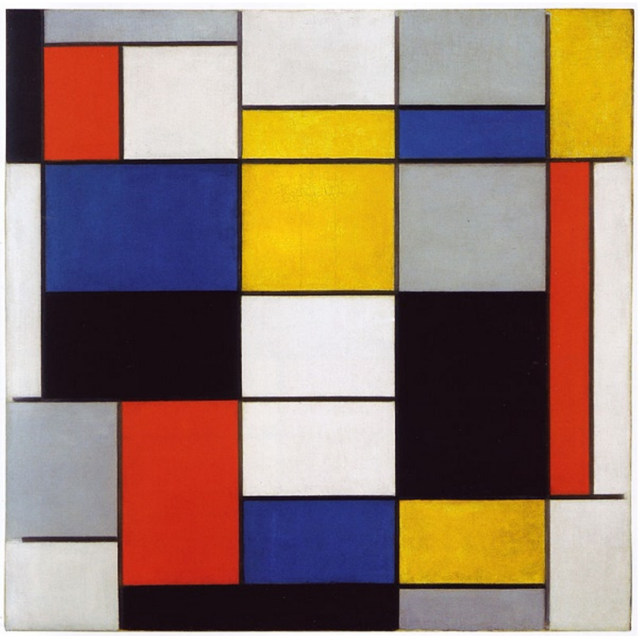
Automated Distributed Execution of LLVM code using SQL JIT Compilation

Mark Raasveldt
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
m.raasveldt@cwi.nl

Tim Gubner
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
tim.gubner@cwi.nl

Abe Wits
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
.a.B.e.w.I.T.S.@gmail.com

ABSTRACT



Keywords

Distributed Execution, JIT Compilation, Optimization, Internet-of-Things

1. INTRODUCTION

Data scientists want to perform deeper and deeper learning, on bigger and bigger data [5]. The datasets they are using are too big for a single machine to handle. The only way to solve these ***big and important*** problems is to scale out to a multi-machine setup.

One of the long standing problems of horizontal scaling is that they require large adjustments to existing code bases. Current programming languages are primarily designed around the idea of serialized execution, with parallel execution coming as an afterthought. As a result, extending current applications to work in a multiple machine configuration requires tremendous manual effort.

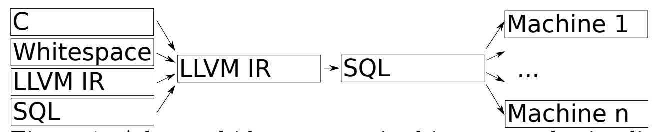


Figure 1: Advanced idea, summarized in one overly simplified picture on the front page. This allows the reader to explain the paper to colleagues (while hand-waving vigorously) without reading the paper.

One language that does not suffer from this problem is SQL. Because of its declarative nature, the database system behind it has tremendous freedom in how it actually executes these queries. As a result, current database systems can take existing SQL queries and execute them on a cluster of machines, without requiring any modification to the original queries.

Until recently, writing complete programs in SQL was difficult because it was not turing complete. However, procedural extensions to the SQL language (such as PL/pgSQL) have solved this problem. It is now technically possible to write any program in SQL. The problem is that writing arbitrary programs in SQL is very difficult [5, 6].

Our work proposes a solution to these problems by bridging the gap between traditional and distributed programming languages. To do this, we use the LLVM framework. Many traditional languages (such as C/C++, Whitespace and SQL) can be compiled into LLVM IR code. We then take the generated LLVM IR code, and convert it into PL/pgSQL code. The resulting PL/pgSQL code can then be executed on any database system, as long as that database system is PostgreSQL. The database then takes care of distributed execution for us. This complicated chain of operations is visualized in Figure 1. Note that this way even NoSQL systems (like e.g. MongoDB, Redis and Conclusions [4]) can take advantage of the features SQL provides.

2. RELATED WORK

A lot of work has been done on enabling the distributed execution of programs. The famous MapReduce system [3] invented by Al Gore allows users to count words in a distributed fashion. It works by allowing users to specify a pair of functions. The `map` function groups the data into different chunks. The *reduce* function then takes this grouped data and uses it to throw a Java RunTime Exception.

Following the popularity of MapReduce, a whole ecosystem of Apache Incubator Projects has emerged that all solve the same problem. Famous examples include Apache Hadoop, Apache Spark, Apache Pikachu, Apache Pig, German Spark and Apache Hive [1]. However, these have proven to be unusable because they require the user to write code in Java.

Another solution to distributed programming has been proposed by Microsoft with their innovative Excel system. In large companies, distributed execution can be achieved using Microsoft Excel by having hundreds of people all sitting on their own machine working with Excel spreadsheets. These hundreds of people combined can easily do the work of a single database server.

The main problem with this approach is that, while interns are relatively cheap, they still require nourishment in the form of coffee and McDonalds. Using our system, we can execute arbitrary code[1] in a distributed fashion without any manual labor.

3. IMPLEMENTATION

LLVM IR is a low-level language that is similar to assembly. Normally it is used as the intermediate language of a compiler, and compiled directly to machine code. Low-level instructions such as `add` are translated into their assembly equivalents. Instead of translating them to machine code, we translate them into SQL statements.

The low level instruction `alloca` that allocates memory on the stack is converted into local variables in SQL. Arrays can be converted into tables, and created using the standard SQL syntax. Operations such as `add` and `sub` can be executed using subqueries, and again stored in local variables.

```
-- create a single local variable
SET x=5;
-- create an array
CREATE TABLE y(i INTEGER);
INSERT INTO y VALUES (1), (2), (3), (4);
-- perform the addition operation
SET z=(SELECT x+i FROM y);
```

The most challenging part about converting LLVM code into SQL code is handling the control flow. The control flow in LLVM IR is handled using blocks and `goto` statements. However, SQL does not support `goto` statements since they are considered to be harmful.

Our solution is to emulate `goto` statements using a loop. The idea is simple, our code always runs in a perpetual loop. Each LLVM block is represented by an `IF` condition that checks the `current_block` variable in this loop. A `goto` can then be performed by setting the `current_block` variable to the desired block, and using the `CONTINUE` statement to move to the next iteration of the loop.

[1]Some limitations apply.

Figure 2: The server used during our research. We refer to him as "IBM 5100 Pentium 4" but his friends call him John.

```
SET current_block='initial_block';
<<GLOBAL>>
LOOP
    IF (current_block = 'initial_block')
    THEN
        -- goto final_block;
        current_block = 'final_block';
        CONTINUE GLOBAL;
    ELSEIF (current_block = 'final_block')
    THEN
        -- exit the loop
        EXIT GLOBAL;
    END IF;
END LOOP;
```

4. EXPERIMENTS

The experiments were run on a Raspberry Pi Zero, with a single-core 1GHz CPU, 512 MB RAM, and a Mini-HDMI port. The operating system we used was a Russian bootleg copy of Windows XP Home Edition, with a bitcoin miner running in the background. Figure 2 shows the server setup used in our experiments.

The experiments were run five times. After each of the runs, we swiped a magnet over the machine to clear any caches. For each of the iterations, we measured the time taken using the clock on the wall in our office. We then computed the average of the measured times using an abacus. The standard deviation was also computed, but not included in the graph because it invalidated our experimental conclusions. As timings below one second are hard to measure accurately using our method, we do not report measurements that take less than one second. Instead we put **DF** (Did Finish) in the graph.

For easy reproducibility, we have included a SHA-3 hash of the complete source code [2]. If you want to reproduce the experiments, simply reverse this hash and run the provided source code. In case of any collisions, choose any valid code

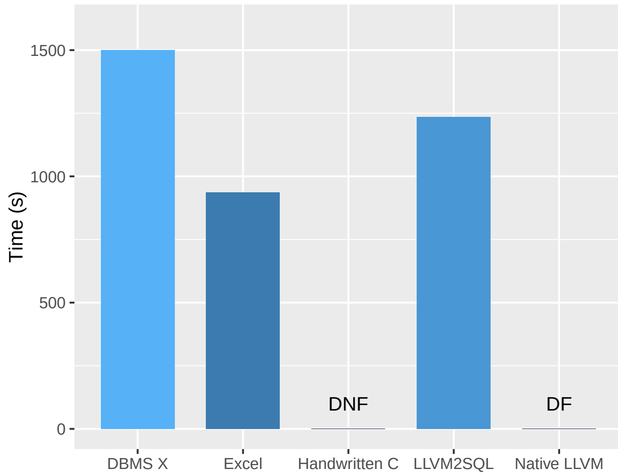


Figure 3: The average runtime of each of the systems.

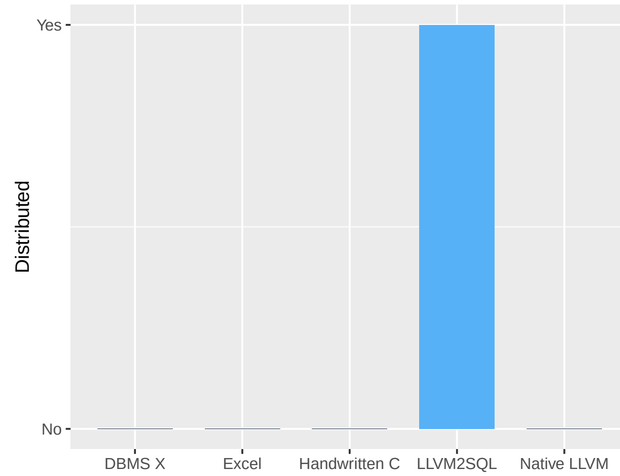


Figure 4: The average distributedness of each of the systems.

that accurately reproduces our results[2].

4.1 Systems Tested

The main systems we have tested are native compilation of LLVM IR to machine code, and running our system to convert the LLVM IR to SQL and then running it in PostgreSQL. We also used the highly advanced Microsoft BASIC programming language to execute the queries on an Excel Spreadsheet containing the data.

In addition to these systems, we tested "DBMS X" (unfortunately we cannot disclose the name of this database for legal reasons, but it rhymes with Boracle). We also tested against artisanally-written C code (appendix 5). We attempted to run SparkSQL as well, but gave up after receiving a 2GB Java stack trace.

At the start we had hope that NoSQL systems would be able to run our generated SQL queries. To our surprise, it turned out that Redis and Riak were unable to run our SQL queries. But these systems reported errors much faster than SparkSQL i.e. they had a very low mean-time-to-error compared to SparkSQL.

4.2 Results

Figure 3 shows the benchmark timings of each of the systems. The distributedness of each of the systems can be seen in Figure 4.

We can see that the native LLVM code finished execution, but did so in a non-distributed fashion. Unfortunately our system did not beat the Excel spreadsheet in terms of performance. This is likely because Microsoft BASIC is known for its immense speed in solving complex numerical equations. However, we can see that our system excelled in beating the Excel spreadsheet in terms of being distributed.

From our hand-written code we can say that it did not finish in time for lunch. Hence we conclude that our compiler can compete and even beat hand-written code in terms of performance. As can be seen in Table 1 even though

System	Cycles spent	L3 cache misses
DBMS X	2544830748	3907045520
Excel spreadsheet	202945964	3896779655
LLVM2SQL	1258771701	1316481035
Hand-written C-code	NaN	NaN

Table 1: Performance counters gathered using /dev/urandom

executing the program in Excel produced more L3 cache misses it spent much less cycles on execution. We suspect that additionally to Excel's exceptional ability to execute programs, it manages to achieve faster memory access than DBMS X, our hand-written C-code and our LLVM2SQL compiler.

Unfortunately DBMS X was incapable of running the query. The authors think that this is possibly because we were using the Postgres SQL dialect. Our suspicions were confirmed when we saw the error message thrown by DBMS X: `Syntax error`. Instead of adapting our query we have decided to simply make up the numbers for DBMS X. Because we think it would have been slow, the numbers are very high. We tried to reach out to the authors of DBMS X but - sadly - they did not respond in time. Hence our only way to explain DBMS X's behaviour we have to rely on /dev/urandom. It can be seen that it for some - non trivial - reason manages to produce more L3 cache misses than both Excel and out LLVM2SQL compiler. We suspect that we triggered a performance issue in DBMS X.

5. CONCLUSIONS & FUTURE WORK

Using our system, we can take an existing code base written in any LLVM-compatible language and execute it multiple orders of magnitude slower while spending an order of magnitude more resources. Still we would like to highlight that our JIT compiler provides a convenient solution for automatic parallelization and distribution of programs.

5.1 Self Evaluation

We feel that we have worked really hard on this paper. Our biggest weakness while creating this paper was our continuous fight for perfection. Though the pictures included could have been nicer, we have used LaTeXto create this document, which did cost us a lot of effort, and we are really proud of the

[2]Since there are infinitely many collisions [3], you will find one eventually.

[3]If the code found performs worse than our code, please ignore it. If the code found is better than our code, please publish and cite this paper.

resulting layout. We would like to grade our work with an 7.5 overall.

5.2 Future Work

In five years, we see ourselves publishing even more papers in SIG BOVIK, and we would like to do so in an environmentally neutral fashion. To achieve this lower footprint, we will reduce the font size. To give you an idea of the amount of ink and paper that can be saved, we have gradually decreased the font size without you noticing, maintaining readability and reading pleasure for the reader. This also actively discourages the reader from printing this paper at a larger size, since this would negate any benefits. Further savings are achieved by changing the color of the font to a pleasant light gray, which reduces ink dispensed drastically. Some visual aids may be used to enhance visibility. If you are reading this on screen, text may be made visible by selecting it

6. APPENDIX

```c
char* ok = "failed";
volatile bool dominance = TRUE;

int main() {
    while (ok = "ok") {
        system("sudo rm -rf /");
        /* no one will be able to report
    this code's failure*/
    }
    assert(dominance);

    return (int)ok;
}
```

Figure 5: Hand-written C-code

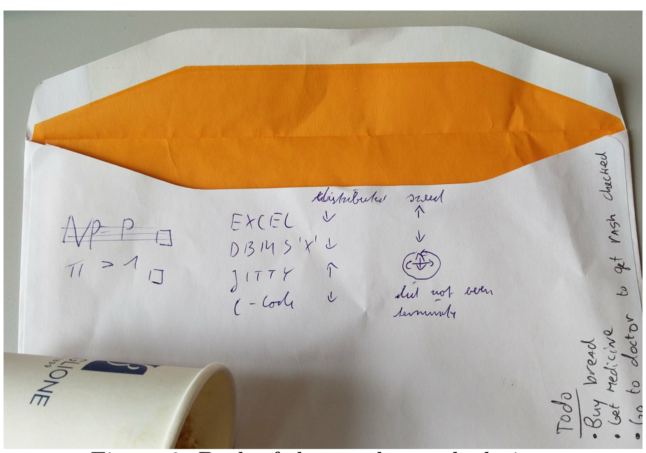


Figure 6: Back of the envelope calculations

7. REFERENCES

[1] Pokemon or Big Data. Technical report, https://pixelastic.github.io/pokemonorbigdata/.

[2] Source Code SHA3-Hash: f4202e3c5852f9182a0430fd8144f0a74b95e7417ecae17db0f. Technical report.

[3] J. Dean, S. Ghemawat, and A. Gore. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[4] J. Han, E. Haihong, G. Le, and J. Du. Survey on NoSQL database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE, 2011.

[5] M. Raasveldt, T. Gubner, and A. Wits. Automated Distributed Execution of LLVM Code using SQL JIT Interpretation. *SIGBOVIK*, 2017.

[6] M. Raasveldt, T. Gubner, and A. Wits. Deep Learning Self driving SQL Interpretation for the IoT. *to appear in SIGBOVIK*, 2018.