  ^^^^Fallthrough
^Ins7:
  vdbe.Halt
```

Figure 3.6: Implementation of the Jump behaviour in VDBE-IR. Operation 6 is `OP_IfPos` and may jump to Operation 2 or fallthrough to Operation 7 (`OP_Halt`).

3.1.4 Allowing jumps to other operations

Many VDBE operations jump to another operation using the programme counter. In the interpreter, that is done by modifying the value of the programme counter; the interpreter loop then continues execution at another point during the next iteration. LLVM IR does not provide a mechanism for jumping to arbitrary programme locations or instructions, which means reproducing this feature in MLIR (or LLVM IR) is not straightforward.
The only way to jump to an instruction in LLVM IR is to use an unconditional branch (which jumps to another basic block). That is why, when populating the MLIR module with VDBE-IR operations, the start of each operation is placed in its own basic block (lowering passes may expand the region to more than one block, if the implementation has several branches).
That way, it is possible to easily jump from one operation to the other, simply by branching to the corresponding block (see Appendix 6.4 and Fig. 3.6).

The concept of *fallthrough* does not exist in LLVM IR nor MLIR. A *terminator* (as a branch operator or a return operator) must always terminate its block (*i.e.* no other operation should follow it).
That is why, for operations that may have a fallthrough behaviour (for instance, `IfPos` which jumps to a specified operation only if the value of a specified register is positive), the VDBE-IR operation takes two successors: the jump target (for the 'positive register' case in `IfPos`) and the fallthrough (immediate next block) (for the 'non-positive register' case in `IfPos`). Fig. 3.6 illustrates this.

Like in the default VDBE implementation, running multiple steps of a prepared statement is done by calling the `Step` function multiple times. The SQLite interpreter can resume execution in a straightforward manner because the state of the VDBE is not lost when exiting from the interpreter function (it simply reads the `pc` attribute from the VDBE). As explained before, all the data is stored in the VDBE (the `Vdbe*` passed to the JIT-compiled function).

3.1.5 Conclusion and optimisations

One of the goals of just-in-time compilation is to lift some operations and eliminate irrelevant branches at JIT compile-time when they can be.
Doing this kind of optimisation requires to take decisions regarding the end to which things should be 'optimised in'. As an example, we take the `Integer` operation, which stores a given integer into a register. Three levels of optimisation can be imagined:

- No optimisation. The value of the integer that should be stored in the register is read at run-time from the VDBE bytecode. This requires one memory read per execution;
- Partial optimisation. The value of the integer that should be stored is assumed to be constant throughout the execution of the prepared statement and is determined (and hopefully constant-folded) at native code generation;
- Complete optimisation. The value of the integer that should be stored is considered to be a characteristic of the bytecode. The same bytecode with a different value for the parameter would be as considered a different VDBE programme. This guarantees constant folding at the IR level.

Choosing the level of constant-folding is a trade-off. Although constant-folding as early as possible may make other optimisations possible, it also means that an already-compiled prepared statement cannot be re-used if the value of at least one parameter differs.
As the cost of creating and optimising an LLVM module is not negligible, this needs to be taken into account. Indeed, a just-in-time compiler that would make VDBE execution much faster but VDBE preparation even slower would be of no practical use.

3.2 Integration in SQLite

The goal of this project is to create a JIT compiler for executing SQLite prepared statements. It thus seems natural that only VDBE-related operations in the SQLite source should be affected by this work.
That is why we propose a drop-in replacement for the SQLite interpreter function, `sqlite3VdbeExec`. This is the only function that higher layers use to interact with VDBEs.

Our custom is only a wrapper around two other functions: the interpreter and the VDBE JIT API. Only the latter compiles VDBEs, keeps state of JIT-compiled statements and executes them.

Fig. 3.7 displays a schematic representation of the LLVMSQLite architecture and shows how its components communicate: the steps taken to create an MLIR module for a prepared statement and run it are the following.

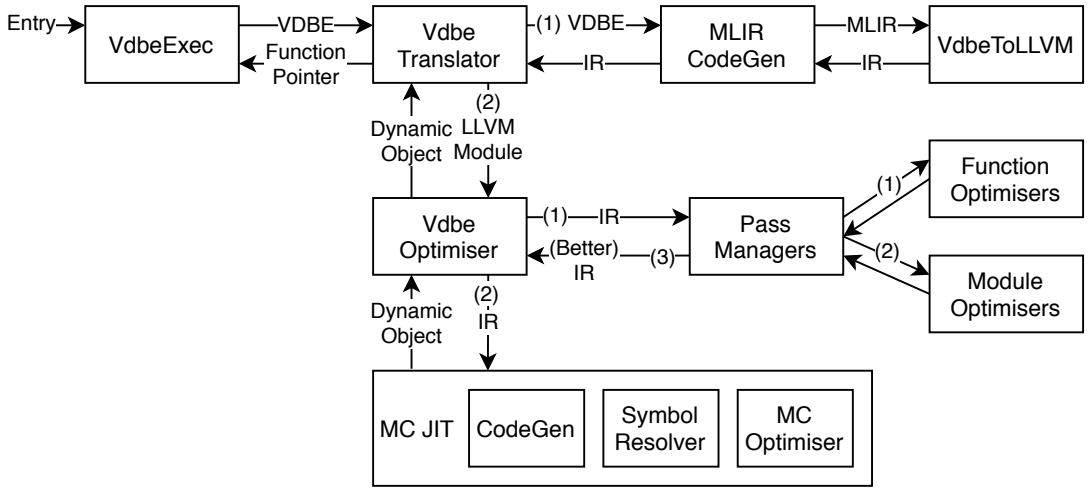


Figure 3.7: The architecture of LLVMSQLite

- A `VdbeTranslator` is constructed with the VDBE instance/prepared statement which holds the bytecode for the query;
- The `VdbeTranslator` creates a VDBE-IR module with a single function. This function will be used to replace the interpreter for that statement. The `VdbeTranslator` then inserts VDBE-IR operations in that function based on the bytecode of the prepared statement;
- VDBE Dialect to LLVM Dialect passes are applied to generate an LLVM-IR-only MLIR module;
- The LLVM-IR-only module is converted to an LLVM module;
- The LLVM module is optimised by the `VdbeOptimiser`;
- The MCJIT API compiles the module and returns a JIT-compiled function pointer that can be called from the high-level API.

From the SQLite high-level API, the integration of the JIT is completely transparent.

Once the LLVM module has been generated and compiled, it is possible (and desirable) to keep a reference to it in order to re-use it for the next VDBE step (on the next `sqlite3VdbeExec` call).

To connect the SQLite APIs and the JIT, we use the `VdbeRunner` data structure. Though all VDBE lowering passes exist on their own and can be seen as part of the LLVM/MLIR infrastructure used by SQLite, LLVMSQLite still needs a *wrapper* to use this API. That is what `VdbeRunner` is.
It is built with a pointer to a VDBE instance and produces an equivalent LLVM module by building an MLIR module, lowering it to LLVM, importing inlined functions, applying optimisations on it and finally generating native code through MCJIT.
`VdbeRunner`'s are not single-use objects. Indeed, to process a statement, the VDBE is iteratively executed while it returns `ROW`, until it returns `OK`. Every time query processing is resumed, the bytecode execution is resumed in its previous state: if

the `ResultRow` operation that caused the interrupt was at programme counter n, resuming query processing will make control flow go at programme counter $n + 1$, without executing operations from 0 to n.
When a `VdbeRunner` is created by the bridging function, it is kept in a cache – a map whose keys are `Vdbe*` and values are `VdbeRunner*`'s – which is probed before running a VDBE. If a `VdbeRunner` is in the cache for that VDBE, it is used and execution is resumed in its previous state. If none exists, a new `VdbeRunner` is constructed and used to build a just-in-time compiled function, and then to execute it.

When the JIT-compiled function yields, the bridge function checks whether the return code is `SQLITE_DONE`. If so, it frees the `VdbeRunner` and the LLVM module. Indeed, a return code of `SQLITE_DONE` indicates that the execution of the prepared statement should not be resumed. Since SQLite pools VDBE instances, deleting the `VdbeRunner` is necessary to force a new LLVM module to be generated if the same VDBE instance is executed again later (with a different bytecode).
This solution matches the constraints set earlier:

- It is totally invisible to the rest of SQLite. No other component of SQLite needs to be modified, the VDBE execution component works in isolation;
- It is possible to run multiple VDBEs concurrently. This happens for instance upon the first query, when SQLite first needs to probe the master record of the database file for existing relations and their definition;
- It supports recycling VDBEs, which once again allows not to change the behaviour of the rest of SQLite.

3.3 Lowering passes

A fundamental concept in MLIR is the *lowering pass*. Like LLVM *optimisation passes*, lowering passes pattern-match instructions in a module and replace them with one or more other operations.
New operations may or may not be from the same dialect, the only condition for a lowering pass to be valid is that when a match is found in the source, running the pass should always replace or remove the original operation.
The VDBE-IR dialect defines one lowering pass per VDBE-IR operation.

3.3.1 Design of the main just-in-time compiled function

In LLVMSQLite, the interpreter and the bytecode it executes are both replaced with a single just-in-time compiled function.
The outline of this function is schematically represented in Fig. 3.8 :

- The first part of the function is the *Resume Switch*, which implements resuming execution and jumping to arbitrary programme counters when stepping through the query results. It is justified and explained more in-depth later;

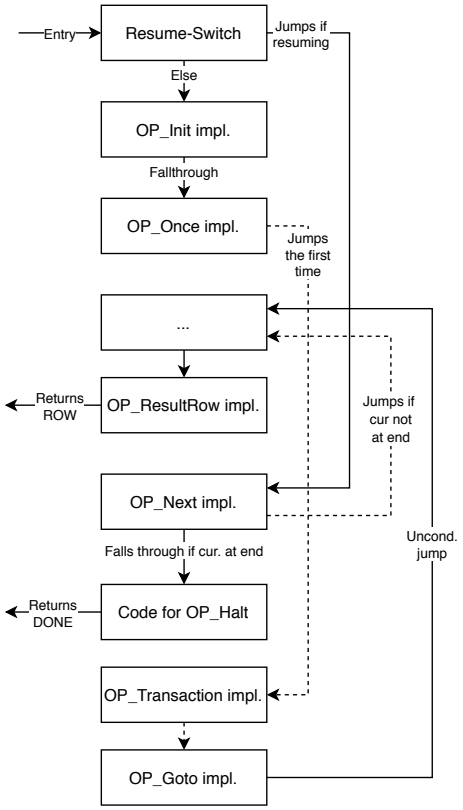


Figure 3.8: Outline of the main JIT-compiled function

- The second part consists of the operations themselves. Each VDBE operation is translated to (at least) one IR basic block[1], which makes it simple to emit jumps from one instructions to others easily at compile-time for operations that need it.

Although LLVM IR allows to jump from a Basic Block to another one, Basic Blocks are compile-time only concepts and cannot be considered as dynamic values. As a result, it is not possible to specify the destination of a `branch` instruction at run-time.

This is an issue for VDBE-IR. Indeed, although most VDBE terminator operations jump to programme counters that are compile-time constants (`Goto 5` always jumps to operation 5), some jump to an operation whose programme counter is only known at run time (for instance, the `Yield` operation interprets the value in register `P1` as a programme counter value and jumps to the corresponding operation). Although the value of this register is limited to the programme counters at which coroutines are called, it is impossible to assume that they are generally constant.

Another case where VDBE needs to jump to an arbitrary programme counter is when execution is resumed after emitting a tuple. When the interpreter returns, it first stores the current programme counter in the VDBE. When it is called the next time,

[1] In LLVM, 'A basic block is simply a container of instructions that execute sequentially.' [32] It can be jumped to and from using branching instructions.

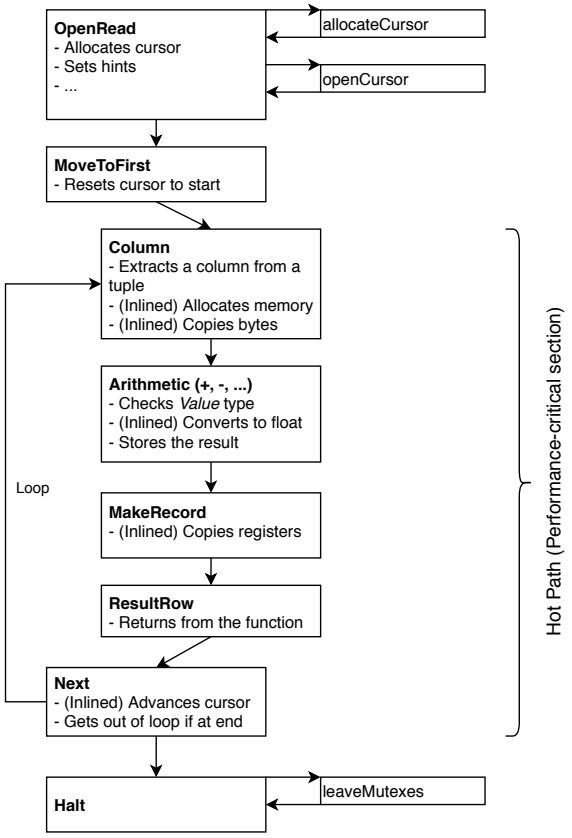


Figure 3.9: Example of the simplified output JIT-compiled query for `SELECT [Column] + 1 FROM [Table]`

execution can be resumed at the next operation. Once again, the value of the programme counter is not completely arbitrary: a VDBE can only resume after emitting a tuple.
As the concept of *programme counter* does not naturally exist in the just-in-time compiled function, we can use the same mechanism that we use for operations that jump to non-constant programme counters.

Translating this simply to LLVM IR is not possible: as explained, the destination of a `branch` (or `condBranch`) instruction needs to be known at compile-time. As a solution to that problem, we propose the *Resume Switch*.

When an operation returns from the just-in-time compiled function, it writes the current value of the programme counter in the VDBE. As a reminder, the programme counter value for each operation in the bytecode is known at compile-time. It then returns with the expected return code, like the interpreter would do.

When the VDBE step function calls back the just-in-time compiled function to resume execution, control flow goes to the start of the function, *i.e.* to the Resume Switch.
The Resume Switch is a large *switch-case* block that has one case per possible programme counter and could be thought of as the following:

```
Entry:
  pc = vdbe->pc
  switch(pc) {
    case 0:
      branch Instr0
    case 1:
      branch Instr1
    # Other cases...
    default:
      # Fallthrough to 0
  }
Instr0:
  ...
```

In the Resume Switch, the destination of each conditional branch is known at JIT compile-time. Though, it also allows to jump to any programme counter at run-time. Thus, it is a valid solution to the problem.

Previous versions of LLVMSQLite used a *Jumps Section*, an if–else-if chain that would match the programme counter against possible values. That was due to the fact MLIR does not support the `switch` instruction by default. This approach was costly – jumping to the last programme counter would first take going through all the others. Although the *switch-case* solution is conceptually equivalent, LLVM's `switch` instruction is more efficient. Indeed, during optimisations, a balanced search tree is generated for cases, which allows to find the target operation in a lower $\log(n)$ complexity (if n is the number of bytecode instructions) [27]. The performance of this approach and its limitations are evaluated in 4.4.1.

3.3.2 Modifications to the MLIR API

The MLIR LLVM exposes all basic LLVM instructions. Unfortunately, this is not sufficient when implementing lowering passes for VDBE-IR.
Indeed, the LLVM IR language also has *intrinsics* that either provide better performance than system calls (replacements for `memset` and `memcpy`) or provide more advanced functionality to LLVM (like memory management features) [33].

Throughout the years, many intrinsics have been added to LLVM IR, some of which are only seldom used. By default, the LLVM Dialect does not support building these instructions. To mitigate that limitation, we propose modifications in the MLIR API to allow JIT-compilation of VDBE bytecode, namely to add support for these intrinsics. In the following paragraphs, we present the modifications brought to the MLIR API.

```
Instr3:
    %record = alloca %UnpackedRecord
    ; ...
    branch Instr4
Instr4:
    ; ...
    branch Instr3
```

Figure 3.10: Stack overflow caused by the `alloca` instruction

Memory Intrinsics LLVM provides replacements for `memset` and `memcpy` (with a `memcpy_inline` specialisation to expand `memcpy` in cases where the memory block width is a *constexpr* [33]). To add support for building these, we add a new TableGen[2] operation class, `LLVM_MemCpyLikeOp`, which takes three arguments and produces nothing. We then use that class to add support for `memcpy` and `memcpy_inline`. We also add a `memset` class for this operation.

Stack management The LLVM stack allocation instruction (`alloca`) allocates stack space for any type by changing the current top-of-the-stack pointer. Stack memory does not need to be freed (as opposed to heap memory), but enlarging the stack can still cause issues when nothing ensures the lifetime of objects is correctly defined.
Fig. 3.10 shows a case where the lack of scoping mechanisms causes an error. Although the `record` value can be seen as *local* to `Instr3`, the `alloca` instruction is executed every time the programme jumps to basic block `Instr3`. Precedent `record`'s are never deallocated, which causes the stack to grow indefinitely.
To solve that, LLVM provides the `stackSave () -> i8*` and `stackRestore i8* -> ()` instructions. High level languages like C can use these intrinsics to implement *scoping*.
In our case, every VDBE operation that allocates memory on the stack has its implementation surrounded with a `stackSave` and a `stackRestore`, to prevent stack overflows. To support building these, we add a new TableGen operation class for each intrinsic (these classes internally rely on the LLVM instruction builder).

[2] LLVM TableGen is a declarative code-generation tool. It allows developers to declare various components – in this case, MLIR Dialects, Operations and even transformation passes – using a high-level DSL. These declarations are then converted into C++ header and source files, which call LLVM APIs. Although the same result could be achieved with manually written C++ code, the rationale behind TableGen is to reduce the verbosity of writing LLVM components. The goal is also to make third-party tools evolve better as LLVM grows and changes. Indeed, each new LLVM version changes its interface – it is even more the case for new projects like MLIR – which can make code that relies on the old interface not work anymore. TableGen allows to consider the LLVM C++ interface as an implementation detail and abstract it away using the higher-level TableGen language.

Support for `switch` To allow resuming VDBE execution, LLVMSQLite uses an LLVM `switch` instruction. Indeed, `switch` LLVM IR instructions can be optimised more than an if–else-if chain. This instruction is not supported by default in the LLVM MLIR Dialect, which is why our custom adds support for it.

Call to arbitrary values Like C, LLVM IR supports indirect calls. A *direct call* is a call instruction whose destination is known at compile time (a function that has been declared or defined previously in the IR, for instance). An *indirect call* is a call whose destination is an arbitrary value, like the return value of a function, and that thus is not known at compile time.
The LLVM Dialect supports direct calls (the first operand is an `mlir::LLVMFuncOp`, an LLVM function declaration), but does not support indirect calls (in which case the first operand is an `mlir::Value` of any type). Though, SQLite uses a lot of these calls – among others, for computing aggregates through its `FuncDef` type.
To add support for calling any value, we modify the `call` TableGen class and allow the constructor to take any `mlir::Value` as a first operand.

3.3.3 Design and implementation of a custom operation builder

The MLIR API provides an `OpBuilder` class that allows to insert operations in MLIR modules. It is based on `OpBuilder::create`, a template function that manages the state of the builder (like its position in the module), makes it consistent and keeps track of objects in memory.
The strength of the `create` function is that it is universal and can be used to emit operations from any dialect – MLIR heavily relies on C++ templates to register new operations in the API. Unfortunately, this strength is also its main weakness: actual calls to operation constructors are hidden behind an `std::forward`, which makes the API completely opaque.
Furthermore, `OpBuilder` comes with a high verbosity, and emitting even seemingly simple operations can span multiple barely readable lines of C++ code.

That is why, in LLVMSQLite, most lowering passes are implemented using an custom wrapper around `OpBuilder` whose primary goals are (1) to reduce the verbosity of operation building (2) to allow some short-hands on the front-end. It only targets the LLVM IR dialect.

From the C++ API, MLIR operations can be seen as `Operation` instances, on which one can act by changing the operands, or as a `Value` (the value resulting from the operation).
Figures 3.11 and 3.12 show C++ usage of an MLIR `OpBuilder` as well as the generated IR. This example demonstrates how much C++ code the API needs to generate a rather short IR programme. Namely, the use of `std::forward` makes the explicit cast to `mlir::ValueRange` necessary, and the use of constants rather cumbersome, as the user needs to first declare them as MLIR values before being able to use them. To mitigate these issues, LLVMSQLite uses different utility classes, defined thereafter.

```
// LOC is a macro that gives the current
// Location in code (file, line, column)
// 32-bit constant of value 0
auto cst0 = builder.create<ConstantOp>(LOC,
    mlir::getIntegerAttr(32, 0));
auto cstOpenMode = ...;
/* Other constants */

auto call = builder.create<CallOp>(LOC,
  f_allocateCursor,
  ValueRange {
    cstX,
    cstOpenMode
  }
);
auto result = call.getResult(0);
auto typeAddr = builder.create<GEPOp>(LOC,
  call,
  ValueRange {
    cst0,
    cst1
  }, 0
);
auto typeVal = builder.create<LoadOp>(LOC, typeAddr);
auto typeValPlus8 = builder.create<AddOp>(LOC, typeVal, cst8);
builder.create<StoreOp>(LOC, typeValPlus8, typeAddr);
```

Figure 3.11: The C++ code required to generate the IR in Fig. 3.12. Note that `typeVal` is an `Operation` instance but is used as (implicitly converted to) a `Value`.

ConstantsManager is a code generator that inserts the necessary MLIR operations to get a constant integer value. Overloading of `operator()` allows elegant usage of constants in code generation. Another overload allows for inserting constant pointers. Although the use of the `IntToPtr` instruction is discouraged by the LLVM documentation – as it hampers alias analysis – it is sometimes necessary and sound, for inserting null pointers for example.

```
mlir::OpBuilder builder(...);
ConstantsManager constant(builder);

/* Code generation */
auto call = builder.create<CallOp>(LOC,
    ...
    // Integer of value 0 and 32 bits wide:
    auto cst0 = constant(0, 32)
    ...
);
```

```
; Allocate a cursor
%1 = call %struct.Cursor* allocateCursor(i32 X, i32 18)
; Get the address of the second struct element (called type)
%2 = getelementptr %struct.Cursor* %1, i32 0, i32 1
; Load the current value of type
%3 = load i32* %2
; Add 8 to it
%4 = add i32 %3, i32 8
; Update the cursor structure
store i32 %4, i32* %2
```

Figure 3.12: The IR programme generated by the C++ code in Fig. 3.11

Printer is a code generator that inserts the boilerplate required for printing information at run-time. As explained earlier, calling mechanisms in the MLIR C++ API are rather cumbersome for simple functions, let alone ones that use strings or static data in general.
Though, printing run-time information is invaluable, especially as debuggers cannot read debug information from the just-in-time compiled function [51]. Solving this issue takes another wrapper, which draws its simplicity from the fact that just-in-time compiled modules and host processes share the same memory. This avoids having to copy strings into the module.
When called, the printer creates an `i8*` (equivalent of a `char*` in C) by emitting an `IntToPtr` operation whose integer constant is the (static) `char*` passed to `print`, cast to a wide integer type. Then, it uses just-in-time static type information on the passed MLIR Value to determine what printer function should be called – overloading does not exist in LLVM IR. After determining which function should be called, it emits the `CallOp` with previously specified values so that the printing function is called at run-time.

```
mlir::OpBuilder builder(...);
ConstantsManager constants(builder);
Printer print(builder, constants);

/* Code generation */
auto myPointer = rewriter.create<GEPOp>(...);

// LOCL gives the location in file and the file name as a char*
print(LOCL, "Address in myPointer:", myPointer);
```

Assert works in a way similar to `Printer`, but has a role analogous to the `assert` macro in C. When called, it inserts the code to call a C-defined function with two parameters: the condition value (which should be null or not null) and the location information.
Calling a C-defined function (rather than defining `assert` in a fashion similar to its C definition) allows to insert a breakpoint in its C implementation and stop JIT execution when an assertion fails.
Once again, passing location information to the `assert` operator is done to improve the debugging experience: having it allows to print what call to `assert` in the lowering passes caused the crash, and thus where the bug should be primarily looked for.

```
mlir::OpBuilder builder(...);
ConstantsManager constants(builder);
Assert myAssert(builder, constants);

/* Code generation */
// Value is an i32
auto value = rewriter.create<LoadOp>(...);
{ // Assert that value != 0
  auto valueNotNull = rewriter.create<ICmpOp>
    (LOC, ne, value, constant(0, 32));
  myAssert(LOCL, valueNotNull);
}
```

CustomOpBuilder completes this collection of utilities by making code generation much less verbose and much easier thanks to automated constant value generation, implicit casts and custom operation wrappers. The following paragraphs describe the general usage of the custom operation builder. For a complete specification, see Appendix 6.5.

First, `CustomOpBuilder` defines wrappers for all LLVM IR operations useful in this project. All these wrappers return MLIR `Value`'s, which is the only use LLVMSQLite makes of `OpBuilder::create` results. Second, all lowering passes use a macro that defines several lambdas that wrap the wrappers to make their usage less verbose.
Fig. 3.13 displays a usage example for the `load` wrapper. Note that, using the classic `mlir::OpBuilder`, the loaded address would have had to be cast to a `Value*` first.

Second, `CustomOpBuilder` can take C values instead of MLIR `Value`'s. This is especially useful to guarantee type correctness of values. `CustomOpBuilder` defines a method to determine the width of an MLIR integer (or pointer-to-integer) `Value` based on its static type information. For example, the `Store` operation inserter uses this information to let the user pass a C integer, from which the inserter can build a constant MLIR `Value` whose width is determined automatically based on the type of the pointer. Using this information makes the use of the just-in-time compiler more robust for the future: SQLite internal types are unspecified and may change,

```
Value* CustomBuilder::insertLoad(Location loc, Value* addr) {
  // Builder is an MLIR::OpBuilder
  return builder.create<mlir::LLVM::LoadOp>(loc, addr);
}

/* Lambda definitions */
#define MYBUILDER_OPERATORS \
auto load = [&](Location loc, Value* addr) -> Value * { \
  // customBuilder has been captured \
  return customBuilder.insertLoad(loc, addr); \
};

/* Usage in codegen */
auto pointer = ...;
auto value = load(LOC, pointer);
```

Figure 3.13: Custom builder `load` operation example along with lambda-wrapper definition for the `load` wrapper and a usage example

in which case the compiler will be able to adapt. (For more information about SQLite types and how LLVMSQLite interacts with them, see 3.3.4.)
Last, `CustomOpBuilder` comes with utility lambdas that help achieving common operations, such as `x++` or `x += y`. These functions were added when needed and when they would make code much more readable than the equivalent sequence of instructions.

A usage example of the `CustomOpBuilder` is visible in Fig. 3.14. It shows how much less verbose it makes IR generation.

3.3.4 Implementation of lowering passes

VDBE types in the JIT

Lowering a VDBE operation to LLVM IR can be done by translating its default implementation from the SQLite interpreter implementation file `vdbe.c`.
Fig. 3.15 shows a simplified VDBE interpreter that works of integer-typed registers only. Like in SQLite, a `for` loop reads through the bytecode and a `switch` on the operation code jumps to the right implementation. Some operations always cause an interrupt (`ResultRow`, `Halt`), some never do (`Add`, `Next`) and some can in the occurrence of an error (`Open`).
Simple operations are executed directly by the interpreter (like `Add`, that adds to registers). On the other hand, operations that rely on auxiliary APIs are wrappers around function calls. In this case, the part actually done in the interpreter is rather simple and connects various APIs. Though, some complex operation codes (like `Column`, which decodes raw data from a B^+-Tree page and copies it as an SQLite Value into a register) contain the whole implementation for deserialising records. In

```
// Get &pMem->flags (5th struct element, an i16)
// We pass C literals, they will be converted to MLIR Constants
auto flagsAddr = getElementPtr(T::i16PtrTy, pMem, 0, 4);

// Load the value of pMem->flags
auto flags = load(flagsAddr);

// Do flags = (flags | MEM_Real)
// MEM_Real is a C #define'd literal, converted here into an MLIR Value
flags = bitOr(flags, MEM_Real);

// myFunction(flags, 5); returns an optional<Value> (void functions)
auto result = call(myFunction, flags, constants(5, 32)).getValue();

// result++; loads the value, adds an MLIR Constant 1, stores the value
result = plusPlus(result);
```

Figure 3.14: (Non-extensive) `OpBuilder` wrapper usage example

all these cases, VDBE needs to interact with many different SQLite data structures.

SQLite uses many custom C types: SQLite Values, various kinds of cursors, database instances, mutexes, records... `struct` definitions alone represent more than 200 internal types. Using these types is rather simple in the C implementation: all necessary definitions are available in header files. Though, the same cannot be done in the just-in-time compiler: in C, all typing information is lost at compile-time. For example, although the function executed by the JIT can be provided with a pointer-to-VDBE argument, it does not know it is a VDBE unless the VDBE type is defined in the LLVM module and the just-in-time compiled code is 'told' to interpret the argument as a pointer-to-VDBE.
Note that even in the case this is done, there is no way for the JIT-compiler to check that its definition for the VDBE type is correct (that is to say, matches the one used by the ahead-of-time compiled C code). Since these types are used extensively, solving this issue is a priority – solving it well even more.

When emitting LLVM IR, custom types can be defined and referred to using the LLVM typing API. It exposes simple types (integers and floats of various widths) and allows to define custom complex (compound) types: function types, structures, fixed-size arrays and pointers. It also is possible to declare *opaque* structures, *i.e.* structures whose precise contents are not known but to which it is possible to manipulate pointers (similar to a forward-declared `struct` in C or C++).

Fig 3.16 shows how types are manipulated through the `Type` API. First, a structure type named `myStructType` is declared. It is opaque, as its body (its fields) have not been defined yet. The pointer-to-`myStructType` is then defined. Finally, a function type is defined: it returns a 32 bits integer and takes a pointer-to-`myStructType` and

```
int interpret(Operation* operations) {
    for operation in operations:
        switch(operation.code)
            case OP_Add:
                registers[operation.P3] = operation.P1 + operation.P2
                break
            case OP_Next:
                advanceCursor(operation.P1)
                break
            case OP_ResultRow:
                copy(&registers[operation.P1], output, n_cols)
                return CODE_ROW
            case OP_Halt:
                return CODE_OK
            case OP_Open:
                cur = allocateCursor(...)
                if cur == nullptr:
                    return CODE_NOMEM
                else:
                    cursors[operation.P1] = cur
                break
}
```

Figure 3.15: A simplified interpreter loop

a 64 bits integer as arguments.

It would be possible to integrate SQLite custom types in the JIT by manually writing their definitions through the C++ LLVM Type API. Though, that would be cumbersome (some types are very complex) and error-prone, as there is no way to verify that a hand-defined type matches the type definition used in C.

Instead of defining types manually with the LLVM Type API, LLVMSQLite loads the pre-generated module and extracts types from it, using their names to identify them. This solution addresses the issue of errors in type definitions, but also makes LLVMSQLite much more sustainable. Should the SQLite types change in the future, making the JIT compatible with them is simply a matter of updating the pre-generated IR module and the `TypeCache` will update accordingly.
The outcome is that any type used in the VDBE Interpreter is readily accessible as a global object in LLVMSQLite as an MLIR `LLVMType` instance. These types are guaranteed to be the exact ones generated by `clang`, which brings another benefit. Since all types allocated by SQLite are defined (in the IR file) by `clang` at compilation, any modifications brought by `clang` to the library code will also be brought on type specifications loaded by the JIT. This allows for more flexibility, whereas manually translating types would have made using Clang optimisations impossible (or limited to no optimisation on type definitions).

```
// Create an opaque struct type named myStructType
auto sTy = LLVMType::createStructTy("myStructType");
// Get the type of s*
auto sPtrTy = sTy.getPointerTo();
// Create the type of a function of prototype
//   i32(s*, i64)
auto funcTy = LLVMType::getFunctionTy(
    /* Return = */ LLVMType::getIntNType(32),
    /* Arguments */ = {
        sPtrTy,
        LLVMType::getIntNType(64)
    }, /* varArg = */ false);
```

Figure 3.16: Type manipulation in the LLVM API

Note In the MLIR LLVM Dialect, the LLVM typing API is abstracted away by the MLIR typing API. The latter uses a thin wrapper called `mlir::LLVM::LLVMType`. Unfortunately, this makes *actual* LLVM types and MLIR `LLVMType`'s not mix together well, as there is no way to produce an `mlir::LLVMType` instance from an already existing `llvm::Type`. This though would be useful in our case, as it would permit us to import types from already existing LLVM IR modules, as mentioned earlier. To mitigate that issue, we modify the MLIR `LLVMType` API so that the front-end can construct `mlir::LLVMType` instances from a pre-existing `llvm::Type`.

Calling SQLite internal API functions from the JIT-compiled code

As already mentioned, many VDBE operations use helper functions to carry out their purpose. These can have many different purposes: allocating memory, changing the state of a cursor, converting data types in registers, computing an aggregate for the current tuple... In order to call them from the JIT-compiled code, we first need to declare them in the generated LLVM IR and to make them externally available to the JIT linker/symbol resolver, as importing their definition is not useful nor desirable (optimising more functions would make JIT compilation much longer, as explained in 2.3.3).

Since many of these functions are used by several operations, the adopted solution is similar to the type cache. Fig. 3.17 shows an SQLite API function definition in the JIT compiler. The JIT contains global variables of type `LLVMFuncOp` (the type used in MLIR to refer to LLVM function definitions).
When starting, the JIT calls an initialisation function that defines these variables. To this end, it defines their type using the MLIR `LLVMType` API then inserts an `LLVMFuncOp` operation in the MLIR module, named after the SQLite API function it defines.
Finally, as symbols from the host process are not always visible to the JIT linker, we manually add the corresponding symbol to the 'LLVM Dynamic Library'. This allows to inject otherwise invisible function into the symbol resolver.

```
// Get function type of sqlite3_value* allocateValue(Vdbe*, i32, void*)
// We make use of the T (type cache) class
    auto funcTy = LLVMType::getFuncTy(
        T::sqlite3_valuePtrTy, {
            T::VdbePtrTy,
            T::i32Ty,
            T::i8PtrTy // void* is translated to char*
        }, /* varArg = */ false
    );

// Declare an LLVM function "allocateValue" with type funcTy
LLVMFuncOp f_allocateValue = b.create<LLVMFuncOp>
                               (..., "allocateValue", funcTy);

// On some OS, functions may not be visible directly to the JIT
// Doing so forces them to be:
llvm::sys::DynamicLibrary::AddSymbol("allocateValue", &allocateValue);
```

Figure 3.17: Example of external function definition in the MLIR module, using the `T` type cache class

It is then possible to call any (defined) SQLite function by using the `f_...` global variables in the lowering passes.

Special operation codes

In SQLite VDBE, some operation have a special behaviour and rewrite themselves at run-time. As opposed to already mentioned operations that change their parameters, these ones change the operation code itself.
Rewriting the operation code allows, in these cases, to execute an initialisation procedure only once, without relying on a more complex mechanism (`Once` → `Goto` → (Initialisation) → `Goto` (back)). In OLAP workloads, two operations use this technique. The next paragraphs describe why they need to use that behaviour and how they are implemented in LLVMSQLite, as the concept of *operation* does not exist in just-in-time compiled code.

The `String8` operation is used to compute the length of a heap-allocated null-terminated string. Upon first execution, the length is computed and the operation code is replaced with `String`, which copies a string into a register and holds the length of the string in parameter `P1` [49].
In LLVMSQLite, the `String8` operation is implemented by computing the length of the string at JIT-compile-time using standard C function `strlen`. Instead of writing the code for a `String8` operation, the equivalent `String` operation is emitted, with the computed length.

```
sum = 0
for i = 0; i < 3; i++
    sum += i
```

(a) Sum in loop form

```
sum = 0
sum += 1
sum += 2
sum += 3
```

(b) Sum in unravelled-loop form

Figure 3.18: An example of loop unrolling for the sum of integers from 1 to 3

The `AggStep` operation computes an aggregate for a tuple. The first time it is executed, the `FuncDef` pointed at by `P4` needs to be converted to an `sqlite3_context`. As this only needs to be done once, the operation code is changed after conversion and subsequent executions only compute the step [49].
In LLVMSQLite, that behaviour is simulated by using the value of `pOp->p4Type` (a field in the VDBE operation structure that indicates the type of the `P4` union), that is changed once `P4` has been converted into an `sqlite3_context`.
When `AggStep` is executed, the value of `p4Type` is loaded. If it is `P4_FUNCCTX`, then the initialisation part is skipped. This requires one more jump than the default implementation (which makes the `switch` directly jump to the `AggStep1` label) but is still more lightweight than reproducing the behaviour with a `Once` / `Goto` or, worse, than initialising the `sqlite3_context` at every execution.

JIT-compile-time optimisations

A prominent advantage of just-in-time compilers over interpreters is their capacity to use properties of the workload available only at run-time to eliminate dead code and branches and proceed to specialisation based on the run-time types [4].
Because of SQLite's *manifest typing*, type specialisation is not straightforward, but some branch elimination can still be done by the just-in-time compiler. Indeed, many SQLite VDBE operation codes use flags (passed in the `P5` parameter) to select – at bytecode-generation time – what branches to take. That means that the branch taken for that operation is always the same during execution. For an operation on the hot path, eliminating useless branches but, even more, lifting branch choice to JIT-compile-time is a promising idea.
For example, the SQLite `Lt` (that checks if a register is *less than* another) accepts the `SQLITE_STOREP2` flag in its `P5` parameter. If this flag is set, the comparison result is not used to jump to another operation but stored in a register specified in parameter `P2`. The `Lt` operation code is often found in the hot path, to check whether a tuple qualifies for a certain predicate. Lifting the `P5` value check to compile-time allows to generate simpler, more predictable code. Everywhere possible, LLVMSQLite proceeds to this kind of lifting.

Another useful optimisation is known at *loop unrolling*. Loop unrolling consists in replacing a loop with equivalent jump-free code. This makes the code much simpler to predict (as branching is eliminated) but also much larger. This kind of optimisation is often omitted in ahead-of-time compilation – because of the lack of information as well as the higher resulting assembly size – but is often useful and reasonable to

use in JIT environments.
As an illustration, consider Fig. 3.18. To sum the numbers from 0 to 3, it is possible to loop over these numbers, increasing a variable (Fig. 3.18.a). Though, that introduces a branch in the code. It is possible to reach the same result without branching, as shown in Fig. 3.18.b.
Many SQLite operations use `for` loops with small numbers of iterations. That is the case of `AggStep` – which executes a function that can take any number of arguments, which need to be copied using a for loop – or `Copy`, which makes shallow copies of n registers. These operations are also often used on the hot path – they need to be executed for a large number of tuples, that is why loop unrolling can be useful in this case and LLVMSQLite is designed to proceed to it whenever possible.

3.3.5 Optimisations on generated IR

One of the most notable benefits of using LLVM as a compiler back-end, and more particularly LLVM IR as a target, is the number of readily available optimisations for IR modules.
Once a module is built, the LLVM API exposes various *optimisation passes* to improve the efficiency (or size) of the module. In this case, we only consider performance optimisations and leave aside size optimisations, as we do not care about memory usage.

Performance optimisations come in several categories:

Function-level optimisations which work on local code. These include local constant folding, conversion from memory to SSA registers, vectorisation, function attribute inference, redundant instruction combination, *etc.* [32];

Module-level optimisations which work on the whole module. These include function inlining or optimisations on global variables.

Creating and populating LLVM `PassManager`'s with adequate transformation passes allows to transform a module into a more efficient one.

Namely, LLVMSQLite uses the inlining pass a lot. It is well known that function calls are costly and generally hurt performance. Calling a function can be costly in several ways: depending on the CPU, it may make the code not as predictable, prevent pipelining, or even jump to addresses which are not in cache – resulting in a cache miss and a long CPU stall. That is why calling a function in a tight loop is generally seen as a performance issue.
These issues can be solved by *inlining*. Inlining a function means to replace a direct function call (a call whose destination is known at compile-time) with the actual function code. This removes control hazards, allows pipelining and makes the code prefetchable. It comes at the cost of a larger binary size and a higher memory footprint at run-time.
Unfortunately, one of the goals of SQLite is to be lightweight, which is why many functions whose inlining is reasonable are explicitly not inlined. LLVMSQLite takes

the opposite direction: many functions that usually are on the hot path are always inlined[3] A list of these functions is available in Appendix 6.1.
Fig. 3.9 illustrates this optimisation. Each operation in the virtual machine programme has has been replaced by equivalent LLVM IR. Functions on the hot path have been inlined to reduce the number of regular and costly function calls.

Inlining a function is only possible if its definition is available. In LLVM, that means externally available functions defined in the module (see 3.3.4) cannot be inlined, as the compiler can only read their native code implementation.
To inline SQLite internal functions, one could compile every SQLite implementation file to IR using `clang`, load it at run-time using the LLVM API and clone functions from the imported module into the JIT-compiled module.
Fortunately, SQLite can be compiled in two ways: each implementation file can be compiled on its own and then linked into a single library or all sources can be merged into a single C file called the *SQLite Amalgamation*. This C file can be compiled to a library file by C compilers, but can also – like any C file – be compiled to a single LLVM module by `clang`. This generated IR module file, called `sqlite3.ll` is our all-in-one solution for importing functions defined in IR into our module at JIT compile time.
Since these function definitions also use the SQLite types, that module is actually loaded when the type cache is initialised (see 3.3.4) and is kept in memory until native code generation when it is used for inlining. Furthermore, using a single IR file for SQLite functions (instead of one per C source file) removes the need for a type mapper (as LLVM does not consider types from different modules with the same definition as identical types).

Finally, the back-end also can optimise the module at code generation level. This includes using more adequate CPU register allocation (reduces the number of memory writes and reads), using SIMD intrinsics for LLVM vector instructions, frame pointer elimination, instruction scheduling for high latency operations [32]...
These native code level optimisations need to be carried out at every execution, as the generated dynamic object is not cached in LLVMSQLite – as opposed to the optimised LLVM module.

3.4 Using static typing in the JIT-compiler

3.4.1 Introduction

The SQLite Value data structure allows tuples to hold any type of data in each column, independently of the relation schema. This data structure is the core of *manifest typing* in SQLite.

As previously mentioned, manifest typing comes with a significant space overhead. That overhead can also have consequences on cache residency and data locality,

[3] Although this is not possible in C, IR functions have the `alwaysinline` attribute.

which have proven to be metrics of importance when targetting high performance [39, 6]. Dynamic typing also causes a overhead in processing time, as more operations are necessary to carry out the same work.

In this part of the project, we propose a limited but sufficient technique to reduce data-type checks and conversions using *a priori* knowledge on the data types found in a column. We focus on numeric types, as strings are the fallback when no numeric type is found. Its goals are the following:

- First, reduce the size of the just-in-time compiled code and reduce the number of considered branches. The implementation of certain operation codes can be made significantly simpler and more predictable if the type of input registers is known at compile-time.

  For instance, the `Compare` operation has handlers for the case where values are `NULL`, the case where they are both numeric, the case where at least one is a string and can be converted to a numeric value, the case where one of them cannot and the other needs to be converted to a string.

  Assuming that only one of these branches is valid from compile time may help performance;

- Second, cut the number of unnecessary conversions. Arithmetic operations take two registers as inputs and may sometimes convert them both to integers before failing and then convert them to reals. This does not only cause conversions, but function calls and checks on flags, which take time;

- Third, solve certain issues that arise from code generation. After profiling the default implementation, it is noticeable that code generation sometimes yields sub-optimal bytecode.

  For instance, it appeared that in some cases, VDBE would try to apply a *Numeric* affinity to date strings. This resulted in many function calls, attempts to convert dates into floats – which inevitably failed – and wasted a lot of time. A major caveat of manifest typing is that even if such an operation has failed on precedent tuples, the fact that typing is independent of schema implies that the conversion attempt can never be skipped, as it may work at least once.

3.4.2 Implementation

LLVMSQLite does not implement extracting type information from the database schema (as mentioned, it cannot be assumed that all columns will have the type given in the schema for various reasons). Rather, it relies on manually entered static data and query numbers (first query is 0, second query is 1, *etc.*). This is necessary, since the first query executed is never a user-defined query but the `sqlite_master` relation probe.

Typing information for each query number is held in a C array of size 255. When the type of register n type can be assumed, the array contains the equivalent flag value

(`MEM_Real`, for instance) at index n. When no type can be assumed – because we do not know it or because it changes over time – the value in the array is 0.
In LLVMSQLite, only two types of operations use static typing knowledge to reduce processing types: the *arithmetic* family (`Add`, `Subtract`, `Multiply`, `Divide`, `Remainder`) and the *compare/compare-and-jump* family (`Ne`, `Eq`, `Lt`, `Gt`, `Le`, `Ge` and `Compare`). These operations have been chosen as they are the only ones that are on the hot path and that use typing information heavily in OLAP workloads.
At just-in-time-compile time, lowering passes probe the static type map and determine whether some branches and conversions can be eliminated based on that knowledge. If so, it uses an alternative, simpler implementation that does not have type guards or conversions.
For example, the `Ge` operation (that jumps if a register is greater or equal to another register) is replaced by a lighter custom implementation when both registers are known to be of the same numeric type (`REAL` or `INTEGER`).
The custom `Compare` implementation (that compares sequences of registers) behaves in a similar way, but is only used if registers in the vector can be compared side-by-side.
If no typing information is available, the default implementation is inserted.

Although LLVMSQLite does not implement automatically extracting type information from the schema, we provide tools to help with benchmarking static typing.
When generating prepared statements for complex queries, such as the ones from TPC-H or SSBM, the resulting bytecode often exceeds 150 operations. Extracting type information from bytecode manually is only reasonable if the bytecode length and the number of registers it uses is contained.
To mitigate that issue, we propose a specialised SQLite interpreter along with a Python script to establish typing automatically.
Our custom interpreter uses a modified `Column` implementation and makes it output the types of extracted values by testing type flags before continuing execution. In 'well-typed' databases and since the VDBE code generator does not re-use registers, it is generally safe to assume that the type the register has after being written by `Column` is the same for the remainder of the processing.

This first tool allows us to get surface information: the type of the data held in each register written by `Column`. We then use a Python script that leverages this information to determine the type of other registers.
Fig. 3.19 illustrates how the script operates. For a given SQL statement, the script executes it using our custom interpreter: it thus knows the type of each register to which a column has been extracted.
It then runs the same SQL statement, prefixed with the `EXPLAIN` keyword, which makes SQLite dump the VDBE bytecode for the statement.
Using the basic typing information obtained from the first run and applying simple inference rules (Type of $A \times B$ is Type of A, Type of a register to which `RealAffinity` is applied is `REAL`...) the script determines the type of most registers of interest for our project.

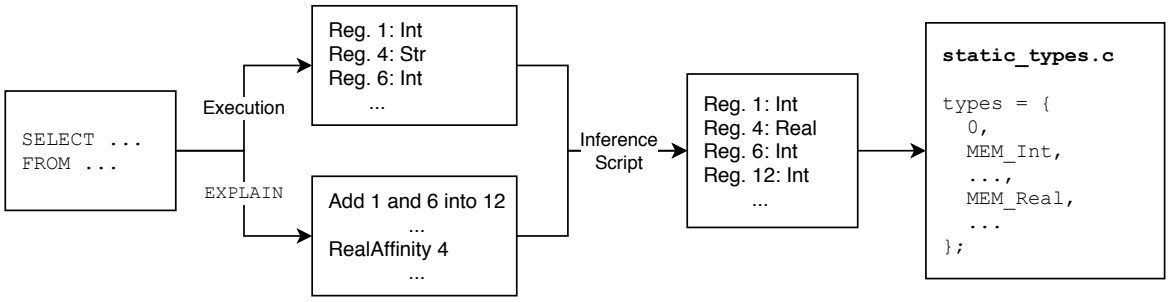


Figure 3.19: Basic type inference through bytecode analysis and custom `Column` operation implementation

The Python script finally generates a C code file which contains a static array indicating the type of each known register. Compiling LLVMSQLite against this file allows to integrate static type knowledge into the JIT compiler.

3.4.3 Limitations

An important aspect of SQLite Values is that a Value can only hold one representation of its numeric value. Furthermore, many typing operations are performed in-place. That means that our knowledge of types only works so far as:

- The type of a register does not change throughout execution – otherwise, our knowledge is wrong at times and it is not possible to eliminate branches. In other words, no cast must be applied in-place on a register for which we provide a static type;
- The register column affinity is be `REAL` or `INTEGER` (not `NUMERIC`). Indeed, when the affinity is `NUMERIC` and for space-saving reasons, SQLite serialises reals with a null decimal part to an integer representation when writing them to disk. As a result, the values extracted from that column can be integers or reals.

  When the `REAL` column affinity is used rather than `NUMERIC`, SQLite forces the affinity by converting the values back to floats at run-time, by inserting a `RealAffinity` operation after the corresponding `Column` operation. In this case, although the type of the Value is not constant throughout execution, it may only change between `Column` and `RealAffinity`, which is not an issue as our implementation does not modify `RealAffinity`.

Although these two conditions may seem very restrictive, they are not necessarily problematic. The SQLite code generation engine is aware of the possible performance issues that manifest typing can cause and automatically attempts to reduce the overhead caused by conversions.
For instance, if a relation `T` contains a column `A` with affinity `INTEGER`, then executing `SELECT A, CAST(A AS REAL) FROM T` will not trigger conversions but rather extract `A` twice into two different registers, only applying the `REAL` affinity to one of them.

3.5 Caching compiled LLVM modules to reduce compile times

3.5.1 Introduction

Although they can bring many performance improvements, just-in-time compilers come with an often non-negligible overhead that interpreters do not have: compilation itself (Fig. 3.1).
In our experiments, we noticed that module creation, optimisation and native code generation could represent a significant cost for simple queries and/or small datasets (typically more than query execution itself). Although this overhead may be deemed acceptable on larger databases and more complex queries, where the time gain brought by JIT-compilation is high, it is not the case for smaller databases and simpler queries, for which compilation costs could make total processing time much higher than the interpreter overhead.
A very representative example of that is the database schema probing query, which is executed every time the user runs a query on a database for the first time. While executing the query itself typically takes less than 1ms, equivalent native code generation takes up to one second.

To mitigate these costs, we propose *compiled prepared statement caching*, in a similar way as Pantilimonov et al. [38] do it for PostgreSQL.

3.5.2 Implementation

A JIT-compiled prepared statement takes different forms.
The first step is an SQLite-generated VDBE instance. It holds allocated and initialised registers, along with a bytecode represented by an array of VDBE operations.

1. LLVMSQLite first translates this bytecode into an equivalent VDBE-IR module. Lowering passes translate VDBE-IR into LLVM IR, and the MLIR framework finally converts it to an in-memory LLVM module. These first steps, *module construction*, take on average 3% of the compilation time.

2. The JIT then applies various optimisations to the generated module, using pre-existing LLVM IR passes. This step, *module optimisation*, takes on average 65% of the compilation time.

3. Finally, the LLVM module is converted to a dynamic object by the LLVM backend (this includes native code generation as well as low-level code generation optimisations). This step, *module compilation*, takes on average 32% of the compilation time[4].

[4] Note though that the *module optimisation time* and *module compilation time* shares may vary depending on the query.

```
prime: unsigned int = 271
hash: unsigned int = 0
For op: Operation in vdbe.operations
    hash = hash * prime + op.operation_code
    hash = hash * prime + op.p1
    hash = hash * prime + op.p2
    hash = hash * prime + op.p3
    hash = hash * prime + op.p4type
    hash = hash * prime + op.p5

Return hash
```

Figure 3.20: Pseudo-code for the `VdbeHash` function

Caching could be done at different levels, but the choice retained in this project is to cache the optimised LLVM module (at the end of Step 2). Indeed, caching a non-optimised module would not be very useful, as optimisation passes represent a large part of compilation costs. On the other hand, caching compiled dynamic objects is rather hard: the LLVM MCJIT does not officially support it [24] and would require to use a custom memory manager in the JIT, a rather cumbersome and error-prone operation.
That is why the optimised module is cached to disk in the form of a bitcode file (LLVM IR in a computer-only-readable format).

Recognising VDBE bytecodes is done through a hash function defined on VDBEs. Before building a new module, the `VdbeRunner` checks whether a bitcode file matches the hash of the VDBE to execute. If so, it loads it and skips module creation and optimisation steps. Otherwise, it writes a new module from scratch using the JIT and then dumps it to disk for future use.
The hash function for VDBEs is defined in Fig. 3.20. The hash is based on the value of the operation code, on the value of the parameters `P1`, `P2`, `P3` and `P5` of each operation and on the type of `P4` in each operation.
Note that the value of parameters `P4` are not taken into account, as they are most of the time dynamic values (pointers to allocated objects or buffers).

In practice, using the VDBE caching feature makes the JIT much faster.
Thanks to the design of the lowering passes, LLVMSQLite does not face the same issues as Pantilimonov et al. do: since the IR generated in PostgreSQL relies on hard-coded ephemeral pointer values (`IntToPtr` LLVM IR instructions), cached modules need to be updated to match the real run-time values. In LLVMSQLite, generated modules only refer to the VDBE run-time instance (including its operations and parameters), which is passed as a function argument to the JIT-compiled function. As a result, there is no need to update any values in the cached module but only to compile and run it.

Chapter 4

Evaluation

4.1 Quality of code assessment through automated regression testing

Assessing the quality of software is of critical importance. It is estimated that about fifty percent of the time on average is dedicated to testing in software engineering [35]. That is well understandable, and especially in the world of data management systems. This software can be used to produce metrics that lead to strategic decisions in companies or have significant economic consequences – for instance, in the case of a faulty database system in a bank. To this end, SQLite is extremely well tested.
Before being able to evaluate LLVMSQLite as a performance competitor to the SQLite interpreter, we carry out several tests to check that both databases produce the same results. Should the results differ, our implementation would be faulty and its performance would not be comparable with those of the SQLite interpreter.

To test the just-in-time compiler itself, we use standard instrumentation tools that can be automatically inserted by a C/C++ compiler into the native code it generates. Memory leaks are checked both with Valgrind[1]. MemorySanitizer (MSan) [48] is also used to check for memory errors. Invalid memory usage is checked with Address Sanitizer (ASan) [46], which checks for reads and writes to invalid memory blocks (uninitialised reads, writes outside of data space, *etc.*). Finally, UndefinedBehavour Sanitizer (UBSan)[2] is also used to check that the compiler does not rely on undefined behaviour. These tools helped detect and address issues that were difficult-to-reproduce and to spot.
Unfortunately, MSan, ASan and UBSan cannot be used on just-in-time compiled IR code, which is much more prone to errors than higher-level languages like C++. Indeed, MSan and ASan are instrumentation emitted by the C compiler and not

[1] The Valgrind Memory Error Detector, often called simply *Valgrind* is a memory error detector that translates machine code to a custom intermediate representation into which instrumentation code can be inserted, before recompiling that representation to machine code. Although these steps slow down execution, Valgrind is helpful when debugging memory issues (see `https://valgrind.org/`

[2] See `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`

by LLVM itself. UBSan is even a higher-level sanitizer that checks for undefined behaviour in C code (and not undefined behaviour in IR). That means although we can ascertain the just-in-time compiler is sound in its memory usage and in its implementation, we cannot check the generated code is with these tools. To mitigate these limitations, we adopt other testing techniques for the just-in-time compiled code.

First, when a JIT-compiled LLVM module is compiled in LLVMSQLite, it is always verified by LLVM before compilation. This allows to catch errors that could make a native code generator fail or emit faulty assembly.
Furthermore, in debug builds, the generated IR module is first run through the `lint` pass[3], which detects bad coding patterns and undefined behaviours in the generated IR[4].

Finally, it is possible to insert a memory address verifier in the generated IR. As mentioned in Section 3.3.4, although LLVMSQLite defines the same types in IR as SQLite does in C, this does not guarantee the generated code behaves as expected when writing in memory.
This comes from the *GetElementPointer* LLVM instruction. *GetElementPointer* fetches the address of an element in an aggregate (array or structure). The IR compiler deduces the return type of `GetElementPointer` at compile-time. If `GetElementPointer` returns an `i32*` that is later used in an expression expecting an `i8*`, an error will be thrown and compilation aborted.
This guarantees type-correctness, but does not guarantee the address of the `i32` returned by `GetElementPointer` is the one sought by the developer. As a result, an incorrect offset in `GetElementPointer` may lead to the wrong field of a structure being changed, with the obvious consequences.
Checking that the element accessed in indeed the right one is not simple. LLVM IR does not name fields and providing a wrong index happens often with the large SQLite data structures. To check that an operation on a structure field is the same in C and in IR, we use templates like the one presented in Fig. 4.1.
We define a global object, constant at JIT-compile-time and never destroyed. We then compute the address of the field at JIT-compile time (in C) and insert the code to compute it at run-time (in IR). As the object is the same, the resulting pointers should be equal. If the index is incorrect (here, if `type` was the fifth element, for example), the assertion will fail as `getElementPointer` will yield a different address.
Note that this testing method should only be used when debugging: it inserts a lot of of useless code (assuming we want to check each field of each structure), but it also makes caching impossible. Indeed, the JIT-compile-time constant address of the static object is not a C compile-time constant but the address of a dynamically allocated variable (on the heap or on the stack), which means accessing the same address in future processes is likely to cause a segmentation fault.

[3] See `https://llvm.org/docs/Passes.html#lint-statically-lint-checks-llvm-ir`
[4] Note this pass does not work like UBSan. Whereas UBSan detects undefined behaviour at run-time and as it happens, the `lint` pass checks for undefined behaviour in IR ahead of time

```
// The cursor we use to check the offsets
static cursor_t* cursor = STATIC_CURSOR_ADDRESS;

// The same cursor_t* as a hard-coded LLVM value
auto llvmCursor = intToPtr(constants((uint64_t)cursor, 64));

// This is computed at JIT-compile-time
auto addressOfCurType = &cursor->type;

// Here, we assume type is the fourth element in cursor_t
// This is computed at execution time
auto irAddressOfCurType = getElementPointer(cursor, 0, 3);

// We can only compare integers
irAddressOfCurType = ptrToInt(irAddressOfCurType);

// If the implementation is correct and "type" is indeed
// the fourth element, this assertion should pass.
auto curTypeAddressValid = iCmp(LOC, Pred::eq,
    irAddressOfCurType,
    (uint64_t)addressOfCurType
);
myAssert(LOCL, curTypeAddressValid);
```

Figure 4.1: Example of structure offset verification at run-time

These measures allow to guarantee the validity and well-behavedness of the code that is run, but do not guarantee it does the same thing as the default implementation. Verifying that the implementation is valid is rather difficult. Some parts of the LLVMSQLite implementation rely on JIT-compile-time information that the default interpreter does not have at query-compile-time, so comparing the IR for the interpreter and the generated IR does not make sense.

For debugging purposes, it may be useful to use the default implementation of an operation by delegating execution of one operation to the interpreter. To this end, we use execution interleaving.
We call *execution interleaving* the capacity to use the JIT implementation for some operation codes and the interpreter for the others. What operations should call into the interpreter is defined statically and the JIT compiler uses this information to either emit the full operation implementation in IR or only a call to the interpreter function instead of the actual implementation.
Fig. 4.2 illustrates it: although the primary execution method of LLVMSQLite is the just-in-time compiled function, that function may call into the interpreter in certain cases (if the operation code is not implemented in the JIT, for example). In that case, the interpreter executes a limited number of operations before giving control back to the JIT-compiled function. Depending on the operation, the JIT-compiled function

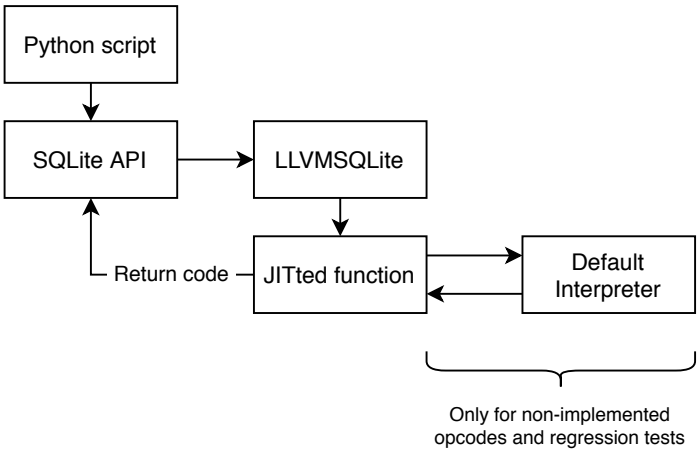


Figure 4.2: Execution interleaving between just-in-time compiled code and the default interpreter

may return the code returned by the interpreter or continue execution.

To allow execution interleaving, we slightly modify the interpreter. By default, the only way to exit the function is to get to an instruction whose operation code jumps to an `vdbe_return` label (`ResultRow`, `Halt`, *etc.*). That is not helpful in our case, as relying exclusively on that behaviour would mean that calling in the default implementation for an operation would get the VDBE stuck in the interpreter until the next interrupting operation.
Changing that behaviour is rather simple and can be done by adding a global `maxVdbeSteps` integer and a local `steps` integer. Before calling into the default implementation, the JIT-compiled code sets `maxVdbeSteps` to the desired value – typically `1` – and updates the `pc` field in the VDBE to the current programme counter (this is known at JIT-compile time for an operation).
When `sqlite3VdbeExec` is called, it uses the execution loop as usual, incrementing the `steps` variable at each operation. At the end of an operation, it checks the value of `steps` against `maxVdbeSteps` (if defined) and, if equal, jump to the `vdbe_return` label, which stops interpreting the programme and gives control back to the JIT-compiled function.

Though these edits allow interleaving for most operations, additional modifications need to be done to allow for full interleaving. First, some operations use local variables defined in the `sqlite3VdbeExec` function. Sharing this variable between the JIT-compiled function and the interpreter is impossible, and even simply calling the interpreter twice will make this value become undefined (as the function will have returned in the meanwhile).
These two issues can be solved by making that variable global in the interpreter translation unit and externally available. When the VDBE wrapper calls the JIT-compiled function, it passes the address of the variable as an argument, so the JIT-compiled function can use it too.

Second, in the interpreter, operations that jump to others do so by changing the value of the local `pOp` (pointer to the current `VdbeOp`). Since the JIT-compiled function uses the value of the programme counter with a `switch` instruction to resume execution, the interpreter now needs to set the `pc` field of the VDBE to the right value when returning after jumping operations – non-jumping operations do not need to do so as their behaviour is to continue execution at the next operation, which the JIT implementation already does.
Furthermore, when calling into the interpreter for a terminator operation, the JIT-compiler does not necessarily know at JIT-compile time what the jump target will be, and thus cannot materialise a valid conditional jump. That is why, in this case, the JIT-compiled function jumps to the *Resume Switch*. That way, the function can jump back to the correct operation. This is not as efficient as a direct jump, but it is designed uniquely as a debug feature, a configuration in which performance does not matter as much.

Interleaving execution allows to discriminate valid operation implementations from faulty ones. Every time a new operation code is implemented, the same queries are executed through the shell with JIT-compilation for the new operation on and off. If the results are the same, then the implementation is deemed valid.
This is not necessarily the case – the branches the queries use may be valid but not others, which we cannot detect – but it helps a lot in finding issues. When outputs differ, the new operation is first switched off and the test is carried out once again. If outputs match, then the error is assumed to lie in the newest operation. If not, then the issues arises from another operation code whose implementation and effects are invalid and used by the newest operation. In this case, all operations are turned off and iteratively turned on until the output discrepancy appears. When it does, we have found one faulty implementation. There may be more, that is why once the operation is fixed, the process is re-run from scratch to check that the regression has been fixed.

Execution interleaving has been especially useful during development. Operations were implemented one by one, and sometimes executing a new query would trigger previously undetected bugs coming from already implemented operations that were thought correct, but had a branch that was not.
In complex queries – like the TPC-H ones – the number of bytecode operations can quickly reach more that 100. Proof-reading and 'print-debugging' (as the JIT does not support other kinds of debugging) that many operations would have been tedious. Thanks to interleaving, it was possible to trigger the default implementation for certain operation codes and keep the JIT one for others.
From a completely default implementation, by iteratively enabling more JIT-compiled operations and running the query, it was fairly simple to locate what operation code was faulty.

4.2 Performance evaluation methodology

Benchmarking SQLite has been shown to be a difficult problem. Purohith et al. [40] explain that most articles that attempt to provide a performance evaluation of SQLite fail to take many performance parameters into account.
This results in sub-par performance which does not reflect SQLite actual potential, reached when configured optimally.

In this section, we present a meaningful comparison: the parameters presented by Purohith et al. [40] act only on transactional workloads. Furthermore, even for transactional workloads, tuning the parameters would not lead to different performance results, as both implementations would equally benefit from it (these parameters change the behaviour of layers that are lower than the VDBE).

Finally, most database systems that use just-in-time compilation provide a mechanism for making databases main memory resident. SQLite is not a main memory database, and profiling on benchmarks show that most time is spent reading data and waiting for disk operations to complete.
To mitigate that effect and provide a better comparison of interpreter vs. JIT approaches, we load benchmarking databases into main memory using `tmpfs` [43]. When a file is copied onto a `tmpfs`-mounted directory, the contents are not stored on the disk but rather in RAM.
This makes all read and write operations from SQLite much faster. Note, though, that this does not make SQLite an in-memory database. Indeed, SQLite still sees RAM as a file system which forces it to trigger system call to read data. It also does not change the architecture of SQLite itself: the in-memory page cache will still be used, even though everything is now in memory (which could make it redundant).

We evaluate the performance of SQLite implementations by measuring the run time of various queries from three different benchmarks: custom microbenchmarks, TPC-H and SSBM. Each query is run several times (to mitigate idiosyncratic noise) and at different moments (to mitigate temporary system load that could bias the time taken for a certain query). The query processing times are then plotted. The baseline is the JIT execution time, the other point for each query is the execution time using the interpreter. Errors on the difference are indicated on the plots (they are computed using an arithmetic average of the NoJIT / JIT ratio).

Our results do not account for module creation and lowering, optimisation and compilation costs, as these are considered an investment to be amortised on future query executions (see 3.5)
Furthermore, we provide results that compare the performance of the two typing implementations: completely dynamic typing (the default in SQLite) and typing with static assumptions made on dynamic type of Values.

In the following subsections, we provide and analyse the performance results of

custom microbenchmarks, and two macrobenchmarks – TPC-H and SSBM – using both the default SQLite VDBE interpreter and the just-in-time bytecode compiler.

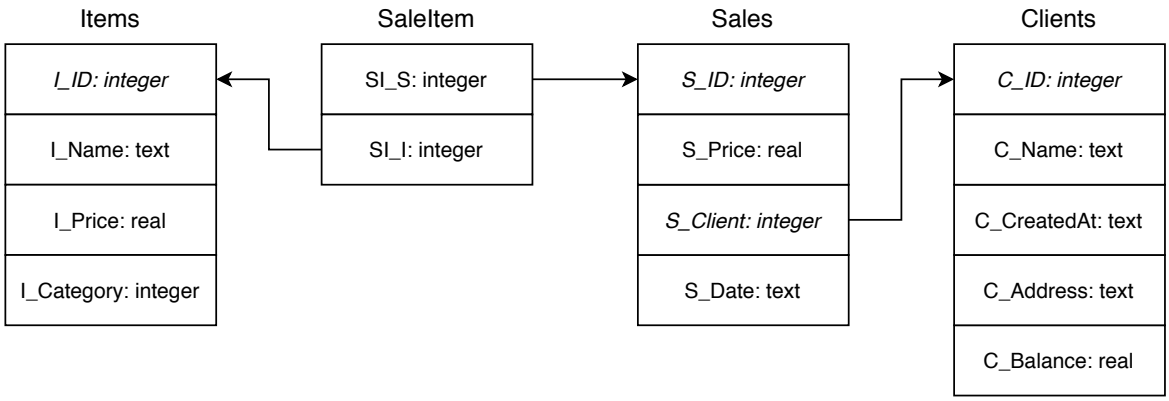


Figure 4.3: Schema of the `Micro` database. Fields in italic are indexed.

4.3 Workloads presentation

In this section, we present a set of micro-benchmarks and two macro-benchmarks for comparing the performance of the VDBE interpreter and that of our just-in-time compiler.

4.3.1 Microbenchmarks

Workload presentation

For benchmarking 'raw' database performance on simple queries, we propose a business database, loosely inspired from the TPC-H schema but simplified for our use-case (Fig. 4.3):

Items stores all items that the store provides;

Clients stores data about all the clients the store has;

Sales stores data about orders made by clients;

SaleItem stores pairs that indicate the contents of each sale in the form of `SaleID, ItemID`.

This database is not designed to allow optimal performance but rather to run many simple queries.
The queries used in the LLVMSQLite microbenchmarks are available in Appendix 6.6 and aim at identifying the strengths and weaknesses of the JIT compiler. To that end, we use simple selections (on indexed and non-indexed data), joins, aggregates with more or less complex expressions, nested queries...

4.3.2 Macrobenchmarks

The macrobenchmarks we use are decision (OLAP) workloads. We use the schema and queries presented in Section 2.4: TPC-H [52] and SSBM [37].

To generate valid TPC-H queries, we use a custom Python script[5] that uses template SQL files and fills them using random parameters that match the constraints defined in the specification.

It is to note that in TPC-H queries, joins are not expressed using the `FROM A JOIN B ON` construct but rather as a restricted cross product (`FROM A, B WHERE A.ID = B.ID`). Equivalent queries were written before our benchmarks but did not perform differently from the cross product-based joins.
As a result, joins are kept as cross products in the benchmarks and we assume that SQLite can optimise them automatically at query level and/or that these joins refer to unindexed columns, on which a nested-loops join is triggered (effectively leading to a cross-product).

For SSBM, we use the 13 queries (built from four query templates) defined in the specification by O'Neil et al., without changing the parameters at each run. Indeed, these 13 queries are 'query flights' that are already parameterised. The parameters have been picked to prevent fortuitous cache effects that could arise if the same tuples were probed twice (which can happen in TPC-H) [37].

4.3.3 Evaluation hardware

The experimental evaluations are executed on a server running Ubuntu 18.04 LTS. The CPU is Intel(R) Xeon(R) E5-2660 v3 (Haswell architecture) running at 2.60GHz and 32GB of RAM are available.
This machine is chosen to act like a machine on which heavy 'back-end' workloads could be executed in commercial environments. Profiles and microarchitectural data are obtained using Intel VTune Amplifier[6].

4.4 Results

4.4.1 Microbenchmarks

Figure 4.4 displays the execution time for microbenchmarks. All queries show an improved execution time when using the just-in-time compiler. Though, the performance gain on most queries is minimal: under 5%.
This can be explained through profiling by finding bounding microarchitectural components. The cardinality of many queries is very high. As a result, in these cases,

[5] See `/llvmsqlite_util/benchmarking/generator_tpch.py`
[6] More information about Intel VTune Amplifier can be found at `https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html`

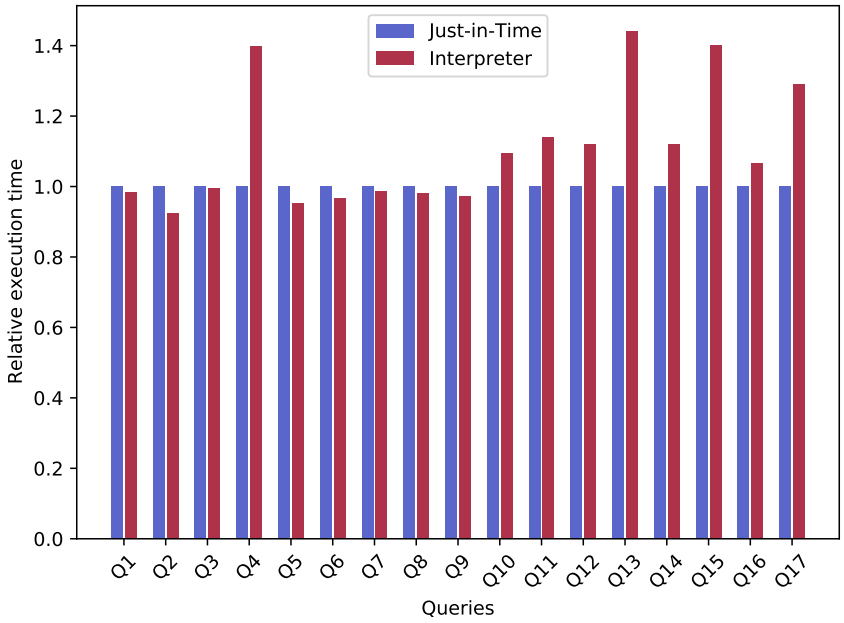


Figure 4.4: Query execution time comparison for microbenchmarks with 25 samples

the dominating costs are the one of function calls (each produced tuple causes a function return and a function call). These costs hide away possible improvements brought by the JIT compiler.

On the other hand, some queries (Q4 Q13, Q15, Q17) show a major execution time enhancement. Looking at the definitions of the microbenchmarks (Appendix 6.6), we notice that the queries for which the improvement is most notable are the ones in which complex arithmetic expressions are evaluated. We also observe that the more complex the expression gets, the higher the performance gain is. This is a place where the benefits of JIT compilation are clearly visible. Whereas computing a value in the interpreter requires multiple jumps to the `Plus / Minus / Divide / Multiply` operation and several memory reads and writes, these can be optimised in the JIT compiled statement. All the operations are executed linearly (no jumps) and values do not always need to be written back to memory. As the implementation of these operations is simple, generated IR shows that LLVM can even omit the intermediate reads and writes in certain cases.

Finally, we notice that, among queries with complex arithmetic expressions, queries that produce more tuples benefit less from just-in-time compilation. Once again, this is due to already-mentioned-before the function call overhead.

Looking at these queries in details (see Appendix 6.6), we notice two things. First, Q5 contains a nested subquery and this may explain the discrepancy. The other degraded queries do not share any specific traits – they are actually very different, one uses sorting, another has a join, some use an index. The only common characteristic among these queries is that their cardinalities are all high. This is also the case for Q5. Other microbenchmarks have a nested statement (Q12 and Q7) but do not suffer from this performance loss, which strengthens the case against the sub-query

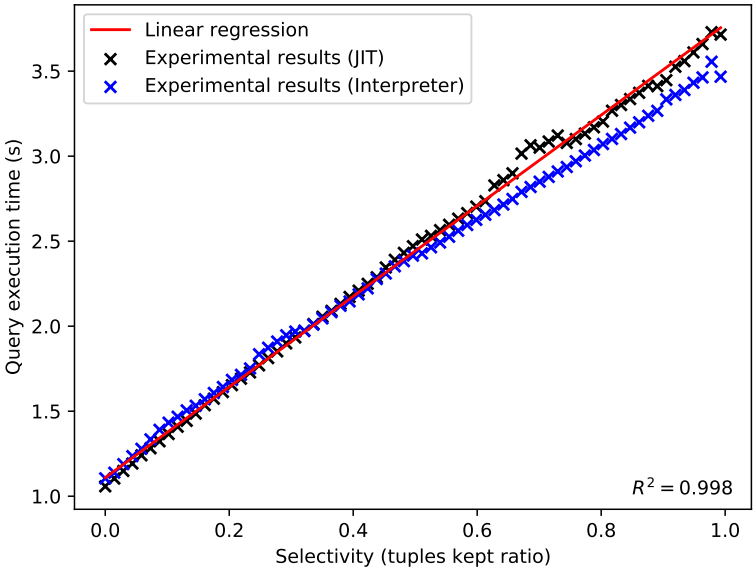


Figure 4.5: Selectivity experiment results

hurting performance hypothesis.
We thus hypothesise that the cause of the slowdown is the cardinality of the query, and that resuming execution in the JIT-compiled function is not as fast as it is in the interpreter. This may be explained by the implementation – the Resume Switch – whose performance may not be as good as the approach the interpreter takes for resuming execution (see 3.3.1).
Although this negative effect also applies in other queries, it may be hidden by the JIT-induced performance improvements. In the case of these simple queries – most of which are simply selections or joins, whose most performance critical components are not in the VDBE – the JIT gains are not sufficient to compensate the cost of resuming execution. It is even worsened by the number of tuples produced (and thus the number of times where resuming execution is required).

To push that analysis further and confirm our hypothesis, we profile the non-benefiting queries. Time profiles show that the time spent in each system call and SQLite internal APIs are the same. We conclude that the difference must come from micro-architectural effects.
To confirm this interpretation, we carry out a microarchitectural profile on the `FRONT END_RETIRED_L1I_MISS` hardware event using the Counters tool of Apple Instruments. This event is triggered when an instruction that retires had caused an L1 instruction cache miss[7] . These cache misses can cause *pipeline bubbles*, which badly affect performance.
Our experiment shows that the number of L1 instruction cache misses on the JIT-

[7] For a complete reference, see `https://oprofile.sourceforge.io/ docs/intel-corei7-events.php`

compiled statement is 28% higher than on the interpreter. This may explain the performance discrepancies on high-cardinality queries (on low cardinality, the number of cache misses is only 6% higher on the JIT compiled statement than on the interpreter).

Finally, we carry out an experiment to confirm that a higher cardinality leads to degraded performance. To assess the impact, we run the query `SELECT I_ID FROM Items WHERE Price < X` with values of `X` ranging from 0 (the minimum) to 2048 (the maximum), with a step of 15. We then measure the query execution time for each value of `X` and plot the resulting points. We do this for both the interpreter and the JIT compiler. Fig. 4.5 displays the experimental results as well as a linear regression.
We conclude that processing time grows linearly with the selectivity for the JIT compiler ($R^2 = 0.998$). We also observe that although execution time are similar for selectivities less than 0.5, discrepancies appear higher up. In these cases, the interpreter has a better performance. We attribute that effect to the Resume Switch. Indeed, these costs cannot be attributed to tuple emission as both implementations emit the same number of tuples/calls.

Other valuable information can be derived from these microbenchmarks.
First of all, one could wonder whether the Resume Switch is a completely unreasonable choice for operations jumping to dynamic destinations. Q7, Q8 and Q12 – which use nested subqueries, a feature that uses coroutines and thus jumps with dynamic destinations – perform better when just-in-time compiled. This proves that jumping back to the Resume Switch before jumping to the destination block can still be an acceptable implementation in the cases where JIT compilation brings other performance improvements. Finding an alternative implementation, though, would make the JIT even more valuable: not only would the queries benefit from the JIT, but they would not suffer from the Resume Switch.

Second, the microbenchmarks are a good place to experiment the effects of certain optimisations on the performance. Namely, one can expect great gains in execution time thanks to function inlining (which the JIT applies in operations found on the hot path). Experimental results show that the effect is in fact barely noticeable. Though, this could still be due to adverse effects (for instance, an improvement induced by inlining along with a worsening incurred by the Resume Switch).
To assess the performance gain induced by inlining in microbenchmarks, we compare the execution time of two SQLite just-in-time compilers, one with inlining enabled and one without. Fig. 4.6 displays the results. We observe that though inlining provides a performance gain in most cases, it is negligible most of the time[8].

[8] One may wonder whether automatic (not forced but decided by LLVM) inlining may still be happening and may explain that very small difference in performance. Automatic inlining cannot happen here as the IR compiler needs the IR definition of the function to be able to inline it; when the just-in-time does not *want* functions to be inlined, it does not copy their definitions across modules, preventing automatic inlining.

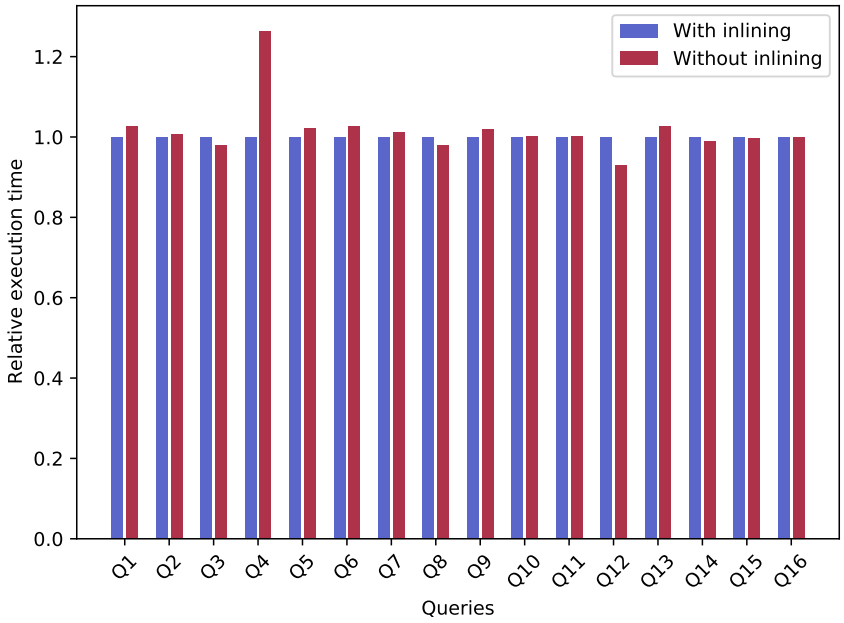


Figure 4.6: Comparison of execution time on microbenchmarks between function inlining on and off (25 samples)

4.4.2 Macrobenchmarks

Figure 4.7 displays a comparison in query processing time between LLVMSQLite and the SQLite interpreter on TPC-H OLAP queries.
A significant part of queries benefit from JIT compilation, with an execution time improvement generally lying between 15% and 25%.
Q11 greatly benefits from JIT compilation (execution time is reduced by more than 40%). This can be explained by the number and the complexity of aggregates in Q11. It consists in a selection of an aggregate `sum(ps_supplycost * ps_availqty)` on a predicate that is based on the result of another aggregate `sum(ps_supplycost * ps_availqty) * fraction`. As we've seen in the microbenchmarks, queries with arithmetic operations and aggregates benefit much more from JIT compilation than others, especially in the case where the cardinality of the query is low. This fact is demonstrated once again in this case.

Figure 4.8 displays the relative performance for JIT (reference) and non-JIT executions on SSBM queries.
We observe that runs on Q1 equally benefit from JIT compilation. We attribute that gain to the arithmetic operation and the aggregate in the SQL statement, which we have already observed benefited from JIT compilation, combined with a low cardinality (only one tuple: the aggregate does not have a `GROUP BY` clause).
The three Q2 runs do not equally benefit from JIT compilation – the difference is actually significant. Whereas Q21 and Q23 run 40% slower on the interpreter, Q22 runs in about the same time on both versions. This can be seen as a consequence of the cardinality of Q22, which is way higher than those of Q21 and Q23. As we have

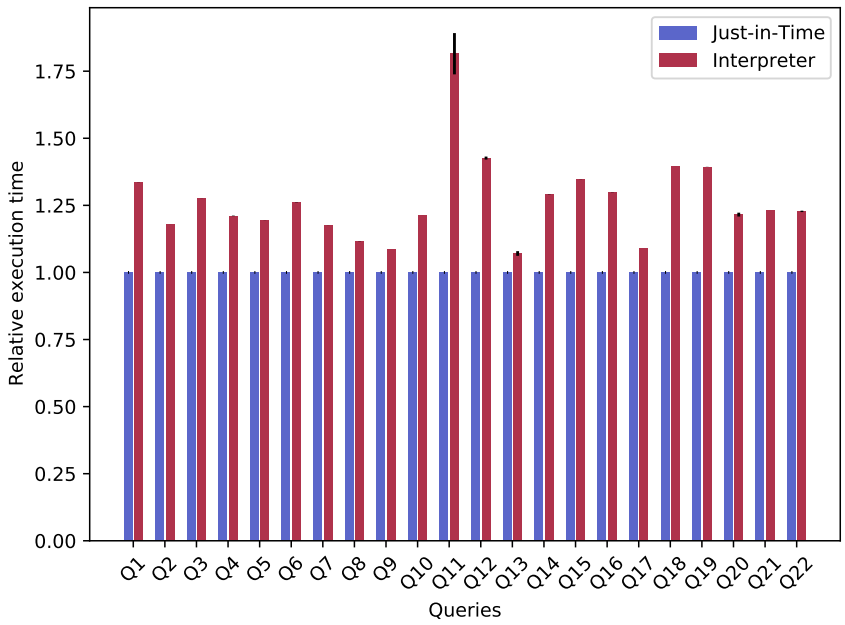


Figure 4.7: Query execution time comparison for TPC-H OLAP workloads with 25 samples

already shown, a higher cardinality plays a major role in reducing the execution time improvements brought by the JIT.
Q31, Q32, Q33 and Q34 equally benefit from JIT compilation. The gain when compared to the interpreter is rather low. We attribute this contained benefit to the fact that query 3 does not have an arithmetic operation. The improvement is thus only due to microarchitectural effects, which are not as significant.
Finally, we notice that Q43 benefits from JIT compilation much more than Q41 and Q42 do. By reading the SSBM specification, we notice that the difference between Q41 and Q42 on the one hand, and Q43 on the other hand, is that Q43 does not have an `ORDER BY` clause. We conclude this discrepancy is due to the sorting cost. As the sorting operation is not carried out directly by the VDBE, JIT compilation does not speed it up. As a result, the gains in execution time brought by the JIT represent a higher share of the total execution time on Q43 than on the other two.

4.5 Effect of static typing

In this section, we analyse the performance gain brought by the modifications on manifest typing presented in 3.4. As a remainder, we propose modifications in three hot path operations that rely on many checks and conversions, the arithmetic family and the compare/compare-and-jump family.

We carry out three experimental evaluations to measure the performance gains of this feature. First, we compare the execution times of simple statements on a basic testing database to assess the raw difference.

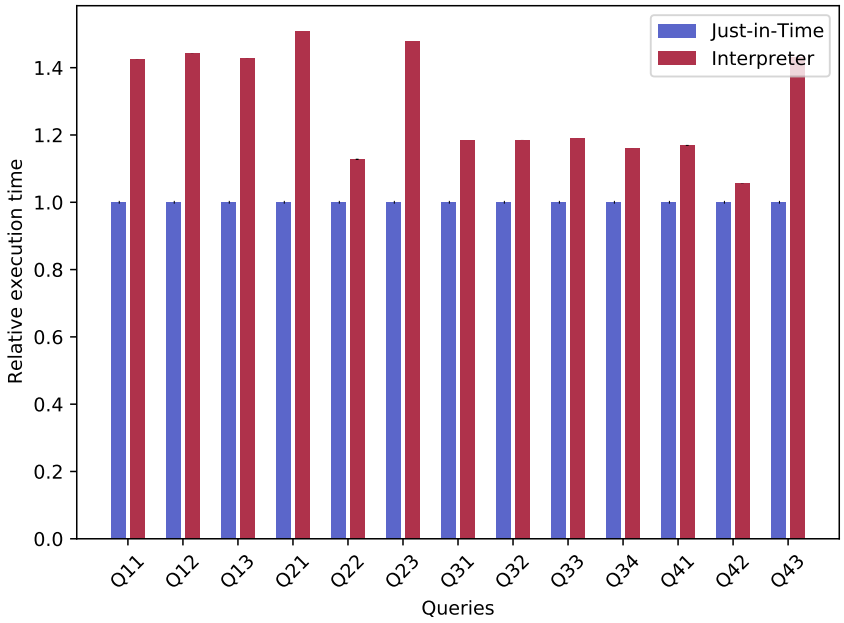


Figure 4.8: Query execution time comparison for SSBM queries with 25 samples

Second, we execute TPC-H queries to evaluate how these differences replicate on more realistic queries.

4.5.1 Raw performance gain

The first evaluation we carry out on the new implementations is the following.
We execute similar queries on two simple tables, `Reals` and `Integers`. Each table has two columns `A` and `B` with the same type (respectively `REAL` and `INTEGER`).
We then execute two simple queries on these tables As we have shown, the cost incurred when producing tuples is high enough to hide away the performance gains brought by LLVMSQLite. We thus choose queries that do not produce many tuples. Furthermore, the optimisations we propose are primarily on the arithmetic and compare-and-jump operations. The queries chosen thus need to evaluate a complex mathematical expression and have a conditional jump to see adequate results.

```
SELECT COUNT(A) FROM [Integers | Reals] WHERE A*B*B*B*B*(1-B) < 10
```

Figure 4.9 displays the relative execution times for the default implementations and the implementations using static typing.
We observe a 40% improvement on `REAL` queries and a 50% improvement on `INTEGER` queries.
We conclude that in the context of JIT-compilation, these new implementations using *a-priori* type knowledge can significantly speed up computations.
We observe that, on these simple workloads, static typing brings a significant performance improvement. Static typing reduces execution time by 50% on the Integer workload, whereas it cuts execution time by 40% on the Real workload.

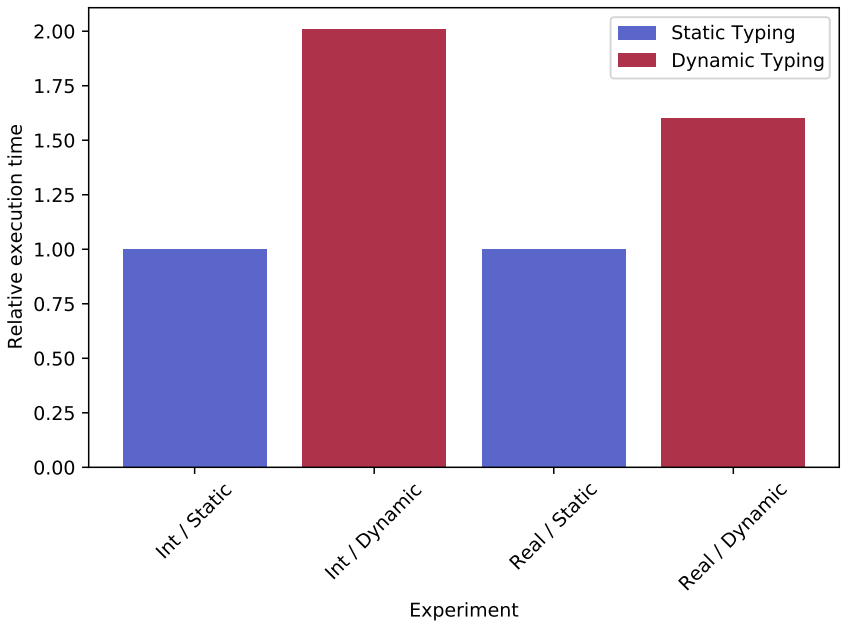


Figure 4.9: Relative performance of static and dynamic typing on basic workloads

These results are promising. To assess the impact of static typing on the performance of more realistic workloads, we execute the TPC-H queries using our static typing implementation and compare the execution times.

4.5.2 Performance gain on TPC-H queries

We continue our evaluation with the same experiment on TPC-H queries[9]. Queries for which our implementations cannot be used – Q11 and Q14 – (for instance, reals multiplied with integers) are skipped in this part of the evaluation.
Fig. 4.10 displays the relative performance of the default implementation and our static typing-based implementations on TPC-H queries.
We notice that the most prominent performance improvements are on the queries that use arithmetic operations and compare-and-jump instructions the most or those for which these instructions represent a majority of states.
For instance, we notice no substantial improvement on Q18 and Q2, whose dominating costs are the full table sorts generated by the SQLite code generator.
On the other hand, Q17 shows a significant performance gain. We attribute that to the large number of compare-and-jump instructions (when compared to other queries) generated by SQLite. We observe a similar effect on Q19 and Q20, which we attribute to the same reason.
We also notice though, that the performance gains brought by static typing in arithmetic operations are minor in comparison with those brought by compare-and-jump.

[9] As TPC-H queries are much more complex than the two previous basic queries, we use the tools presented in 3.4 to automatically establish type information.

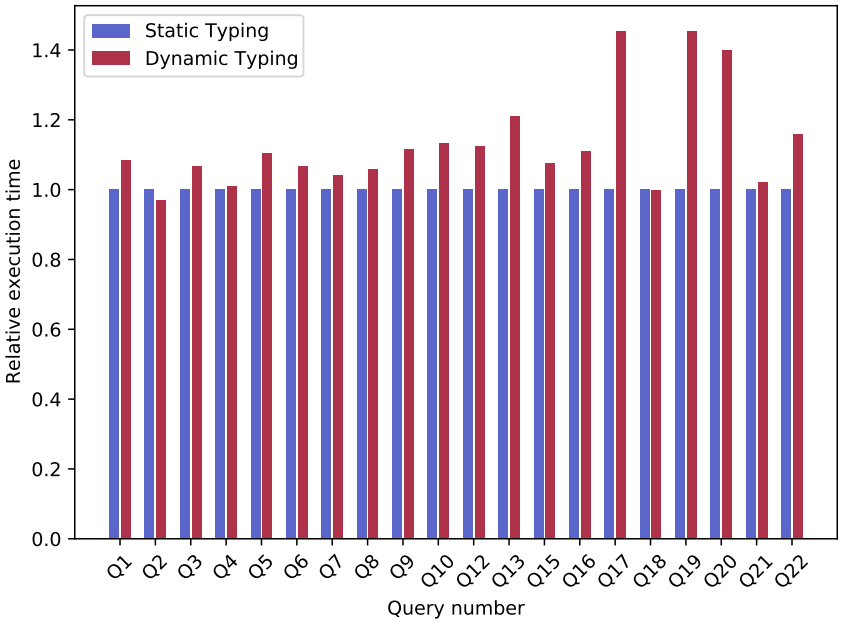


Figure 4.10: Relative performance of static and dynamic typing on TPC-H queries (25 samples)

Although they were significant on raw performance experiments (See 4.5.1), their effect is much less visible here. We attribute this discrepancy to the much lower proportion of arithmetic operations in realistic workloads, when compared to the experiments presented in 4.5.1.

4.6 Comparison with other SQL databases

4.6.1 HyPer

Although HyPer is not publicly available, we can benchmark it on TPC-H queries using an official Web interface[10]. Comparing HyPer on that setup[11] and LLVMSQLite on ours is not necessarily easy. To mitigate the CPU clock frequency difference, we plot another processing time h_{mit} over the raw HyPer measurements h_{raw}, which is given by $h_{mit} = 3.4/2.6 \times h_{raw}$. It represents a hypothetical HyPer processing time running on our server, assuming the bounding factor is the CPU frequency.

We run the TPC-H queries on both SQLite and HyPer in a scale-factor 1.0 TPC-H database. Figure 4.11 displays the relative performance between HyPer and LLVMSQLite.

[10] Queries can be ran at `https://hyper-db.de/interface.html`. Although the page states this interface should not be used for benchmarking – as the server is considered *low-end* and thus non-representative – it is sufficient in our case, as the performance of SQLite does not compete with HyPer, as results show.

[11] The setup for the Web interface is a server with an Intel(R) i7-3770 CPU running at 3.4GHz and 32GB RAM.

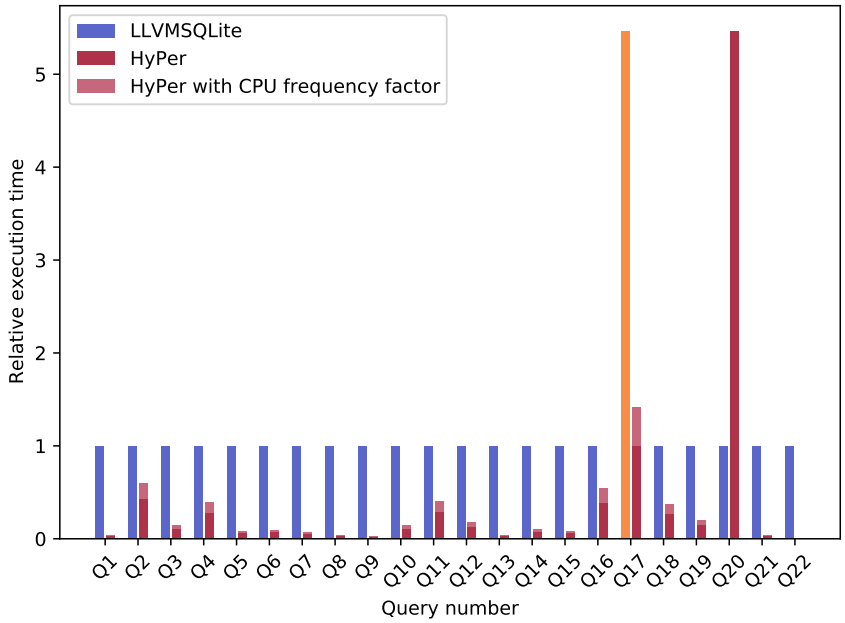


Figure 4.11: Relative performance of LLVMSQLite and HyPer on TPC-H. *Orange bars represent timeouts.*

We notice that, even when taking the difference in CPU frequency, HyPer generally performs much better than LLVMSQLite. This can be explained by the fact that HyPer is designed primarily for performance whereas SQLite, as an embedded database, finds a balance between deployment/execution overhead and query execution performance.

4.6.2 PostgreSQL

We continue our evaluation with PostgreSQL[12]. Like SQLite, PostgreSQL does not compile SQL statements to native code but rather uses a kind of interpreted query plan. Each statement is converted into a query plan (which can be obtained with the `EXPLAIN ANALYZE` token) that is then evaluated like a tree by the engine.

Fig. 4.12 displays the experimental results. We observe that PostgreSQL generally performs better than SQLite. This can be explained by its higher versatility: whereas SQLite often sticks to very simple implementations for operators, PostgreSQL can choose among many different implementations when optimising the query plan.
Still, Q2, Q4, Q19 and Q20 take more than five times the time to execute on PostgreSQL than they do LLVMSQLite.

[12] To execute the TPC-H benchmark on PostgreSQL, we use the instructions provided at `https://ankane.org/tpc-h`.

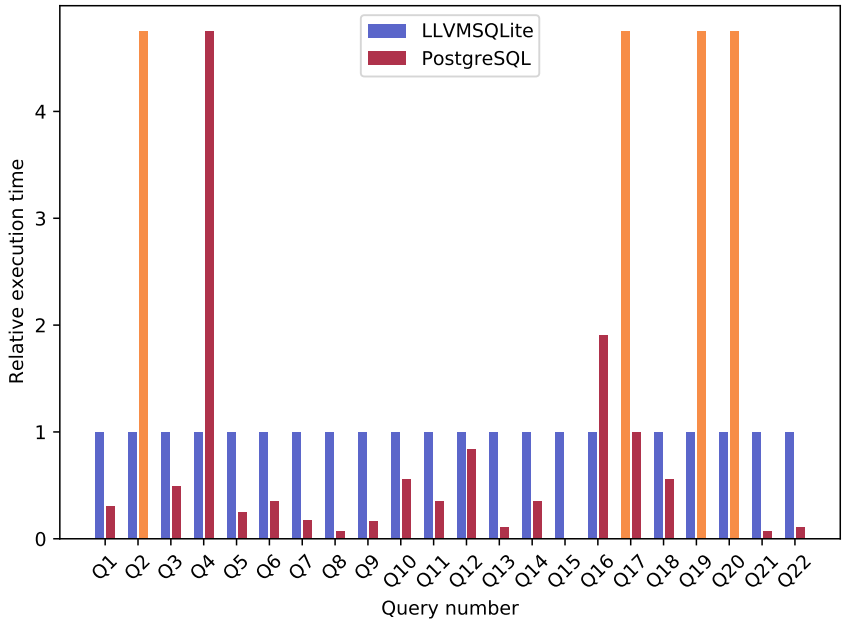


Figure 4.12: Relative performance on TPC-H queries between LLVMSQLite and PostgreSQL. *Orange bars represent timeouts.*

4.7 Discussion

4.7.1 Related work

SQPyte [7] is an attempt at making SQLite faster by using a just-in-time compiler. It is a partial rewrite of the SQLite bytecode interpreter in RPython[13]. SQPyte, as a JIT compiled implementation of the SQLite interpreter, is compiled into C code at runtime, which makes it able to interact with C libraries (using *Foreign function interface*, or FFI) or pre-existing C code (in this case, the default SQLite implementation).
Two hypotheses are the main motive behind SQPyte [7] :

(H1) The barrier between the end-user programme and the API implementation causes a significant overhead. 'Blurring' this barrier would give a significant performance gain, by allowing – for instance – inlining of certain function calls;

(H2) In data management systems that use an interpreted internal language – such as SQLite – replacing the default implementation with a JIT query compiler would greatly help performance.

To validate their hypothesis, Bolz et al. translate the implementation of some SQLite operation codes to RPython in separate functions. These functions are called by the *operation code dispatcher* (equivalent to the SQLite interpreter's `switch-case` statement) and annotated with a profiling decorator. This allows the RPython framework to automatically trace the VDBE state and operation calls.

[13] `https://www.pypy.org`

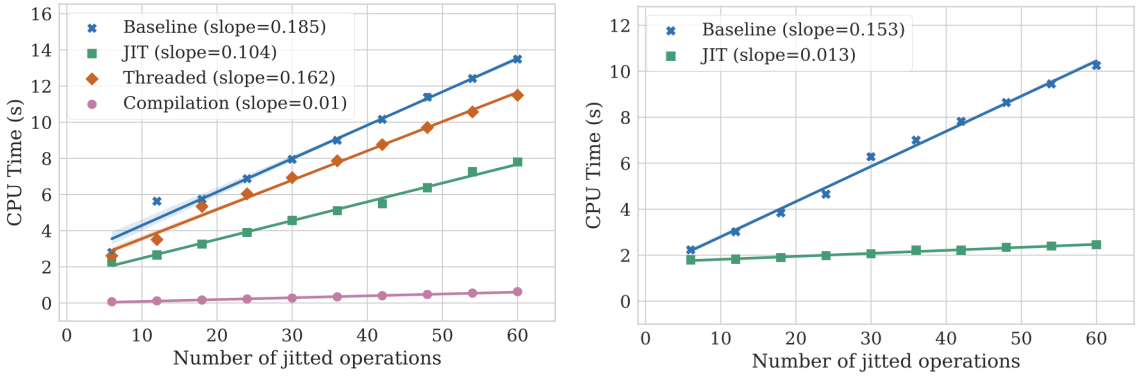
When a loop is called often enough and enough information is known about the context, RPython enters the JIT and compiles a part of the bytecode to native code. This native code is then called every time the VDBE enters the loop, providing better performance for the 'hot path' of the code without incurring the compilation overhead for other rarely called operations.
This also allows the authors to only carry out a partial port to RPython: as RPython allows to call C code, some operation codes are executed using calls to their default C implementation [7].

Bolz et al. conclude that the **H1** – blurring the frontiers between user and API implementation – is strongly validated. On the other hand, they conclude that **H2** – a JIT implementation of operation codes would bring a great performance gain – is not.
It is important to note that the very significant performance benefits claimed in the evaluation were obtained on the authors' benchmarks. Running similar experiments on classical database benchmarks, namely TPC-H [52], shows a performance gain of only 2.4% overall, which is much less significant.
The authors defend their results by arguing that TPC-H is executed on a database in isolation, while the main goal of SQPyte is to reduce overhead between programming languages (end users) and database engines [7].

SQPyte brings ideas for improving performance, some of which were used for the implementation of LLVMSQLite. Namely, they propose the following performance enhancements:

- They state that dynamic types tend to be constant at run-time. Most often, the data type in a column is constant over rows. Assuming the dynamic type of data is done automatically in RPython thanks to meta-tracing capabilities and speculation on the value of the type flag in `sqlite3_value`;
- C functions cannot return more than one element. This sometimes forces the default implementation to read the content of a flag variable, make a function call that may change it and re-read it after. Variadic function return statements allows to immediately get back the new value of the flags;
- Some operation codes need to loop over a number of elements that depends on run-time parameters. For instance, the `MakeRecord` creates a record instance that can hold a variable number of columns. At the moment the prepared statement is generated, the number of columns is already determined (and constant), though the default implementation needs to use a `for` loop to allow for any number of columns to be used. JIT compilation allows for loop-unrolling for values that are constant at prepared-statement generation time.

Kashuba and Mühleisen [23] take another direction to build a just-in-time compiler for SQLite In the case of VDBE 'tight loops' (for instance, iterating over values with a low cardinality, only seldom causing a VDBE interruption caused by the emission



(a) Evolution of the query execution time against number of loop operations [23]

(b) Evolution of the query execution time against the number of loop operations when using specialised operation code implementations [23]

Figure 4.13: Two JIT performance reports from [23]

of a result tuple), the default SQLite VDBE interpreter has a lot of branching which could be removed by compiling the loop body to native code.

To perform these partial compilations, Kashuba and Mühleisen use operation code templates (equivalent to the contents of the `case` labels' blocks in the `switch-case` statement). These templates are C code derived from the default implementation. When a certain loop body is run more than times than a threshold, a C code snippet equivalent to the content of the loop body is generated from the templates. The interpreter then calls a C compiler to generate a dynamic library that can be loaded and called from the SQLite interpreter.

The reason invoked to justify that the JIT-compiled generates C code instead of assembly or intermediate representation is that creating and maintaining even an intermediate representation generator is very cumbersome, while creating C-code templates is rather simple [23].
Unfortunately, this choice comes at a run-time cost. As already mentioned in 2.2.3, optimising C compilers are generally much slower than intermediate-representation compilers [36]. Kashuba and Mühleisen mitigate that limitation by only compiling hot loops (rather than the whole prepared statement) as it reduces the time taken by the compilation effort.

The evaluation carried out by Kashuba and Mühleisen shows that the time taken to execute a query can be reduced by a factor of up to $1.7\times$ (Fig. 4.13). This, though, depends on the query executed as well as its cardinality [23]. Particularly, high cardinality queries benefit less from just-in-time compilation because of the number of function calls involved, an effect we did observe in our evaluation.

A surprising result is the influence of specialised operation code implementations on the performance of the prepared statement. Some SQLite operation codes can act differently based on the value of a flag. This causes run-time stalls caused by control

hazards happening when the interpreter checks the value of these flags (See 3.1.5). As these flags are constant throughout VDBE execution and determined when the bytecode is generated, it is possible to generate specialised branch-free C code. According to Kashuba and Mühleisen, this does not only reduce control hazards, but also improves cache line utilisation, as only useful instructions are loaded.

4.7.2 Future work

Although LLVMSQLite lays a ground towards building an efficient JIT for SQLite, many limitations that stem from how SQLite is built remain. Furthermore, some optimisation whose potential was discovered in the evaluation have not been tackled. The following paragraphs propose various aspects that could be explored in the future for improving the performance of LLVMSQLite and similar just-in-time compilers.

Introducing bulk-processing in SQLite

As the evaluation part shows, JIT tends to improve execution time much more significantly on low-cardinality queries. That is a consequence of how SQLite emits tuples. Every time a tuple is emitted, the interpreter/JIT-compiled function is returned from and control is given back to the user (in our experiments, to the shell). To continue processing the statement, another function call to the interpreter/JIT-compiled function must be issued. This pattern has been pointed as a performance bottleneck by Kashuba and Mühleisen [23] and appeared once again in LLVMSQLite.

Several query execution engines have taken into account that dominating costs on modern systems are more caused by control hazards and pipeline bubbles than storage media access. Query processing in SQLite does not break this trend, and the performance of LLVMSQLite suffers from it.

To reduce that overhead in SQLite, bulk processing could also be used. In order to check the viability of this project, one could change the way tuples are emitted by making the interpreter copy records to a temporary buffer instead of stopping execution, and return only after a certain number of tuples have been emitted to the buffer.
This could be done transparently for higher-level APIs by wrapping the interpreter in a function that would extract tuples from the buffer and emit the codes that the interpreter would return for each tuple. Doing so would allow to add bulk processing to SQLite and assess the potential performance gains it would bring to query processing without disrupting the remainder of SQLite.

Reducing typing overhead

Section 3.4 explains why using static typing in SQLite is a difficult task. Manifest typing is seen as a first-class feature and to this end, the liberties it brings are used to their most and some choices in the implementation of VDBE are made to keep some parts simple – and efficient – at the cost of many type conversions.

To mitigate performance costs incurred by manifest typing, several measures could be imagined :

- Making SQLite Values able to hold several representations of a value at once. Currently, Values can hold a numeric value and its string representation. Allowing Values to hold at least the integer and float representation at the same time would remove the need for many costly conversions or duplicate extractions. In the search for best performance, the trade-off of consuming more memory – the size of a `double` for each Value – for faster operations should be considered;
- Changing the semantics of the `Affinity` operation code. Making it work out-of-place (on a shallow copy of the register, which is almost free) could be a way to mitigate that cost. The currently implemented solution – extracting the value twice – seems suboptimal as it triggers as many deserialisations as there are extractions.
- Making column types strict. By adding a *strict typing pragma* to the database, which would check the types of inserted tuples, SQLite could assume they are the expected ones when processing SQL statements. Though this would generate more branches in the interpreter, it would be useful in the JIT – correct typing could be assumed and less branches would be generated.

Reducing read times by loading the database in main memory

As mentioned in 2.1.3 and 4.3, SQLite is not an in-memory data management system, as opposed to HyPer [25] or VoltDB[14].
As a result, many operations in SQLite are slower than their in-memory counterparts by nature: scanning a relation, for instance, means reading each B^+-Tree node (and each page that node consists of) from the disk.

An in-memory caching mechanism exists, for which the number of cached pages can be set to an arbitrary number (only limited by RAM capacity) at run-time through a `pragma` command [28]. To try and replicate the behaviour of an in-memory database, we set the number of cached pages to a value higher than the size of database (in pages). Experiments show that it does not lead to a much better query processing time[15].

[14] VoltDB is an in-memory database which targets big data analytics and decision assistance with high performance originally based on H-Store [22]. More can be read at `https://www.voltdb.com/`

[15] Although these results are not presented in this report, they can be reproduced simply by adding `pragma cache_size = 524288` (or more) at the start of the aggregated microbenchmarks SQL script for a 2 GB database (the default page size in SQLite 3 is 4096 bytes).

Once again, that stems from the design of SQLite: it is not an in-memory database. HyPer uses mechanisms much more complex than just caching pages – for example, by using memory snapshots for transactions rather than writing to disk like SQLite does.

Pattern-based optimisations on VDBE / IR

One of the primary goals of MLIR is to provide a clean framework for high-level languages compilers for the future. To this end, it makes optimisation (as well as tracking code origins throughout optimisation) a first class priority. As well as providing *lowering passes*, a compiler writer can provide *transformation passes* that act on the IR and generate other IR from the same dialect that is presumably more efficient.
VDBE-IR does not make us of this part of MLIR. Indeed, the primary goal of this project is to provide a JIT compiler for SQLite prepared statements. SQLite itself, like most relational data management systems, already embeds a query optimiser[16]. This optimiser is able to perform most common query optimisations (using indices for `WHERE` clauses, order of tables in joins, *pushing down* some queries for compound queries...). These optimisations are at the relation algebra level. Once the bytecode has been generated, no optimisation is done on it.
Optimisations on the VDBE bytecode could be explored and implemented using MLIR and its optimisation passes, though we present no work on that.

[16] See `https://www.sqlite.org/optoverview.html`

Chapter 5

Conclusion

In this project, we discuss query execution in the SQLite embedded data management system and propose a design for a just-in-time compiler for the SQLite virtual machine. We then implement it using the LLVM and MLIR compiler infrastructures. Our approach allows to reuse most of the SQLite code base. It also brings support for future interface and data structure changes in SQLite by having the just-in-time compiler read the SQLite source code rather than rely on hardcoded definitions. As a result, it does not rely on the types of a specific SQLite version.

The approach chosen in LLVMSQLite to compile queries to native code is to generate an LLVM IR module based on bytecode using the MLIR API. This function is generated using transformation passes derived from the VDBE interpreter source code. We then apply various LLVM optimisations to the module and compile it just-in-time using the LLVM MCJIT API. This function behaves exactly like the interpreter would for the given VDBE and can be used in the exact same way.

The default SQLite implementation is completely dynamically typed. To reduce the cost incurred by type checks and conversions in query execution, we propose a partial static typing based on *a priori* knowledge of the data types. This can be seen as a lifting optimisation where the dynamic typing checks are performed at compile-time rather than at run-time. We integrate that design in LLVMSQLite and show that it can greatly reduce execution time in comparison and arithmetic operations.

As compilation costs are significant in JIT compilers, we propose a method for caching JIT-compiled optimised IR modules. Thanks to this approach, generated IR functions can be reused in subsequent executions of a query. As opposed to caching mechanisms found in the literature, our method does not require cached modules to be refreshed before reusing them. This significantly reduces the processing time by removing the costly compilation step from the execution pipeline.

To verify the correctness of the IR generated by the compiler, we propose automated regression tests based on *execution interleaving*. To ascertain that the implementation of a VDBE operation is valid, we run several queries using the interpreter then using the just-in-time compiler. The results are tested against those of the default

implementation. If they do not match, just-in-time operations are disabled one after the other until the faulty implementation is found.
This gives us confidence in the validity of our implementation, but also makes benchmarking meaningful. Failing to prove validity on the benchmark queries could indeed make comparisons irrelevant.

In order to evaluate the performance of our just-in-time compiler and replacement for the SQLite Virtual Machine, we carry out three different benchmarks: a custom set of microbenchmarks and two macrobenchmarks, TPC-H [52] and SSBM [37].

The microbenchmarks show that although not all basic statements benefit from just-in-time compilation, those which are computationally heavy tend to benefit more as the complexity of the expressions grows.
On the other hand, we notice that the cardinality of the queries has the opposite effect. Experiments show that high cardinality queries tend to run much slower than those whose cardinality is low. To explain this effect, we profile queries of varying selectivity. We observe that query execution time grows linearly with selectivity in SQLite. As a result, the performance improvements brought by the JIT are hidden away by the costs incurred by returns and calls in queries that produce many tuples.

Macrobenchmarks tend to give more promising results, and we generally observe a speedup of on average 15% when compared to interpreter-based processing.
Finally, although all queries do not benefit from just-in-time compilation, it is to note that it almost never hurts performance, which makes it a reasonable alternative for query processing.

Chapter 6

Appendices

6.1 Functions always inlined in LLVMSQLite

The following lists SQLite functions which are cloned and marked `alwaysinline` by LLVMSQLite when compiling the IR module.
They have been chosen based on profiling. The number of calls to each function, for each query was computed and most called functions (which we concluded were on the hot path) are the ones chosen for systematic inlining.

- `sqlite3VdbeMemIntegerify`
- `applyNumericAffinity`
- `sqlite3AtoF`
- `alsoAnInt`
- `sqlite3VdbeIntegerAffinity`
- `doubleToInt64`
- `sqlite3Atoi64`
- `sqlite3VdbeMemRealify`
- `sqlite3BtreeCursorHintFlags`
- `sqlite3VdbeMemShallowCopy`
- `sqlite3VdbeMemNulTerminate`
- `sqlite3VdbeRecordUnpack`
- `sqlite3GetVarint`
- `sqlite3VdbeSerialGet`
- `sqlite3AddInt64`
- `sqlite3SubInt64`
- `numericType`
- `sqlite3VdbeRealValue`
- `sqlite3MulInt64`
- `sqlite3IsNaN`
- `sqlite3VdbeIntValue`
- `applyAffinity`
- `sqlite3VdbeMemShallowCopy`
- `sqlite3VdbeMemExpandBlob`
- `sqlite3VdbeMemTooBig`
- `sqlite3BtreeLast`
- `sqlite3PutVarint`
- `sqlite3VarintLen`
- `sqlite3VdbeMemClearAndResize`
- `sqlite3VdbeSerialPut`

6.2 New `Print` operation for the VDBE language

`Print` is a new VDBE operation that prints the contents of a dynamically-typed SQLite Value: the register `P1`. It uses the `flags` field to determine what format specifier to use.
It is a relevant example of how the `sqlite3_value` data structure is used in the source of the VDBE interpreter.

```
switch (op.opcode) {
    // ...
    case OP_Print:
        sqlite3_value* reg = vdbe->aOp[op.p1];

        if (reg->flags & MEM_Int) {
            printf("Register %d holds %lld\n", op.p1, reg->u.i);
        } else if (reg->flags & MEM_Real) {
            printf("Register %d holds %f\n", op.p1, reg->u.r);
        } else if (reg->flags & MEM_Null) {
            printf("Register %d holds NULL\n", op.p1);
        } else if (reg->flags & MEM_Str) {
            printf("Register %d holds %s of length %u\n",
                    op.p1, reg->z, reg->n);
        } else {
            printf("Register %d cannot be printed.", op.p1);
        }

        break;
}
```

6.3 Volcano Processing Model Hash Join

The following pseudo-code is an example implementation of a `HashJoin` operator in the Volcano-style processing model.

```
class HashJoin : Operator {
    // Input A (Build side)
    Operator A;
    // Input B (Probe side)
    Operator B;

    // Hash index
    HashTable table;

    // Fills the hash table with tuples from A
    void generateTable() {
        do {
            tuple = A.next()
            table.insert(tuple.id, tuple);
        } while tuple != None
    }

    // Fetches next tuples from B (probe side) while it is not empty
    // and no corresponding tuple is found.
    // If a tuple from B matches one from A, returns the joined tuple.
    // Returns None if no tuple is left
    Tuple next() {
        if table.empty() {
            generateTable()
        }

        do {
            tuple = B.next();
        } while tuple != None and tuple.id not in table

        if tuple {
            return join(tuple, table[tuple.id]);
        } else {
            return None;
        }
    }
}
```

6.4 Example query as compiled to VDBE-IR

The following is the VDBE-MLIR IR for the `SELECT * FROM Relation` query, assuming the database schema has already been loaded.

```
func @jittedFunction(%arg0: !llvm<"%struct.Vdbe*">, %arg1: !llvm<"i8*">,
  ;                  ^^^^^^ Vdbe instance           ^^^^^^ &sqlite3CtypeMap
                     %arg2: !llvm<"i32*">,      %arg3: !llvm<"i8*">,
                     ^^^^^^ Pointer to iCompare ^^^^^^ Reference to "%s"
                     %arg4: !llvm<"i8*">) -> i32 {
                     ^^^^^^ &sqliteSmallTypesSize
    ; Load the value of vdbe->aMem
    %0 = llvm.mlir.constant(0 : i32) : !llvm.i32
    %1 = llvm.mlir.constant(19 : i32) : !llvm.i32
    %2 = llvm.getelementptr %arg0[%0, %1] : (!llvm<"%struct.Vdbe*">, !llvm.i32, !llvm.i32)
            -> !llvm<"%struct.sqlite3_value**">
    %3 = llvm.load %2 : !llvm<"%struct.sqlite3_value**">
    ; Load the value of vdbe->aCur
    %4 = llvm.mlir.constant(0 : i32) : !llvm.i32
    %5 = llvm.mlir.constant(21 : i32) : !llvm.i32
    %6 = llvm.getelementptr %arg0[%4, %5] : (!llvm<"%struct.Vdbe*">, !llvm.i32, !llvm.i32)
            -> !llvm<"%struct.VdbeCursor***">
    %7 = llvm.load %6 : !llvm<"%struct.VdbeCursor***">
    ; Load the value of vdbe->aOp
    %8 = llvm.mlir.constant(0 : i32) : !llvm.i32
    %9 = llvm.mlir.constant(23 : i32) : !llvm.i32
    %10 = llvm.getelementptr %arg0[%8, %9] : (!llvm<"%struct.Vdbe*">, !llvm.i32, !llvm.i32)
            -> !llvm<"%struct.VdbeOp**">
    %11 = llvm.load %10 : !llvm<"%struct.VdbeOp**">
    ; Load the value of vdbe->db
    %12 = llvm.mlir.constant(0 : i32) : !llvm.i32
    %13 = llvm.mlir.constant(0 : i32) : !llvm.i32
    %14 = llvm.getelementptr %arg0[%12, %13] : (!llvm<"%struct.Vdbe*">, !llvm.i32, !llvm.i32)
            -> !llvm<"%struct.sqlite3**">
    %15 = llvm.load %14 : !llvm<"%struct.sqlite3**">
    br ^resume
  ^resume: // pred ^resume
    ; Each possible PC value as a constant
    %16 = llvm.mlir.constant(0 : i32) : !llvm.i32
    %17 = llvm.mlir.constant(1 : i32) : !llvm.i32
    ...
    %29 = llvm.mlir.constant(13 : i32) : !llvm.i32
    ; The Resume-Switch.
    ; Syntax is switch(Val1, ..., ValN, DefaultBlock, Block1, ..., BlockN)
    "llvm.switch"(%16, %17, ..., %29, ^bb3, ^bb3, ^bb4, ..., ^bb16)
  ^bb3:        // 2 preds: ^resume, ^bb3
    ; VDBE Instruction 0: Init
    "vdbe.Init"()[^bb15] {jump_to = 0 : si64} : () -> ()
  ^bb4:        // 2 preds: ^resume, ^bb16
    "vdbe.Noop"() {pc = 1 : ui64} : () -> ()
    br ^bb5
  ^bb5:        // 2 preds: ^resume, ^bb4
    ; Open a cursor for reading table 1
    "vdbe.OpenRead"() {P4 = 5 : si32, P5 = 0 : ui16, counter = 2 : ui64,
```

```
                          curIdx = 0 : si32, database = 0 : si32, rootPage = 1 : si32} : () -> ()
  br ^bb6
^bb6:        // 2 preds: ^resume, ^bb5
  ; Put cursor 0 (just opened) at start
  "vdbe.Rewind"() {curIdx = 0 : si32, jumpTo = 11 : si32} : () -> ()
  br ^bb7
^bb7:        // 3 preds: ^resume, ^bb6, ^bb13
  ; Extract column 0 into register 1
  "vdbe.Column"() {column = 0 : si32, counter = 4 : si64, curIdx = 0 : si32,
                        defaultValue = 0 : ui64, extractTo = 1 : si32, flags = 0 : ui16} : () -> ()
  br ^bb8
^bb8:        // 2 preds: ^resume, ^bb7
  ; Extract column 1 into register 2
  "vdbe.Column"() {column = 1 : si32, counter = 5 : si64, curIdx = 0 : si32,
                        defaultValue = 0 : ui64, extractTo = 2 : si32, flags = 0 : ui16} : () -> ()
  br ^bb9
^bb9:        // 2 preds: ^resume, ^bb8
  ; Extract column 2 into register 3
  "vdbe.Column"() {column = 2 : si32, counter = 6 : si64, curIdx = 0 : si32,
                        defaultValue = 0 : ui64, extractTo = 3 : si32, flags = 0 : ui16} : () -> ()
  br ^bb10
^bb10:        // 2 preds: ^resume, ^bb9
  ; Extract column 3 into register 4
  "vdbe.Column"() {column = 3 : si32, counter = 7 : si64, curIdx = 0 : si32,
                        defaultValue = 0 : ui64, extractTo = 4 : si32, flags = 0 : ui16} : () -> ()
  br ^bb11
^bb11:        // 2 preds: ^resume, ^bb10
  ; Extract column 4 into register 5
  "vdbe.Column"() {column = 4 : si32, counter = 8 : si64, curIdx = 0 : si32,
                        defaultValue = 0 : ui64, extractTo = 5 : si32, flags = 0 : ui16} : () -> ()
  br ^bb12
^bb12:        // 2 preds: ^resume, ^bb11
  ; Emit a tuple with registers 1 ... 5
  "vdbe.ResultRow"() {counter = 9 : si64, firstCol = 1 : si32, nCol = 5 : si32} : () -> ()
^bb13:        // pred: ^resume
  ; Advance the cursor.
  ; Note the only predecessor is ^resume, as ResultRow never falls through
  "vdbe.Next"()[^bb7, ^bb14] {P5 = 1 : ui16, advancer = 4506194000 : ui64,
                                    curHint = 0 : si32, curIdx = 0 : si32} : () -> ()
^bb14:        // 2 preds: ^resume, ^bb13
  ; Exit VDBE execution
  "vdbe.Halt"() : () -> ()
^bb15:        // 2 preds: ^resume, ^bb3
  "vdbe.Transaction"() {P3 = 11 : si32, P4 = 0 : si64, P5 = 0 : si16, dbIdx = 0 : si32,
                            isWrite = 0 : si32} : () -> ()
  br ^bb16
^bb16:        // 2 preds: ^resume, ^bb15
  ; Unconditionally jump back to ^bb4
  "vdbe.Goto"()[^bb4] : () -> ()
}
```

6.5 `CustomOpBuilder` specification

The `CustomOpBuilder` used for emitting LLVMDialect operations in MLIR has the following uses. Each `insertXOp` operation has a macro-defined lambda function `x` counterpart which does the same thing with less verbosity and simply forwards its argument to the `insertX` method.

For instance, for `insertAllocaOp(Type ty)`, a lambda function `alloca(Type ty)` exists and is defined as follows:

```
auto alloca = [&builder](Location loc, Type ty) {
              // ^^^^^^^ CustomOpBuilder instance
    return builder.insertAllocaOp(loc, ty);
};
```

`insertAllocaOp(Type ty)` inserts an `alloca` instruction of size `1` with an alignment of 8 bytes and returns it.

`insertCallOp([LLVMFuncOp|Value] f, ValueRange args)` inserts a `call` instruction to the `LLVMFuncOp` or `Value f` whose arguments are given in the `ValueRange` args. It also checks for the type of `f` and returns an `Optional` which is `nullopt` if the return type of `f` is `void` and the return value of `f` otherwise.
A variadic specialisation exists for the `call` lambda which accepts several `Values` instead of a single `ValueRange` and uses them to construct a `valueRange`.

`insertICmpOp(ICmpPredicate pred, Value lhs, Value rhs` inserts an `icmp` instruction between `lhs` and `rhs` with comparison predicate `pred` (`eq`, `slt`, *etc.*). It also has a specialisation that accepts a C pointer as `rhs` to easily compare pointer-typed values with `nullptr`, in which case it checks that the type of `lhs` is a pointer type.

`insertBranchOp(Block* b)` inserts a `branch` instruction to block `b`.

`insertCondBranchOp(Value cond, Block* ifTrue, Block* ifFalse)` inserts a `condBr` instruction. The condition operand to `condBr` is the 1-bit wide value `cond`, the *true branch* is `ifTrue`; the *false branch* is `ifFalse`.

`insertGEPOp(Type ty, Value base, ValueRange indices)` inserts a `getelementptr` instruction on `base` with indices `indices`. The `ty` parameter is the type that should be returned by the `getelementptr` instruction. In LLVM, this type can be omitted (in which case it is deduced) but not in MLIR.
A variadic specialisation exists for the `getElementPointer` lambda which accepts several C integers instead of a single `ValueRange` and uses them to construct MLIR constants which are in turn used to construct a `ValueRange`.

`insertLoad(Value addr)` inserts a `load` instruction for address `addr` (which can be the result of a GEP operation or an `alloca`).

`insertPtrToIntOp(Value p)` inserts a `ptrtoint` operation for pointer-value `p`. It also checks that the type of `p` is a pointer-type.

`insertIntToPtrOp(Type ty, Value x)` inserts an `inttoptr` instruction for integer `x` to type `ty`. It also checks that `ty` is a pointer-type. A specialisation exists that takes a C integer and builds an MLIR constant from it.

`insertBitCastOp(Value x, Type ty)` inserts a `bitcast` instruction (similar to the C++ `reinterpret_cast`) for value `x` to type `ty`.

`insertZExtOp(Value x, size_t width)` inserts a `zext` instruction (zero-extension) for value `x` to integer of width `width`.

`insertTruncOp(Value x, size_t width)` inserts a `trunc` instruction (truncate) for value `x` for integer of width `width`.

`insertOrOp(Value x, Value y)` inserts an `or` instruction for integers `x` and `y`. A specialisation exists that takes a C integer and builds an MLIR constant from it (for places where literals are used: `flags = bitOr(flags, MEM_Real)`.

`insertAndOp(Value x, Value y)` inserts an `and` instruction for integers `x` and `y`. A specialisation exists that takes a C integer and builds an MLIR constant from it (for places where literals are used: `flags = bitAnd(flags, MEM_IntReal)`.

`insertStoreOp(Value x, Value into)` inserts a `store` instruction for storing value `x` into address-like Value `y`. A specialisation exists that takes a C integer or a C pointer and builds an MLIR constant from it (which is useful for initialising stack-allocated elements: `iAddr = alloca(T::i32PtrTy); store(0, iAddr);`). This function type-checks the arguments to check at MLIR level that the pointer `into` is well typed for value `x`.

`insertISDivOp(Value divided, Value by)`, `insertIMulOp(Value a, Value b)`, `insertISRemOp(Value divided, Value by)`, `insertISubOp(Value a, Value b)` respectively insert an integer signed division, an integer multiplication, an integer signed modulus and an integer subtraction. Specialisations exist and accept C integers.

`insertSaveStack` inserts a `stacksave` instruction.

`insertRestoreStack(Value state)` inserts a `stackrestore` instruction for restoring the stack to a previously saved state.

6.6 Microbenchmarks queries listing

The following is an aggregation of all queries used in microbenchmarks (see 4.3.1).

```
--- Query 1
--- Simple select
-- This allows to test whether the performance of
-- going through the entire table and generating many
-- tuples differs greatly.
SELECT * FROM Items;


--- Query 2
--- Restricted SELECT on indexed column
-- This allows to check if tree traversal
-- is more / less efficient on the JIT.
SELECT * FROM Items WHERE I_ID < 30000;


--- Query 3
--- Restricted SELECT on non-indexed column
-- This allows to see whether relation traversal
-- is more / less efficient on the JIT.
SELECT * FROM Items WHERE I_Price < 231964.11;


--- Query 4
--- Aggregation
-- This allows to check whether aggregating tuples
-- is more / less efficient in the JIT.
SELECT SUM(I_ID * I_Category + I_Price + (I_ID * I_Price)) FROM Items;


--- Query 5
--- Join on indexed column
-- This allows to see if indexed joins are more / less
-- efficient in the JIT.
SELECT * FROM Sales JOIN Clients ON Sales.S_Client = Clients.C_ID;


--- Query 6
--- Join on non-indexed column
-- This allows to check whether non-indexed joins are more / less
-- efficient in the JIT.
SELECT S_ID, S_Price, S_Client, S_Date, SI_I
FROM Sales JOIN SaleItem ON Sales.S_ID = SaleItem.SI_S;


--- Query 7
--- Nested query
-- This allows to check whether nested subqueries (coroutines) are more / less
-- efficient in the JIT.
SELECT S_ID, S_Price, S_Client, S_Date, I_ID, I_Name, I_Price, I_Category
FROM Items JOIN
        (
```

```
                SELECT S_ID, S_Price, S_Client, S_Date, SI_I
                FROM Sales JOIN SaleItem ON Sales.S_ID = SaleItem.SI_S
        ) As Temp
ON Temp.SI_I = Items.I_ID;


--- Query 8
--- Two nested queries
-- More complex nested query benchmark to see the influence of the number of
-- nested queries.
SELECT S_ID, S_Price, S_Client, S_Date, I_ID, I_Name, I_Price, I_Category
FROM Items JOIN
        (
                SELECT S_ID, S_Price, S_Client, S_Date, SI_I
                FROM
                        (
                                SELECT * FROM Sales WHERE S_ID < (SELECT MIN(S_ID) + 5000 FROM Sales)
                        ) AS Sales
                JOIN SaleItem ON Sales.S_ID = SaleItem.SI_S
        ) As Temp
ON Temp.SI_I = Items.I_ID;


--- Query 9
--- Query with ORDER BY
-- This allows to check if the Sorter module
-- is more / less efficient in the JIT.
SELECT SI_I FROM SaleItem ORDER BY SI_S;


--- Query 10
--- Query with HAVING
SELECT SI_I FROM SaleItem
GROUP BY SI_S HAVING SUM(SI_I) + SI_S > 5000000;


--- Query 11
--- Query with GROUP BY, ORDER BY and HAVING
-- Example of a more complex query.
SELECT AVG(SI_I) FROM SaleItem
GROUP BY SI_S HAVING SUM(SI_I) + SI_S > 5000000
ORDER BY SI_S;


--- Query 12
--- Nested query with two ORDER BY's and HAVING
-- More complex query with a nested subquery.
SELECT AVG(SI_I) FROM (
    SELECT SI_I, SI_S
    FROM SaleItem
    WHERE SI_S > 1000
    ORDER BY SI_S + SI_I * 2
) SaleItem
GROUP BY SI_S HAVING SUM(SI_I) + SI_S > 5000000
ORDER BY SI_S;
```

```
--- Query 13
--- Aggregation
-- This allows to check whether aggregating tuples
-- is more / less efficient in the JIT.
SELECT SUM(
        I_ID * I_Category + I_Price + (I_ID * I_Price) - (I_Category * (I_ID - I_Price * I_ID))
    )
FROM Items;


--- Query 14
--- Query with HAVING
SELECT * FROM (
        SELECT SI_I FROM SaleItem
        GROUP BY SI_S
        HAVING SUM(SI_I) + SI_S > 5000000
) GROUP BY SI_I HAVING SUM(SI_I) > 5000;


--- Query 15
--- Aggregation
-- This allows to check whether aggregating tuples
-- is more / less efficient in the JIT.

SELECT SUM(I_ID * I_Category + I_Price + (I_ID * I_Price)) FROM Items;


--- Query 16
--- Aggregation
-- This allows to check whether aggregating tuples
-- is more / less efficient in the JIT.

SELECT I_ID * I_Category + I_Price + (I_ID * I_Price) FROM Items;


--- Query 17
--- Aggregation
-- This allows to check whether aggregating tuples
-- is more / less efficient in the JIT.
SELECT SUM(
        I_ID * (I_Category - 23.0) * (I_ID / 4.0) + I_Price + (I_ID * I_Price - 4)
    )
FROM Items;
```

Chapter 7

User's guide

7.1 Getting LLVMSQLite

LLVMSQLite consists in two code bases:

- The LLVMSQLite code itself, which contains a slightly modified version of the SQLite source code and the MLIR definitions (dialect, passes, *etc.*) for the VDBE-IR dialect;
- A fork of the LLVM repository[1] (dating back to the 21st of April 2020) on which modifications have been made to the LLVM MLIR dialect to allow more operations (see 3.3.2 Modifications to the MLIR API).

The LLVMSQLite code needs to be compiled and linked against the custom LLVM version in order to run properly. Trying to compile LLVMSQLite using a stock version of LLVM will fail at compile-time because of missing LLVM Dialect operation definitions.
LLVMSQLite has been tested on the latest versions of Ubuntu and macOS.

To get the necessary source code, clone the following two Git repositories:

```
# Get LLVMSQLite
git clone https://gitlab.doc.ic.ac.uk/tk1319/llvmsqlite
# Get the custom LLVM version
git clone https://gitlab.doc.ic.ac.uk/tk1319/llvmsqlite_llvm \
    --depth 1

# (Optional) Get the benchmarking tools
git clone https://gitlab.doc.ic.ac.uk/tk1319/llvmsqlite_dbgen
```

[1] The official LLVM repository can be found at `https://github.com/llvm/llvm-project`

7.2 Compiling LLVM

LLVM can be compiled using CMake and any C++ compiler.
The following example uses `clang` and `clang++` but other values can be chosen for `CC` and `CXX`. The recommended build tool is Ninja[2] for speed reasons. Compiling with `make` can be done by removing `-G Ninja` but is recommended against.
This only builds the `MLIR` and `Clang` target in LLVM, as they are the only ones required by LLVMSQLite. Compiled tools and libraries are installed to sub-directories in `~/llvm`.

```
# After cloning the custom LLVM in CWD
# or extracting it from the archive
mkdir ~/llvm/
cd llvmsqlite_llvm
mkdir build && cd build

export CC=clang
export CXX=clang++

cmake -G Ninja ../llvm -DLLVM_ENABLE_PROJECTS="mlir;clang" \
-DLLVM_TARGETS_TO_BUILD="X86" \
-DCMAKE_BUILD_TYPE=RelWithDebInfo \
-DLLVM_ENABLE_ASSERTIONS=ON

DESTDIR=~/llvm ninja install
```

Note If you install LLVM in another directory than `~/llvm`, edit the `LLVMSQLite/src/mlir/CMakeLists.txt` file and update the following line to match your LLVM install directory:

```
# The directory where LLVM libraries are installed
set(PREFIX "ENV{HOME}/llvm/usr/local/")
```

7.3 Compiling LLVMSQLite

Once LLVM has been compiled, LLVMSQLite can be compiled and linked against it. Once again, you may use a different compiler by changing `CC` and `CXX`. Ninja is once again the recommended build tool for performance reasons.

[2] Official website: `https://build.ninja`

Note Using the just-built `clang(++)` may not work because it has not necessarily been linked against the C++ standard library. It is recommended that you use your system toolchain rather.

```
# After cloning LLVMSQLite in CWD
# or extracting it from the archive
cd llvmsqlite
mkdir release && cd release

export CC=clang
export CXX=clang++

cmake                                                \
    -G Ninja ../                                     \
    -DCLANG11=home/you/llvm/usr/local/bin/clang      \
    -DCMAKE_BUILD_TYPE=RelWithDebInfo

ninja shell_default
ninja
```

Defining the `CLANG11` constant to the `clang` that has just been built allows the CMake project to generate dependencies (`sqlite3.ll`) using a compiler that complies with the IR generated and parsed by the LLVM library embedded in LLVMSQLite.

Note When profiling, you may enable Intel VTune support (on Linux only) by adding `-DVTUNE=ON` at the end of the CMake invocation.
Profiling with Apple Instruments does not require adding anything other than compiling with debug information.

Note Other instrumentation tools can be enabled at compile-time: `-DSANITIZER=[address|memory|undefined]` enables respectively ASan [46], MSan [48] and UBSan[3]. `-DXRAY=ON` enables the LLVM X-Ray[4] profiling tool.

7.3.1 Using LLVMSQLite

LLVMSQLite comes in the form of a modified version of the SQLite shell. After compilation, the `build` folder should contain three binaries:

`shell_default` The default, interpreter-based, SQLite shell;

`shell_jit` The LLVMSQLite shell, powered by JIT-compiled prepared statements;

`profilable_shell` A mix of the two first shells. JIT can be switched on or off by passing `-jit` or `-nojit`. This shell allows profiling SQLite with VTune, which refuses to compare profiles obtained on unrelated binaries.

[3] `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`
[4] `https://llvm.org/docs/XRay.html`

Both shells work with all SQLite database files (the JIT does not introduce file incompatibilities).

Note Only a subset of the VDBE bytecode language has been implemented in LLVM-SQLite. As a result, JIT shells may fail to execute queries for which it does not implement the operations.
Namely, no transactional/OLTP operations are supported.

7.4 Profiling LLVMSQLite

The best way to profile SQLite is to generate database files using the tools provided in `LLVMSQLite_DBGen`.

```
cd LLVMSQLite_DBGen
make -f MakefileSSBM # Generates SSBM-*.db
make -f MakefileTPCH # Generates TPC-H-*.db
```

This will generate four databases of varying sizes for each benchmark (`small`, `med`, `big` and `huge`).
SQL queries and statement templates can be found in the `LLVMSQLite/llvmsqlite_util/benchmarking/[tpch|ssbm|micro]` folders.
To generate an SQL statement for TPC-H query q, use the following:

```
cd LLVMSQLite/llvmsqlite_util
python3 generator_tpch.py --query q
# Pipe into shell...
    | ../../build/shell_jit /path/to/tpch.db
# ... or dump to file
    > query.sql
```

Statements for microbenchmarks and SSBM are static and can directly be read by the interpreter.

Bibliography

[1] Allen, G. and Owens, M. (2010). *The Definitive Guide to SQLite*. Apress. pages 14

[2] Ancona, D., Ancona, M., Cuni, A., and Matsakis, N. D. (2007). RPython. In *Proceedings of the 2007 Symposium on Dynamic languages - DLS '07*. ACM Press. pages 17

[3] Armstrong, J. (2014). *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Royal Institute of Technology. pages 18

[4] Aycock, J. (2003). A Brief History of Just-in-Time. *ACM Computing Survey*, 35:97–113. pages 15, 17, 58

[5] Barata, M., Bernardino, J., and Furtado, P. (2015). An Overview of Decision Support Benchmarks: TPC-DS, TPC-H and SSB. In Rocha, A., Correia, A. M., Costanzo, S., and Reis, L. P., editors, *New Contributions in Information Systems and Technologies*, pages 619–628, Cham. Springer International Publishing. pages 28, 29

[6] Bastoul, C. and Feautrier, P. (2003). Reordering methods for data locality improvement. *HAL Archive*. pages 61

[7] Bolz, C. F., Kurilova, D., and Tratt, L. (2016). Making an Embedded DBMS JIT-friendly. In Krishnamurthi, S. and Lerner, B. S., editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:24, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. pages 84, 85

[8] Boncz, P. A., Zukowski, M., and Nes, N. (2005). MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*. pages 10

[9] Bondhugula, U. (2020). Polyhedral Compilation Opportunities in MLIR. pages 26

[10] Brooks, G., Hansen, G. J., and Simmons, S. (1992). A New Approach to Debugging Optimized Code. *SIGPLAN Not.*, 27(7):1–11. pages 22

[11] Buda, T., Cerqueus, T., Grava, C., and Murphy, J. (2017). Rex: Representative Extrapolating Relational Databases. *Information Systems*, 67. pages 29

[12] Chang, M., Bebenita, M., Yermolovich, A., Gal, A., and Franz, M. (2007). Efficient just-in-time execution of dynamically typed languages via code specialization using precise runtime type inference. Technical report, Citeseer. pages 16

[13] Cohen, A. (2019). An Overview of Loop Nest Optimization, Parallelization and Acceleration in MLIR. pages 26

[14] Costa, I., Alves, P., Santos, H. N., and Quintão Pereira, F. M. (2013). Just-in-Time Value Specialization. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11. pages 16

[15] Cramer, T., Friedman, R., Miller, T., Seberger, D., Wilson, R., and Wolczko, M. (1997). Compiling Java just in time. *IEEE Micro*, 17(3):36–43. pages 15, 17

[16] Ezginci, Y. and Karadas, M. (2019). Access Control and Recording System Using SQLite and RFID. *International Journal of Scientific and Technological Research*. pages 4

[17] Graefe, G. (1994). Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6:120–135. pages 10

[18] Haldar, S. (2007). *Inside SQLite*. O'Reilly. pages 5, 6, 7, 11, 12, 15

[19] Hipp, R. D. (2020). SQLite. pages 3

[20] ISO/IEC JTC 1/SC 32 Data management and interchange (1992). Information technology – Database languages – SQL. Standard, American National Standards Institute. pages 3

[21] Kallas, K. and Sagonas, K. (2018). HiPErJiT: A Profile-Driven Just-in-Time Compiler for Erlang. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages*, pages 25–36. pages 2, 17, 18

[22] Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. P. C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., and Abadi, D. J. (2008). H-Store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499. pages 88

[23] Kashuba, A. and Mühleisen, H. (2018). Automatic Generation of a Hybrid Query Execution Engine. arXiv:1808.05448v1. pages 12, 13, 85, 86, 87

[24] Kaylor, Andy and Andric, Dimitry (2016). MCJIT Design and Implementation. Technical report, LLVM Project. pages 22, 24, 65

[25] Kemper, A. and Neumann, T. (2011). HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. *2011 IEEE 27th International Conference on Data Engineering*, pages 195–206. pages 88

[26] Kimball, R. and Ross, M. (2013). *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*. Wiley Publishing, 3rd edition. pages 30

[27] Korobeynikov, A. (2007). Improving switch lowering for the llvm compiler system. In *Proceeding of the 2007 Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE)*. pages 47

[28] Kreibich, J. (2010). *Using SQLite*. O'Reilly, Sebastopol, Calif. pages 4, 88

[29] Lattner, C. (2003). *LLVM: An Infrastructure for Multi-Stage Optimization*. PhD thesis, University of Portland. pages 2, 16, 20, 21

[30] Lattner, C., Pienaar, J., Amini, M., Bondhugula, U., Riddle, R., Cohen, A., Shpeisman, T., Davis, A., Vasilache, N., and Zinenko, O. (2020). MLIR: A Compiler Infrastructure for the End of Moore's Law. Google Research. pages 2, 23, 24, 26, 36

[31] Lee, M., Lee, M., and Kim, C. (2016). A JIT Compilation-Based Unified SQL Query Optimization System. In *2016 6th International Conference on IT Convergence and Security (ICITCS)*, pages 1–2. pages 18

[32] LLVM Project (2020a). *LLVM Documentation*. pages 45, 59, 60

[33] LLVM Project (2020b). *LLVM IR Language Reference*. pages 21, 47, 48

[34] Melnik, D., Buchatskiy, R., Zhuykov, R., and Sharygin, E. (2017). Dynamic Compilation of SQL Queries in PostgreSQL Using LLVM. In *PGCon '17 Proceedings*. pages 19, 20

[35] Myers, G. J. and Sandler, C. (2004). *The Art of Software Testing*. John Wiley & Sons, Inc., Hoboken, NJ, USA. pages 66

[36] Neumann, T. (2011). Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4:539–550. pages 2, 18, 19, 20, 86

[37] O'Neil, P., O'Neil, B., and Chen, X. (2009). The Star Schema Benchmark (SSB). Technical report, University of Massachusetts Amherst. pages 28, 30, 74, 91

[38] Pantilimonov, M., Buchatskiy, R., Zhuykov, R., Sharygin, E., and Melnik, D. (2019). Machine Code Caching in PostgreSQL Query JIT-Compiler. In *2019 Ivannikov Memorial Workshop (IVMEM)*, pages 18–25. IEEE. pages 20, 64, 65

[39] Pirk, H. (2010). *Cache Conscious Data Layouting for In-Memory Databases*. Humboldt-Universität zu Berlin. pages 61

[40] Purohith, D., Mohan, J., and Chidambaram, V. (2017). The Dangers and Complexities of SQLite Benchmarking. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, New York, NY, USA. Association for Computing Machinery. pages 71

[41] Reames, P. (2017). Falcon: An Optimizing Java JIT. pages 2

[42] Rigo, A. and Pedroni, S. (2006). PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications - OOPSLA '06*. ACM Press. pages 17

[43] Rohland, C., Dickins, H., and Motohiro, K. (2007). *TMPFS Documentation*. pages 71

[44] Rohou, E., Swamy, B. N., and Seznec, A. (2015). Branch prediction and the performance of interpreters — Don't trust folklore. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 103–114. pages 15

[45] Sanchez, J. (2016). A Review of Star Schema Benchmark. pages 31

[46] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. (2012). AddressSanitizer: A Fast Address Sanity Checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, Boston, MA. USENIX. pages 66, 105

[47] Shanley, K. (1998). Origins of the TPC and the first 10 years. Technical report, Transaction Processing Council. pages 28, 31

[48] Stepanov, E. and Serebryany, K. (2015). MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 46–55, San Francisco, CA, USA. pages 66, 105

[49] The SQLite Project (2020a). *SQLite Documentation*, 2020 edition. pages 1, 3, 4, 5, 11, 12, 15, 57, 58

[50] The SQLite Project (2020b). *The SQLite License*, 2020 edition. pages 31

[51] Thomas Kowalski (2020). Private conversations with Lang Hames over the course of this project. pages 51

[52] Transaction Processing Performance Council (2018). TPC Benchmark H (Decision Support). Technical report, TPC. pages 28, 29, 74, 85, 91

[53] Winslett, M. and Braganholo, V. (2019). Richard Hipp Speaks Out on SQLite. *SIGMOD Rec.*, 48(2):39–46. pages 4

[54] Yibo, X., Bifeng, Y., Shangchang, M., and Yanjun, Z. (2019). Research and Application of Integrated SQLite Based on Ground Meteorological Observation. In *2019 International Conference on Meteorology Observations (ICMO)*, pages 1–4. pages 4

[55] Zhao, X., Ding, J., Ma, R., Gong, Q., and Yang, Y. (2017/09). Research and Improvement of SQLite's Concurrency Control Mechanism. In *Proceedings of the 2017 5th International Conference on Mechatronics, Materials, Chemistry and Computer Engineering (ICMMCCE 2017)*, pages 1547–1555. Atlantis Press. pages 3

N.B. All URLs included in the bibliography and in footnotes have been saved and can be seen in the state they were when this report was written using archive.org.