An introduction to compilers for databases people

MAX BERNSTEIN, Northeastern University, USA

NOTE: This is a draft. I wrote it for a course and I have not reviewed it since. Please do not take it too seriously or post it publicly. Feel free to email me if you have questions or comments. Thanks.

There is significant overlap in the problem spaces for databases and compilers. Compiler people can learn from databases people and vice versa. In this article we propose an abbreviated syllabus for an introductory compilers course designed specifically for databases people. We also present some (potentially naïve) suggestions for optimizing query evaluation engines.

Additional Key Words and Phrases: databases, compilers, programming languages, optimization, SSA

1 INTRODUCTION

Databases people and compilers people don't talk with one another as much as they should. Compilers and programming languages folk, engineers and academics alike, manage more data than they would like to admit. Database herders begrudgingly use a programming language instead of relational algebra for their incantations over huge and varying tables.

This common programming language and its variants, collectively dubbed Structured Query Language (SQL), has remained roughly in the same form since its birth in the 1970s. The language is fine; much as with Excel, people use it with success every day. But the underlying implementations can sometimes ignore what is considered "common knowledge" to those who regularly work with compilers.

This proposal outlines a curriculum for an imaginary literature review course targeted at databases folk. It:

- draws parallels between existing database (DB) and programming language (PL) terms;
- gives a PL perspective of query plans and query optimization;
- and points to further advanced literature that is beyond the scope of an introductory course.

For each optimization technique, we will provide a description, an example of a program that could benefit from the optimization, and an imaginary optimized version of that same program, if applicable. Depending on the optimization pass, these examples might be in SQL or some other programming language.

Note that we will assume that no database indices exist; while they can be used as inputs to greatly speed up query planning and execution, they are unrelated to the main points we hope to make here.

We hope that this is illuminating and reduces some of the mystery shrouding programming languages and compilers. As with many fields, we have invented many terms, languages, and imbued many existing objects with new meaning.

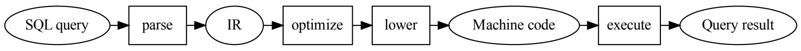
1.1 Prerequisite knowledge

This paper assumes general programming knowledge and familiarity with common data structures like trees. It also leans heavily on Python and SQL syntax for example code snippets. It is otherwise fairly readable for any programmer.

Some of the SQLite bytecode snippets may appear impenetrable, but they are not meant to be fully read and understood. Glance at them once or twice, confirm that it kind of looks like the relevant *thing* is present in the listing, and move on.

Author's address: Max Bernstein, bernstein.ma@northeastern.edu, Northeastern University, #202, West Village Residence Complex H, 440 Huntington Ave, Boston, Massachusetts, USA, 02115.

Fig. 1. Phases of a hypothetical optimizing query compiler. Ovals represent program representations and rectangles represent stages of compilation and execution. This is similar enough to the structure of major compiler projects in the wild.



2 COMMON IDEAS WITH DIFFERENT NAMES

Query. We call this a **source program** or **expression**. It is an input from the programmer, designed to be optimized, executed, and maybe also read by other engineers. It will eventually be transformed into another representation that a machine can directly execute.

Query planner. We call this a **compiler**. In broad terms, a compiler is a function that receives a program as an input and produces a program as an output: $f : P \longrightarrow P'$. The programs may be in the same language or they may not be; a source to source optimizer for Python is just as valid as a TypeScript to JavaScript compiler is just as valid as a C to machine code compiler. There is often some confusion about this, especially when a compiler's source and target languages are languages traditionally written by humans (for example, Python to JavaScript). People may call this a "transpiler". But there is no real meaningful distinction.

Figure 1 shows the phases of a hypothetical SQL query compiler. The query gets optimized—likely first using laws from relational algebra, then using standard PL techniques—then turned into machine code, at which point is executed.

Not many languages are as tied to a mathematical domain as SQL is to relational algebra[1], so they do not have these two distinct phases of optimization. The PingCAP documentation (JoyinQ and Yi [27], Oreoxmt [31]) describes them as logical optimization and physical optimization, respectively.

Contemporary query planners tend to lean heavily on the laws from relational algebra, graph theory, and so on. This is great: it gets the query, as it is written, to the theoretical optimum based on the state of the art in databases (Group [23]). We can, however, push further using techniques from the PL world. Query plans might benefit from canonicalizing using PL techniques before *and* after query planning.

Query plan. A query planner produces a **query plan**. This is not quite an executable form; it is more of a sketch. It is then used in conjunction with the original query to produce an executable program. The most common form of executable program we have seen in database management systems (DBMSes) is some kind of abstract syntax tree (AST) or data-flow graph (DFG). More advanced database management systems further compile this into a bytecode (see Section 3.2) for their evaluator. These different forms of the same program are all called **intermediate representations** and should represent the same abstract idea of the program—be semantically equivalent.

Evaluator. We call this an **interpreter**: a function that receives a program as an input and executes it, producing a value as an output: $f : P \longrightarrow V$.

We call an imaginary function that takes some representation of code and specifies how the code would be executed an *abstract machine*. It's the ideal version of a thing, not the thing itself. If you're

[1]At the time of writing, the only other very prominent ones that come to mind are constraint solving languages, regular expressions, and machine learning (ML)-related languages. Regular expressions, for example, get compiled to a theoretically optimal nondeterministic finite automaton (NFA)/deterministic finite automaton (DFA) for use in parsing. But this parser implementation can still benefit from traditional PL techniques.

reading a book about it, it's probably an abstract machine. Examples include: the x86 hardware architecture, the Java virtual machine (Java virtual machine (JVM)), Python, and C.

An interpreter is a real implementation of an abstract machine. An actual program that takes code in and executes it, often producing data and side-effects. Interpreters can operate on any of the representations in Section 3 (or others!), but generally they do not work on the raw text. Examples include: the Intel or AMD processor you likely have inside your computer, the OpenJDK JVM, CPython, and LLVM.

In database-land, a program is generally a *query* and the interpretation result—the value—is an array of *rows*. There is not a fundamental difference between returning one value and multiple values; the latter can be abstracted as returning a one value: a collection of others. Loop fusion (or, related, stream fusion (Bosboom et al. [9], Coutts et al. [18])) is a common enough optimization that, among other things, can optimize returning multiple values from a function. There are also other classes of optimizations like storage strategies (Bolz et al. [6]) that make working with collections faster.

3 INTERMEDIATE REPRESENTATIONS

Compilers tend to have some additional internal intermediate representations to the ones mentioned above. They are collectively referred to as "intermediate representation (IR)", which is very helpful. IRs in general tend to fall into one of two camps: a high-level IR ("HIR") and a low-level IR ("LIR") (Burke et al. [11]). These so-called levels refer to the average semantic meaning imbued into each instruction in the representation[2]. Lower levels have less meaning per instruction and generally more instructions; they are similar to the instructions that hardware can process[3]. Higher levels are closer to the source language semantics. Higher-level IRs are generally used for optimizing program semantics; lower-level IRs are generally used for optimizing the existing program for the hardware.

The design of a new optimizing compiler for a DBMS should include a domain-specific IR. Compiling to a generic IR like LLVM (not an initialism) is all fine and good, but it's a local maximum, since LLVM was defined for C-like (read: lower level) languages. Encoding the semantics of your high-level source language into your HIR helps with bigger picture optimizations. In this section, we give a brief tour of the various common IR designs, their benefits, and their drawbacks.

3.1 Abstract syntax trees (ASTs)

ASTs are the (often fairly direct) in-memory representation of the syntax of the program. The vertices in these trees represent expressions and the edges represent pointers to other expressions. Figure 2 shows a hypothetical corresponding AST for a small program.

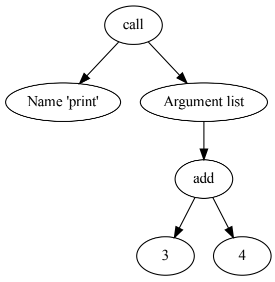
Interpreting ASTs directly tends to be slow; all of the pointer chasing and memory reads and name lookups incur high interpreter overhead. For this reason, interpreter authors tend to *compile* the ASTs to bytecode (Section 3.2) first, then interpret the bytecode. This frequently reduces interpreter overhead, even with the compilation stage.

3.2 Bytecode

Bytecode is a compact binary representation of a program suitable for interpretation. Generally bytecode eschews names and labels for indices and offsets; this strictly numeric form removes some

[2]This is, we think, common knowledge, and not often expressly discussed in the literature.
[3]Incidentally, hardware central processing units (CPUs) are another kind of interpreter. There are many concrete CPUs that, for example, implement the AMD64 abstract machine. And if you look even closer, CPUs are also kind of like compilers. To learn more, search for "x86 microcode" with your favorite search engine.

Fig. 2. An example AST for `print(3+4)`. There is an argument list because in Python, `print` can have any number of arguments. In this case, there is only one, and it is an expression subtree.



interpreter overhead[4]. At the core of DBMSes like SQLite is one very hot loop written in C that reads one bytecode operation at a time and executes it (authors [1]).

These operations are small atomic chunks like `LOAD_CONSTANT` or `BINARY_OR` or `PROJECT`. Listing 1 is an example of a program written in stack-based bytecode. It is a very simple, somewhat contrived program, which pushes two constant numbers onto the stack, adds them together, and then prints the result.

Listing 1. Imaginary stack-based bytecode for a program that adds two numbers and prints them. This could have been compiled from the AST in Figure 2. See how the arguments to the function call are recursively evaluated first, then the call itself. This is visible because they are pushed on the stack first. Compiletion is a post-order traversal.

```
program = [PUSH_CONST, 3,
           PUSH_CONST, 4,
           ADD,
           PRINT]
interpret(program)
# => 7
```

Writing an interpreter (Listing 2) for the small bytecode language is very simple. The basic form is a loop that iterates over the program and executes the instructions one at a time.

The important parts here are the instruction pointer (IP)/program counter (PC)—in this case, `ip`—and the value stack. Typical interpreters also have control-flow instructions like `JUMP_FORWARD` and `JUMP_IF_FALSE` that change the IP. These are used to implement if-statements, loops, and other control-flow.

Listing 2. An interpreter for the stack-based bytecode from Listing 1. It loops over instructions, processing them one at a time. This is structurally very similar to SQLite's internal bytecode loop.

```
def interpret(program):
  ip = 0
  stack = []
  while ip < len(program):
    instr = program[ip]
    ip += 1
    if instr == PUSH_CONST:
```

[4]**Overhead** is the time decoding the program and shuffling data around that is not spent actually fetching data or adding it to the result set. While this is dramatically smaller in bytecode interpreters than in AST interpreters, it still exists. Compiler and interpreter authors try to shrink overhead as much as possible.

```
      arg = program[ip]
      ip += 1
      stack.append(arg)
      continue
    if instr == ADD:
      rhs = stack.pop()
      lhs = stack.pop()
      result = lhs + rhs
      stack.append(result)
      continue
    if instr == PRINT:
      value = stack.pop()
      print(value)
      continue
```

Bytecode is usually generated once and then not modified, because modifying it would require adjusting a bunch of the hard-coded offsets inside the bytecode (jump destinations, etc). Bytecode can be high-level or low-level, but the lower the level of the average opcode, the more time is spent inside the interpreter fetching and decoding the low-level instructions—more overhead.

3.3 Control-flow graphs (CFGs)

Here on out, when we refer to "IR", we will generally mean something different: a CFG modeled with **basic blocks**[5] that contain instructions. Unlike bytecode, which is linear and has hard-coded offsets, this kind of IR is very amenable to modification: it uses structs or classes, pointers, and so on. It is not generally meant to be interpreted directly, but instead optimized and then translated to some lower-level representation.

Listing 3 shows a small Python function and Listing 4 shows an example CFG that could be made from the Python code. Note how in Listing 4 every branch is explicit: there is a `CondBranch` in `bb 0` that goes to either `bb 1` or `bb 2` depending on the truth value of `v3`.

Listing 3. A small Python function that changes its behavior depending on its argument's value. This demonstrates both control flow and variable reassignment.

```
def test(x):
  x = x * 2
  if x:
    return True
  else:
    return False
```

This lets the compiler reason about basic blocks—jump-free regions of code—as entire units. It also greatly simplifies compiler reasoning about, for example, non-local jumps. Imagine a language with a `goto` construct. Looking at a piece of an AST, it would be hard to tell that it is a jump target for that `goto`. But in CFG representation, it is easy: the `goto` is just another block predecessor. It is available inside of the data structure that describes the code.

Listing 4. A hypothetical CFG for the Python program in Listing 3. The syntax and semantics of the IR in the CFG are not so important (it is adapted from the Cinder project (Cinder Team [14])); the important parts are the flattened expressions and the splitting of the function into components with no internal control-flow.

```
fun test {
  bb 0 {
    v0 = LoadArg<0; "x">
    v1 = LoadConst<LongExact[2]>
    v2 = BinaryOp<Multiply> v0 v1
    v0 = Assign v2
```

[5]There are other designs, like basic block arguments and sea of nodes (Click and Paleczny [16]), that are structured differently but are semantically equivalent.

```
    v3 = IsTruthy v0
    CondBranch<1, 2> v3
  }
  bb 1 (preds 0) {
    v4 = LoadConst<Bool[True]>
    Return v4
  }
  bb 2 (preds 0) {
    v5 = LoadConst<Bool[False]>
    Return v5
  }
}
```

3.4 Static single-assignment form (SSA)

Let's look once more at the IR in Listing 4. We see that `v0` maps logically to `x`, `v2` to `x * 2`, and ... hm, `v0` refers now to something new: also `x * 2`. Now any optimization that involves `v0` will have to be *context aware* and that's a huge pain.

Compiler authors used to use many kinds of data structures (def-use chains, etc) to represent code in a way that is amenable to optimization. These days, the field has largely settled on representing this data directly in the IR in a form known as SSA (Braun et al. [10], Cytron et al. [19], Rosen et al. [35]).

Consider: what if every variable was only ever defined exactly once? Or, said slightly differently, what if you could attach names to every expression? Science fiction authors and computer scientists agree: names have power (Engelbart [21], Le Guin [29]).

If one name only ever describes one expression, then we significantly reduce the amount of context any given analysis must carry around. We have more precise data dependencies, can do easier flow typing, etc. Most compiler passes these days are especially written for code in SSA form for this reason.

So let's look at the code from Listing 4 again, but this time in SSA. Turn your eyes to Listing 5.

Listing 5. A hypothetical SSA CFG for the Python program in Listing 3 and CFG in Listing 4. Note that we only have one definition for `v0` and the result of the assign now goes to `v6`.

```
fun test {
  bb 0 {
    v0 = LoadArg<0; "x">
    v1 = LoadConst<LongExact[2]>
    v2 = BinaryOp<Multiply> v0 v1
    v6 = Assign v2
    v3 = IsTruthy v6
    CondBranch<1, 2> v3
  }
  bb 1 (preds 0) {
    v4 = LoadConst<Bool[True]>
    Return v4
  }
  bb 2 (preds 0) {
    v5 = LoadConst<Bool[False]>
    Return v5
  }
}
```

Since names can only ever be defined once, diamond control-flow graphs (CFGs) get trickier. Consider: what are you supposed to do if both branches of an `if` statement assign to a variable? Let's look at an example in Listing 6.

Listing 6. A small Python function that assigns to a variable in two places. SSA must provide a solution for multiple assignment, and it does: ϕ nodes.

```
def test(cond):
  if cond:
    x = 3
  else:
    x = 4
  return x
```

The answer in SSA is a ϕ node (*phi* node). A ϕ is a pseudo-instruction that indicates that a value could be either one of its arguments depending on the control flow path at runtime. The type—the set of potential values—of the output of a ϕ is the *union* of its arguments.

Let's take a look at Listing 7 to see what this looks like.

Listing 7. A SSA CFG for the source code in Listing 6. See the ϕ node in bb 3 which merges the expressions v2 and v3.

```
fun test {
  bb 0 {
    v0 = LoadArg<0; "cond">
    v1 = IsTruthy v0
    CondBranch<1, 2> v1
  }
  bb 1 (preds 0) {
    v2 = LoadConst<LongExact[3]>
    Branch<3>
  }
  bb 2 (preds 0) {
    v3 = LoadConst<LongExact[4]>
    Branch<3>
  }
  bb 3 (preds 1, 2) {
    v4 = Phi v2 v3
    Return v4
  }
```

Placing ϕs is not always obvious, especially if you want to have the fewest number possible—*minimal SSA*. Most compiler passes that produce SSA from some other form of IR try to produce minimal SSA, but littering the code with ϕs is also a totally valid approach. It just might take some more memory to encode the same program and more time in each optimization pass.

The optimizations described in the rest of this paper assume input in SSA form.

4 COMMON OPTIMIZATIONS

Right. Let's finally get down to brass tacks. You've finally dragged your SQL query through six different IRs all the way to SSA form (losing a lot of hair in the process), and you are at long last ready to optimize the heck out of it. This section presents the high-level ideas behind several compiler passes and how they work together to optimize your code.

4.1 Common subexpression elimination (CSE)

It's not always possible or even very readable to make sure that every calculation is only executed as many times as it needs to be. Sometimes code contains redundant computation, whether it is through carelessness, incompetence, compiler complexity (macros, for example), or simplification by other compiler optimizations.

Keeping those computations around would slow the program down, so compilers have a pass called CSE. CSE, also known as *value numbering*[6], identifies semantically equivalent expressions and caches them in local variables.

A very simple example from Maziarz et al. [30] is `(a + (v+7)) * (v+7)`. CSE should be able to hoist `v+7` into its own variable and rewrite the expression to something like `let x = v+7 in (a + x) * x`.

We can also look at an example in databases. We have an example table, Table 1. Someone might reasonably write a query to find the maximum value in the table by writing `select max(value) from numbers;`.

Table 1. A table *numbers* containing a large amount of real numbers. The content isn't important; it's just expensive to query.

value
1110.1
10.1
23.113231
...

The normal plan for this query is to search linearly over the table to find the maximum.

```
sqlite> explain query plan select max(value) from numbers;
QUERY PLAN
`--SEARCH numbers
sqlite>
```

But what happens if we try to retrieve this maximum value twice in the same query? This SQL query gets the maximum value twice and then adds it to itself:

```sql
select x+y
from (select max(value) x from numbers) a,
     (select max(value) y from numbers) b;
```

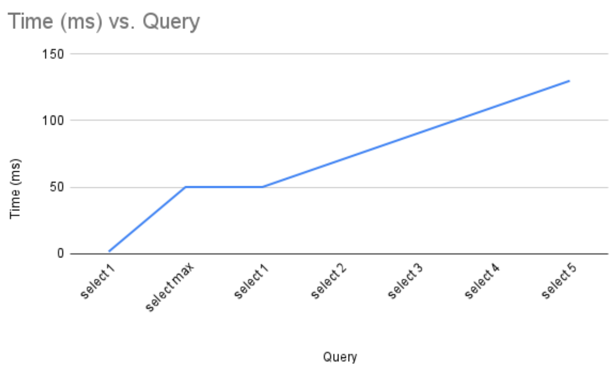
One might naïvely assume that because `max` is a built-in function, the SQL optimizer would perform some kind of CSE, but this is not the case:

```
sqlite> explain query plan select x+y
from (select max(value) x from numbers) a,
     (select max(value) y from numbers) b;
QUERY PLAN
|--MATERIALIZE a
|  `--SEARCH numbers
|--MATERIALIZE b
|  `--SEARCH numbers
|--SCAN a
`--SCAN b
sqlite>
```

The SQLite query plan for this query is to search the table `numbers` twice: once for each subquery. And if you inspect the bytecode with `EXPLAIN` (drop the "QUERY PLAN"), you will see that confirmed with two separate loops over `numbers`.

[6]This is a more extreme form of SSA, if you think about it. One where each subexpression has one unique name.

Fig. 3. A plot of execution time in PostgreSQL as a function of number of identical common subexpressions in a SQL query over a large table of numbers. The time increases linearly with the number of subexpressions.



The same is true for PostgreSQL (Figure 3), where we have plotted execution time as a function of number of common subexpressions (note, *not* number of rows in the table).

This kind of optimization is table stakes for “normal” compilers for functional programming languages: programmers expect the SQL query to be rewritten into something using `WITH`, as in Listing 8.

Listing 8. Software engineers are used to compilers transforming their code to do “the obvious thing” and hoist identical subexpressions into a variable that can be re-used. Here, the `WITH` creates a new temporary table `maximum` to hold the result of the (potentially expensive) maximum value lookup.

```
with maximum(x) as (select max(value) from numbers)
  select x+x from maximum;
```

See how the subquery will only be executed *once*, and then its result used twice in the subsequent `select`.

CSE must not be too eager to merge subexpressions that look identical. For example, two very simple textually identical expressions might yield different results:

```
a = 1
b = a + 2
a = 7
c = a + 2
```

Folding `a` and `c` into the same expression is wrong because `a` no longer means the same thing due to the re-definition. This is one advantage of IR representations like SSA.

Another confounding factor is expressions that might have side effects. For example, the `print` function has an I/O side effect:

```
print("hello")
print("hello")
```

Folding those two would change the externally visible behavior of the program, so CSE must take into account effects.

Even though CSE has some complications, we were not sure why SQLite did not include CSE. We looked into the matter and found a mailing list answer. Dr. Richard Hipp, creator of SQLite, notes (Hipp [24]):

> Our belief is that CSE would be not worth the extra memory, CPU, and code space required to implement it since CSE simply does not come up that often in SQL statements. If, in the future, we find that people begin coding more complex SQL statements which will more often benefit from CSE, then we might revisit this decision.

This is completely fair, and this section does not argue that SQLite *should* include CSE, merely that it *could*.

However, Hipp's argument does not generalize to other kinds of more complex program optimization. Consider the following very common case: a web server that executes more or less the same queries on each page load. The only difference between the queries might be part of the `WHERE` clause (`SELECT * FROM users WHERE users.name == "Wolfgang";`). The rest of the query could be optimized once, stored, and re-used. This treats queries a little more like functions with parameters, and less like whole expressions.

4.2 Loop-invariant code motion (LICM)

LICM is the compilers term for the database world's *selection pushing*. Our technique is a little bit more general, though: it tries to hoist any computation that can be done outside a loop to immediately before the loop.

It's tricky to come up with an example that does not look contrived, because software engineers are trained to avoid this kind of redundant computation manually. A common example, though, is computing the length of an unchanging data structure in the loop condition.

In Listing 9, we see a Python function that adds up the decimal values of each character in a string. It iterates over the string using an integer index `i` until `i` reaches the string length, `len(some_string)`.

Listing 9. A function, sum_chars that adds up the decimal values of every character in a string. Some of the computation is redundant and can be automatically fixed.

```python
def sum_chars(some_string):
  result = 0
  while i < len(some_string):
    result += ord(some_string[i])
    i += 1
  return result
```

It may not be clear, but with every iteration of the loop, the expression `i < len(some_string)` gets evaluated, which calls `len`, which calls a method on the `str` class, and so on. If you inspect the loop manually, it's clear that none of this is necessary: `some_string` is never modified in the loop[7]. The length should be calculated once, and not again.

Remark *Did you notice that the same problem happens in the source code for the bytecode interpreter in Listing 2? The interpreter code itself could benefit from LICM!*

LICM can detect this and move the length calculation before the loop. It normally works on IRs, but the resulting optimized function might look something like Listing 10.

Listing 10. An optimized version of Listing 9 that does not compute the length of the string with every iteration of the loop. It is only computed once, before the loop begins.

```python
def sum_chars(some_string):
  result = 0
```

[7]Incidentally, string objects are immutable in Python, so the length will never change. What could happen, though, is that someone might redefine the local variable to point to some other object. If you are also going to nitpick about Python semantics, yes, the variable `some_string` could technically be any type and therefore the length function could have a side effect. But we're going to assume that it is a string for the purposes of this demonstration.

```
  string_length = len(some_string)
  while i < string_length:
    result += ord(some_string[i])
    i += 1
  return result
```

Note that the comparison still needs to happen with every loop iteration, because `i` changes over time.

A more relatable SQL-esque example is left as an exercise for the reader. What could be optimized in this rendering of the conceptual evaluation order (Listing 11)?

Listing 11. A Python-syntax function that joins two relations. Something about it can be optimized using LICM.

```
def filter_join(A, B):
  result = []
  for a in A:
    for b in B:
      if a < 10 and b < 20:
        result.append((a, b))
  return result
```

To learn more, see Jing et al. [26], an excellent blog post from CS 6120, the Cornell compilers course blog.

4.3 Constant propagation

Let's look at a couple of SQL snippets and the bytecode that SQLite produces for the query. For this section, we will use the Chinook database (Unknown [41]), which primarily contains metadata about music.

In Listing 12, we have a SQL query that iterates over the `albums` table and collects the results. We have used `explain` to make SQLite produce its internal bytecode representation of the query for us.

In this bytecode, we can see that we will open the table, loop over it, and then halt. Pretty straightforward and expected.

Listing 12. The SQLite bytecode compiler has compiled the query into a simple loop over the albums table. This is expected, and the optimal solution.

```
sqlite> explain select * from albums;
addr  opcode         p1    p2    p3    p4             p5  comment
----  -------------  ----  ----  ----  -------------  --  -------------
0     Init           0     9     0                    0   Start at 9
1     OpenRead       0     2     0     3              0   root=2 iDb=0; albums
2     Rewind         0     8     0                    0
3       Rowid          0     1     0                    0   r[1]=rowid
4       Column         0     1     2                    0   r[2]=albums.Title
5       Column         0     2     3                    0   r[3]=albums.ArtistId
6       ResultRow      1     3     0                    0   output=r[1..3]
7     Next           0     3     0                    1
8     Halt           0     0     0                    0
9     Transaction    0     0     35    0              1   usesStmtJournal=0
10    Goto           0     1     0                    0
sqlite>
```

Let us compare to Listing 13, where we have a SQLite query that collects the results, filtering on an always false condition. In this case, the optimizer has (correctly) detected that this query will never return anything, and injected a `Goto` opcode (addr 1) to finish early.

Listing 13. The SQLite bytecode compiler has injected a `Goto` to skip the loop because the filtering condition is always false (0). This is expected, and nearly optimal.

```
sqlite> explain select * from albums where 0;
addr  opcode         p1    p2    p3    p4             p5  comment
----  -------------  ----  ----  ----  -------------  --  -------------
```

```
0     Init            0     10    0                0   Start at 10
1     Goto            0     9     0                0
2     OpenRead        0     2     0     3          0   root=2 iDb=0; albums
3     Rewind          0     9     0                0
4       Rowid           0     1     0                0   r[1]=rowid
5       Column          0     1     2                0   r[2]=albums.Title
6       Column          0     2     3                0   r[3]=albums.ArtistId
7       ResultRow       1     3     0                0   output=r[1..3]
8     Next            0     4     0                1
9     Halt            0     0     0                0
10    Transaction     0     0     35    0          1   usesStmtJournal=0
11    Goto            0     1     0                0
sqlite>
```

There is apparently no dead code elimination (DCE) pass, since the rest of the loop body is left around, but that's not very important here. The important thing is that the flow of control will always short-circuit the loop.

Let's push the complexity a little further. In Listing 14, we have a query, except that this time, the filtering condition is `0 != 0`, not just the literal `0`. Inside the bytecode, we no longer get a `Goto`. Instead, we have an `Eq` that will check *once* at run-time if the condition is true, and if so, skip the loop.

Listing 14. The SQLite bytecode compiler has injected a check to short-circuit the loop if at runtime, zero is not equal to zero (addr 1: Eq). For some reason unknown to the authors, the optimizer can't eliminate that check at compile time.

```
sqlite> explain select * from albums where 0!=0;
addr  opcode         p1    p2    p3    p4             p5  comment
----  -------------  ----  ----  ----  -------------  --  -------------
0     Init            0     10    0                0   Start at 10
1     Eq              1     9     1                80  if r[1]==r[1] goto 9
2     OpenRead        0     2     0     3          0   root=2 iDb=0; albums
3     Rewind          0     9     0                0
4       Rowid           0     2     0                0   r[2]=rowid
5       Column          0     1     3                0   r[3]=albums.Title
6       Column          0     2     4                0   r[4]=albums.ArtistId
7       ResultRow       2     3     0                0   output=r[2..4]
8     Next            0     4     0                1
9     Halt            0     0     0                0
10    Transaction     0     0     35    0          1   usesStmtJournal=0
11    Integer         0     1     0                0   r[1]=0
12    Goto            0     1     0                0
sqlite>
```

This is starting to look a little bit like LICM, actually, since SQLite was smart enough to hoist a condition that does not depend on any one row outside of the loop.

We can confirm that sqlite is just not doing constant folding or propagation by isolating the `0!=0` (Listing 15).

Listing 15. The SQLite optimizer does not constant fold, apparently. See the Ne instruction that compares the two values at runtime.

```
sqlite> explain select 0 != 0;
addr  opcode         p1    p2    p3    p4             p5  comment
----  -------------  ----  ----  ----  -------------  --  -------------
0     Init            0     6     0                0   Start at 6
1     Integer         1     1     0                0   r[1]=1
2     Ne              2     4     2                64  if r[2]!=r[2] goto 4
3     ZeroOrNull      2     1     2                0   r[1] = 0 OR NULL
4     ResultRow       1     1     0                0   output=r[1]
5     Halt            0     0     0                0
6     Integer         0     2     0                0   r[2]=0
7     Goto            0     1     0                0
sqlite>
```

Constant propagation (more precisely, constant folding) would be able to turn that `Eq`/`Ne` into a `Goto`, making the queries in Listing 14 and Listing 13 produce equivalent bytecode.

For further reading, check out specific constant propagation algorithms, like Sparse Conditional Constant Propagation (Wegman and Zadeck [43]) and Cliff Click's PhD thesis (Click and Cooper [15]). These generally require monotone analyses with fixpoints and data structures like lattices.

4.4 Inlining

Inlining has been described as "one of the most important compiler optimizations". It is the process of copying the body of a procedure into the body of that procedure's caller. This has two main advantages:

- Reducing function call overhead
- Expanding the scope of the optimizer

While function calls can be expensive—and therefore good to eliminate—the latter is the bigger benefit. Expanding the view of the optimizer can yield better results on all other optimizations.

Consider for example Listing 16. Written as-is, the bytecode for the query is a big loop over the `albums` table. Even though you as a reader know that clearly `select 0` and `0` are equivalent, the SQLite query optimizer does not. It looks at one query at a time, and the subquery is not in its metaphorical "field of view" when optimizing the outer query.

If the inner query were to be inlined into the outer query, the optimizer could reason about them as a unit and identify that `select 0` is indeed always equal to `0`.

Listing 16. The SQLite optimizer only has visibility into one compilation unit (subquery) at a time. Since it can't know anything about what the opaque subquery returns, it must iterate over the `albums` table. This is a current limitation to SQLite, not an inherent constraint, and could be improved.

```
sqlite> explain select * from albums where (select 0)!=0;
addr  opcode         p1    p2    p3    p4             p5  comment
----  -------------  ----  ----  ----  -------------  --  -------------
0     Init           0     17    0                    0   Start at 17
1     OpenRead       0     2     0     3              0   root=2 iDb=0; albums
2     Rewind         0     16    0                    0
3       Integer        9     2     0                    0   r[2]=9; return address
4       Once           0     9     0                    0
... ... ... ... ... ... ...
13      Column         0     2     8                    0   r[8]=albums.ArtistId
14      ResultRow      6     3     0                    0   output=r[6..8]
15    Next           0     3     0                    1
16    Halt           0     0     0                    0
17    Transaction    0     0     35    0              1   usesStmtJournal=0
18    Integer        0     5     0                    0   r[5]=0
19    Goto           0     1     0                    0
sqlite>
```

While the code in Listing 16 may look contrived—hopefully nobody writes code like this—consider that this could be an intermediate program in a pipeline of optimizations. Perhaps `select 0` is leftover from an earlier optimization on a query that returned the size of the table. Inlining that `select 0` would have significant performance impact on the outer query. An ideal end result would look similar to Listing 13, but Listing 14 would also be acceptable.

Inlining isn't all peaches and cream, though. The primary drawback is the main feature: the compiler copies code. This increases code size, which can be a problem for larger programs.

It is also tricky to decide which function calls to inline; compilers use compile-time ("static") and run-time ("dynamic") information to make decisions about inlining. While optimizers can generally use some context clues and manual programmer annotations (the `inline` keyword in C, for example), it is not obvious what the best heuristic is. Also, inlining the wrong functions could cause a performance *slowdown*. Inlining heuristics are an active area of research. We suspect optimal inlining is similar to a graph contraction problem.

We also see this in Listing 17 (foreshadowing!), where without inlining, we never would have been able to elide the allocation. For more reading on inlining in the Cinder just-in-time (JIT) Python compiler, see Bernstein [2].

4.5 Code/allocation sinking

Allocation sinking (Pall [32]), also known as allocation removal (Bolz et al. [5]), escape analysis (Blanchet [4], Choi et al. [13], Deutsch [20], Gay and Steensgaard [22], Kotzmann and Mössenböck [28], Park and Goldberg [33], Vivien and Rinard [42], and more!), and sometimes points-to analysis (Steensgaard [38]), is the process of moving, changing, or eliminating code that creates and writes to objects with short lifespans. It has several main benefits:

- Reducing allocations
- Reducing memory traffic
- Helping constant propagation
- Reducing code size

It is one of those optimizations that, like inlining, make everything better. To illustrate, we will look at a small Python-esque program[8]. It contains a class and a free function that operates on an instance of the class.

Listing 17. A small Python program that contains a a Point class, a Euclidean distance function on that Point class, and somewhat contrived function to calculate a Point's distance from the origin. `distance_from_origin` is a great candidate for optimization.

```
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y
  def distance(self, other):
    return math.sqrt((self.x-other.x)**2 + (self.y-other.y)**2)
def distance_from_origin(p: Point):
  return Point(0, 0).distance(p)
```

Though it may not look like it, `distance_from_origin` is actually quite expensive for what it does. Though it's not asymptotically a problem, little expenses add up. One might imagine a ray tracer, for example, has to do similar calculations for *every pixel* for *every frame* of an animation.

While math can be slow, the primary expense comes from allocation and memory traffic. It is important to note that unlike languages like C and C++, where creating an instance of a class can happen on the stack *or* the heap, managed languages such as Python and Java **only** have heap-allocated objects.

This means that not only is the creation of the origin point (`Point(0,0)`) a heap allocation, but so is *every* immediate integral or floating point value in the distance computation.

While these objects will not end up contributing to the so-called "high watermark" memory usage (they do not live long enough), heap allocations and memory traffic are still expensive.

Implementors of the JVM, LuaJIT, and PyPy runtimes have worked for years to invent and implement algorithms to reduce as much allocation as possible. The mechanics can be slightly complicated, so we will not discuss them here, but you can read more in Bolz-Tereick [8]. We will, however, demonstrate in Listing 18 a step-by-step visualization of the `distance_from_origin` function as it is transformed by a hypothetical compiler with an allocation removal pass.

Listing 18. The definition of `distance_from_origin` is transformed over time by multiple sequential compiler passes. Each is described in a comment above the function definition.

[8] We're not expressly trying to stick to full Python semantics, just close enough.

```
# The original function
def distance_from_origin(p: Point):
  return Point(0, 0).distance(p)
# Inline the construction of Point
def distance_from_origin(p: Point):
  v0 = Alloc<Point>
  v0.x = 0
  v0.y = 0
  v1 = v0.distance(p)
  return v1
# Inline the definition of Point.distance
def distance_from_origin(p: Point):
  v0 = Alloc<Point>
  v0.x = 0
  v0.y = 0
  v1 = math.sqrt((v0.x-p.x)**2 + (v0.y-p.y)**2)
  return v1
# Forward the stores to and loads from the origin Point
def distance_from_origin(p: Point):
  v0 = Alloc<Point>
  v0.x = 0
  v0.y = 0
  # Notice that v0.x and v0.y have been transformed into 0.
  v1 = math.sqrt((0-p.x)**2 + (0-p.y)**2)
  return v1
```

The load and store forwarding are an important part of the optimization pass. They help constant propagation, strength reduction, and reduce (or eliminate) dependencies on the allocated object. For more complex examples that involve constructing and then destruction collections of objects, they may even help with flow-based type analysis.

Next, we will see how other optimization passes like DCE can further optimize this function.

4.6 Dead code elimination (DCE)

Allocations are not the only code that should be deleted. In general, *any* code that need not or will not be executed should be deleted. This generalization is called DCE.

Dead code is not always visible to the programmer. For example, it might come as the result of platform specialization (Listing 19), or as the result of some other compiler pass (Listing 18), or something else entirely.

Listing 19. Low-level code often contains different paths for different operating systems. Some of this can be completely deleted if the operating system is known at compile-time.

```
import sys
def root_directory():
  if sys.platform == "linux":
    return "/"
  elif sys.platform == "windows":
    return "C:\\"
  else:
    raise Exception("Unknown operating system!")
```

If you were reading the section on allocation sinking closely, you might have noticed that in the last example snippet, the allocation for `v0` was completely unused after it was created. The `Point` is not obviously dead from the original Python code as it was written, but compiler transformations have optimized the IR in the background.

Listing 20 shows the conclusion in the exciting optimization series of this Euclidean geometry function. Since the allocation and the writes to said allocation are completely unused, they can be removed.

Listing 20. With DCE we can also completely eliminate the allocation of the Point and the writes to it, since it is unused in the rest of the function. This has a first-order performance impact inside this function and second-order impact in the garbage collector and micro-architectural caches.

```
# Completely eliminate the allocation of v0, since it is not used
def distance_from_origin(p: Point):
  v1 = math.sqrt((0-p.x)**2 + (0-p.y)**2)
  return v1
```

Dead code elimination (DCE) is a cleverer optimization than it first appears. We mentioned that `v0` was "completely unused", but this is not strictly true. The allocation *is* actually used—the `x` and `y` attributes get written to. So how does DCE eliminate it?

Well, it's similar to garbage collection (GC). Yes, that's right, the compile-time elimination of code is pretty similar to the run-time deallocation of objects.

The algorithm in Listing 21 starts from the required instructions that build the skeleton of the function: the terminators of the basic blocks, the phi instructions, and any instruction that might have a side effect (maybe a function call, or some I/O)—this is equivalent to root finding in GC. Then it does a graph traversal of all of the instructions needed to provide those skeleton instructions with operands—this is called marking in GC. Then it deletes the rest—sweeping.

So even though we write to the attributes `x` and `y` of `v0`, none of the allocations are either useful or needed by a useful instruction. So the whole bunch gets deleted.

Listing 21. A Python-esque sketch of the DCE algorithm. It is adapted from the Cinder JIT for Python (Cinder Team [14]).

```
def is_useful(instr):
  return instr.is_terminator() or instr.is_phi() or
     instr.has_side_effect()
def eliminate_dead_code(func):
  # Find the roots in the graph
  worklist: Queue[Instr] = Queue()
  for block in func.cfg.blocks:
    for instr in block:
      if is_useful(instr):
        worklist.push(instr)
  # Traverse the dependencies of the roots
  live_set: Set[Instr] = set()
  while not worklist.empty():
    live_instr = worklist.pop()
    if live_instr in live_set:
      continue
    live_set.add(live_instr)
    for operand in live_instr:
      if operand.defining_instr() not in live_set:
        worklist.push(operand.defining_instr())
  # Delete the unnecessary code
  for block in func.cfg.blocks:
    for instr in block:
      if instr not in live_set:
        instr.remove()
```

Like garbage collection (GC), dead code elimination (DCE) can be tricky. Bugs in the `is_useful` function might end up removing too much code, or maybe the dataflow dependencies in the graph traversal track too little. Things like that. But it is a very powerful compiler optimization that can remove whole subgraphs of your CFG that need not be there.

Fun fact *In 2022, Theodoridis et al. [40] wrote a paper about using DCE to find missed opportunities in compiler optimization.*

5 SUGGESTIONS FOR DATABASE COMPILERS

Many (most, perhaps) databases and query optimizers treat queries as one-off programs to be executed with the lowest latency possible. For one-off queries, this is absolutely the right model; reducing end-to-end latency for a single query is the path to analyst happiness. But this is not the only usage pattern.

There are other ways to use a database that might want databases to spend more time in the query optimizer before caching the result and executing:

- Long-running batch jobs (such as extract, transform, load (ETL) jobs)
- Applications with high read traffic (such as web applications)
- Applications with limited resources (such as mobile applications or applications on embedded devices)
- Applications with infrequent deploys and static queries (such as enterprise applications)

These kinds of applications might want to either compile queries ahead of time, or at least JIT them with an optimizing compiler. There are some challenges with this, for which we present suggested solutions.

How do we indicate to the database that a query should be optimized? Define the query as a stored procedure. That is a good indication that it will be called often, and also indicates what parts of the query will vary over time: the parameters.

If stored procedures are not an option, which queries should be optimized? The database can keep track of what queries (or groups of queries, if the queries are parameterized by some value) are executed the most. This is a very common feature in state-of-the-art JIT compilers.

What if the query should be optimized differently based on its parameters? This is handled reasonably neatly by both inlining and polymorphic specialization. There is also an active research interest in lazy basic block versioning (Chevalier-Boisvert and Feeley [12]) for dynamic languages.

What if the compiler speculated wrong when it first optimized the query? Deoptimize and recompile it. At some point it may be worth completely bailing out and not compiling the function anymore so that you have some amount of performance convergence. In this case, we recommend loudly reporting this to the engineer for inspection.

How do I ensure my complicated compiler is correct? You can write and check formal proofs of correctness for your compiler passes, but this is extraordinarily time consuming and does not give you everything. People tend to write loads of tests for compiler IR transformations and also maintain end-to-end test suites for overall program behavior across all optimization levels and passes. Extensive test suites can be equivalent to program specifications and be used to compare implementation for defects. For example, when we were developing the Skybison Python runtime (Skybison Team [37]), our comprehensive test suite found several bugs in the extremely widely deployed but less tested reference implementation, CPython.

6 FURTHER READING

There are too many areas of compiler research to list here in this paper. We will list other topics with extreme brevity here for interested readers.

- Loop fusion and stream optimization (Bosboom et al. [9], Coutts et al. [18]), which can be handy in reducing or eliminating intermediate results between loops
- E-graphs and equality saturation (Tate et al. [39], Willsey et al. [44]), which help eliminate pass-ordering problems and allow more flexible constraint-based code generation

- Partial evaluation, which can be used to specialize any chunk of code for a given value
- Other CFG representations such as regionalized value state dependence graphs (RVSDGs) (Reissmann et al. [34]), which track more information than normal SSA does in the IR itself
- Abstract interpretation (Cousot and Cousot [17]), which is the general idea of evaluating code to get some other result than a value (for example: a type, some kind of safety check, etc)
- Determinacy and counterfactual execution (Schäfer et al. [36]), which can be used to augment DCE or strength reduce very dynamic language constructs like `eval`
- Vectorization, which can take advantage of the hardware's ability to do multiple math operations at once (Hofer [25])
- *Making an Embedded DBMS JIT-friendly* (Bolz et al. [7]) is a superb read after learning the basics from this paper and is a tour of effective SQLite optimization using a JIT compiler

We hope you use this as a springboard for more learning. For more links and writing, check out our compilation of compilers and programming languages resources (Bernstein [3]).

7 COMPLEXITY

Adding optimization passes to a compiler is not all roses. There are some thorns, too. Many compiler engineers write and deploy compiler passes without rigorous formal proofs of correctness, and instead ship new software tests. This is the industry standard. As such, there exist bugs in all compilers, no matter how well tested they are and how well used they are. Heck, even if they did write proofs, proofs only go so far.

For example, Alexis King, prominent functional programmer and compiler engineer, recently tweeted about a SQL optimizer bug in SQLite. We have reproduced the SQL code in the tweet here in Listing 22.

Listing 22. Two SQL queries that should be equivalent. The first one returns two equal numbers, as expected. The second one triggers a bug in the optimizer that causes two entirely different numbers to be returned.

```
sqlite> select value, value from (select random() as value);
-6485496119127326647|-6485496119127326647
sqlite> select value, value from (select random() as value from (select 1));
-3291086315575408646|-3264740897758797789
sqlite>
```

Not all bugs are correctness bugs, either. Some bugs, such as pass ordering bugs, are silent because they only result in subpar code generation. Occasionally a compiler pass will have an implicit dependency on another because it expects to optimize away a certain pattern that another generates. For example, consider two potential optimizations of calling `type(x)` (which returns the type of `x`, whatever it is, at run-time) in Python.

If the type of `x` is known at compile time, it is possible to replace the call `type(x)` with the type itself (say, `list`). If it is not known, however, one can still optimize the code by replacing `type(x)` with a memory load of the `PyObject.ob_type` field. Seems simple enough, but sometimes the type of `x` may become known somewhere in the middle of the compiler optimization pipeline. If the compiler has already done the second transformation, we've lost some information and can no longer do the first transformation.

This kind of problem shows there is no total order on compiler passes. It's tricky business and there is no well-recognized "right answer".

Project maintainers must decide what their level of risk tolerance is when introducing additional complexity into their projects. With every new compiler, *especially* with JIT native code compilers, come bugs—some more severe than others.

High-risk projects like web browsers (which run untrusted code from the internet) have occasionally decided to completely forgo native code generation in their JIT compilers. This has a performance penalty, but the maintainers have deemed that penalty acceptable compared to the security risk.

All of this is to say that compilers are not “free money”. Like any other software, they have bugs, and these bugs can have weird and hidden second-order effects.

REFERENCES

[1] SQLite authors. 2022. The SQLite Bytecode Engine. https://www.sqlite.org/opcode.html.

[2] Max Bernstein. 2022. How the Cinder JIT’s function inliner helps us optimize Instagram. https://engineering.fb.com/2022/05/02/open-source/cinder-jits-instagram/

[3] Max Bernstein. 2023. Programming languages resources. https://bernsteinbear.com/pl-resources/

[4] Bruno Blanchet. 1998. Escape Analysis: Correctness Proof, Implementation and Experimental Results. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL ’98)*. Association for Computing Machinery, New York, NY, USA, 25–37. https://doi.org/10.1145/268946.268949

[5] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. 2011. Allocation Removal by Partial Evaluation in a Tracing JIT. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Austin, Texas, USA) *(PEPM ’11)*. Association for Computing Machinery, New York, NY, USA, 43–52. https://doi.org/10.1145/1929501.1929508

[6] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. 2013. Storage Strategies for Collections in Dynamically Typed Languages. *SIGPLAN Not.* 48, 10 (oct 2013), 167–182. https://doi.org/10.1145/2544173.2509531

[7] Carl Friedrich Bolz, Darya Kurilova, and Laurence Tratt. 2015. Making an Embedded DBMS JIT-friendly. *CoRR* abs/1512.03207 (2015). arXiv:1512.03207 http://arxiv.org/abs/1512.03207

[8] Carl-Friedrich Bolz-Tereick. 2022. Allocation Removal in the Toy Optimizer. https://www.pypy.org/posts/2022/10/toy-optimizer-allocation-removal.html

[9] Jeffrey Bosboom, Sumanaruban Rajadurai, Weng-Fai Wong, and Saman Amarasinghe. 2014. StreamJIT: A Commensal Compiler for High-Performance Stream Programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA ’14)*. Association for Computing Machinery, New York, NY, USA, 177–195. https://doi.org/10.1145/2660193.2660236

[10] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. 2013. Simple and Efficient Construction of Static Single Assignment Form. In *Proceedings of the 22nd International Conference on Compiler Construction* (Rome, Italy) *(CC’13)*. Springer-Verlag, Berlin, Heidelberg, 102–122. https://doi.org/10.1007/978-3-642-37051-9_6

[11] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. 1999. The Jalapeño Dynamic Optimizing Compiler for Java. In *Proceedings of the ACM 1999 Conference on Java Grande* (San Francisco, California, USA) *(JAVA ’99)*. Association for Computing Machinery, New York, NY, USA, 129–141. https://doi.org/10.1145/304065.304113

[12] Maxime Chevalier-Boisvert and Marc Feeley. 2014. Simple and Effective Type Check Removal through Lazy Basic Block Versioning. *CoRR* abs/1411.0352 (2014). arXiv:1411.0352 http://arxiv.org/abs/1411.0352

[13] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape Analysis for Java. *SIGPLAN Not.* 34, 10 (oct 1999), 1–19. https://doi.org/10.1145/320385.320386

[14] Cinder Team. 2020. Cinder. https://github.com/facebookincubator/cinder.

[15] Cliff Click and Keith D. Cooper. 1995. Combining Analyses, Combining Optimizations. *ACM Trans. Program. Lang. Syst.* 17, 2 (mar 1995), 181–196. https://doi.org/10.1145/201059.201061

[16] Cliff Click and Michael Paleczny. 1995. A Simple Graph-Based Intermediate Representation. *SIGPLAN Not.* 30, 3 (mar 1995), 35–49. https://doi.org/10.1145/202530.202534

[17] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL ’77)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

[18] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream Fusion: From Lists to Streams to Nothing at All. *SIGPLAN Not.* 42, 9 (oct 2007), 315–326. https://doi.org/10.1145/1291220.1291199

[19] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (oct 1991), 451–490. https://doi.org/10.1145/115372.115320

[20] Alain Deutsch. 1997. On the Complexity of Escape Analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France) *(POPL '97)*. Association for Computing Machinery, New York, NY, USA, 358–371. https://doi.org/10.1145/263699.263750

[21] Douglas C. Engelbart. 1990. Knowledge-Domain Interoperability and an Open Hyperdocument System. In *Proceedings of the 1990 ACM Conference on Computer-Supported Cooperative Work* (Los Angeles, California, USA) *(CSCW '90)*. Association for Computing Machinery, New York, NY, USA, 143–156. https://doi.org/10.1145/99332.99351

[22] David Gay and Bjarne Steensgaard. 2000. Fast Escape Analysis and Stack Allocation for Object-Based Programs. In *Proceedings of the 9th International Conference on Compiler Construction (CC '00)*. Springer-Verlag, Berlin, Heidelberg, 82–93.

[23] The PostgreSQL Global Development Group. 1996. PostgreSQL: Documentation: 15: 52.5. Planner/Optimizer. https://www.postgresql.org/docs/current/planner-optimizer.html.

[24] Richard Hipp. 2012. Common subexpression optimization of deterministic functions. https://sqlite-users.sqlite.narkive.com/1pJDsmOS/sqlite-common-subexpression-optimization-of-deterministic-functions#post8

[25] Phil Hofer. 2022. Building a SQL VM in AVX-512 Assembly. https://sneller.io/blog/2023/03/22/sql-vm-in-avx-512/

[26] Yi Jing, Zhijing Li, and Neil Adit. 2019. Loop Invariant Code Motion and Loop Reduction for Bril. https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/loop-reduction/

[27] JoyinQ and Keke Yi. 2020. SQL Logical Optimization. https://docs.pingcap.com/tidb/stable/sql-logical-optimization

[28] Thomas Kotzmann and Hanspeter Mössenböck. 2005. Escape Analysis in the Context of Dynamic Compilation and Deoptimization. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments* (Chicago, IL, USA) *(VEE '05)*. Association for Computing Machinery, New York, NY, USA, 111–120. https://doi.org/10.1145/1064979.1064996

[29] Ursula K. Le Guin. 2012. *A Wizard of Earthsea*. HarperCollins. https://books.google.com/books?id=hDtjOj5FL8MC

[30] Krzysztof Maziarz, Tom Ellis, Alan Lawrence, Andrew Fitzgibbon, and Simon Peyton Jones. 2021. Hashing modulo Alpha-Equivalence. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 960–973. https://doi.org/10.1145/3453483.3454088

[31] Oreoxmt. 2022. SQL physical optimization. https://docs.pingcap.com/tidb/dev/sql-physical-optimization

[32] Mike Pall. 2012. Allocation sinking in git HEAD. https://www.freelists.org/post/luajit/Allocation-sinking-in-git-HEAD

[33] Young Gil Park and Benjamin Goldberg. 1992. Escape Analysis on Lists. *SIGPLAN Not.* 27, 7 (jul 1992), 116–127. https://doi.org/10.1145/143103.143125

[34] Nico Reissmann, Jan Christian Meyer, Helge Bahmann, and Magnus Själander. 2020. RVSDG: An Intermediate Representation for Optimizing Compilers. *ACM Trans. Embed. Comput. Syst.* 19, 6, Article 49 (dec 2020), 28 pages. https://doi.org/10.1145/3391902

[35] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 12–27. https://doi.org/10.1145/73560.73562

[36] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Dynamic Determinacy Analysis. *SIGPLAN Not.* 48, 6 (jun 2013), 165–174. https://doi.org/10.1145/2499370.2462168

[37] Skybison Team. 2020. Skybison. https://github.com/facebookexperimental/skybison.

[38] Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 32–41. https://doi.org/10.1145/237721.237727

[39] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. *SIGPLAN Not.* 44, 1 (jan 2009), 264–276. https://doi.org/10.1145/1594834.1480915

[40] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding Missed Optimizations through the Lens of Dead Code Elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 697–709. https://doi.org/10.1145/3503222.3507764

[41] Unknown. 2022. SQLite Sample Database. https://www.sqlitetutorial.net/sqlite-sample-database/

[42] Frédéric Vivien and Martin Rinard. 2001. Incrementalized Pointer and Escape Analysis. *SIGPLAN Not.* 36, 5 (may 2001), 35–46. https://doi.org/10.1145/381694.378804

[43] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (apr 1991), 181–210. https://doi.org/10.1145/103135.103136

[44] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434304

Received 01 March 2023; revised 20 April 2023; accepted 24 April 2023