Technische Universität München

TUM School of Computation, Information and Technology

Efficient and Safe Integration of User-Defined Operators into Modern Database Systems

Moritz-Felipe Sichert

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:

Prof. Dr. Alexander Pretschner

Prüfer*innen der Dissertation:

1. Prof. Dr. Thomas Neumann
2. Prof. Alfons Kemper, Ph.D.
3. Prof. Dr. Torsten Grust

Die Dissertation wurde am 29.06.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 15.12.2023 angenommen.

Abstract

Modern relational database systems are able to execute SQL queries on state-of-the-art hardware very efficiently. They are designed to run on hundreds of CPU cores and effectively utilize terabytes of main memory.

Complex data mining and machine learning algorithms have become increasingly common in data analytics. However, most data scientists prefer not to use classical database systems for data analytics. The main reason why relational database systems are not used is that SQL is difficult to work with due to its declarative and set-oriented nature, and is not easily extensible. Instead, several specialized systems which are easier to use exist to evaluate these algorithms.

However, using these various systems comes at a price. Moving data out of traditional database systems is often slow as it requires exporting and importing data, which is typically performed using the relatively inefficient CSV format. Additionally, database systems usually offer strong ACID guarantees, which are lost when adding new external systems. This disadvantage can be detrimental to the consistency of the results.

We present User-Defined Operators as a concept to include custom algorithms in modern query engines. Users can write idiomatic code in the programming language of their choice, which is then directly integrated into our compiling database system Umbra and its generated code. The system must be guarded against potentially malicious user code. We show how WebAssembly can be used as an intermediate language to guarantee the safety of the execution.

Zusammenfassung

Moderne relationale Datenbanksysteme können SQL-Anfragen sehr effizient auf aktueller Hardware ausführen. Sie zeichnen sich dadurch aus, dass sie Systeme mit hunderten von CPU-Kernen und mehreren Terabyte an Hauptspeicher voll auslasten können.

Komplexe Algorithmen für Data-Mining und maschinelles Lernen werden in der Datenanalyse immer häufiger eingesetzt. Üblicherweise werden in Data-Science-Anwendungen jedoch keine klassischen Datenbanksysteme zur Datenanalyse verwendet. Der Hauptgrund warum relationale Datenbanksysteme nicht verwendet werden, ist, dass SQL eine deklarative und mengenorientierte Sprache ist, die nicht leicht erweiterbar ist. Stattdessen werden spezialisierte Systeme eingesetzt, die einfacher zu verwenden sind, um solche Algorithmen auszuführen.

Die Verwendung dieser verschiedenen Systeme hat jedoch ihren Preis. Das Importieren und Exportieren von Daten aus traditionellen Datenbanksystemen ist oft langsam, da die Daten üblicherweise in das relativ ineffiziente CSV-Format konvertiert werden. Außerdem bieten Datenbanksysteme in der Regel starke ACID-Garantien, die verloren gehen, wenn weitere Systeme hinzugefügt werden. Dadurch kann die Konsistenz der Ergebnisse beeinträchtigt werden.

In dieser Arbeit stellen wir das Konzept von User-Defined Operatoren vor, mit denen benutzerdefinierte Algorithmen in moderne Datenbanksysteme integriert werden können. Benutzer können idiomatischen Code in der Programmiersprache ihrer Wahl schreiben, der dann direkt in unser kompilierendes Datenbanksystem Umbra und seinen generierten Code integriert wird. Das System muss gegen potenziell bösartigen Benutzercode geschützt werden. Wir zeigen, wie WebAssembly als Zwischensprache verwendet werden kann, um die Sicherheit der Ausführung zu gewährleisten.

Acknowledgments

I am deeply grateful to Prof. Dr. Thomas Neumann, my professor and doctoral advisor, for his invaluable guidance. I extend my thanks to the members of my thesis committee, Prof. Alfons Kemper, Ph.D., Prof. Dr. Torsten Grust, and Prof. Dr. Alexander Pretschner.

Special thanks to my colleagues, friends, and family whose unwavering support and motivation have been crucial throughout my academic journey.

Preface

The author has published excerpts of this thesis in advance.

Chapter 2 has previously been published in:

> Moritz Sichert and Thomas Neumann. "User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases." In: *Proc. VLDB Endow.* 15.5 (2022), pp. 1119–1131

In addition, the author of this thesis also co-authored the following related work, which is not part of this thesis:

> Magdalena Pröbstl, Philipp Fent, Maximilian E. Schüle, Moritz Sichert, Thomas Neumann, and Alfons Kemper. "One Buffer Manager to Rule Them All: Using Distributed Memory with Cache Coherence over RDMA." in: *ADMS@VLDB*. 2021, pp. 17–26

> Maximilian Rieger, Moritz Sichert, and Thomas Neumann. "Integrating Deep Learning Frameworks into Main-Memory Databases." In: *4th International Workshop on Applied AI for Database Systems and Applications*. 2022

> Christian Winter, Moritz Sichert, Altan Birler, Thomas Neumann, and Alfons Kemper. "Communication-Optimal Parallel Reservoir Sampling." In: *BTW*. vol. P-331. LNI. Gesellschaft für Informatik e.V., 2023, pp. 567–578

Contents

List of Figures

List of Tables

List of Listings

CHAPTER 1

Introduction

Excerpts of this chapter have been published in [SN22].

Modern data analytics has evolved to include complex algorithms for data mining and machine learning. Specialized systems have been designed that are able to handle ever-growing amounts of data to solve a larger variety of problems. Unfortunately, traditional systems, especially relational database management systems (RDBMS), seem difficult to adapt to state-of-the-art data analytics [Yin+21].

Still, most data that is eventually analyzed in special-purpose systems is originally sourced from RDBMS. Thus, a common approach is to create ETL (Extract, Transform, Load) workflows that can accommodate the use of different systems [Zha+21b]. ETL workflows usually collect data in a data warehouse, which is built on top of an RDBMS that supports SQL. The data is then exported to be further processed by the data analytics systems. This extraction process can be implemented naively by exporting to a CSV file from the data warehouse and subsequently importing the data into the analytics systems. Additionally, some systems support more efficient data transfer by directly communicating with the RDBMS. As the last step, the results of the analysis are often transferred back to the data warehouse so that they can be further processed, for example, the data displayed in a dashboard. Exporting and importing data can often consume considerable time, especially when the data must be serialized to and parsed from a text format like CSV.

When data is extracted from an RDBMS, many beneficial characteristics of the system are lost. In particular, the ACID properties – Atomicity, Consistency, Isolation, and Durability – which are essential for correct transactional processing [HR83] can no longer be guaranteed. The atomicity property ensures that a transaction is either fully committed to the database or not at all, even if it contains multiple operations, to avoid incomplete or potentially incorrect

data. A database system that respects the isolation property must ensure that concurrent accesses to the same data by different users of the system do not lead to unexpected or invalid results. In particular, when multiple users are updating or writing to the database at the same time, the database system must guarantee that the updates do not interfere with each other.

One could argue that the atomicity and isolation properties may be of minor importance for systems that mainly process read-only OLAP workloads since a system processing only read-only transactions cannot violate these properties. But the durability property, which guarantees that data cannot be lost accidentally once it is committed to the database, and the consistency property, which mandates that the data must always be in a consistent state even if a transaction or the entire database system fails, are essential to provide accurate results. Additionally, to provide near real-time data analytics, data warehouses must be periodically updated. Since periodical updates may run in combination with longer-running analytics queries at the same time, an analytics system must inevitably support proper transaction management and observe all ACID properties.

1.1 Modern Relational Database Management Systems

Modern RDBMS additionally offer excellent execution speeds on modern hardware. Systems such as DuckDB [RM19] or Umbra [NF20] are able to fully saturate the available resources on modern systems. Traditional RBDMS, such as Postgres [SR86] or IBM DB2 [HJ84], were developed under the assumption that most data must be loaded from spinning hard disks or even tape drives whereas modern systems try to optimize their query execution for in-memory processing. The bandwidth of hard disks and even modern flash-based SSDs is lower than the bandwidth of main memory by orders of magnitude; hard disks can reach a read bandwidth of a few hundred of MB/s, SSDs connected to the CPU via NVMe several GB/s, while DDR4 or newer main memory reaches several hundred GB/s. Also, it is now feasible to build systems with several hundreds or even thousands of GiB of main memory, which means that the working set of an analytical query will often completely fit in main memory.

Also, while the clock rate of CPUs has increased dramatically up to a few GHz in the last decades, the growth of CPU clock rates in the last century has mostly stagnated. This phenomenon has been coined “The End of Moore’s Law” [TW17]. Instead, CPU vendors now build chips with an increasing amount of cores and develop new techniques to better utilize the available clock rate.

Modern, commercially available CPUs contain up to 100 cores and use simultaneous multithreading (SMT) to execute up to 200 threads on a single CPU. The high degree of parallelism in these CPUs necessarily revolutionized software development towards concurrent processing [SL05].

Naturally, traditional systems run faster on modern hardware than on forty-year-old hardware. However, to fully utilize the available resources, a database system must specifically be designed with the capabilities of modern hardware in mind. Especially the execution engine of modern database systems rely on techniques such as code-generation [Neu11] or vectorized execution [BZN05], to be able to saturate the memory bandwidth, instead of relying on traditional Volcano-style execution [Gra94].

This thesis builds on the database system Umbra. Umbra is a main-memory-first system that tries to load as much data as possible into main memory but can gracefully fall back to fast flash-based storage. It employs code-generation using a custom intermediate representation to achieve low latencies and high throughput at the same time [Ker+18; KLN18; KLN21a]. In Umbra, all queries are fully parallelized using morsel-driven parallelism [Lei+14] so that all available CPU cores can be saturated. Additionally, Umbra's query optimizer uses several novel techniques that allow it to generate efficient query plans even for deeply nested queries or queries with thousands of joins [Fre+20; NK15; NR18] and it can choose from many specialized join and aggregation algorithms [BGN21; FN21; KLN21b; RN22]. The optimizer relies on samples of the data, which are used to predict selectivities and cardinalities [BRN20; FN19; Win+23]. Umbra also has optimized several components found in traditional database systems or other data analytics systems for modern multi-core systems [Böt+20; DLN19; DLN21; Win+20; Win+22].

1.2 Data Analytics in Database Systems

In practice, in data analytics, the advantages of using RDBMS do not outweigh the disadvantages. Data scientists often prefer using systems such as Spark [Zah+12], TensorFlow [Aba+16], or systems using the MapReduce paradigm [DG08], even if they cannot reach main-memory performance. In these systems, algorithms can be formulated in procedural programming languages such as Java, Scala, or Python. In comparison to SQL – the primary query language for RDBMS – those languages are better suited to formulate algorithms used in modern data analytics. Such algorithms are often iterative and can be expressed naturally by using code that contains assignments to variables and control-flow statements such as loops.

However, SQL is declarative, set-oriented, and generally very different from most programming languages. It was initially proposed over 40 years ago by Chamberlin and Boyce [CB74], but has remained the most widely used query language for RDBMS until today. Precisely because of its differences from procedural programming languages, SQL gives query engines a lot of flexibility in deciding how to execute a query.

When a database system receives a SQL query, it first translates the query to an algebraic representation which is based on the relational algebra originally proposed by Codd in 1970 [Cod70]. Each algebraic operator generates a multiset of tuples and takes the multisets of any number of input operators. Hence, database systems based on the relational algebra conceptually process multisets or streams of tuples.

SQL imposes no additional requirements on database systems to execute queries. In particular, a database system can freely choose the order in which the algebraic operators are evaluated, and how exactly each algebraic operator is implemented as long as the semantics of the result is preserved. Thus, SQL makes it possible to design efficient query engines that are tailored to old mainframe systems that mainly read from tape drives, as well as state-of-the art query engines that rely on fast NVMe SSDs or even run entirely in main memory to achieve processing speeds of several hundred gigabytes per second. All query engines use the same SQL queries but can be tuned individually to the features and capabilities of the hardware they are running on.

1.3 Extensible Database Systems

The feature set of SQL is closely tied to the concept of stream processing, which supports filtering, several join types, and a wide range of aggregation and window functions. However, adding new functions is not easily possible. The execution of SQL queries is tightly coupled to the specific query engine of each database system, so adding new features to SQL requires deep knowledge of database internals.

In 1990, Carey and Haas noticed that "a DBMS must be extensible at all levels" [CH90] to support new types of workloads that SQL does not cover. Even earlier, the EXODUS system was designed by Carey et al. [Car+86] with the explicit goal of being an extensible DBMS. For query processing, EXODUS supports adding new algebraic operators, which must be written in its own programming language "E". E is based on C and adds several new types and functions that allow a program to interact with the other components of the system, such as the query optimizer or the storage engine. During that time,

other projects such as Starburst [Sch+86] and Postgres [SR86], which is still a widely-used RDBMS, set their focus on extensibility, as well.

Most of these projects can be extended by using a low-level C interface. Often, extensions can be loaded by the database system dynamically, which means that the database system does not need to be recompiled from its source code and not even restarted to add new functionality. Because every system defines its own API, however, programmers that develop extensions need to learn the API for each specific system. Also, data scientists often prefer to use higher-level programming languages to implement their algorithms as opposed to low-level languages like C.

Another downside of a low-level interface is the lack of security; if an extension runs in the same process as the rest of the database system, a bug in the extension can bring down the entire database system, or even worse, a malicious extension can gain unrestricted access to the system. So, extensions must be *fenced* from the database system, for example, by running them in a separate process.

Current database systems also offer some extensibility directly in SQL. In general, imperative control flow can be formulated in SQL by using recursive CTEs [DHG20; HG21]. Since SQL with recursive CTEs is turing-complete, it is possible to translate arbitrary imperative control flow directly to SQL. Also, some RDBMS offer imperative extensions to SQL such as T-SQL in Microsoft SQL Server [22c] or PL/pgSQL in Postgres [22a]. Imperative extensions allow users to write User-Defined Functions (UDFs), which can be called from standard SQL queries. The query engine can directly execute queries containing calls to UDFs written in this extended language. In theory, this functionality makes it much easier to write more complex algorithms when compared to recursive CTEs. However, the execution of functions written in these imperative languages tends to be very slow [GR21].

Further, data analytics pipelines usually rely on several pre-defined algorithms or at least a common standard library, as found in many programming languages. Data scientists typically use existing frameworks, such as TensorFlow or Spark, to implement their custom analytics workflow in Python or Scala, respectively. For high-performance applications that require better control of memory, languages such as C++ or Rust are often used. Using an imperative extension of SQL is mostly infeasible for data scientists. These extensions offer no built-in functionality to support data analytics, do not have large communities that provide solutions to many common problems, and often have poor debugging support.

Additionally, UDFs are usually not allowed to take entire streams of tuples as arguments or return them. Instead, UDFs are applied to each tuple of one particular stream, which is similar to how a map operation works in other

systems. Clearly, algorithms used in data analytics cannot generally be expressed exclusively by map-like operations; clustering, regression, and training machine learning models all process entire sets of data and must be able to access the data arbitrarily. Thus, UDFs are unsuitable for efficiently integrating custom algorithms in database systems.

1.4 Challenges and Contributions

Clearly, data analytics plays an essential role in the era of ever-growing amounts of data. Relational database systems have evolved to run efficiently on modern hardware with terabytes of main memory and hundreds of CPU cores. However, data scientists tend to use specialized data analytics systems because RDBMS do not fit their needs. To summarize, we identify the following problems with modern data analytics:

(a) Processing data is inefficient due to costly import and export processes between different systems. Additionally, many systems cannot use modern hardware to its full capacity.

(b) Data may not be consistent; especially when extracting from an RDBMS, ACID properties can no longer be guaranteed.

(c) The most efficient systems have poor usability and use SQL as a query language that is difficult to extend for data analytics.

(d) Extensible database systems are based on traditional architectures and older hardware. Also, extensions must be written using low-level APIs, which can often lead to security issues.

Naturally, problems (a) and (b) can be solved by running all data analytics algorithms directly in a modern RDBMS. However, using an RDBMS directly leads to problems (c) and (d). Since most data originates from an RDBMS which is used to safely and consistently store critical data, leaving the data in the database system and directly analyzing it is a sensible choice. Therefore, we focus on the issues (c) and (d).

In this thesis, we present the concept and implementation of *User-Defined Operators* (UDOs). UDOs allow users to easily add custom algorithms to modern query engines. Users can write idiomatic code in the programming language of their choice, which is then directly integrated into existing database systems.

We demonstrate how UDOs can be integrated efficiently into our modern database system, Umbra. UDOs can benefit from all the advantages of Umbra, such as its code-generating query engine and efficient parallelization strategy.

When data analytics algorithms are executed in Umbra using UDOs, they can easily outperform existing specialized data analytics systems such as Spark. To demonstrate the usability of UDOs, we also show how they can be integrated into the traditional database system Postgres. That way, users need to write UDO code only once and can use it in any database that supports UDOs.

Further, we show how UDOs can be executed safely even if they contain arbitrary user code. Since our database system Umbra generates code that is compiled into one program for every query, it is not possible to easily separate the execution of user code from the rest of the query by executing it in a separate process. Instead, we leverage WebAssembly to ensure that running user code cannot lead to security issues. WebAssembly is an assembly-like language that was initially designed for the safe execution of arbitrary code in web browsers. We integrate a WebAssembly translator between the UDO code and the intermediate representation used by Umbra's code-generating query engine, which enables safe execution of UDOs at near-native speed.

CHAPTER 2

User-Defined Operators

Excerpts of this chapter have been published in [SN22].

In this chapter, we present the concept and implementation of *User-Defined Operators* (UDOs). UDOs integrate user-written code directly into existing database systems which lets them benefit from existing features of a database system. The user code can directly process data which does not need be exported from the system, first. Since the data remains under control of the database system, running queries with UDOs also maintains all ACID properties which enables transaction-safe analytics of data.

Users can conceptually write UDOs in any programming language of their choice and are not limited to SQL. We provide a simple API that lets user code interact with the database system. The API is independent from internals of the query engine and the same user code can be used in any database system that implements UDOs. We describe how UDOs can be implemented in our code-generating database system Umbra [NF20] and in Postgres.

2.1 Related Work

Many database systems offer imperative extensions to SQL that can be used to write UDFs, which enables easier implementation of iterative algorithms and also allows the users to reuse existing database functionality. Gupta et al. [GR21] show that the extensions tend to be slow and cannot reach main-memory speeds. With their findings they encourage more research to improve the execution of UDFs.

A very popular approach for large-scale data analytics is MapReduce, which is often used for Big Data analytics but can also be applied to UDFs. Friedman

et al. present an approach to formulate UDFs in MapReduce and integrate them into SQL queries [FPC09].

Furthermore, separating UDFs into calls to a few predefined functions like map and reduce can be used to compile UDFs in the database and integrate them into the query execution as shown by Crotty et al. [Cro+15] who extend the MapReduce concept by adding more functions such as selection, join, or loop so that UDFs can address more different use cases. To execute this functionality efficiently, they make use of the LLVM framework [LA04] to be able to apply low-level optimization techniques to the generated queries.

Zou et al. [Zou+21] present an optimized approach to automatically partition workloads that contain UDFs. This approach allows users to develop highly scalable data analytics pipelines without in-depth knowledge of writing scalable code.

Palkar et al. [Pal+18] propose the Weld framework for data analytics, which combines code from different systems, such as queries written in SQL and programs written in Python. This framework uses a novel intermediate representation that can be lowered to LLVM.

Timely Dataflow is a novel concept for scalable data analytics presented in the Naiad system by Murray et al. [Mur+13]. This concept provides a low-level interface to assemble computational graphs, upon which high-level libraries and applications can be built. Murray et al. also present an implementation in the Rust programming language [Mur+16].

Writing code manually for a specific problem can lead to rapid execution but may not be easily combined with existing modern database systems. Passing et al. [Pas+17] present *lambda functions*, which are used to customize specific code-generating operators with greater ease. Schüle et al. [Sch+20] show how this technique can be applied to just-in-time compilation of user-written lambda functions in Postgres.

Techniques used in modern main-memory databases such as code generation and vectorized execution can also be used for data analytics. Zhang et al. [Zha+21a] present a system that automatically analyzes code to dynamically generate optimal query plans at runtime.

Duta et al. and Hirn et al. [DHG20; HG21] describe an approach that allows users to write code in the procedural programming language PL/SQL. Such PL/SQL programs are transformed entirely to recursive CTEs. This technique allows procedural programs to be interpreted by any SQL engine that supports recursive CTEs, and works well for many use cases. The result of the transformation is standard SQL, which can theoretically be executed in a main-memory database, provided it has an efficient implementation of recursive CTEs.

2.2 The User-Defined Operator

To achieve good usability and performance while maintaining ACID properties, we present the novel *User-Defined Operator* (UDO), which represents user-written algorithms as *algebraic operators*, which extend the relational algebra utilized by existing RDBMS. UDOs solve the problems in data analytics mentioned in Chapter 1 as follows:

Performance: Our implementation can optimize queries containing UDOs very efficiently. When those queries are executed in our code-generating database system Umbra, we can generate code that is as efficient as complex native operators written by database experts. As UDOs are directly executed in the database, the costly exporting and importing of data is not required.

Consistency: As UDOs are directly integrated into the query engine of existing RDBMS, they preserve all the ACID properties guaranteed by the system. User-written code is not required to take any precautions regarding consistency or isolation; the tuple streams utilized as input by UDOs follow the standard semantics of the isolation levels in SQL.

Usability and Extensibility: Even though UDOs are deeply integrated into query and execution engines of existing RDBMS as algebraic operators, users can use a simple API in the programming language of their choice. The complexity of writing efficient query engines is entirely hidden, and no knowledge of database internals is required. As query engines treat UDOs as regular algebraic operators, they can process arbitrary streams of tuples to generate a new output stream. This approach also interacts nicely with SQL; the input streams given to an UDO can be the result of arbitrary SQL queries containing joins and aggregations. Similarly, the output of the UDO can be further processed by using SQL.

2.2.1 Overview

Figure 2.1 shows the overview of our architecture to compile and integrate UDOs into a query engine. The user writes a standard SQL query, and the query uses the UDO given by the user as source code of an imperative programming language, such as C++ or Rust.

First, the UDO code is processed and analyzed to detect errors in the code. Next, the UDO must be translated into a representation that the query engine can use. We call this part of the system the *UDO User Compiler*; as an additional

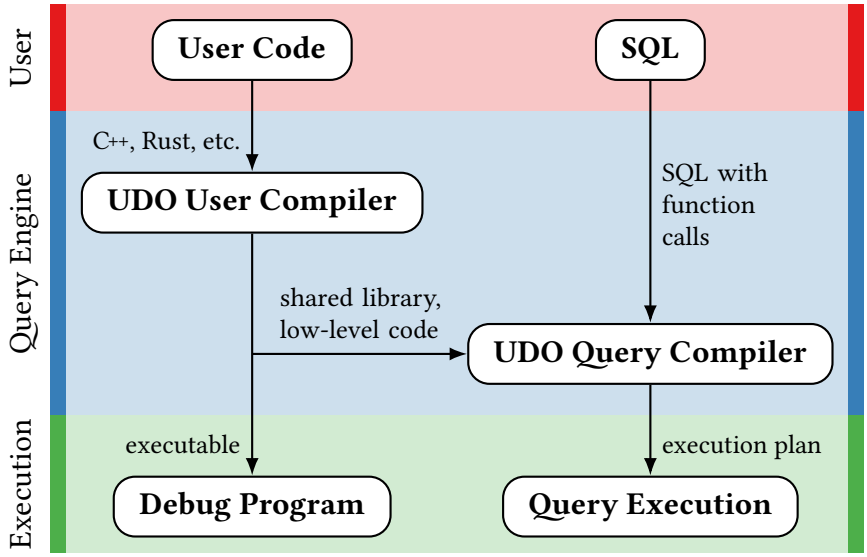


Figure 2.1: Architecture of the UDO User Compiler and UDO Query Compiler

feature it can directly generate a debug program that allows flexible debugging of the user code independent of the database system.

In a second step, the query engine takes the processed user code and integrates it into the query plan that is generated from the SQL query. This part of the system is called the *UDO Query Compiler*, which implements the representation of the UDO as an algebraic operator.

Both compilers can be developed separately. Once a suitable API for user-written code is defined, several implementations for different query execution models can be developed. Different UDO User Compilers can accept code written in different programming languages, while the UDO Query Compiler will usually be tightly connected to the rest of the database system.

In the following, we explain the concepts of an UDO with the help of the following example. A blog website uses a relational database system to store its blog posts. The writer of the blog wants to analyze the posts by category, and is interested in how many blog posts of the category "lifestyle" were written and how many posts of the other categories exist. Listing 2.1 shows a query that uses a UDO written in C++ that can answer the writer's questions. The query uses existing SQL syntax; the create function statement creates a new function with the name count_lifestyle Ⓒ, which takes a table as an input that must match the schema specified by InputTuple Ⓢ. Thus, any subquery with a string attribute with the name "word" can be used as an input to this UDO. The function also returns a table, which means that it can be used like a relation in the from part of a SQL query Ⓕ. Because the language is set to UDO-C++, the

```
create function count_lifestyle(table) Ⓒ
returns table language 'UDO-C++' as $$
class CountLifestyle : public UDOperator {
    uint64_t lifestyle = 0, other = 0;

    public:
    struct InputTuple { udo::String word; }; Ⓢ
    struct OutputTuple { udo::String word; uint64_t n; };

    void accept(ExecutionState state, const InputTuple& tuple) { ①
        if (tuple.word == "lifestyle") lifestyle++;
        else other++;
    }

    bool process(ExecutionState state) { ③
        vector<OutputTuple> output = {
            {"lifestyle", lifestyle}, {"other", other}
        };
        for (auto& tuple : output)
            emit<CountLifestyle>(state, tuple); ④
        return true;
    }
};
$$, CountLifestyle;
select * from count_lifestyle( Ⓕ
    table (select category as word from blog_posts)
);
```

Listing 2.1: SQL syntax to define and use UDOs: create function defines a function of the language UDO-C++. The C++ code is directly included in the statement, and the function can be called like any other table function by using the table keyword for table arguments.

SQL parser knows that this function is an UDO written in C++. The user-written code is included in the SQL statement (①, ③, and ④) and will eventually be processed by the UDO User Compiler and UDO Query Compiler.

2.2.2 The UDO User Compiler

In this section we introduce the *UDO User Compiler*. We define a high-level API that allows user-written code to be used by the User-Defined Operator. This API consists of only three functions, and it is not specific to any particular program-

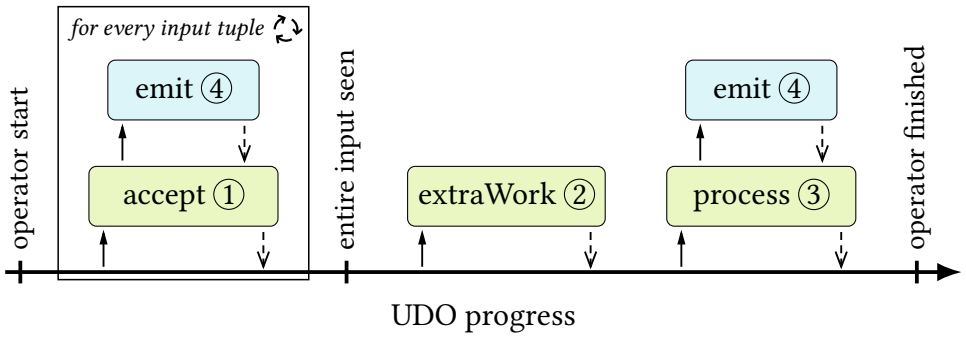


Figure 2.2: The UDO User Compiler: A user can write code in the accept and process functions and call the provided emit function. Conceptually, accept is called once for each input tuple, and process is called after the entire input was seen.

ming language. Our implementation provides a C++ API (refer to Section 2.3) but it is general enough to be implemented for most programming languages.

Figure 2.2 shows the conceptual overview of the functions that an UDO can implement and use. As the query execution progresses, the user-written functions accept ①, extraWork ②, and process ③ are executed. There are no additional limitations with the implementation of these functions. As such, the user can create arbitrary control flow by using conditionals and loops. The functions accept and process can also call the function emit ④ as provided by the UDO User Compiler.

These functions are used to integrate the UDO into the tuple stream of the query. To interoperate with other algebraic operators of the query, the user-written code must:

1. Optain tuples from its input(s), which are other relational operators,
2. Process the tuples, i.e., do the actual work, and
3. Generate the tuples as output so that the parent operator of the UDO can continue processing the query.

These requirements map nicely to the functions ① to ④ from above:

Accept The accept function takes a single tuple from an input and is called repeatedly for each tuple of the input and implements per-tuple processing. This function can, for example, materialize the tuple by storing it in a temporary data structure, or directly compute a (partial) result as shown in ① in Listing 2.1.

ExtraWork and process After all tuples from the input were seen by accept, the functions extraWork and process are executed. They can be used to post-process the input, such as aggregating or sorting it. As the user-written functions can contain arbitrary code and thus arbitrary control flow, more complex data analytics algorithms that require the entire input to be available upfront can be implemented as well.

The function extraWork allows the user code to guide the parallel execution of the UDO. Conceptually, the execution of the extraWork function is separated into *phases*. The phases are represented as integers, i.e., every phase has its own arbitrary integer value. The extraWork function has a parameter that takes the current phase and it returns the phase that should be executed next. If the UDO is then executed in parallel, the database system makes sure that all parallel invocations of extraWork all execute the same phase. Only when all threads have finished executing the current phase, the next phase will be started.

The function process on the other hand conceptually only has a single phase. It has no parameter for the phase and also cannot return a phase like extraWork. However, process is allowed to call emit (see below). The strict separation between extraWork and process allows implementations to more efficiently parallelize UDOs. We discuss this in more detail in Section 2.3.4.

Emit The function emit is provided by the UDO User Compiler and can be called by the accept and process functions to generate a tuple of the output of an UDO. It takes a single tuple of the output as an argument. Conceptually, this tuple is then passed to the parent operator of the UDO.

All four functions have a parameter for an implementation-specific execution state. When the user code calls emit, it must pass along the execution state it receives as argument in its accept or process functions. The main purpose of the execution state is to hold auxiliary information that the database system needs to execute the UDO. The user code does not need to interact with the state at all other than passing it to emit. However, database systems may implement additional functionality using the state.

The functions accept, extraWork, process, and emit can be used to easily implement iterative algorithms. The user code in the accept function should store all tuples of the input. The implementation of the extraWork and process functions should then use a loop to iteratively compute the result by repeatedly accessing the stored tuples. When the result is computed, process should call emit for every tuple of the result.

In the example shown in Listing 2.1, only process ③ calls emit ④ because the tuples for the output can only be computed once the entire input was seen. The example also shows that inputs and outputs do not have to be equal in their

schema or their cardinality. The code only uses a string attribute of the input with an unknown number of tuples while it generates exactly two tuples with a string and an integer attribute as its output.

Usability and Debugging

As the execution of UDFs is usually interleaved with the rest of the query and runs in the same process as the database system, it is not obvious how to enable debugging of the user code. This either requires the database system to implement a debugger itself, e.g., like in DBToaster [AK09], or the user requires access to the database process to use common debugging tools. Both approaches are not optimal. With the former approach users are forced to use DBMS-specific debugging tools instead of their own, and with the latter approach users need low-level privileged access to the database process.

For UDOs there is a more elegant solution: The accept, extraWork, and process functions are implemented by the user as separate functions in the programming language of their choice. Thus, a test program can be written in that language that provides a tuple stream and calls those functions. Additionally, the test program is required to implement the emit function, which is usually provided by the UDO User Compiler. For debugging purposes, the provided function could print its argument, for example.

Our implementation provides a standalone program for C++. It is completely independent from the database system and therefore can be used for any UDO independent of the system the UDO will eventually be used in. It reads the input data from regular CSV files and prints the output of the UDO. To match the characteristics of the execution in a database system more closely, our standalone programs also run multi-threaded and concurrently call the accept, extraWork, and process functions.

2.2.3 The UDO Query Compiler

The UDO Query Compiler is the part of the system that takes the artifacts generated by the User Compiler, such as a shared library, an object file, or a program of some low-level intermediate language, and integrates it into existing query plans. Additionally, the UDO Query Compiler must implement a standard algebraic operator. This operator must behave just like any other operator in the database system, such as selection, aggregation, or join, so that it can be used in conjunction with other operators in arbitrary queries. As such, the operator must be directly implemented by the query engine of the database system and by design requires the use of database internals.

There are four points where the UDO interfaces with the query engine, which are the four functions accept, extraWork, process, and emit. The first three functions are implemented by the user, but the emit function must be provided by the UDO Query Compiler. Conceptually, this function takes a tuple that was generated by the UDO and passes it on to the parent operator of the UDO. Depending on the actual execution engine used by the database system, different strategies may be used to implement this function so that it efficiently interacts with the existing engine.

Modern code-generating database systems based on the produce-consume model [Neu11], for example, do not need to implement the emit function at all. They could replace all function calls to emit by the actual code that is generated in the parent operator. We provide a detailed description of an implementation of UDOs in a code-generating query engine in Section 2.3.

Database systems that employ the iterator model [Gra94; Lor74] such as Postgres can implement the emit function by storing the emitted tuples in a temporary buffer. When the execution of the UDO is finished, the buffer can be used to return the tuples to the parent operator of the UDO. The same strategy can be used in vectorized query engines that are used in systems like MonetDB [BZN05], Vectorwise [ZWB12], or DuckDB [RM19].

Materializing all tuples in a temporary buffer in the iterator model may add significant runtime overhead. This overhead can be avoided by interrupting the user code as soon as it calls the emit function. Then, the single generated tuple can be passed to the parent operator. When the UDO should generate more tuples, the interrupted code must be continued. We explain how this approach can be implemented in more detail in Section 2.4.

2.3 User-Defined Operators in Code-generating Query Engines

In this section we present our implementation of UDOs in our database system Umbra [NF20]. Umbra is a main-memory first system, which is geared towards working primarily in memory but also supports storing relations on disks or preferably fast SSDs. After translating SQL to relational algebra, Umbra uses the produce-consume model [Neu11] to generate efficient code for the query. As code generation can be relatively expensive, especially for ad-hoc queries that complete quickly, Umbra does not directly generate machine code but uses the intermediate representation *Umbra IR*. This low-level language that was inspired by LLVM [LA04] can then be executed in multiple ways: A virtual machine that interprets the IR, a direct translation from Umbra IR to x86 assembly

```
enum Phase {
  Initialize = 0,
  AssociatePoints = 1,
  RecalculateMeans = 2,
  Finish = -1,
};

Phase extraWork(ExecutionState state, Phase phase) {
  switch (phase) {
    case Initialize:
      // Initialize the cluster centers
      // [...]
      return AssociatePoints;
    case AssociatePoints:
      // Iterate over all points to associate them with the closest
      // cluster center
      // [...]
      // Afterwards, continue by recalculating the means of all
      // clusters
      return RecalculateMeans;
    case RecalculateMeans:
      // Calculate the new means for all clusters
      // [...]
      if (/* k-Means exit condition */)
        // If some exit condition for the clustering is met, we finish
        // the execution of extraWork.
        return Finish;
      else
        // Otherwise, continue by associating the points to the new
        // cluster centers.
        return AssociatePoints;
  }
}
```

Listing 2.2: Implementation of extraWork for a UDO that implements k-Means clustering. The user code defines phases in an enum and can switch between them by returning the phase that should be executed next.

called the *Flying Start* backend [KLN21a], and a more sophisticated translation that uses LLVM to generate optimized machine code. Umbra uses *adaptive execution* [KLN18] to dynamically switch between all three approaches to achieve low latency for short queries and high throughput for long-running analytical queries. Finally, to enable good scalability across many CPU cores and even NUMA nodes, our execution engine employs *morsel-driven parallelism* [Lei+14].

2.3.1 UDO User Compiler

The goal of our implementation is for queries containing UDOs to run as fast as "native" queries that consist of only Umbra IR. To achieve this goal, our implementation supports C++ as the programming language for user code. As a systems language, it can be directly compiled to efficient native machine code. The second reason why we chose C++ specifically is that it can also be compiled to LLVM IR with the Clang compiler. As our compilation framework can use LLVM IR as well, this makes it possible to completely inline user code into the generated machine code, thus enabling native query speed.

Our UDO User Compiler for C++ provides some definitions of classes and functions that users can build on. The user code must define a subclass of the provided UDOperator base class. The conceptual functions accept, extraWork, and process functions are realized as member functions of the subclass which the user code can define. The emit function is a predefined member function of the base class UDOperator. As the functions that the user writes are member functions of a class, the UDO code can also make use of member variables to manage state across invocations of the accept, extraWork, and process functions. To enable parallelism within the UDO, our system potentially calls accept, extraWork, and process concurrently. Therefore, users must ensure that these functions are thread-safe.

The initial example in Listing 2.1 shows C++ code that our UDO User Compiler can use. The user-written class CountLifestyle is a subclass of UDOperator, which uses two member variables that track the number of occurrences of the word "lifestyle" and all other words. The variables are updated in accept ① and used for the generated output in process ③. The emit function is called with a single output tuple as an argument ④. The UDO CountLifestyle in this example does not require any additional parallel processing, so it does not define an extraWork function.

Listing 2.2 shows an excerpt of the implementation of extraWork for a UDO that implements k-Means clustering. The user can use arbitrary integers to define different phases, for example by defining an enum in C++. When extraWork is called with the current phase, it performs the operation according to the current phase and returns the phase that should be executed next. The

```
1 fn Γ.produce():
2   genCode("ht := initialize ht")
3   child.produce()
4   genCode("for r, aggr in ht:")
5   parent.consume("r⊕aggr")
6 fn Γ.consume(t):
7   genCode("ht.update("+t+")")
8 fn σ_p.produce():
9   child.produce()
10 fn σ_p.consume(t):
11  genCode("if p("+t+"):")
12  parent.consume(t)
13 fn R.produce():
14  genCode("for r in R:")
15  parent.consume("r")
```

Listing 2.3: Implementation of the produce and consume functions to demonstrate an aggregation (Γ), a selection (σ_p), and a table scan (R) in the produce-consume model.

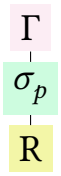
iterative k-Means algorithm needs to alternate between associating points to the nearest cluster centers and recalculating the cluster centers with its newly associated points. So, the user code defines the enum values AssociatePoints and RecalculateMeans to represent the steps of the k-Means algorithm as phases. The listing also shows that the user code can dynamically decide which phase to execute next. When the k-Means algorithm is finished according to some exit condition, extraWork returns the Finish value to indicate that the database system should proceed with calling the process function.

For the UDO Query Compiler our User Compiler generates two artifacts generated from the C++ code: An object file and an LLVM module. The object file uses the ELF format and could theoretically also be generated by any other C++ compiler that supports ELF. The LLVM module is specific to the Clang compiler. The module is generated, so that the UDO Query Compiler can potentially inline the UDO code with the rest of the query code generated by the database system.

Even completely different programming languages that do not use LLVM at all could be used. Our UDO Query Compiler in Umbra can work with any code that can be compiled to ELF object files. Some optimizations are not possible when the LLVM module is not available, nevertheless the UDO can be executed.

2.3.2 UDO Query Compiler

Our UDO Query Compiler takes the object file and the LLVM module from the UDO User Compiler and integrates it into a query plan. In the query, the UDO is represented as a relational algebra operator, which we will call UDO_a. The implementation of UDO_a is not specific to one particular UDO. UDO_a can take any UDO that was processed by the UDO User Compiler and integrate it into existing queries.




```
Γ.produce():
  R.produce():
σp.consume(r):
Γ.consume(r):
  Γ.produce():
```

```
1 ht := initialize ht
2 for r in R:
3   if p(r):
4     ht.update(r)
5 for r, aggr in ht:
6   output r⊕aggr
```

Figure 2.3: Code generated by a query compiler using the produce-consume model. The code generated by the algebraic operators is interleaved which leads to better data locality.

Because it is treated like any other existing algebraic operator, UDO_a integrates seamlessly with all DBMS components, such as the optimizer. While general optimizations across UDO_a are not possible since the optimizer does not have any information about the semantics of the UDO, the subtree that represents the input of UDO_a and the subtree that contains UDO_a can still be optimized.

Produce-Consume Model

Umbra uses the produce-consume model [Neu11] to generate code for a relational algebra tree. In this model, every algebraic operator implements the two functions produce and consume that generate code. Listing 2.3 shows how these functions could be implemented for an aggregation that uses hash tables (Γ), a selection with a predicate p (σ_p) and a table scan for the relation R. In general, the produce function is called when an operator should start generating its output. For Γ, this function first generates code to initialize a hash table, and then calls the produce function on its input. When an operator wants to pass a tuple of its output to its parent, it calls the consume function of the parent. The Γ operator generates code to update the hash table with the new tuple in its consume function. The remainder of Γ.produce then iterates over the hash table and calls the consume function of its parent with the aggregated result. The implementation of the σ_p operator is simpler; the selection does not need to initialize any data structures, so it only calls the produce function of its input in σ_p.produce. σ_p.consume generates code to evaluate the predicate and calls the consume function of its parent that will generate code in the true-branch of the predicate.

Figure 2.3 shows a query that uses those three operators and the generated code. Code from different operators tends to be interleaved. This approach leads to good data locality and very efficient execution on modern hardware but makes it difficult to integrate “foreign” code.

```
1 fn UDO_a.produce():
2   child.produce()
3   if UDO has extraWork:
4     genCode("phase = 0")
5     genCode("while phase != -1:")
6     genCode("  phase = extraWork(state, phase)")
7   if UDO has process:
8     genCode("process(state)")
9   genCode("fn emit_helper(t):")
10  parent.consume("t")
11
12 fn UDO_a.consume(t):
13  genCode("accept(state, " + t + ")")
```

Listing 2.4: Implementation of UDO_a in the produce-consume model. The produce and consume functions of UDO_a generate code to call the UDO functions accept, extraWork, and process.

Generating Code for UDOs

To integrate the functions ① – ④ of an UDO into a query plan, the UDO Query Compiler must generate code that acts as an interface between the user code and the code generated by built-in operators. As we model UDO_a like any other algebraic operator, there are two choices to emit code: the produce function and the consume function.

Integrating accept is straightforward; the consume function must generate code to call accept. Conceptually, the consume function generates code that processes a single tuple and we defined accept as a function that takes a single tuple, so they are a perfect match.

Similarly, the extraWork and process functions must be called in the code generated by produce. Only then, UDO_a can make sure that the entire input was seen and accept was called with all tuples from the input before the extraWork and process functions are called for the first time. To call extraWork, UDO_a generates code that calls it in a loop. The loop repeatedly calls extraWork keeping track of the current phase until extraWork returns -1 which indicates that it is finished.

Listing 2.4 shows the implementation of UDO_a in the produce-consume model. As described above, UDO_a.produce generates code to call extraWork and process and UDO_a.consume generates a call to accept.

To interoperate with other operators in the query plan, UDO_a.produce also first calls the produce function of its child operator. In the produce-consume model, the produce function of the child operator is then responsible for even-

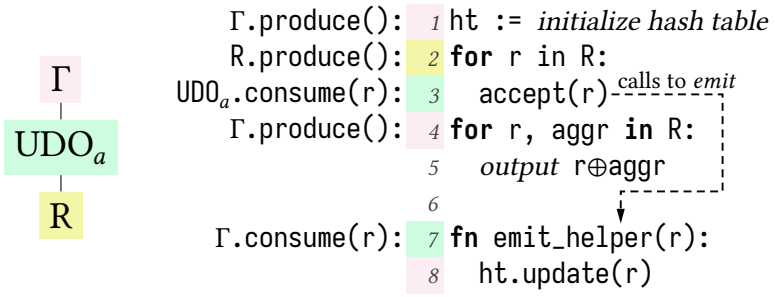


Figure 2.4: Code generated by the UDO Query Compiler and UDO$_a$ for a UDO that behaves like a selection (i.e., "pipelined operator"). Calls to emit in the user code are replaced by calls to the generated emit_helper function.

tually calling UDO$_a$.consume with an argument that contains the location or name of the variable that contains the tuple passed to UDO$_a$.

To implement the emit function in the produce-consume model, UDO$_a$.produce also generates code that defines a new, separate function emit_helper. Every call to emit in the user code means that a new tuple of its output is generated. This tuple must eventually reach the parent of UDO$_a$ so that the query processing can continue. In the produce-consume model, the parent receives an input tuple in the code generated by the consume function. To bridge the gap between the user code and generated code, UDO$_a$ emits the code of its parent into the separate function emit_helper. Then, the UDO Query Compiler replaces all calls to emit in the user code by calls to emit_helper.

With this approach, the UDO is entirely transparent both to the user code and the other relational algebra operators; they do not require any implementation details about the other. Built-in operators can call produce and consume of UDO$_a$ as usual, and the user code uses the API functions ① – ④.

Figure 2.4 shows the algebra tree and the generated code for a query that contains an UDO. This query is similar to the query from Figure 2.3; only the selection was replaced by an UDO. For this example, we assume that the UDO only calls the emit function in accept and does not implement extraWork or process. While the generated code defines the function emit_helper, it does not contain calls to emit_helper, because emit_helper is only indirectly called every time the UDO code calls emit.

Figure 2.5 shows another algebra tree and its generated code. Again, the query from Figure 2.3 is shown but we replaced the aggregation with an UDO. Additionally, the UDO now calls emit only in the process function. Writing UDOs that call emit only in process is useful for algorithms that can only generate their output once the entire input is seen. The generated code contains

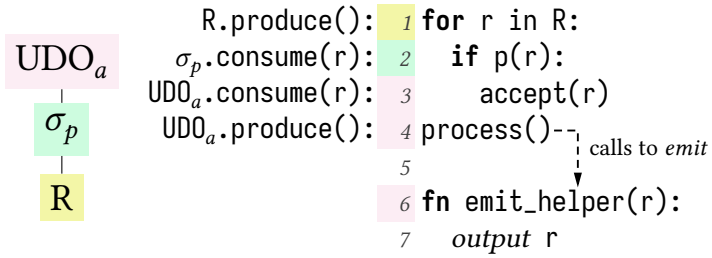


Figure 2.5: Code generated by the UDO Query Compiler and UDO$_a$ for an UDO that behaves like an aggregation (i.e., "pipeline breaker"). The generated code also calls accept and additionally makes sure that process is called only after the entire input is seen.

a call to process after the loop of the table scan. Thus, the process function will only be called after the UDO received all tuples of its input.

2.3.3 UDO Function Inlining

The examples in Figures 2.4 and 2.5 show that, initially, UDO$_a$ introduces several low-level function calls that will be executed for each tuple of the input and output; the accept function is called in line 3 for every tuple and the user code calls emit and indirectly emit_helper for every tuple of its output. When UDOs are executed in main-memory systems running on modern hardware, the best performance can only be achieved when the number of function calls is reduced. As data can be processed so quickly, every function call adds noticeable overhead. To solve this issue, we use a very common approach used in compiler optimization; we inline the function calls to eliminate them.

This approach is possible in our implementation because we can compile the user code written in C++ to LLVM. Umbra supports different execution modes, which includes LLVM, as well. Hence, we generally compile the user code to native machine code, which is stored as object files. To prepare the UDO code to be eventually inlined, the C++ code is also compiled to LLVM.

When the query is executed, the object file is loaded into memory and the code generated by UDO$_a$ uses real low-level function calls into the functions that are located in the object file. Since the object file is generated only once when the user runs create function, this compilation incurs no compilation overhead from the high-level compiler, which for languages like C++ could take up to several seconds. In fact, since the object file is generated only once for every UDO, we can afford running all available compiler optimizations on the user code. This means that even though the compiled query code will execute

lots of function calls to accept, extraWork, process, and emit, the UDO code is very efficient.

When our framework for adaptive execution [KLN18] detects a long running query, our compilation engine generates LLVM code for the entire query plan. When that happens, we search for all call instructions that call the accept, extraWork, process, or emit functions from the object file and replace them by their LLVM representation.

Note that in Umbra, switching to the LLVM mode is not a static decision that must be made before executing the query. This decision is made by the execution engine at runtime. The execution of the UDO code is transitioned smoothly with no downtime from the unoptimized implementation, which uses real functions calls into the object files, to the optimized and inlined LLVM code.

Listing 2.5 shows a part of the LLVM code, which is generated by our UDO Query Compiler. This code was generated from the SQL query shown in Listing 2.1 that selects from a relation and uses an UDO. The part shown here contains the table scan and the code for accept. The resulting LLVM code has no clear boundaries between the user code and the code generated for the table scan and thus has good locality. The first part of the code loads the string value from the column of the relation. This code is directly generated by the query engine for the table scan. It is followed by the inlined UDO code which first checks if the string has exactly 9 characters, because it compares it with the word “lifestyle”. If it does, it then actually compares the string by using the memcmp function and increments the corresponding counter. The rest of the code, again generated for the table scan, increases the tuple index, and repeats the loop if any tuples are left.

Independent from the actual implementation, the concept of function inlining can quickly lead to an increase in code size especially when inlined functions contain calls to other functions that are inlined. In the UDO Query Compiler, this issue can occur when a single query contains multiple UDOs. To avoid any code explosion caused by inlining, we only inline calls to the functions of a UDO if they are called only exactly once syntactically. As a result, there is no potential for code to be duplicated; therefore, code explosion can be avoided. This does not mean that it is impossible to call the emit function multiple times semantically. Wrapping the call to emit in a loop, for example, still satisfies our requirement but at runtime the function is called multiple times.

2.3.4 Parallel Execution

Generally, Umbra generates code that processes all queries concurrently on all available CPU cores. To achieve the best performance, UDO_a must also generate code that can be executed concurrently.

table scan

```
scan_loop_head:
  %tuple_index = phi i64 [ i64 0, %start_scan ],
      [ %next_index, %string_eq_block ]
  %attr_ptr = getelementptr { i64, i64 },
      { i64, i64 }* %column_ptr, i64 %tuple_index, i32 0
  %str_header = load i64, i64* %attr_ptr, align 8
  %str_body = getelementptr inbounds i64,
      i64* %attr_ptr, i64 1
  %str_offset = load i64, i64* %str_body, align 8
  %str_len = trunc i64 %str_header to i32
  %is_long_str = icmp ugt i32 %str_len, 12
  %str_raw_ptr = select i1 %is_long_str,
      i64 %str_data, i64 0
  %str_ptr = add i64 %str_raw_ptr, %str_offset
  store i64 %str_header,
      i64* %local_var_str_header, align 8
  store i64 %str_ptr,
      i64* %local_var_str_body, align 8
```

User-Defined Operator

```
  %has_len_9 = icmp eq i32 %str_len, 9
  br i1 %has_len_9, label %cmp_str_block, label %else_block
cmp_str_block:
  %string_cmp = call @memcmp(i8* %local_var_str,
      i8* @str_literal, i64 9)
  %string_eq = icmp eq i32 %string_cmp, 0
  br i1 %string_eq, label %string_eq_block,
      label %string_ne_block
string_ne_block:
  br label %string_eq_block
string_eq_block:
  %var = phi i8* [ %dbs_var, %string_ne_block ],
      [ %nondbs_var, %cmp_str_block ]
  %counter = bitcast i8* %var to i64*
  atomicrmw add i64* %counter, i64 1 monotonic
```

table scan

```
  %next_index = add i64 %tuple_index, 1
  %at_end = icmp eq i64 %next_index, %relation_size
  br i1 %at_end, label %finish_scan,
      label %scan_loop_head
```

Listing 2.5: LLVM code generated by the UDO Query Compiler after inlining the UDO from Listing 2.1. The user code is directly interleaved with the rest of the query code which removes all potential function call overhead.

Our code generation framework and the morsel-driven scheduler ensure that code generated by the consume function of any relational algebra operator is called concurrently. As our implementation of consume for UDO_a generates a call to accept, this means that the user-written code in accept is executed concurrently as well. Similarly, we generate code that calls the extraWork and process functions concurrently on all available CPU cores.

Since the extraWork function does not receive any tuples and cannot call emit and therefore does not interact with any other relational operators in a query, our UDO Query Compiler can generate more efficient code for it. The implementation of relational operators usually requires the allocation of state for intermediate results, such as a hash table to compute a hash-join or an aggregation. Since the execution of the code generated by UDO_a and all other relational operators in the produce-consume model is usually interleaved, it is necessary to keep alive the states of all relational operators that are currently being executed. However, while extraWork is called, all children of UDO_a have already finished, so their states can be deallocated, but the execution of the parent operators has not started, yet. So, the allocation of the states of the parent operators can be delayed until process is called for the first time.

To parallelize the execution of the extraWork function, our implementation stores the integer value of the current phase in a global variable. This global variable is then used as an argument to the call of extraWork. The return value of extraWork is then compared to the value stored in the global variable. If they are equal, the execution continues without any synchronization. Otherwise, our implementation waits until all other concurrent calls to extraWork in other threads have finished. Then, it updates the global variable with the new phase. We repeat this until one call to extraWork returns -1 to indicate that the execution should proceed with the process function.

For this parallelization to work, the functions accept, extraWork, and process must be thread-safe. Therefore, the user must ensure that those functions can be called in parallel. Only thread-safe data structures or common idioms for synchronization such as mutexes or atomic operations should be used. Furthermore, since these functions may contain calls to emit, our UDO Query Compiler ensures that emit is thread-safe, as well. When functions are implemented as UDOs to compete with code-generating operators directly implemented in Umbra, the user code must be written very well and use state-of-the art synchronization.

2.3.5 Implementation Considerations

In this section we discuss some additional technical details and considerations for our implementation. While they do not extend the theoretical framework

of the UDO User Compiler and UDO Query Compiler, they are still useful to obtain a full picture of our implementation.

Allocation and Initialization of the UDO State

A UDO usually needs to maintain state during the execution. For UDOs written in C++, for example, users can define a class that can contain arbitrary data members. The UDO functions are then implemented as member functions of this class so they can access these data members. Thus, the UDO Query Compiler needs to make sure that the state of a UDO is handled correctly.

When a UDO is created by a user, the UDO User Compiler must determine the size and alignment of the UDO class and pass it to the UDO Query Compiler. With this information, the UDO Query Compiler generates code that allocates the memory for the UDO state at the beginning of the query execution and stores the memory address of the UDO state in a variable. It also generates code that initializes the allocated memory. In case of UDOs written in C++, the state is initialized by calling the constructor of the UDO class.

Similarly, a call to the destructor is generated at the end of the query program. Our UDO Query Compiler ensures that the destructor is always called even when the query is cancelled. Therefore, all state created in the constructor is correctly cleaned up by the C++ destructor which may contain implicit calls to destructors of member variables of the UDO class.

All member functions of a C++ class have an implicit this argument which is a pointer to an object of the class. Our UDO Query Compiler uses the variable that is created when the object is initialized as an argument for all calls to member functions of the UDO class.

Interestingly, passing the pointer to the UDO object explicitly to all UDO member functions allows the execution engine to execute different implementations of the same UDO on the same state. In our implementation in Umbra we have the choice between using the compiled object file of a C++ UDO or directly inlining the LLVM code of the UDO into the LLVM code generated for a query as discussed in Section 2.3.3. Now, when we switch from the code containing calls to the object file to the optimized inlined LLVM code, no synchronisation between threads is required. Since both versions of the code are semantically identical and operate on the same state, they can run concurrently. Thus, we can seamlessly switch between both versions of the code.

Passing SQL Values to UDOs

Programs written in high-level languages use basic types such as int and float, and complex types such as std::string in C++. SQL on the other hand uses its

SQL Type	C++ Type	
smallint	int16_t / uint16_t	*signed and unsigned integers have identical storage layout*
integer	int32_t / uint32_t	
bigint	int64_t / uint64_t	
double precision	double	
text / varchar	udo::String	*udo::String is converted to text*

Table 2.1: SQL types supported for C++ UDOs. Values can be converted in both directions. Nullable types are not supported.

own set of types such as integer, double precision, and varchar. Additionally, all value types in SQL can also be nullable. To pass SQL values to the user code and translate values generated by the user code back to SQL values, our implementation can convert values between the representation used to process SQL queries and the types of the high-level language used for the user code.

Table 2.1 shows the list of SQL and C++ types supported by our implementation. In general, no nullable types are supported and potentially nullable values cannot be passed to UDOs. To circumvent this limitation, a query can use the SQL function coalesce, for example, to ensure that all NULL values are replaced.

The SQL integer types map directly to integer types in C++. In SQL, all integers are signed, but C++ UDOs may also use unsigned integers. The storage layout for both kinds of integers is identical and uses twos-complement to represent signed integers. When a UDO generates an unsigned integer, it is converted to the corresponding signed integer type in SQL which can lead to negative values in case of a wrap-around. For floating-point numbers, no conversion is necessary. Values of double precision in Umbra's query engine and C++ use the same IEEE 754 representation [19].

For SQL strings, the UDO User Compiler provides the class udo::String. This class wraps the internal representation of SQL strings that is used by the query execution engine. It supports some basic string operations but more importantly can be converted cheaply to std::string_view. The std::string_view value then only references the string data and does not require any memory to be copied. Similarly, a std::string_view value can be converted cheaply to udo::String to generate a string as a result in a C++ UDO.

Global State and Concurrent Queries

As user code can be arbitrary code, it can also make use of global variables. While it is generally considered bad practice to rely heavily on global state, especially when global state is mutated, it is sometimes useful or even necessary to use. The standard library of C++, for example, uses global state to implement

some functions more efficiently. Another use of global state are *thread-local* variables, which are often used to make parallel implementations of algorithms more efficient.

As we allow users to write arbitrary code, our UDO User Compiler does not prohibit the use of (thread-local) global variables. Because we also want to maintain the isolation of separate queries running concurrently to not violate ACID properties, our UDO Query Compiler maintains entirely separate global and thread-local states for every instance of an UDO that is being executed. Thus, when a new query that contains an UDO starts, its global state will always reflect a clean state and is not affected by any other queries that use the same UDO.

To allow users to implement parallel algorithms more efficiently, we also provide a small area of thread-local state in every call to a UDO function. The functions accept, extraWork, and process all have a parameter for an unspecified query state. In our implementation for C++ UDOs, the query state provides the member function getLocalState which returns the pointer to the local state, which is a small memory area that is reserved for the current thread. The UDO Query Compiler passes the same local state as argument when a UDO function is called multiple times in the same thread. This allows the user code to recognize when a thread accesses a data structure multiple times and optimize theses accesses.

Linking of Runtime Dependencies

Executing C++ and even C code requires several dependencies to be loaded at runtime. Most notably the C standard library, also called *libc*, and the C++ standard library must be loaded so that the user code can use all features provided by the language. The trivial approach of loading them as shared libraries into the database process is not feasible as this would interfere with the rest of the system. Additionally, this approach does not allow separating the global states of the libraries. Just like global variables explicitly written by the user, we also want to provide full isolation of all runtime dependencies so that two UDOs running concurrently can never interfere with each other.

To solve this issue, our system contains a custom runtime linker, which can load the required runtime libraries as object files or static libraries, and load them into the existing database process. This linker also takes the object file that is generated from the UDO code and links it with the runtime dependencies. Finally, the linker also supports allocating global and thread-local state which allows us to implement the strict separation for global states of concurrent UDOs.

2.4 User-Defined Operators in the Iterator Model

The concept of UDOs is not limited to being implemented in code-generating databases. Query engines based on the iterator model can achieve very efficient execution of UDOs, as well. This approach allows the efficient execution of custom algorithms in traditional disk-based database systems. In this section, we present our implementation of UDOs in Postgres.

In the iterator model, every algebraic operator defines a next function. This function generates a single tuple of the output of the operator. It is called repeatedly until the entire output is generated, which establishes a *pull-based* control flow, that is, the parent operator decides when the child operator should generate the next tuple. This stands in contrast to the architecture of UDOs; the user code can decide by itself when to generate a new tuple of the output and call emit. Thus, UDOs have a *push-based* control flow; children operators push their outputs to their parents. An implementation of UDOs in a query engine based on the iterator model must bridge the gap between the push-based user code and the pull-based execution of the query in which the UDO is contained.

2.4.1 UDO User Compiler

As the UDO User Compiler can generally be independent of the actual query engine of the existing database, it does not have to be reimplemented for every database. Our Postgres implementation uses the exact same UDO User Compiler as our implementation in Umbra (see Section 2.3.1). As such, it supports user-written C++ code. The UDO User Compiler in Umbra can also make use of the LLVM IR to further optimize the code generated for queries that contain UDOs. Postgres does not generate code for entire queries, so it only uses the object files generated by the UDO User Compiler.

2.4.2 UDO Query Compiler

As mentioned above, the main problem that the UDO Query Compiler, which is integrated into the iterator model, must solve is the mismatch between the pull-based query engine and the push-based user code. Conceptually, the implementation of the algebraic operator UDO_a must first call accept with all tuples of the input and then call extraWork and process. In the context of the iterator model, this is in implemented in the next function of UDO_a.

Listing 2.6 shows how the functions of the UDO are called in the iterator model. The next function must call the accept function for every tuple of its

```
1 fn UDO_a.next():
2   if UDO not finished:
3     if state is empty:
4       state = init next_coro()
5     resume state
6     if state is suspended:
7       return state.tuple

8 coro UDO_a.next_coro():
9   while child has tuples:
10    accept(child.next())
11  phase = 0
12  while phase != -1:
13    phase = extraWork(phase)
14  process()
15
16 fn UDO_a.emit_helper(tuple):
17  state.tuple = tuple
18  suspend state
```

Listing 2.6: Implementation of UDO_a in the iterator model. Calls to emit are redirected to emit_helper which suspends the execution of the user code until the next invocation of UDO_a.next.

child operator. These tuples are fetched by calling the next function of the child in the loop in lines 9 and 10. After all tuples are fetched, the extraWork and process functions are called in lines 13 and 14.

The iterator model mandates that the next function should return a single tuple. To avoid memory and runtime overhead, UDO_a.next should not store any intermediate results in temporary buffers. However, the user code can decide arbitrarily when to call emit either in accept or process. Hence, the implementation of UDO_a.next does not have control over when a new tuple is generated. To solve this issue, our implementation *suspends* the execution of the user code as soon as it calls emit. It saves the execution state, such as the stack and the instruction pointer, of the user code and jumps back to where the execution of the user code was first started in line 5. It also remembers the tuple argument that the user code passed to emit so that it can return the tuple in line 7.

Conceptually, we treat the next_coro function (lines 8 to 14) as a *coroutine*. Instead of calling it once and getting a result value once, we first *initialize* it in line 4. This sets up the execution state for the coroutine but does not yet execute the function. In line 5, the coroutine is *resumed*, which means that the execution starts (for a new coroutine) or continues (for an old coroutine that was suspended before). The execution continues until the coroutine is *suspended*. In our implementation, this happens precisely when the user code calls emit.

Note that the coroutine is an implementation detail of the UDO Query Compiler in the iterator model. The user code is not modified in any way to enable the execution of the coroutine. As our implementation in Postgres uses

the same UDO User Compiler as in Umbra, the exact same user code can be used in both systems.

Like in Umbra, our UDO Query Compiler in Postgres has only one implementation for UDO_a that is used for all UDOs. Because different UDOs can use different types for the attributes of the tuples of their input and output, the implementation for UDO_a must be able to call the accept function and implement the emit function for any attribute types. In Umbra, UDO_a generates code, so it can generate the correct code for the corresponding attribute type. However in Postgres, the implementation of UDO_a directly contains a call to accept and it also directly defines the emit_helper function. To be able to still handle UDOs with different attribute types, our UDO Query Compiler in Postgres uses a custom function calling sequence to call the UDO functions and to implement emit_helper. We use low-level machine code to be able to precisely move the UDO tuple attributes from the internal representation in Postgres to the specific registers mandated by the calling convention of the system. Our custom machine code also handles suspending and resuming the execution of the user code by saving and restoring all relevant registers and by switching to a separate CPU stack.

2.5 Evaluation

To evaluate our implementation, we implemented several different functions as UDOs and executed them in Umbra and Postgres. We compared their runtime with equivalent queries implemented in standard SQL or "native" operators of Umbra that generate code. We also ran some queries on Spark, a data analytics engine, and DuckDB, an in-memory database system that uses vectorized execution. Our results show that the runtime of queries containing UDOs is always similar to or faster than all competing approaches while allowing the user to write standard C++ code.

We ran all our benchmarks on a NUMA machine with two Intel® Xeon® E5-2680 CPUs with 14 cores and 28 hyper-threads each and 128 GiB of DRAM per node. Unless otherwise noted, we ran all benchmarks on all 56 hyper-threads.

We ran all queries 10 times and report the root mean square. We ensured that the data sets were loaded into the file system caches before running the queries. Postgres is configured to use 128 GiB of DRAM. Spark is configured to use 64 GiB of main memory each in the driver and executor.

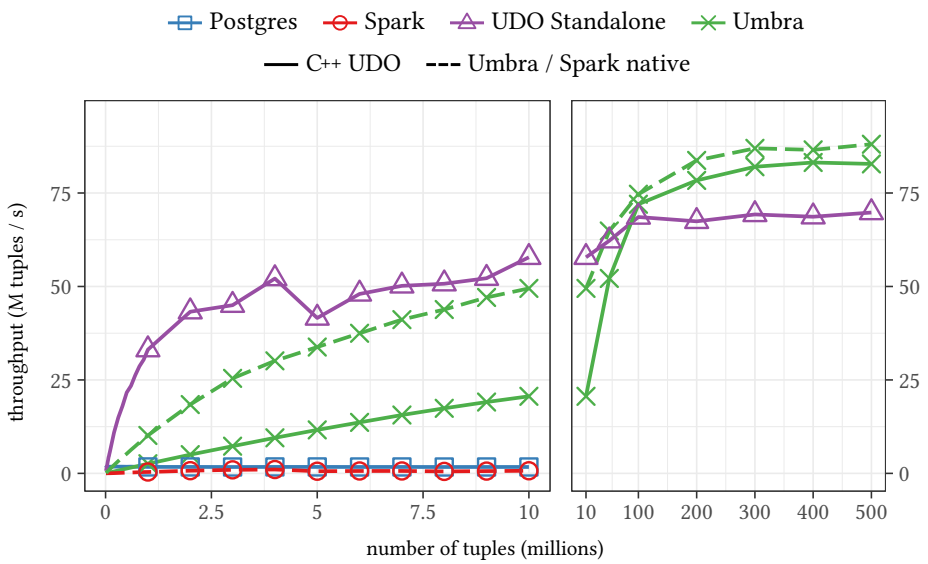


Figure 2.6: End-to-end throughput of different implementations of k-Means. The measurements include the compilation time of the UDO Query Compiler. Overall, the UDO implementations surpass the throughput of Spark and can reach the same throughput of a native, code-generating implementation in Umbra.

2.5.1 Complex Iterative Algorithm: k-Means

One of the main use cases of UDOs is to integrate complex data analytics algorithms that are written in imperative code into the database system. We want to show that implementing such an algorithm as an UDO can achieve the same performance as if it were implemented directly into the database by generating code. For nonexperts of our database system, it is not feasible to implement a new operator on the relational algebra level, especially for complex algorithms. To be able to understand the performance characteristics of UDOs, however, it is best to have a direct comparison of UDO vs. native code-generating code.

For this we chose the comparatively simple k-Means algorithm and implemented it both as a C++ UDO and as a native operator directly into Umbra. Conceptually, both implementations follow this simplified algorithm:

1. Initialize cluster centers by randomly sampling k points.

2. For each point select the cluster center with the smallest distance and update the point's cluster id.

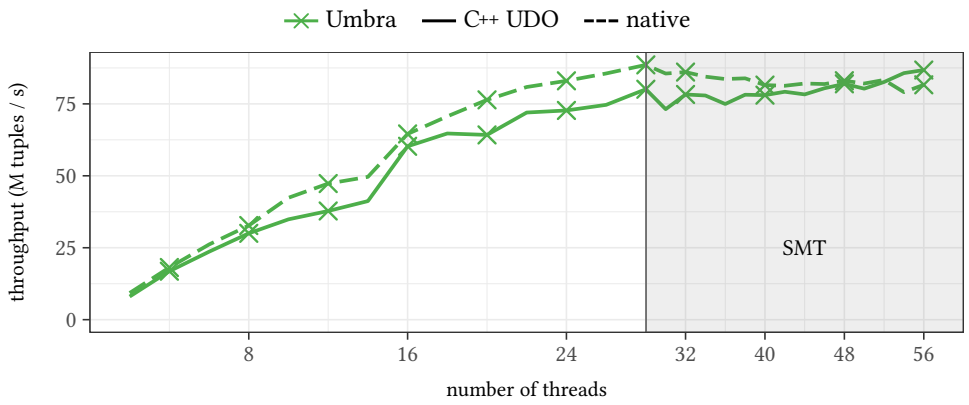


Figure 2.7: Scalability of runtime throughput of k-Means in Umbra. Here, the significant compilation time of C++ UDOs is not included. In pure runtime, the C++ UDO can scale just as well as the native code-generating implementation in Umbra.

3. For each cluster find all points with the same cluster id and calculate the new cluster center.

4. Repeat steps (2) and (3) 10 times.

Usually, the iteration is stopped when a cancellation criterion is met, such as a required minimum movement of the cluster centers. As this depends heavily on the initialization of cluster centers, which is random in our implementation, instead we always iterate exactly 10 times to ensure that the runtimes of all approaches are directly comparable.

We use synthetically generated two-dimensional points that are clustered in eight clusters as the data-set for our benchmarks. The points within each cluster are drawn from a normal distribution where each cluster has separate means and variances. As the number of iterations is fixed to ten, all executions always must scan all points exactly ten times and compare them to the current cluster centers of all eight clusters.

The implementation effort required for both approaches – one natively in Umbra, using code-generating code and database internals, and the other as a self-contained UDO – highlights the qualitative advantage of using UDOs. The Umbra implementation extends the relational algebra by adding a new algebraic operator that implements k-Means. Adding a new operator requires modifying several parts of the system such as the SQL parser, the optimizer, and of course the query engine. In its core, the implementation consists of about 500 lines of code of which many generate one or more instructions. The UDO implementation has 400 lines of self-contained C++ code. This code only uses

features from the C++ standard library and some auxiliary data structures. As the UDO does not use any database-specific code, it can easily be compiled into a standalone executable program which also makes debugging much easier.

Spark allows for even more straightforward implementation of data analytics algorithms such as k-Means as they are directly built-in into the system. At its core, the Spark code consists of only one call to the existing k-Means clustering function and a few more lines to set up the experiment. When compared to writing C++ code for UDOs, Spark enables users without in-depth knowledge of programming languages such as C++ or Rust to do data analytics. However, Spark cannot reach the performance of queries using UDOs.

The main advantage of using UDOs in existing database systems instead of specialized data analytics systems like Spark is their execution speed. Figure 2.6 shows the throughput of different implementations of the k-Means algorithms as described above for data sets containing up to 500 million tuples. We ran the C++ UDO in our UDO implementations for Umbra (multi-threaded) and Postgres (single-threaded). Additionally, we tested the performance of the standalone executable that is mainly used for debugging (see also Section 2.2.2). As a comparison, we provide measurements of a native k-Means operator of Umbra, which generates efficient code and makes heavy use of database internals to achieve good parallelization.

For small data sets that contain up to 10 million tuples, the C++ UDO executed in Umbra cannot yet compete with the native implementation. Because executing the C++ UDO incurs a compilation overhead of almost 400 ms while the entire query only runs for 480 ms, the end-to-end throughput is dominated by the compilation overhead. The native implementation in Umbra, on the other hand, only has a compilation overhead of approximately 80 ms. When the same queries are executed on a larger data set, the compilation overhead decreases relative to the total query runtime. Thus, the C++ UDO in Umbra can reach a throughput similar to the native Umbra implementation for the large data sets with 400 million tuples or more.

Since the standalone executable is compiled in advance, it has no compilation overhead. For small data sets this allows the standalone executable to achieve a higher end-to-end throughput than Umbra. However, for larger data sets Umbra's morsel-driven parallelism strategy and leads to unmatched performance.

The execution of the C++ UDO in Postgres is not able to reach Umbra's performance because Postgres only runs single-threaded whereas Umbra uses all available 56 hyper-threads. Nevertheless, the execution of the UDO in Postgres achieves a single-threaded throughput that is comparable to the throughput of each thread in the Umbra implementations; the total throughput of the execution in Umbra is between 50 and 100 times higher than in Postgres while using 56 times as many CPU threads. Since both Umbra and Postgres use the exact same

UDO, both execute mostly the same machine code, which allows Postgres to achieve similar performance on its single thread.

Even though Postgres only uses a single thread, it still achieves a higher throughput than Spark which also uses all hyper-threads. As for the other implementations, in Spark we select random points to initialize the cluster centers and iterate exactly 10 times. The large performance difference between Spark and the other approaches has several reasons. The C++ UDO executes efficient machine code which is generated from an optimizing C++ compiler from a C++ program that stores all its state in main memory. Spark needs to make sure that the k-Means algorithm can scale out onto large clusters for data sets which cannot easily fit in main-memory. Thus, it often needs to materialize intermediate states to synchronize all workers and to allow for the states to potentially be written to disk. The C++ UDO implementations on the other hand can heavily benefit from cheap intra-process synchronization.

To demonstrate the scalability of the UDO implementation onto many threads, we compare the runtime of the k-Means query that processes 500 million tuples while varying the number of threads. Figure 2.7 shows the throughput excluding the compilation overhead for this query for the native and the UDO implementations in Umbra. Since the UDO is also executed using Umbra's morsel-driven parallelism strategy, it scales similarly to the native Umbra operator. In general, user-written queries have the potential to scale as well as native database operators as long as their implementation is reasonably efficient and makes use of lock-free data structures as opposed to relying on mutual exclusion.

Overall, the experiments show that a k-Means UDO can reach a throughput comparable to a native operator even though the k-Means UDO is implemented using only standard C++ constructs. The C++ code is compiled by using the Clang compiler, thereby enabling all available optimizations. Umbra generates the low-level code for the native k-Means operator directly without going through a higher-level language first and uses fewer expensive optimizations than the Clang compiler. For UDOs, spending more time on compiling C++ is not a performance issue as this is only done once when the create function statement is executed. A code-generating database must balance the trade-offs between spending more time to generate faster code.

2.5.2 Linear Regression

To benchmark another algorithm used in data analytics, we tested simple linear regression using the least squares error function. We implemented it as an UDO, natively in Umbra, Spark, and SQL. SQL offers the functions regr_slope and regr_intercept, which can be used for simple linear regression on a linear

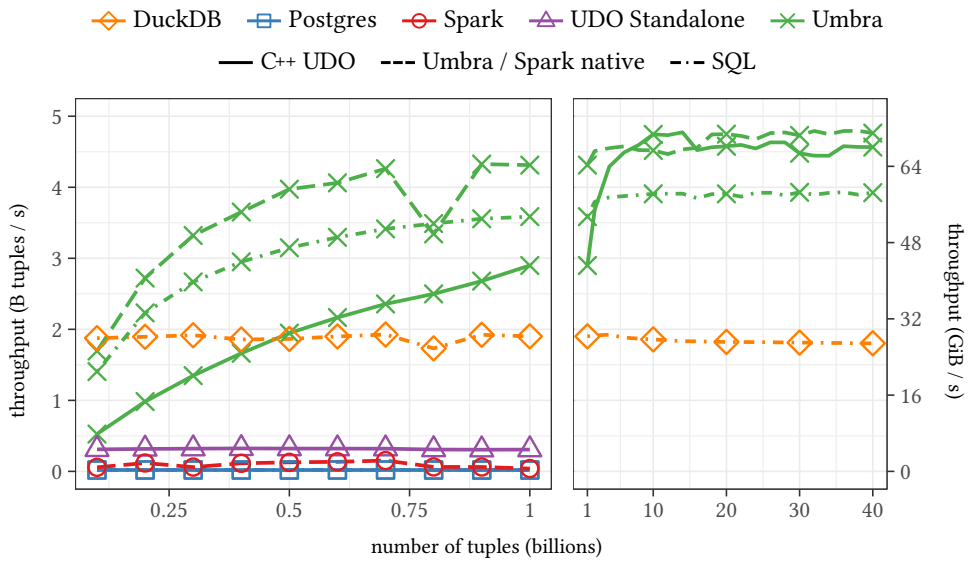


Figure 2.8: End-to-end throughput of different implementations of simple linear regression. For aggregation-like algorithms, all approaches using code-generation perform best for larger data sets.

function. All Non-SQL implementations instead use a polynomial of degree 2 as a target function to highlight the use case for hyperparameter tuning, which often requires slight changes to existing algorithms.

Our implementations solve the following problem: For given pairs of values x, y choose a, b, and c while minimizing the error term $\sum_i (a + bx_i + cx_i^2 - y_i)^2$. This problem has the following closed-form solution:

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} \sum 1 & \sum x & \sum x^2 \\ \sum x & \sum x^2 & \sum x^3 \\ \sum x^2 & \sum x^3 & \sum x^4 \end{pmatrix}^{-1} \cdot \begin{pmatrix} \sum y \\ \sum xy \\ \sum x^2 y \end{pmatrix}$$

To compute this efficiently, first, all sums need to be calculated. Calculating the sums requires little synchronization and should scale very well. When multiple threads compute the sums, each thread can independently compute partial sums of the part of the input data it sees. The values for a, b, and c can be determined at the end by summing up the partial sums of all threads and calculating the inverse of the matrix.

Figure 2.8 shows the end-to-end throughput for all implementations. As the linear regression is essentially an aggregation, the very fast code-generating query engine used in Umbra surpasses all other approaches. This result, of course, is not due to the use of UDOs but because generating code is the most efficient approach to compute this algorithm. Still, the throughput of the UDO

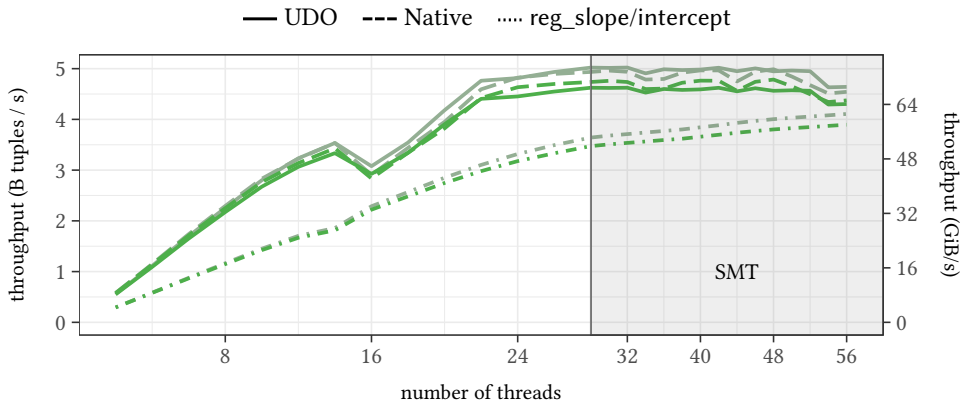


Figure 2.9: Scalability of simple linear regression in Umbra. The faded lines show the raw throughput without compilation overhead. The UDO query reaches a raw throughput of over 70 GiB/s but its compilation overhead brings down the total throughput.

implementation in Umbra is very similar to the native, code-generating operator. Especially for larger data sets were the compilation overhead of using a UDO becomes insignificant relative to the total query runtime, the UDO implementation in Umbra reaches the same throughput as the native operator. Both can process almost five billion tuples per second. Each tuple consists of the values x and y which are represented as 8-byte double precision floating-point numbers, so the raw memory throughput is close to 70 GiB/s. The maximum available memory bandwidth on our system is approximately 110 GiB/s which means that we can nearly fully utilize the available resources of modern hardware.

The in-memory database system DuckDB with its vectorized query engine can calculate aggregations very efficiently, as well. As vectorized query engines have no compilation overhead, computing the linear regression in DuckDB can be faster for small datasets where the significant compilation overhead of a query containing a UDO still dominates the execution time. For larger data sets, DuckDB's memory throughput of 27 GiB/s can still utilize a large portion of the available memory bandwidth.

The performance of the linear regression in Postgres, for both the SQL regression functions and the UDO function, is significantly worse. In Postgres, the cost of fetching every tuple from the base table that contains the input data makes up most of the runtime. Since for every tuple only few numeric operations are required, using an UDO does not improve the performance. In fact, in Postgres when using the UDO implementation, every input tuple of the UDO results in an additional function call which makes the UDO query even slightly slower than the SQL query. The standalone UDO executable faces a

similar problem; it reads all the input data from a CSV file. Parsing a tuple from a line of the CSV line is significantly more expensive than computing the partial sum for the linear regression. Therefore, the runtime of the standalone executable is dominated by the time it takes to parse up to 30 GB of CSV data.

In Spark, the linear regression is implemented using one map operation followed by one reduce operation. These two primitives make it very easy for the Spark executor to distribute the query execution automatically onto large clusters. However, on a single machine, Spark cannot compete with systems that keep their intermediate state in main-memory and use cheap inter-process synchronization.

We also tested the scalability of the different implementations of linear regression in Umbra, as can be seen in Figure 2.9. We ran the query on the data set with $40 \cdot 10^9$ tuples (640 GB) and varied the number of threads. The throughput of the UDO and the native implementation increase nearly linearly with the number of cores in the beginning but decreases when executing the query on more than 14 CPU cores. Our benchmark system is a NUMA system with two CPU sockets that have 14 cores each, so when more than 14 cores are used, the query runs on both sockets. When a query runs on multiple sockets, the synchronization overhead increases significantly due to the communication latency between the sockets. The communication overhead can be compensated when more than 18 cores are used.

For more than 24 cores the throughput does not increase significantly anymore, even when all 56 available threads are used. Umbra generates very efficient code which can fully saturate the available resources. The generated code contains several floating-point instructions that fully saturate the execution units on the CPUs. Since all execution units are used when executing the query without SMT, adding more threads using SMT cannot significantly increase the throughput. Also, as mentioned above, the query execution is close to hitting the limit of the main memory bandwidth, as well.

Figure 2.9 also again shows the negative effect of using UDOs; when the total throughput (faded lines) is compared to the throughput excluding compilation overhead (green lines), the UDO query can reach the same performance as the native implementation only when excluding the compilation overhead. Like in the k-Means experiment, using a UDO leads to a higher compilation overhead which brings down the total throughput slightly.

2.5.3 Imperative Programming

To highlight the advantages of using an imperative language to process queries, we tested a query that generates multiple output tuples for every input. The input contains a string that is a comma-separated list of numbers and words.

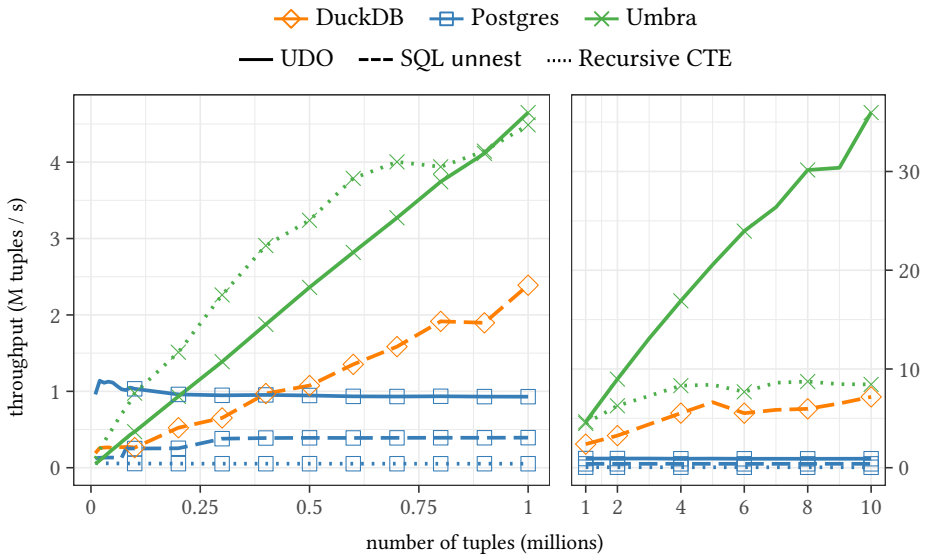


Figure 2.10: End-to-end throughput of queries that split comma-separated values into individual tuples. When the data set is large enough so that the compilation overhead becomes insignificant, the UDO in Umbra outperforms all other approaches.

The UDO parses this string, takes out all the numbers, and generates a new tuple for each number.

While the C++ code for this algorithm is only a few lines, implementing this query in SQL is very tedious. Listing 2.7 shows how recursive CTEs can be used to formulate this conceptually simple query. Every input tuple can generate an arbitrary number of outputs; however, since SQL does not support loop statements, the SQL query uses recursion instead. To ensure that the cast is not evaluated for invalid strings, it uses a case when statement. Moving the similar to expression to the where clause would allow the database to reorder the expressions which could lead to runtime errors when the cast is evaluated before the similar to expression.

In DuckDB, this recursion can be prevented by using the special function unnest, which directly converts an array of values into multiple tuples. Postgres has a similar function called string_to_table, which splits a string by a given separator into multiple tuples.

For such an algorithm that is inherently imperative, the UDO can easily outperform the SQL versions for larger data sizes. Figure 2.10 shows the throughput for ad-hoc queries, so it includes the compilation time. For data sets containing at least one million tuples, the UDO achieves a much better throughput than all other approaches. The C++ code for the UDO contains a single loop that can

```
with recursive split_arrays(name, value, tail) as (
    select c.name, NULL, c.values as tail
    -- schema: array_values(name text, values text)
    from array_values c
    union all
    select s.name,
        case
            when s.comma = 0 then s.tail
            else left(s.tail, s.comma - 1)
        end as value,
        case
            when s.comma = 0 then ''
            else right(s.tail, -s.comma)
        end as tail
    from (
        select s.*, position(',' in s.tail) as comma
        from split_arrays s
    ) s
    where s.tail != ''
)
select name,
    case
        when value similar to '[0-9]+' then cast(value as bigint)
        else null
    end as value
```

Listing 2.7: Splitting comma-separated strings into individual integer tuples using recursive CTEs in SQL.

efficiently extract all numbers from a string and create a new tuple for every number. All other approaches cannot use such an efficient loop.

For smaller data sets, some implementations can reach a better performance. Even when the control flow from the imperative C++ program must be simulated by using a recursive CTE, Umbra is able to execute the query very efficiently. Umbra optimizes all string functions used by the query shown in Listing 2.7 (left, right, and position) so that they do not create copies of the string and has a specialized implementation for recursive SQL queries. Also, the recursive SQL query in Umbra has a compilation time of only 80 ms while compiling the UDO query takes 2 s, which leads to a higher total throughput for smaller data sizes when the compilation overhead still dominates the execution of the UDO query.

Data Set	Tuples	Size	Generate	Insert
points (`double`, `integer`)	10M	354 MiB	0.4 s	5 s
points (`double`, `integer`)	100M	3.5 GiB	2 s	52 s
comma-separated (`text`)	10M	700 MiB	13 s	15 s
comma-separated (`text`)	100M	7 GiB	123 s	148 s

Table 2.2: Runtimes to generate and insert different benchmark data sets by using UDOs.

In Postgres, the UDO always performs better than the other approaches. Even a query that uses the special function string_to_table cannot reach the same performance as the UDO query. Thus, using UDOs can be beneficial even in traditional database systems. However, other modern database systems such as DuckDB can outperform UDOs executed in Postgres.

2.5.4 Data Generation

Our implementation does not require UDOs to have any inputs. Therefore, it is possible to write "output-only" functions that can take scalar arguments and then return a stream of tuples, that behave similar to functions like generate_series in SQL.

When writing benchmarks, it is very common to synthetically generate all data that is used by the test queries. Such data generators are usually written in an imperative language. Examples for this are C for the widely used benchmarks TPC-H and TPC-DS, and Python, which has a broad range of libraries for that purpose. The disadvantage of this approach is that the data is generated first by a program and then it must be inserted into a database. With UDOs this can now be implemented directly in the database. No extra steps to export and import data are required.

All test data for our benchmarks was created by using UDOs which were directly used in INSERT statements. Table 2.2 shows the runtime of some insert statements for different data sets. The sizes shown in the table refer to the size the generated data set would have if exported as a CSV file. For the "comma-separated" data set, which contains only strings, insert queries using an UDO as a source can process around 40 MiB/s to 50 MiB/s while the "points" data set can process around 70 MiB/s. This result is mainly a limitation caused by the insert statement, not by the UDOs themselves. The "Generate" column shows how long it takes to just generate the tuples and immediately discard them. The UDO that makes heavy usage of string operations can reach up to 80 MiB/s. The points dataset that does not use any strings reaches about 1.7 GiB/s which means it can keep up with modern SSDs and the runtime of the data generation

will most likely be dominated by the actual insertion of tuples into a physical relation.

All cases have in common that they do not require writing any intermediate files to disk that must then be read again by the database system. Especially for larger data sets, this approach prevents wasting space and time to store data on disk that is quickly discarded. This approach yields no disadvantages for the user as this data generation can be written in standard, imperative code.

2.6 Summary

In this chapter, we presented *User-Defined Operators* (UDOs) – a novel framework to efficiently integrate and execute custom algorithms in modern databases. UDOs can achieve very high throughput, which is competitive with main-memory databases. Furthermore, because UDOs are integrated into query engines of existing RDBMS, all ACID properties can be preserved. Nevertheless, users are not required to know any database internals. Instead, they are provided with an easy-to-use API.

We implemented UDOs in Postgres and Umbra – our code-generating database. Our evaluation shows that queries containing UDOs can achieve throughputs similar to main-memory databases. Even in disk-based systems such as Postgres, the execution of UDOs is very efficient. Thus, UDOs enable users to integrate custom algorithms for data analytics directly into databases very efficiently.

In Umbra, the excellent performance of UDOs is achieved by directly inlining user code into the code generated by the database system for other parts of the query. The code generated by the built-in relational operators is written by experts and is carefully tested. This means that bugs tend to be rare so that arbitrary SQL queries can be safely executed. UDOs, however, are potentially written by users less familiar with the system or even malicious users. Since the user code is executed with the same capabilities and privileges as the entire query execution system, bugs in an UDO can crash the entire database system, and malicious actors can use this as an easy privilege escalation. To prevent memory bugs, users could use a programming language such as Rust [22b]. The Rust compiler uses LLVM, and can guarantee memory-safety. However, even for Rust programs, a static analyzer cannot guarantee that it does not contain potentially malicious code. Therefore, UDOs as presented in this chapter must not be accessible to untrusted users.

CHAPTER 3

Safe Execution of User-Defined Operators with WebAssembly

In Chapter 2 we presented User-Defined Operators which allow integrating arbitrary user code into existing database systems. UDOs make it very easy to add new functionality into RDBMS that interoperates seamlessly with existing SQL features. In compiling database systems, the user code can be integrated directly with the code generated by the database system. Integrated code leads to near zero-overhead execution of arbitrary algorithms.

Because the code generated for queries containing UDOs has no clear boundaries between user code and system-generated code, it is also not easily possible to effectively restrict the capabilities of user code. The user code is essentially executed with permissions of the database superuser. Thus, malicious users can potentially gain unrestricted access to the database system if they are allowed to create new UDOs. Even trusted users can accidentally bring down the entire database system if their code contains bugs that would normally lead to program termination, such as invalid memory accesses.

The database system needs to be shielded from the execution of arbitrary, untrusted code. With the introduction of UDFs in IBM's database system DB2, IBM established the concept of *fenced* execution of untrusted code [Cha96, Ch. 4.4]. Fenced UDFs are executed in an entirely separate process with its own address space. The database system uses an explicit communication protocol to send data to and retrieve data from the UDF. In contrast, so-called *unfenced* functions – UDFs that have unrestricted access to the database process, such as our UDOs – can communicate implicitly with the system as they can directly access all memory that contains the input for the UDF.

The explicit communication and execution in a separate process add significant overhead to fenced UDFs. Our UDOs, on the other hand, are very efficient

but cannot be fenced. Directly integrating UDOs into the query execution engine is a design choice that enables their efficient execution, whereas fencing untrusted code is necessary in existing systems to prevent security issues. Currently, database administrators need to chose between slow but secure, and fast but insecure UDFs.

In this chapter, we present an approach to safely execute UDOs without resorting to classical, fully fenced execution. Ideally, we want users to be able to add custom algorithms to the database with little performance overhead while offering the same strict security guarantees as existing systems. Also, we want database systems in the cloud to be able to use UDOs. As cloud providers manage the storage and processing of data on behalf of their customers, they must ensure that customers cannot harm their systems even when the cloud system executes arbitrary code provided by potentially adversarial customers.

As discussed above, using fenced execution is not an option if we want to maintain the efficiency of UDOs. Instead, we must guarantee that executing user code is safe even when it is combined with code generated by a database system. Because we want users to be able to write UDOs in any programming language, it is not feasible to statically analyze user code for safety prior to execution. In fact, it has been shown that analyzing arbitrary programs to rule out denial-of-service vulnerabilities requires solving the halting problem which is undecidable [Coh87].

We propose using *WebAssembly* to guarantee safe execution of arbitrary code. WebAssembly is a low-level language that was originally designed for safe execution of untrusted code in web browsers [Ros+18]. Interestingly, web browsers faced issues similar to the execution of UDFs in database systems; browser engines need to execute arbitrary, untrusted code. Before the introduction of WebAssembly, JavaScript was the only programming language supported by all major browsers. JavaScript is a garbage-collected, interpreted programming language which requires a runtime to be executed. As such, it does not allow access or branches to arbitrary memory locations. However, similar to fenced execution of UDFs, the execution of JavaScript is significantly slower than native machine code. Even though modern runtimes such as Mozilla's SpiderMonkey[1] and Google's V8[2] have very sophisticated just-in-time compilation engines, they are not able to keep up with the ever growing demand of high-performance code execution in browsers. Rossberg et al. specifically mention audio and video processing as well as games as the main motivation for the development of WebAssembly.

[1] `https://spidermonkey.dev/` (Accessed: 21 June 2023)
[2] `https://v8.dev/` (Accessed: 21 June 2023)

The WebAssembly language is designed to be a compilation target for high-level languages. Today, C, C++, and Rust can be compiled to WebAssembly. Both the Clang compiler (for C and C++) and the Rust compiler use the LLVM project which supports WebAssembly as a target language. Also, all major browsers support WebAssembly and it has gained popularity in several non-web environments [SM21]. Thus, we believe WebAssembly to be a good choice as it is already supported by a few widely-used programming languages and is likely to become even more relevant in the future.

In the following, we first show an overview of the WebAssembly language. We present the structure of a WebAssembly program and the stack-based semantics of WebAssembly instructions. Then, we briefly discuss how WebAssembly can be used to safely execute UDOs in compiling database systems.

3.1 The WebAssembly Language

WebAssembly is a low-level language that is designed to be used mainly by compilers. It consists of individual instructions and has no concept of more complex expressions, user-defined types, function overloading, and other features commonly found in higher-level languages. Instead, higher-level languages can be compiled to WebAssembly automatically, as mentioned above. In this section we will show an overview of the WebAssembly language and its unique properties. Our work is based on the W3C First Public Working Draft 19 April 2022 of the WebAssembly Core Specification [Ros22].

3.1.1 WebAssembly Modules

A WebAssembly program consists of a WebAssembly *module*. A module contains all the information that is required to execute the program. The compiler usually outputs a single WebAssembly module when it compiles a higher-level program. Modules are stored in a binary format that efficiently encodes all its contents. As WebAssembly has been designed primarily for execution in web browsers, the binary format is designed to be space efficient to reduce the amount of data that is transferred over the network. Also, module files are structured internally to allow for streaming compilation, i.e., web browsers can start executing a WebAssembly module before its contents are completely downloaded.

Functions: The most important components of a module are its *functions*. Each module can contain multiple functions which are referenced by a unique index within the module. Unlike functions in high-level languages, WebAssembly functions usually do not have names. Every function in a module specifies

its argument and return types, its local variables, and its implementation. The implementation of a WebAssembly function consists only of a sequence of instructions. The instructions operate on the function arguments, the local variables, the global variables defined in the module, and the memory. They can also call other functions from the module.

Global Variables: A module can define *global variables*. Every global variable has a type, and its value can be read and modified by all functions. Like functions, global variables have no names but are identified by a unique index. One notable difference to global variables in other programming languages is that conceptually global variables in WebAssembly are not stored in addressable memory. As such, WebAssembly programs cannot take the address of a global variable or use memory instructions to access them. Global variables can only be accessed by the special instructions global.get and global.set.

Exports: To use a WebAssembly module, code outside of WebAssembly needs to interact with WebAssembly modules, such as calling a WebAssembly function or accessing a global variable. For that purpose, WebAssembly modules can define *exports*. An export definition references either the unique index of a function or of a global variable. The export also has a name so that a program using the WebAssembly module can find the index of an exported function or global variable.

Imports: Similarly, WebAssembly modules can define *imports* to access functions from outside of the WebAssembly module. An import specifies the name of the imported function, and its argument and return types. Like regular functions, every function import receives a unique index that can be used to identify the imported function within the WebAssembly module. Imported functions do not have to be WebAssembly functions defined in other modules. In fact, imports are intended to be used mostly to import functions that allow a WebAssembly program to directly interact with the system outside of its module. For example, an imported function could be a print function that allows a WebAssembly program to write text to the terminal, or a UDO function such as accept that interacts with the outside query execution engine.

3.1.2 WebAssembly Memory

WebAssembly instructions conceptually operate on one contiguous memory area. The memory is byte-addressed by 32-bit addresses, and the memory instructions have an additional 32-bit immediate that is added to the address.

Thus, the maximum addressable memory size is 2^{33} B = 8 GiB, though the upper 4 GiB can only be accessed using constant instruction immediates, making only the lower 4 GiB generally accessible. A module can specify an additional limit for the memory size of less than 8 GiB and the minimum memory size required to execute the module.

At runtime, the size of the WebAssembly memory is set to the minimum specified in the module or to zero if the module has no minimum. After that, the memory can be grown at runtime by special memory instructions as long as the new size does not exceed the limits. However, there are no instructions to decrease the memory size.

An additional constraint for the size of the WebAssembly memory is that it must be a multiple of the WebAssembly page size. The page size is defined as 2^{16} B = 64 KiB by the specification. This makes it similar to the page sizes of existing CPU architectures: On x86-64, the page size is usually 4 KiB, modern ARM architectures (ARMv7-A, ARMv8-A) support 4 KiB and 64 KiB pages.

Since the WebAssembly memory is contiguous, the first memory address is always 0. When a new page is added, the next 2^{16} memory addresses become usable implicitly. The specification requires that all programs that implement the execution of WebAssembly must make sure that invalid memory accesses must terminate the execution the WebAssembly program immediately.

3.1.3 WebAssembly Stack

In addition to reading from and writing to memory, WebAssembly programs mostly use values from the *stack*. The WebAssembly language is stack-based, which means that its instructions do not use registers or explicit operands. Instead, every instruction conceptually pushes and pops values to and from a stack. This differentiates WebAssembly from modern CPU architectures which are register-based. While CPUs do usually have a stack and a stack register, they use the stack as a special memory region with cheap allocation and deallocation. CPU instructions still operate on registers and the stack must be accessed explicitly with load and store instructions. However, stack-based low-level languages are often used in interpreters such as the Java Virtual Machine, Microsoft's CLR, which is used for .NET programming languages like C#, or the CPython interpreter.

The WebAssembly stack has two kinds of entries: values and labels. Values are pushed and popped by arithmetic and memory instructions. A memory load instruction, for example, first pops a value from the stack that contains the memory address. Then, it reads the value at that address and pushes that value to the stack. Labels are used by control-flow instructions: Every block of WebAssembly instructions starts by pushing a label to the stack. A branch

instruction can then pop entries from the stack until it pops the label it is targeting.

The WebAssembly stack has a few interesting properties that guide implementations in ensuring the safety of the execution. Unlike CPU stacks, the WebAssembly stack does not lie in addressable memory. Values that live on the WebAssembly stack cannot be referenced by any memory address, so load and store instructions cannot access the stack. Thus, common stack overflow vulnerabilities that modify the control flow by overwriting return addresses on the stack are not possible.

Also, all stack accesses can be statically analyzed without executing the program because every instruction operates on a known, fixed number of stack entries. Therefore, a static analyzer can determine the maximum number of stack entries required for every function and can analyze whether the program is guaranteed to never pop from the stack if it is empty. This also includes the stack state across control-flow instructions. After a conditional branch instruction, for example, the stack state must be consistent in all cases. When the branch is not taken, the execution continues with the same stack. If it is taken, the stack is unwound to the target label. All branch instructions that target the same label must leave the stack in the same layout, i.e., the same number of entries with the same types.

3.1.4 WebAssembly Types and Values

As described above, the WebAssembly stack can contain values. Every value has one of the following types: one of the four *number types*, a *vector type* that extends a value type, or a *reference type*. No other types are supported by the stack or the instructions directly. User-defined types defined in higher-level languages must be accessed by combining multiple instructions, like when these languages are compiled to real machine code.

Table 3.1 shows all supported types. For numbers, only 32-bit and 64-bit integers and floating-point values are supported. Arithmetic WebAssembly instructions cannot operate on integers smaller than 32 bits. However, some memory instructions support smaller integer sizes and implicitly convert them to one of the supported sizes. Thus, higher-level languages that support smaller integer types can still be compiled to WebAssembly.

Since little-endian two's complement integers and IEEE 754 floating-point numbers are the de-facto standard in modern CPU architectures, the number types of WebAssembly are based on them as well. Integer types do not specify the signedness of their values. Instead, most instructions that operate on integers specify whether they interpret the value as signed or unsigned integers.

Name	Description
Number Types	
i32	32 bit integer (little endian, two's complement)
i64	64 bit integer (little endian, two's complement)
f32	32 bit floating-point number (IEEE 754)
f64	64 bit floating-point number (IEEE 754)
Vector Types	
v128	128 bit vector containing values of any number type
Reference Types	
funcref	A reference to a function, used for indirect calls
externref	A reference to an "external" object

Table 3.1: Value types supported by WebAssembly

Values of type v128 are 128 bit wide. Vector instructions that operate on these values interpret them either as four i32/f32 values, or as two i64/f64 values. They are designed so that modern CPUs can store a WebAssembly vector value in a single CPU vector register, e.g., XMM0-15 on x86-64, or v0-v31 on ARMv8-A.

Function reference values are used by the indirect call instruction. A regular call instruction encodes the function it calls directly, while an indirect call determines the called function using a funcref value. Since funcref values are guaranteed to always be valid, it is not possible to corrupt the execution by calling into an arbitrary address. All instructions that create function references ensure that the new reference is either null, i.e., it references no function, or references an existing function with known argument and return types. To make this possible, no instructions exist to convert arbitrary memory addresses to function references.

Similarly, externref values reference objects that live outside of the WebAssembly module. WebAssembly has no instructions that can directly inspect externref values or convert them to memory addresses. However, functions can take externref values as arguments and return them. Since WebAssembly programs can also import functions, an implementation can provide functions that can handle these references. A use case envisioned by the specification is that a browser could pass references to complex JavaScript objects to a WebAssembly function as an externref value. The WebAssembly function can then extract information from this object by calling functions provided by the browser that take the reference as an argument.

```c
// Example function written in C
double load_add(double* ptr) {
    return ptr[0] + ptr[1];
}
```

```
;; Define a WebAssembly function called "load_add" with one para-
;; meter of type i32, which is used as a pointer by the function.
(func $load_add (result f64) (param i32)
  local.get 0 ;; Push the argument onto the stack
  f64.load 0 ;; Load a 64 bit float from memory
  local.get 0 ;; Push the argument onto the stack again
  f64.load 8 ;; Load a 64 bit float from memory with offset 8
  f64.add ;; Add the two values loaded from memory
  ;; Implicitly return the result of f64.add
)
```

Listing 3.1: Example of a small function in C and its translation to WebAssembly. It loads two f64 values from memory, adds them, and returns the result.

3.1.5 WebAssembly Instructions

All WebAssembly functions contain a list of WebAssembly instructions. Like most low-level assembly languages, each individual instruction operates on only a few values and generally performs exactly one operation. WebAssembly has arithmetic instructions that operate on integers and floating-point values, as well as control-flow instructions. However, WebAssembly instructions also have several properties that set them apart from common CPU architectures, such as *structured control flow*, i.e., control-flow instructions cannot jump to arbitrary code, and their *stack-based semantics*, i.e., most instructions operate on an implicit stack of values.

Numeric Instructions

To operate on values of number type, the WebAssembly specification defines several numeric instructions. For integer values, WebAssembly has instructions for all basic arithmetic and comparison operators. In addition, several bitwise operations are defined, including advanced instructions such as bitwise rotation, clz/ctz (“count leading/trailing zeroes”), and popcnt (counts the number of 1-bits). It is also possible to convert between integer sizes by using conversion instructions.

All integer instructions are designed to be able to be executed in a single or few CPU instructions. All reasonable CPUs support basic arithmetic and

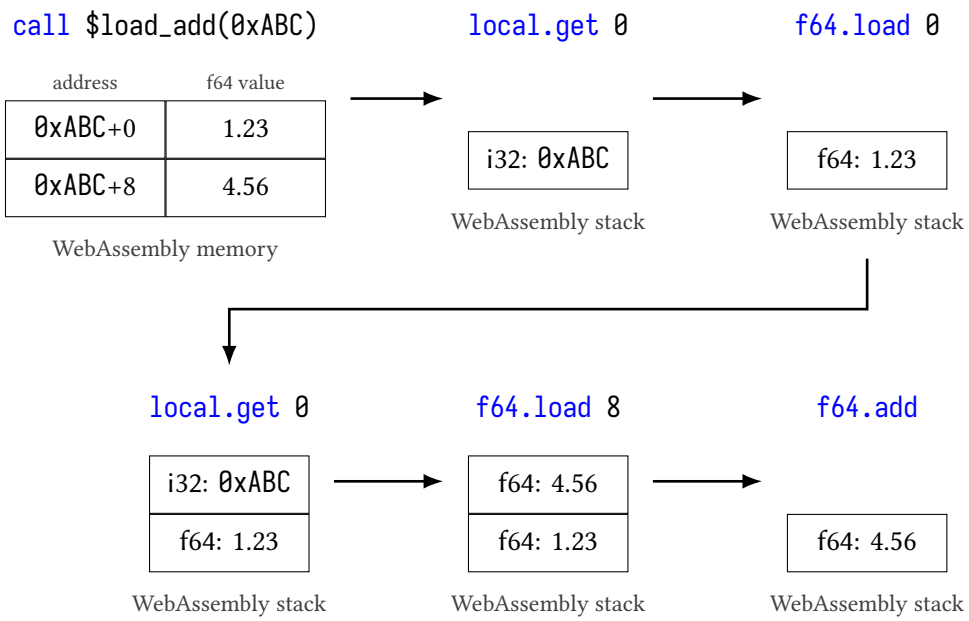


Figure 3.1: State of the conceptual WebAssembly stack after every instruction when executing the function load_add from Listing 3.1.

comparison, and even special instructions such as clz, ctz, and popcnt are found on most modern architectures. Again, x86 and ARM in particular have a CPU instruction for every WebAssembly integer instruction.

Basic arithmetic and comparison operations are also defined for floating-point values. They are based on the semantics of the IEEE 754 standard [19]. Thus, the bit representation of floating-point values in WebAssembly is compatible with their bit representations on real hardware.

To ensure safety, WebAssembly imposes some additional restrictions on the numeric instructions. As for all instructions, it can be statically proven that every numeric instruction receives the correct number of values with the correct types: Integer instructions, for example, usually pop up to two integer values from the stack and push the result back onto the stack. Also, no instructions leads to unpredictable results. The WebAssembly specification achieves this by precisely defining the behavior of instructions in cases where real hardware offers little or no guarantees. For example, arithmetic operations on integers have well-defined overflow, division by zero must immediately terminate the execution of the WebAssembly program, bitwise shift operations have predictable results even when shifting by more than 64 bits, and all instructions operating on floating-point values have well-defined results when operating on NaN or infinity values.

Memory Instructions

To interact with the WebAssembly memory, functions can use several different load and store instructions. These instructions can pop integer, floating-point, or vector values from the stack and write them to memory, or read them from memory and push them onto the stack. To support reads and writes smaller than the size of the smallest integer type in WebAssembly (32 bits / 4 B), load and store instructions can also read and write individual bytes by implicitly converting them from or to 32-bit integers.

All load and store instructions use 32-bit addresses which refer to the contiguous WebAssembly memory that always starts at address 0. The address of a memory instruction is computed by popping an i32 value from the stack and adding the 32-bit immediate that is encoded directly in the instruction. As mentioned above, semantically, every load and store instruction is also required to ensure memory safety by checking that the address it uses is valid before performing the memory access.

New memory can be allocated by using the memory.grow instruction. It increases the size of the WebAssembly memory by a multiple of the WebAssembly page size (64 KiB). The new pages always contain zero bytes and their address range always extends the current usable address range so that the entire WebAssembly memory is always one contiguous area of memory.

Listing 3.1 shows an example of a function that loads two 64-bit float values from memory, adds them, and returns the result. In C, this function can easily be written in one line. As a low-level language, WebAssembly naturally uses several instructions to define an equivalent function. The example shows that the local.get instruction pushes the function argument, which in this function contains a memory address, onto the stack, the f64.load instruction loads a 64-bit float value from memory, and the f64.add instruction adds two f64 values.

The example also shows that the instructions do not have explicit operands. Figure 3.1 illustrates how the implicit operands of the instructions are passed via the WebAssembly stack. The memory load instruction f64.load in line 9, for example, reads the memory address from which it should load the f64 value from the stack. This address originates from the instruction local.get in line 8, which takes the argument of the function, `0xABC` in the figure, and pushes it onto the stack.

Control Flow Instructions

Naturally, WebAssembly also has control-flow instructions in order to support arbitrary algorithms. Similar to other low-level languages, WebAssembly defines conditional and unconditional branch instructions. It also has instructions to

```c
int func1(); int func2(int);
// Example function written in C
int control_flow(int a) {
    int result;
    if (a == 123) {
        if (func1()) goto end;
        result = 456;
    } else {
        result = func2(a);
    }
    return result;
end:
    return -1;
}
```

```
(func $control_flow (result i32) (param i32)
  block ;; Start a block
    local.get 0 ;; Push the argument onto the stack
    i32.const 123 ;; Push the constant 123 onto the stack
    i32.eq ;; Compare two i32 values for equality
    if ;; Start an if-block
      call $func1 ;; Call the function func1
      br_if 1 ;; Conditional branch to the second parent (Line 30)
      i32.const 456 ;; Push the constant 456 onto the stack
    else
      local.get 0 ;; Push the argument onto the stack
      call $func2 ;; Call the function func2
    end ;; End of the if-block
    return ;; Return from the function
  end ;; End of the block
  i32.const -1 ;; Push the constant -1 onto the stack
  ;; Implicitly return the value -1
)
```

Listing 3.2: Example of a function in C and WebAssembly demonstrating how different control-flow constructs are represented in WebAssembly's structured control flow.

call and return from functions. However, again motivated by the safety of the execution, branch and call instructions cannot jump to arbitrary memory addresses.

Instead, WebAssembly employs so-called *structured control flow* by defining the three special structured instructions block, if/else, and loop. These instructions define a block of execution in a function and contain one or more other instructions, including structured instructions. Thus, WebAssembly functions are not defined as a single sequence of instructions but as strictly nested sequences. Conceptually, all block instructions push a *label* onto the WebAssembly stack when they are executed. At the end of each block, indicated by an end instruction, the topmost label in the stack must be the label initially pushed at the start of the block. Additionally, the stack may contain an arbitrary but statically known number of values on top of the label, which are considered the *result values* of the block.

To jump between blocks, WebAssembly defines the regular branch instructions br and br_if, as well as the instructions br_table and return which can be used in many cases to reduce code size. The instruction br represents an unconditional branch, br_if a conditional branch. The condition of a br_if or an if block is popped from the WebAssembly stack when the instruction is executed and is stored in an i32 value. A condition is considered true if the value is non-zero. To efficiently represent switch/case statements from higher-level languages, the br_table acts as a sequence of br_if instructions; an i32 value is popped from the stack which represents the "switch label". The br_table instruction can contain one branch target for each value of the switch label and must have a default branch target.

Note that all blocks of instructions in WebAssembly are not equivalent to basic blocks in other low-level languages such as LLVM IR or Umbra IR. Conceptually, a basic block is a directed acyclic graph of non-terminator instructions and one terminator instruction. The edges of the graph represent data dependencies between the instructions. A terminator instruction is an instruction that transfers the control flow to another basic block or terminates the execution. Since every basic block has exactly one terminator and branch instructions are only allowed to branch to the beginning of a basic block, a compiler can assume that all instructions within the same basic block are always executed together. Therefore, to analyze the control flow of a program, the compiler can consider entire basic blocks at once instead of every individual instruction, which often simplifies optimization of programs. However, a WebAssembly block can contain multiple branch instructions. Branch instructions can still only branch to the beginning of a WebAssembly block, but a block can have multiple terminators.

To statically ensure well-formed control flow, WebAssembly branch instructions can only jump to blocks they are transitively contained in. In particular, branch instructions cannot jump to arbitrary other instructions or even arbitrary memory locations. It is also not possible to branch to "sibling"-blocks, i.e., other blocks that are not direct ancestors of the branch instructions. The set of target blocks of every branch instruction is statically known; indirect branch instructions do not exist.

To ensure that all branch instructions adhere to these rules, every branch instruction specifies its target as an immediate integer operand n. Then, the target of a branch instruction is the $n + 1^{\text{st}}$ parent block in which it is contained. Also, because every structured instruction pushes a label onto the stack and structured instructions are strictly nested, a branch instruction targets the $n+1^{\text{st}}$ label entry of the stack. Thus, the instruction br 0 is an unconditional branch to the block this instruction is contained in and the topmost label on the stack, and br_if 2 is a conditional branch to the third parent block and the third label entry on the stack.

The purpose of label entries in the stack is to encode the expected return values of a block and the label's *continuation*. The continuation determines where the execution continues when a branch instruction targets a label. For labels generated by the block and if/else instructions, the continuation specifies that the execution should continue after the end instruction of the block. Therefore, branch instructions that target regular blocks or if/else-blocks jump *out* of the block. Labels generated by the loop instruction, on the other hand, set the continuation to the loop instruction itself. Thus, branch instructions targeting a loop-block repeat the loop from the beginning. An end instruction always continues execution with the instruction immediately following it, which also includes the end instruction of a loop. So, unlike loops in higher-level languages, loop blocks are only executed repeatedly when they contain at least one branch instruction that targets them.

A static analyzer can easily check for every branch instruction whether its branch target is actually valid; it can statically count the number of parent blocks a given instruction has and check whether the target value of the branch is smaller than the number of parents. Alternatively, it can count the number of label entries on the stack which is always equal to the number of parents of the current instruction.

To allow for efficient static analysis, the state of the WebAssembly stack before and after a branch instruction must be consistent. To be consistent, all branch instructions targeting a particular block and label must leave the stack in a state compatible with the continuation of the label. The expected return values in a label specify which values the stack must contain when a branch instruction targeting this label is executed. This expected stack state is always compatible

with the state of the stack when the end instruction of the label's block is reached normally without branch instructions. Only then, the consistency of the stack is ensured in the presence of branch instructions. Conditional branches must additionally ensure that the stack is consistent even if the branch is not taken because the condition is false. Again, this property can be statically proven since all instructions consume and generate a statically know number of values from the stack.

Similarly, the call instruction can only call functions that are statically defined in advance. It is also possible to call functions indirectly. The indirect call instruction encodes the expected function type of the function it will call. At runtime, the execution must ensure that the function that is called is valid and has the correct expected type. Just like for branch instructions, indirect call instructions cannot call arbitrary addresses. Instead, function target addresses are represented by values of type funcref. These values can only be created from existing functions and cannot be written to memory or inspected in any way. Thus, it is not possible for WebAssembly programs to generate invalid funcref that are non-null, and therefore no unexpected control flow can occur.

An example of WebAssembly's structured control flow can be seen in Listing 3.2. The WebAssembly function begins with a block instruction in line 16. This block is targeted by the conditional branch instruction in line 22 which means that the execution continues after the end instruction of this block in line 29 when the condition evaluates to true, i.e., the value is non-zero. Since the function returns exactly one value of type i32, the stack must contain exactly one value of that type at the end of the function. Since the last instruction of the function in line 30 pushes an i32 value onto the stack, the stack state immediately before executing the i32.const instruction must be empty. So, at the end of the block that starts at line 16 and ends at line 29, the stack must also be empty, which means that the block has no return values. Therefore, all branch instructions that have this block as target, such as the conditional branch in line 22, must leave the stack empty as well.

We can verify this by following the instructions from the start of the function until the conditional branch: The instructions local.get and i32.const in lines 17 and 18, respectively, each push one value to the stack. In line 19, i32.eq pops two value from the stack, compares them, and pushes a result back onto the stack. So, before the if instruction the stack contains exactly one value. The if instruction then pops one value representing the condition from the stack. Then, when the condition is true, the execution continues at line 21. The call instruction calls a function that has no arguments and returns an i32 value, so, it pops nothing from the stack and pushes the return value of the function onto the stack. Then, the br_if instruction in line 22 pops the condition value from the stack leaving the stack empty. Therefore, the conditional branch is well-formed.

As can be seen for this example, verifying that all branch instructions are valid and leave the stack in a valid state only requires a single pass over the function; no complex control-flow analysis is necessary. Also, since the control-flow graph generated by WebAssembly program consists only of strictly nested blocks and loops that have a single entry point, WebAssembly programs cannot contain irreducible loops [All70]. While programs containing irreducible loops are not inherently less secure than programs without them, analysis of programs containing irreducible loops is generally harder. As a result, machine code generated for programs containing irreducible loops is generally less efficient.

3.1.6 Safety

The WebAssembly specification is carefully designed to rule out any potential safety issues when executing WebAssembly programs. For every instruction, the specification exhaustively lists which behavior may occur at runtime; all instructions only have few precisely defined outcomes. While the outcome of an instruction may not be deterministic, all outcomes are well-defined in order to prevent an unpredictable or unsafe execution.

One possible outcome for every instruction is a WebAssembly *trap*. When a trap is raised, the execution of the WebAssembly program must immediately terminate. It is not possible for WebAssembly programs to react to a trap in any way, in particular a trap does not behave like an exception in other programming languages as it cannot be caught. WebAssembly traps usually occur due to unrecoverable programming errors, e.g., when a memory instruction uses an invalid memory address, explicitly when the instruction unreachable is executed, or at any time when an implementation decides to terminate the execution.

All other possible outcomes for all instructions are precisely defined. Most instructions have only one deterministic outcome other than raising a trap. Integer instructions, for example, deterministically compute their result using twos-complement arithmetic and all edge cases are covered by the specification as well; for example, division by zero always raises a trap, and the instructions clz and ctz have a deterministic result if their input is zero. Similarly, every floating-point instruction precisely defines its outcome in case one of its operands has one of the special values NaN, positive infinity, or negative infinity.

3.1.7 Multi-Threading in WebAssembly

The current WebAssembly specification assumes that only one function is executed at a time and that all global variables and the memory are not accessed concurrently. An extension to WebAssembly is currently being proposed in

```
(func $lock-mutex (param i32)
  loop ;; the loop of the spin-lock
    local.get 0 ;; push the function argument onto the stack
    i32.const 1 ;; push the constant 1 onto the stack
    ;; atomically exchange the current flag with the constant 1
    i32.atomic.rmw.xchg 0
    br_if 0 ;; continue the loop if the old value was 1
  end ;; break out of the loop otherwise
)
```

Listing 3.3: Implementation of a mutex using a spin-lock in WebAssembly using atomic instructions introduced by the Threads Proposal.

the "Threads Proposal"[3]. While the proposal is still under active development and has not been standardized, yet, many implementations already support it. The Clang compiler can generate thread-safe WebAssembly and Mozilla's and Google's runtimes support executing thread-safe WebAssembly modules.

The Threads Proposal describes two main additions to the WebAssembly core specification; it allows the WebAssembly memory to be accessed concurrently, and it adds several new memory instructions, most importantly atomic memory instructions, that help with multi-threaded programming. However, it does not describe how a WebAssembly program can create new threads. Instead, starting and managing threads is up to the WebAssembly runtime that implements the Threads Proposal.

When a module should be executed concurrently in multiple threads, the proposal describes that every conceptual thread has its own, separate *module instance*. A module instance contains the WebAssembly stack and the values of all global variables. Naturally, every thread needs its own separate WebAssembly stack to prevent threads interfering with each other. Interestingly, global variables are also specific to a module instance which means that the variables are not shared between threads; each thread has its own separate set of global variables. Thus, global variables in WebAssembly can be compared to thread-local variables used in programming languages like C++ and Rust.

Only the WebAssembly memory is shared between all threads. So, all synchronization between threads is done via the memory. An implementation of a mutex using a simple spin lock, for example, must specify a memory address that stores a boolean flag. When a lock should be acquired, the flag can be set to true using atomic instructions. This must be executed in a loop to make sure

[3] `https://github.com/WebAssembly/threads` (Accessed: 22 March 2023)

that the value of the flag prior to setting it was actually false. Listing 3.3 shows the WebAssembly code for such a spin lock.

Currently, the Threads Proposal requires all atomic instructions to have *sequential consistency* [WRP19]. Therefore, all threads must observe all modifications of atomic memory locations in a single total order. In particular, it is not possible to execute atomic instructions with weaker consistency guarantees, such as "acquire" and "release" which are available in most programming languages. However, the proposal does allow unsynchronized non-atomic memory operations on the same memory location. Usually these data races are forbidden by higher-level languages; in C++ and Rust, for example, the behavior of unsynchronized data races is undefined. The Threads Proposal, on the other hand, defines all non-atomic loads and stores to behave like byte-wise atomic loads and stores with relaxed consistency. Because non-atomic memory instructions are only defined to be byte-wise consistent, data races will potentially lead to torn reads and other similar issues.

3.2 WebAssembly in Compiling Database Systems

Our goal is to use WebAssembly to safely execute UDOs in database systems. We will mainly focus on compiling database systems such as our database system Umbra. Only in a compiling database system can we inline user code into the code generated by the system. On the one hand, this leads to the best performance of UDOs as they are integrated into the query execution with near zero overhead. On the other hand, the lines between trusted and untrusted code are blurred, which means that ensuring the safety of the untrusted code is crucial.

Our database system Umbra generates code for every SQL query. Instead of generating real machine-code directly, Umbra uses a special intermediate representation called *Umbra IR*. Umbra IR shares many properties with LLVM's IR: It is a register-based language, which means that all instructions operate on explicit register operands, and it uses *static single assignment* (SSA) which means that every register can be assigned to exactly once syntactically.

To actually execute the generated code, Umbra has different options: It can interpret the IR using an integrated VM, it can compile the IR directly to less efficient x86 or ARM assembly using its Flying Start backend [KLN21a], and it can transform the Umbra IR to LLVM IR and use the LLVM framework to generate optimized machine code. The three options trade off compilation latency and execution speed; the VM generally has very low latency but is also

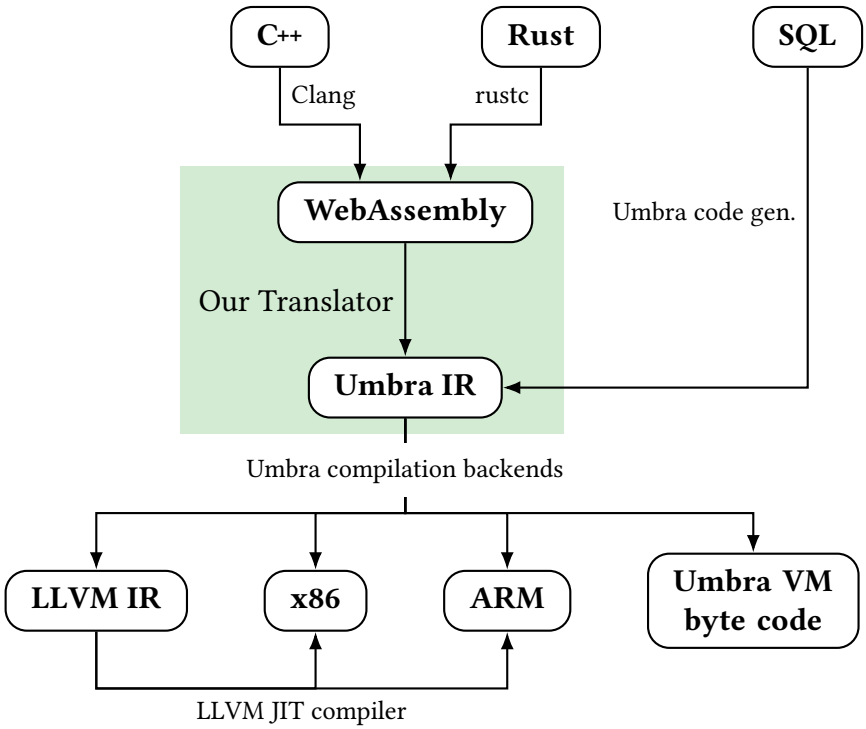


Figure 3.2: Translation and compilation phases of a WebAssembly UDO in Umbra. C++ or Rust source code is compiled to WebAssembly and our translator translates it to Umbra IR.

slowest while the LLVM framework has a large compilation overhead but it generates very fast machine code. The Flying Start backend lies in-between the other options; it has a higher latency than the VM but faster execution speeds and has lower latency than using LLVM but generates less efficient code.

When executing SQL queries that contain calls to WebAssembly UDOs, Umbra should still be able to use all its different execution options. As discussed in Chapter 2, to integrate UDOs written in C++ into Umbra, we exploit the fact that C++ can be compiled to LLVM IR which is one of the compilation modes also supported by Umbra. However, when a query containing a C++ UDO is executed using one of Umbra's other compilation modes that do not use LLVM, we need to fall back to calling functions of the pre-compiled object-file of the UDO. Using the pre-compiled object file incurs more execution overhead as the UDO cannot be inlined into the code generated for a query which leads to at least one additional function call per input and output tuple of the UDO.

To enable the efficient execution of WebAssembly UDOs, we propose translating WebAssembly directly to Umbra IR. By design, the instruction set of WebAssembly is small enough so that developing a WebAssembly runtime can

be accomplished in reasonable time. Also, its semantics is general enough, yet designed with the capability of existing hardware in mind, so that most WebAssembly instructions can be translated to a single Umbra IR instruction.

Compilers for complex high-level languages such as C++ or Rust are very sophisticated, are written by large teams of compiler engineers, and often consist of several millions lines of code. Thus, we deem it infeasible to design a compiler for these languages to Umbra IR. However, as mentioned earlier, both C++ and Rust can be compiled to WebAssembly. If our system can execute WebAssembly, it can therefore also execute WebAssembly modules that were generated from C++ and Rust. Then, we can not only execute untrusted user code safely, because we rely on the safety properties of WebAssembly, but we can also execute all programs written in higher-level languages that can be compiled to WebAssembly.

Additionally, since all WebAssembly code is translated to Umbra IR, all queries containing WebAssembly UDOs can directly inline the user code. In contrast to our approach when inlining LLVM IR when compiling C++ UDOs, the inlining of WebAssembly is possible independent of the compilation backend used by Umbra. Thus, we can benefit from all properties of the different compilation backends with all WebAssembly UDOs.

Figure 3.2 shows an overview of how a UDO originally written in C++ or Rust is compiled, translated, and executed. The proposed WebAssembly to Umbra IR translator is used as an intermediate step to tie together the world of high-level language compilers and Umbra. Existing compilers such as Clang or rustc must be used by the user to compile the high-level source code to WebAssembly. In our system we provide a C++ header file that defines the declarations for a UDO such as the UDOperator base class (see Section 2.2.2). Still, the compilation from C++ to WebAssembly is entirely external to the database system and occurs prior to submitting a `create function` statement.

The database system receives the compiled WebAssembly module when a new UDO is created. As discussed earlier, when a SQL query is executed that contains a call to a WebAssembly UDO, the SQL query is translated to Umbra IR. Our proposed WebAssembly translator can then also translate the UDO code to Umbra IR which results in a single Umbra IR module for the entire query. The Umbra IR module can then be executed using one of Umbra's several backends.

3.3 Related Work

Since WebAssembly was originally designed for safe and efficient execution of untrusted code in web browsers, the first WebAssembly runtimes were implemented in browser engines. The JavaScript engines of both Firefox and Google

Chrome – SpiderMonkey[4] and V8[5] – implement very sophisticated just-in-time compilation for JavaScript. As such, both engines already contained a compilation backend that generates machine code. The engines now also have a WebAssembly runtime that reuses most of the existing compilation backends.

As WebAssembly is becoming increasingly popular for use-cases outside of a web browser, several projects exist that develop independent WebAssembly runtimes. Wasmtime[6] and Wasmer[7] are two popular WebAssembly runtimes. Both projects can compile WebAssembly programs to machine code using the Cranelift[8] compilation framework. Cranelift's intermediate representation is similar to Umbra IR; it is register-based and uses SSA. Wasmer can also translate WebAssembly programs to LLVM IR.

Ménétrey et al. have explored how WebAssembly can be used to run programs on untrusted systems [Mén+21]. Especially for cloud systems where users may process confidential data on servers controlled by the cloud provider, it is desirable for users to ensure that cloud providers have no access to the data. The authors present Twine, a WebAssembly runtime that runs inside of a Trusted Execution Environment of a CPU, such as Intel SGX, which allows to run arbitrary WebAssembly programs confidentially. To demonstrate the usability of Twine for data processing, they compile the embeddable database system SQLite to WebAssembly and run several benchmarks on SQLite running in Twine.

[4] `https://spidermonkey.dev/` (Accessed: 21 June 2023)
[5] `https://v8.dev/` (Accessed: 21 June 2023)
[6] `https://wasmtime.dev/` (Accessed: 21 June 2023)
[7] `https://github.com/wasmerio/wasmer` (Accessed: 21 June 2023)
[8] `https://cranelift.dev/` (Accessed: 21 June 2023)

CHAPTER 4

Translating WebAssembly to Umbra IR

Our goal is to safely integrate UDOs into compiling database systems. We identified that WebAssembly can be used as an intermediate compilation target to execute untrusted user code safely. Then, we need to translate WebAssembly to a format understood by the database system. To integrate WebAssembly in our system Umbra, we translate WebAssembly to Umbra IR.

As a low-level language that is designed to be used as a compilation target for higher-level languages, WebAssembly shares several properties with Umbra IR. In fact, most WebAssembly instructions can be translated to an equivalent Umbra IR instruction. The WebAssembly instruction i32.add, for example, can be translated directly to the Umbra IR instruction add i32.

One main difference between WebAssembly and Umbra IR is that WebAssembly is a *stack-based* language while Umbra IR is *register-based*. Listing 4.1 shows how the function load_add from the example in Listing 3.1 can be written in Umbra IR. The load instructions in lines 3 and 4 take the function argument which is stored in the conceptual register ptr as explicit operand. Umbra's load instructions also support immediate operands to specify a fixed offset which should be considered to calculate the actual memory address. In this example, it is specified as the array index 1 while in WebAssembly the corresponding instruction f64.load 8 specifies its fixed offset in bytes. The function returns the result of the add double instruction which is stored in the register ret explicitly by using a return instruction in line 6.

So, to translate WebAssembly programs into Umbra IR, we must translate all indirect operands implicitly passed via the stack into explicit register operands. Fortunately, translating a stack-based language into a register-based language has been studied before. Especially in the context of Java and its well-studied

```
define double @load_add(double* %ptr) {
body:
  %ptr0 = load double %ptr
  %ptr1 = load double %ptr, int32 1
  %ret = add double %ptr0, %ptr1
  return %ret
}
```

Listing 4.1: Example function load_add from Listing 3.1 written in Umbra IR. Because Umbra IR is register-based, all instructions have explicit operands.

JVM, the benefits of register-based languages have been researched. Gregg et al. [Gre+05] show that translating JVM byte-code to a register-based language first can lead to significant benefits while interpreting the new register-based language. Shi et al. [Shi+08] expand on that and describe in detail how implicit accesses to the stack can be translated to explicit register accesses. The general idea is to assign one register to every stack slot. Since in JVM byte-code, like in WebAssembly, the height and state of the stack can be determined statically, a compiler can also statically determine for every instruction which stack slot(s) it reads from or writes to. Now, every instruction can be transformed to an instruction with explicit register operands by using the registers assigned to the stack slot(s) used by the original instruction.

The transformation using one register per stack slot assumes that every register can be written to multiple times since multiple instructions can use the same stack slot for their result, mostly purely coincidentally. However, Umbra IR uses *static single assignment* (SSA) for its registers. Like LLVM, Umbra IR uses SSA to simplify lifetime analysis of registers and therefore better register allocation when translating to real machine code that only has a limited number of registers.

In SSA, the instruction set has access to an infinite number of registers. Every register must be statically assigned to exactly once. “Statically once” means that exactly one instruction must exist that stores its result in a given register. Since an instruction can be executed multiple times, e.g., when it is contained in a loop, a register still can be assigned to multiple times at runtime. But because registers cannot be assigned to multiple times statically when using SSA and thus for Umbra IR, the approach described by Shi et al. cannot be used directly.

It is possible to modify arbitrary programs written in a register-based language into SSA form. Cytron et al. [Cyt+91] show that by introducing so-called *phi nodes* at the edges of the control-flow graph, the program can be trans-

formed into an equivalent program that satisfies the SSA requirement. The transformation requires analyzing all registers and how they are used. Taking into account all possible control flow of the function, the so-called dominator relation between all instructions and registers is calculated. An instruction X dominates an instruction Y if and only if every path through the program that ends at Y also contains X.

To transform a program to SSA form, all assignments to registers are analyzed. Every time an existing register R would be overwritten by an instruction I, thereby violating the SSA property, a new register R' is introduced instead. Every instruction U that is dominated by I and uses the register R as operand is then updated to use R' instead. All instructions that dominate I remain unchanged.

It is possible for an instruction N that uses R to neither dominate I nor be dominated by I, e.g., when I appears inside of an if/else branch that may not always be taken. In that case, a phi node is introduced at the first position in the control-flow graph after I that dominates N. A phi node acts like an instruction that has two or more inputs, for example the registers R and R', and writes one of the inputs to its output register. It chooses which input register to use according to which branch was taken before the phi node and stores it in a new register Φ. All other instructions that are dominated by the phi node and use any of the operand registers of the phi node are then updated to use Φ instead.

Listing 4.2 shows how a program that is not in SSA form is transformed into an equivalent program that satisfies it. The function f_no_ssa contains three assignments to the register ret in lines 6, 9, and 12 which violates the SSA property. Also, the add instruction in line 6 does not dominate the instruction in line 12 which also uses the register ret because the add instruction lies in the if_true branch that may not always be taken. Likewise, the sub instruction in line 9 writes to the register ret but does not dominate line 12, either. To transform this function into SSA form, first, new registers are introduced so that no register is assigned to twice; lines 20, 23, and 27 now use the separate registers ret1, ret2, and ret3, respectively. Eventually, the registers ret1 and ret2 must be combined again so that all other instructions that used the register ret before the transformation still work correctly after the transformation. The registers ret1 and ret2 are combined into the register ret.phi by using a special phi instruction. This instruction must be inserted in the function such that it dominates all following instructions that used the old register ret, which in the example is at the beginning of the end block. The phi instruction specifies that its result should be the value of the ret1 register if during the execution this instruction was reached via the if_true block, or the value of ret2 when coming from the if_false block. Finally, all instructions that used the register ret are updated to use ret.phi instead, such as the add instruction in line 27.

```
define int32 @f_no_ssa(int32 %a, int32 %b) {
body:
  %is_42 = cmpeq i32 %a, 42
  condbr %is_42 %if_true %if_false
if_true:
  %ret = add i32 %a, %b
  br %end
if_false:
  %ret = sub i32 %a, %b
  br %end
end:
  %ret = add i32 %ret, 123
  return %ret
}

define int32 @f_ssa(int32 %a, int32 %b) {
body:
  %is_42 = cmpeq i32 %a, 42
  condbr %is_42 %if_true %if_false
if_true:
  %ret1 = add i32 %a, %b
  br %end
if_false:
  %ret2 = sub i32 %a, %b
  br %end
end:
  %ret.phi = phi i32 [%ret1 %if_true, %ret2 %if_false]
  %ret3 = add i32 %ret.phi, 123
  return %ret3
}
```

Listing 4.2: Transformation of the function f_no_ssa with a conditional branch that violates the SSA property into an equivalent function f_ssa that satisfies it.

The transformation to SSA form requires detailed and expensive analysis of each function. To find the dominator relation, the control-flow graph of each function must be built. Since registers are updated during the transformation, it is also necessary to track all usages of every register throughout a function. Finally, the control-flow graph must be used to find suitable locations to add new instructions representing phi nodes. Adding new instructions at arbitrary locations may also require moving around neighboring instructions.

This expensive transformation is necessary because programming languages usually do not require the strict SSA property. In general, SSA languages are not designed to be written by humans. Instead, SSA is chosen as a property, for example, by LLVM IR or Umbra IR, because it simplifies the optimization of the program. WebAssembly is also designed as a target language for compilers but is a stack-based language without registers and, therefore, does not satisfy the SSA property. However, because of its strict structured control flow, it is efficient to reason about the control flow and dominator relations of WebAssembly programs.

In the remainder of this chapter, we present how WebAssembly can be translated directly to a register-based language in SSA form to eventually safely execute UDOs in compiling database systems. First, we show how WebAssembly can be efficiently translated into Umbra IR. We also explain how the translator ensures all safety guarantees of WebAssembly programs when they are translated to Umbra IR. Then, we present how our WebAssembly translator can be integrated into a UDO Query Compiler to eventually execute UDOs using WebAssembly. To evaluate the feasibility of using WebAssembly for safe execution of UDOs, we measure the efficiency of our translator and its generated code running in our database system Umbra.

4.1 Translation to SSA with a Virtual Stack

We want to directly translate the stack-based WebAssembly language to a register-based language such as Umbra IR which requires that all programs satisfy the SSA property. We cannot assign each stack slot a fixed register as proposed by Shi et al. as that potentially generates multiple different instructions that write to the same register. Furthermore, all values in WebAssembly have a specific type but the same stack slot could contain values of different types at different points in the program.

Instead, we will track a *virtual stack* while translating the program. When translating a WebAssembly instruction, we generate the corresponding Umbra IR instruction and push the name of the register that the Umbra IR instruction uses for its result into the virtual stack. Similarly, when a WebAssembly stack

WebAssembly	Umbra IR	Virtual Stack
`func $load_add (result f64) (param i32)`	`define double @load_add(int32 %arg1)`	
`local.get 0`	*(no instruction)*	i32: `%arg1`
`f64.load 0`	`%r1 = load double %arg1`	f64: `%r1`
`local.get 0`	*(no instruction)*	i32: `%arg1` f64: `%r1`
`f64.load 8`	`%r2 = load double %arg1, int32 1`	f64: `%r2` f64: `%r1`
`f64.add`	`%r3 = add double %r1, %r2`	f64: `%r3`
implicit return	`return %r3`	

Figure 4.1: Translation of the function load_add from Listing 3.1 from WebAssembly to Umbra IR by using a virtual stack.

would need to pop values from the stack at runtime, we pop values from the virtual stack while translating the instruction and use the registers stored in the stack as input operands for the new Umbra IR instruction. The virtual stack is only used while translating the program; at runtime, no stack is required.

Figure 4.1 shows how the example function load_add from Listing 3.1 is translated to Umbra IR by using a virtual stack. For the first WebAssembly instruction local.get, no Umbra IR instruction is generated. However, after translating the instruction, the virtual stack now contains an entry that references the register arg1. Semantically, the local.get instruction gets the first function argument and pushes it onto the stack. When translated to an Umbra IR function where the function argument is stored in the register arg1, we only need to remember that the result of the local.get instruction will be available in the register arg1 at runtime. Now, when the next WebAssembly instruction f64.load would need to pop a value from the stack at runtime, we instead pop a value from the virtual stack while translating the instruction. Because the stack contains the register arg1, we know that the translated instruction load double must use it as an operand. For its result, the translator allocates the new register r1 and pushes it onto the virtual stack. When another instruction eventually pops the entry from the virtual stack again, such as f64.add in our example, it will then use r1 as an operand.

The example shows that the translator is able to generate an Umbra IR program that is identical to the manual translation shown in Listing 4.1. Also, the translation happens in a single pass over the WebAssembly program. In particular, no analysis of the control-flow graph or the dominator relation is necessary here. It also shows that the translator does not need to generate any Umbra IR instructions for some WebAssembly instructions; all instructions that are used to manipulate the WebAssembly stack without any other operations do not lead to the generation of any Umbra IR instruction. Instead, the translator only remembers the new structure of the virtual stack after processing such a WebAssembly instruction. Examples for WebAssembly instructions that do not lead to Umbra IR instructions are local.get which pushes the value of a function argument or local variable onto the stack, as seen in the example, and drop which pops a value from the stack and discards it.

As long as the virtual stack contains only registers, the translation of WebAssembly to Umbra IR is very straight-forward; for every generated instruction, the translator can immediately determine all operand registers. However, the WebAssembly stack can also contain labels when a program contains control-flow instructions. In the next section we explain how WebAssembly labels can be represented in the virtual stack to be able to translate arbitrary control flow.

```
1 (func $f (result i32) (param i32 i32)
2   local.get 0 ;; Push the first argument onto the stack
3   local.get 1 ;; Push the second argument onto the stack
4   local.get 0 ;; Push the first argument onto the stack
5   i32.const 42 ;; Push the constant 42 onto the stack
6   i32.eq ;; Compare two i32 values for equality
7   if ;; Start an if-block
8     i32.add ;; Add two i32 values
9   else
10    i32.sub ;; Subtract two i32 values
11  end
12  i32.const 123 ;; Push the constant 123 onto the stack
13  i32.add ;; Add two i32 values
14  ;; Implicitly return the result of i32.add
15 )
```

Listing 4.3: A WebAssembly function containing an if/else-block. The function is equivalent to f_ssa shown in Listing 4.2 and the translator should be able to generate exactly the same code.

4.2 Translation of Structured Control Flow

Naturally, when a program contains control flow, it is not possible to statically determine where the operands from a given instruction originate from. When an instruction in a stack-based language pops a value from the stack, or an instruction in a non-SSA register-based language uses a register, the actual value used by the instruction potentially could have been generated by several different instructions. Especially when two sides of a conditional branch which depends on an unknown runtime value contain two different instructions that write to the same stack slot or register, a static analyzer must assume that at runtime either branch can be executed.

4.2.1 Translation of if/else Blocks

Listing 4.3 shows a small WebAssembly function that contains an if instruction. This function is equivalent to the function f_ssa shown in Listing 4.2. The goal for the translator is to directly generate Umbra IR, which is in SSA form, so the resulting code should be equal to f_ssa. The result of the function depends on the value generated by either the instruction i32.add in line 8 or the instruction i32.sub in line 10. After the end instruction, the WebAssembly stack contains exactly one value, which is then used by the last i32.add instruction in line 13.

We extend the approach shown in Section 4.1 so that it can handle control-flow instructions. The translator generates two new registers when translating the instructions within the if and else instructions in lines 8 and 10. At the end of the if/else-block in line 11, the translator must generate a new register that represents the result value of the if/else-block. Unifying the values from different branches into a single register is precisely why the phi instruction in SSA languages exists. So, a WebAssembly translator must generate a phi instruction at the end of every if/else-block.

The translation of an if-branch is shown in Figure 4.2. The function arguments have been translated to be stored in the registers arg1 and arg2. After the translator reaches and translates the instruction i32.eq in line 6, the virtual stack contains the first and second function argument, which were pushed earlier when translating the instructions in lines 2 and 3, as well as the result of the comparison.

When the if instruction needs to be translated, the translator first needs to determine how many values from the stack will be used inside the if/else-block, which is called the *block input type* in WebAssembly. In our example, the if-block contains a single i32.add instruction. This instruction needs to pop two i32 values from the stack and pushes one to the stack again. So, the input type of the if-block has two i32 values. Also, since at the end of the block one i32 value remains on the stack, its result type is i32. Note that WebAssembly requires that the input and result types of an if- and its else-block must always match. In our example, this obviously holds since the else-block also contains only exactly one instruction that pops two values from and pushes one onto the stack.

Then, the translator pops values from the stack according to the input type, pushes a new label entry onto the virtual stack, and pushes the values back onto the stack. The stack entry for the label contains additional information such as the input and result type of the block, and the names of the basic blocks in the generated code – ifblock, elseblock, and ifend in our example – so that the translator can later generate branches to these basic blocks. In general, every branch instruction terminates a basic block in Umbra IR. Since translating an if/else-block requries three branch instructions – the conditional branch that checks the condition, and the two unconditional branches from the end of the if- and else-blocks to the end – the translator creates three new basic blocks.

Before translating the first instruction inside the if-block, the translator also creates a copy of the current stack. The copy is used to later translate the else-block. Since after executing the conditional branch the execution could continue at either the if- or the else-block, the virtual stack must have the same state at the beginning of the translation of both branches.

Since an if/else-block can occur at any point in a program, in particular at points where the virtual stack contains many entries, it is important to imple-

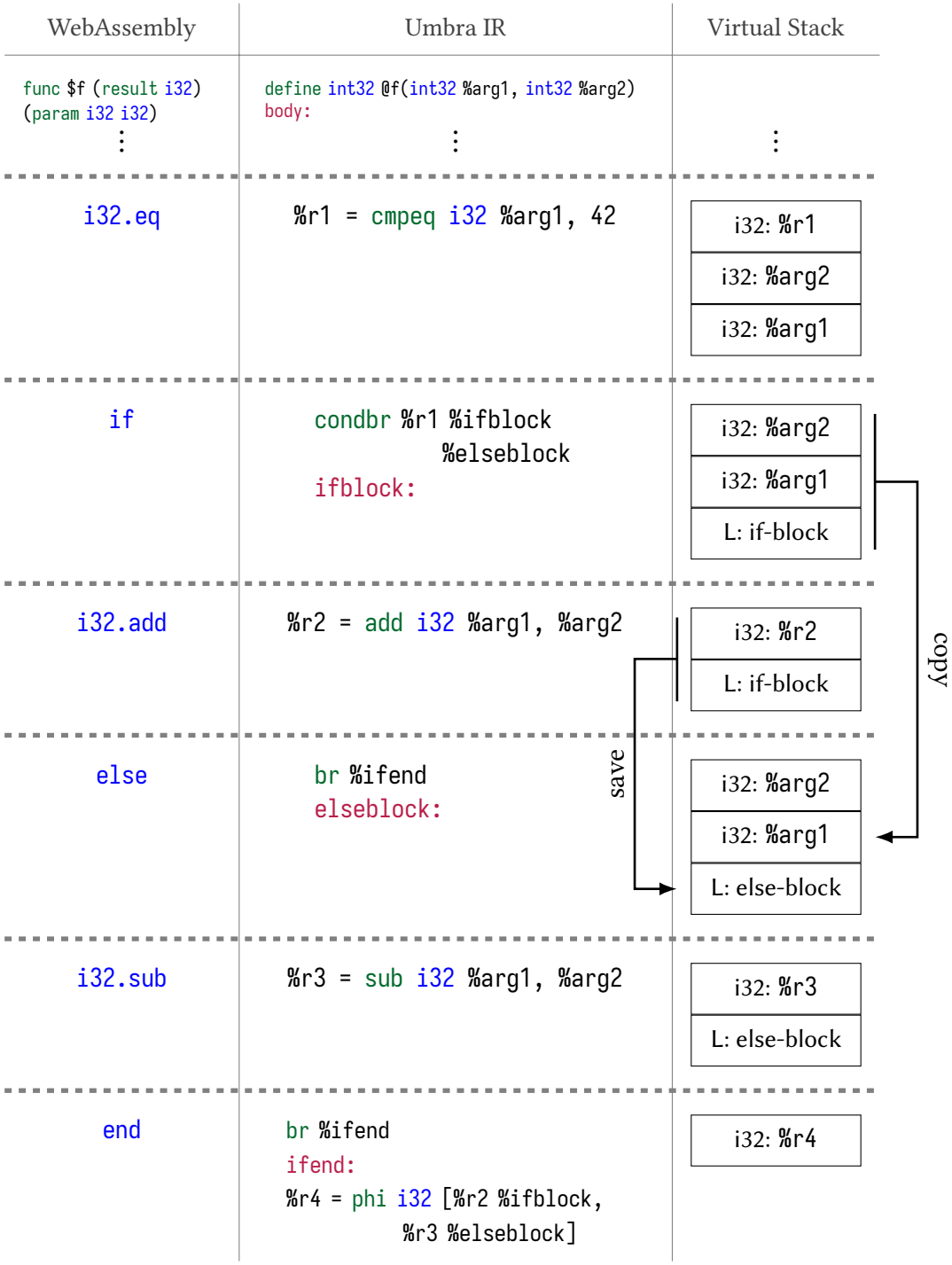


Figure 4.2: Translation of the if-block from the function shown in Listing 4.3 to Umbra IR. Additional bookkeeping of the stack is necessary to be able to generate the phi instruction eventually at the end of the if-block.

ment the copying of the stack efficiently. Naively copying the entire stack every time an if/else-block is translated can easily lead to quadratic complexity in the number of instructions. So, instead, we employ reference counting on the stack entries so that for every translated if/else-block only the values representing its input type are copied. Since an input type of a block in WebAssembly can only contain a fixed number of values, copying only the input values and increasing the reference count on the topmost entry of the remaining stack guarantees that the storage complexity of the translator remains linear in the number of if/else-blocks in a function. Fortunately, because of WebAssembly's design goal to facilitate efficient translation, every block instruction in WebAssembly directly encodes its input type and result type. So, a translator can immediately determine the required input type without having to traverse the block to find out how many stack entries the instructions in the block pop and push.

After the label representing the if-block and all input values of the block are pushed onto the stack, the translation continues normally. In our example, the next instruction i32.add is translated by popping two values from the virtual stack and using the registers contained in the stack entries as operands. The translation routine for the instruction i32.add requires no knowledge of the fact that it is contained in an if-block.

When the translator encounters an else instruction, it first needs to post-process the if-block; it must collect the result values from the stack, create a branch instruction to the end of the entire if/else-block and create a new basic block for the else-block, called elseblock in the example. To collect the result values, the translator scans the stack, without modifying any entries, to find the label of the if-block this else instruction belongs to. The label entry stores the result type of the block, so the translator can pop values from the stack accordingly. Because the result values of the if-block need to be used later to generate phi nodes, they are saved in a new label entry that is created for the else-block. All additional information that was stored in the label entry of the if-block is saved in the label entry for the else-block as well. Finally, the translator restores the copy of the stack that it made when translating the if-block and replaces the label entry for the if-block with the new label entry for the else-block. Then, the translation can again proceed as usual.

The first instruction inside the else-block sees the virtual stack in the same state as the first instruction if-block when it is translated. This is necessary, because when the function is executed eventually, the execution can reach either of these two instructions from the same condbr instruction. Again, when the i32.sub instruction is translated, the translator can pop and push values without special handling for the translation in an else-block.

The translator detects the end of the else-block when it reaches the end instruction. Again, the translator scans the virtual stack to find the topmost

label entry. Then, it post-processes the else-block similar to the if-block; it generates a new branch instruction and then creates the basic block for the end of the entire if/else-block, called ifend in our example. It also pops values from the stack according to the result type of the block. Now, the translator needs to generate a phi instruction to reconcile the result values of the if-block that were saved into the label entry for the else-block and the result values that it just popped from the virtual stack. In the example, the result value of the if-block is an i32 value that is stored in register r2 and the result value of the else-block is stored in register r3. Therefore, a new phi instruction is created that evaluates to r2 if during execution the path over the ifblock was taken, or to r3 for the elseblock. The result of the phi instruction is stored in the new register r4. As the last steps, the translator pops the label entry from the stack as it is not needed anymore, and pushes a new entry that contains the register r4 representing the result of the entire if/else-block.

To summarize, if/else-blocks are translated by duplicating the virtual stack for the if-block and the else-block. The entry on the stack that contains the label for the blocks contains auxiliary information that allows the translator to generate the necessary branch and phi instructions.

This strategy for translating if/else-blocks to Umbra IR while maintaining the SSA property and correctly creating phi instructions is facilitated by WebAssembly's design; the structured control flow guarantees that the state of the stack at the end of the if-block and its else-block must be compatible which means that a translator does not have to analyze the control flow further to generate all required phi instructions.

4.2.2 Translation of Branch Instructions

In addition to its block instructions, WebAssembly also has branch instructions which a translator needs to handle. Similar to if/else-blocks, the conditional branch instruction br_if can lead to program states during execution where an entry for a value in the WebAssembly stack can have multiple sources. Again, to represent control flow where the value of a register can have multiple sources in a SSA language, phi instructions must be inserted correctly.

To implement conditional branch instructions efficiently in a translator, two observations are crucial: First, a branch instruction in WebAssembly cannot target arbitrary other instructions in the same function but only labels created by block instructions, and second, in SSA languages phi nodes are only necessary when two diverging branches need to converge again. Both observations lead to the conclusion that special handling for translating branch instructions is only necessary at the end of every block instruction, i.e., every time an end instruction is translated. In WebAssembly, only at an end instruction two different blocks

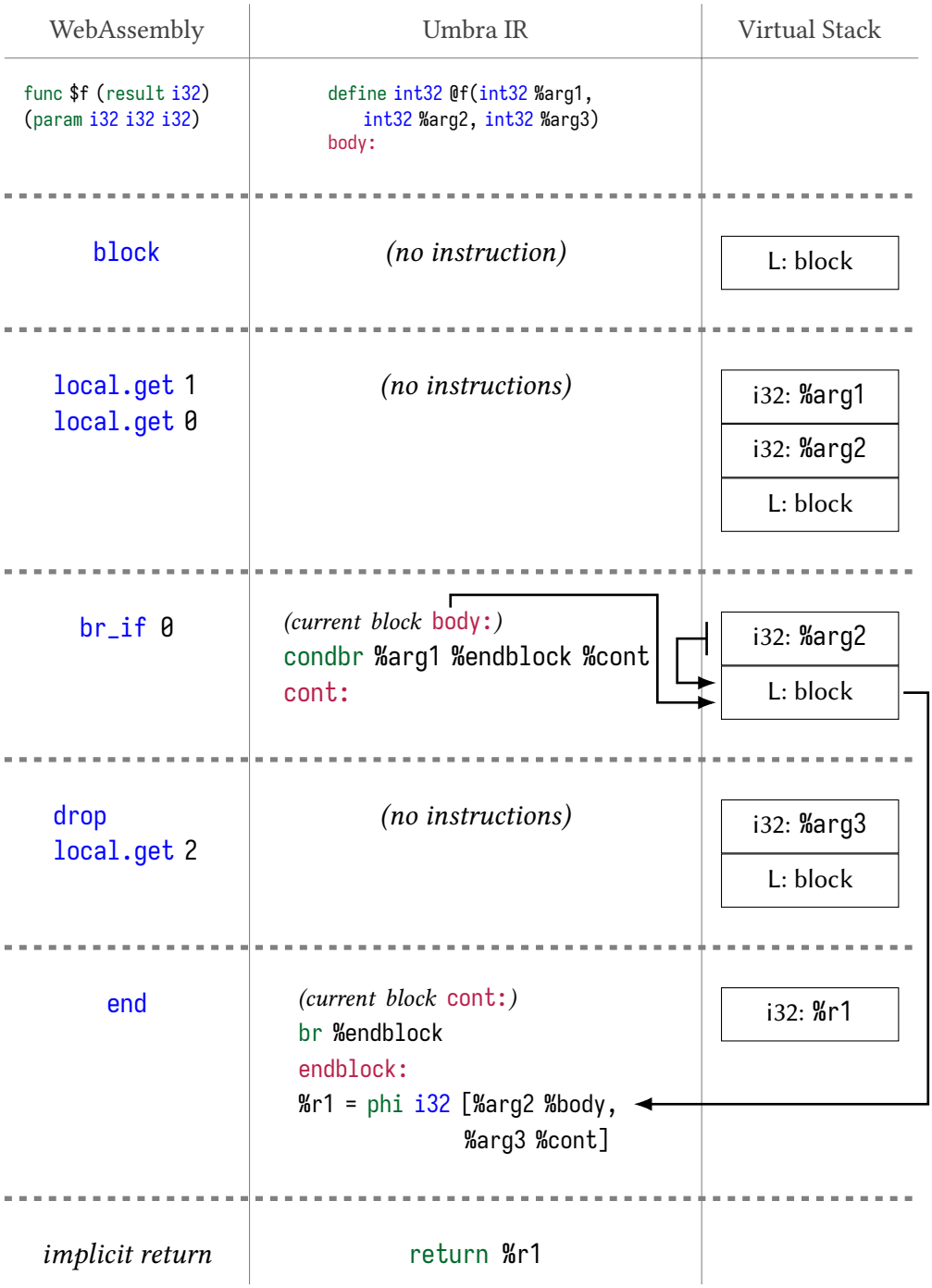


Figure 4.3: Translation of a function containing a condbr branch instruction to Umbra IR. For every branch instruction the current basic block and the result values from the stack are saved to the label entry for the target block.

can converge again, such as an if- and an else-block as shown before. So, phi instructions only need to be created at that point in the translation.

Nevertheless, when a branch instruction is translated, the translator must store enough information so that the end instructions can eventually correctly create all required phi instructions. Figure 4.3 shows the translation of a function that uses a block instruction which contains a br_if branch instruction that targets the block. When the translator reaches the br_if instruction, it must first determine the target of the instruction. In the example, the branch instruction has the index 0 so the translator must scan the virtual stack to find the first label. Then, it extracts the result type of the block from the label entry. Since a branch instruction conceptually skips ahead to the end of its target block, the translator must ensure that the result values of the block are taken from the state of the stack at the point of the branch instruction. So, using the extracted result type, all stack entries representing the result type of the block in case the branch is taken are saved to the label entry of the block. To be able to create phi nodes later, the name of the current basic block must be saved as well.

The actual conditional branch is implemented by generating a condbr instruction which uses the topmost value on the stack as its condition. The target of the conditional branch is the basic block stored as continuation in the label entry that was found for the target of the br_if instruction. In our example, the conditional branch targets the end of the WebAssembly block for which the translator generated a basic block named endblock. Since every branch instruction terminates a basic block, the translator needs to create a new basic block that represents the path of the program that is taken when the condition is false, called cont in the example.

The structure of the virtual stack only changes by the br_if popping one value representing the condition – the register arg1 in the example. All following instructions can again be translated normally; the drop instruction pops a value from the stack and discards it, and the local.get instruction pushes a new value onto the stack.

At the end of every block, when its end instruction is translated, the translator needs to generate phi instructions for all return values of the block. When translating if/else-blocks as shown in Figure 4.2, the phi instructions are generated for the if-block and the else-block. Similarly, Figure 4.3 shows how a phi instruction is generated, even for regular block instructions. The approach is the same as when translating if/else-blocks; for every “incoming” branch instruction, i.e., all branch instructions that were added to the label entry for the current block, the resulting phi node contains one entry. In our example, when the translator translated the br_if instruction, it saved the name of the current block (body) and the register containing the return value (arg2). Thus, the generated phi instruction has one entry for the register arg2 and the basic

```
define void @f(int32 %arg1) {
body:
  ; [...]
  br %loop_head
loop_head:
  %var1 = phi i32 [%arg1 %body, %var1next %loop_end]
  ; [...]
  br %after_loop
loop_end:
  %var1next = phi i32 [...]
  br %loop_head
after_loop:
  ; [...]
}
```

Listing 4.4: Umbra IR generated by the translator for a function containing a loop. The translator generates two sets of phi instructions to correctly handle the loop variables.

block body. It also has a second entry for the implicit incoming edge from the regular execution up to the current end instruction – the register arg3 and the basic block cont here. The register that contains the result of the phi instruction is then also pushed back to the stack so that following instructions can use it.

Overall, for every translated end instruction our translator creates one phi instruction per result value with one entry per incoming edge. In if/else-blocks, naturally there are at least two incoming edges, so the generated phi instructions have at least two entries. Of course, the if- and else-block could contain more branch instructions thus leading to more entries in every phi instruction. Similarly, regular block instructions have at least one incoming edge at the end, which is the edge from the last instruction of the block. In the special case where at the end instruction only one incoming edge exists because no branch instructions were used inside the block, the translator does not need to generate any phi instructions. Instead, it can directly use the registers from the virtual stack as result values.

4.2.3 Translation of Loops

When a branch instruction targets a loop-block, the execution continues at the beginning of the loop, i.e., these branch instructions act like continue statements in higher-level languages instead of breaking out of the loop. In WebAssembly, loop-blocks can also have input values, just like if/else and regular blocks. The

input values of a loop-block can be thought of as "loop variables"; they have one initial value when the loop is executed for the first time. For every subsequent iteration, the value of the loop variables depends on the previous iteration which may modify them arbitrarily. Since the values of the loop variables can come either from the code before the loop-block, or from code inside the loop-block executed before a branch instruction, a translator must generate phi-instructions at the beginning of a loop.

However, all incoming edges of the phi nodes at the beginning of the loop are only known *after* all instructions inside the loop-block are translated. But the instructions in the loop may already use the registers that contain the results of the phi instructions from the beginning of the loop. To break the circular dependencies of the phi registers while translating loop-blocks, our translator first generates two sets of phi instructions; one set at the beginning of the code generated for the loop-block and one set at its end instruction. The set of phi instructions generated for the end instruction are very similar to the phi instructions generated for the end instruction of regular and if/else blocks; the translator creates one phi instruction for every input value of the loop-block (as opposed to creating one for every result value like in regular and if/else blocks) which has one entry for every incoming edge. At the beginning of the loop-block, the translator also generates one phi instruction for every input value. These phi instructions have exactly two entries: One entry for the basic block that is executed immediately before the loop with the initial value for the loop variable, and another entry for the basic block that contains the branch instruction back to the top of the loop, i.e., the *back edge* of the loop, with the value of the loop variable for the next loop iteration.

Listing 4.4 shows the Umbra IR function generated by the translator when it translates a function containing a loop-block with one loop variable. In line 10, it generated the phi instruction for the end of the loop. This instruction is generated using the same approach as shown in Figures 4.2 and 4.3. The result of this phi instruction is stored in the register var1next. Thus, in the next iteration of the loop, this register should be used as loop variable.

Note that the br instruction in line 8 does not jump to the basic block loop_end. As mentioned before, at the end of a WebAssembly loop-block, the execution continues with the first instruction after the loop, i.e., the basic block after_loop. Only explicit branch instructions inside the loop-block that could occur in line 7 can potentially jump to loop_end.

When the translator generates the code for the beginning of the loop-block, naturally, it does not know the name of the registers that will contain the values for the loop variables for the next iteration. This register is only created when translating the end of the loop-block. The translator still generates a phi instruction for each loop variable and sets the first entry of each instruction to

contain the initial value, but sets the second entry to dummy values. Only when the phi instructions at the end of the loop are created, the translator updates the second entry of the phi instructions at the beginning of the loop, as shown in line 6.

4.2.4 Unreachable Blocks

It is possible to write WebAssembly programs that contain instructions that cannot be reached during execution. In some cases, a static analyzer can easily prove that some parts of the program are unreachable. For example, all instructions within the same block following a return instruction can never be executed. Only the instructions after the next end instruction are potentially reachable again, for example when the execution branches to the end instruction using a branch instruction before the function returns.

The WebAssembly specification generally requires all programs to be well-formed. In particular, since every instruction pops and pushes a statically known number of values from and onto the stack, WebAssembly programs may not contain sequences of instructions that would lead to an invalid stack state. However, there are three exceptions; a block of instructions may contain arbitrary, malformed sequences of instructions as long as they follow one of the instructions which guarantee the unreachability: return, as motivated above, br, since an unconditional branch is always taken, br_table which is a specialization of br that can be used to better represent switch/case statements from higher-level languages, and the special unreachable instruction which immediately terminates the execution if it is reached.

When a WebAssembly program is interpreted, unreachable instructions which are potentially malformed, cause no problems. But our translator relies on the fact that all instructions leave the stack in a valid state to be able to create the virtual stack during translation. Naturally, if the translator translates an unreachable instruction and tries to pop a value from the virtual stack even though the stack is empty, the translator is not able to generate any sensible Umbra IR instruction.

Therefore, the translator must track the reachability of every instruction. If an instruction is known to not be reachable due to one of the exceptions in the WebAssembly specification, the translator must silently skip it. Fortunately, this can be implemented cheaply; as soon as one of the instructions that make all the following instruction unreachable is translated, the translator must set a flag indicating that it is currently in an unreachable state. Then, it can skip over all following instructions as long as the flag is set. Only when the end instruction of the block that contains the instruction that caused the flag to be set is reached, the flag needs to be unset again so that the translation can continue as usual.

When translating an if/else-block, the translator additionally needs to consider that the else-block may be reachable even if the end of the if-block is not.

In some cases, skipping over unreachable WebAssembly instructions can lead to phi instructions with no incoming edges. For example, when a regular block has a result type with more than one value but it contains only a return instruction, its result will never be created. So, a translator must be able to handle the end of a block even if it cannot be reached. In Umbra IR, phi instructions are allowed to have no entries, which means that the translator requires no special handling for these cases.

Of course, all code following a block which by itself is determined to never reach its end, is also unreachable, even if the WebAssembly specification requires it to be well-formed. Thus, a translator could omit translating these instructions as well. However, in general, finding these unreachable instructions requires complex reasoning about the control flow of the program whereas only handling the exceptions made by the WebAssembly specification can be implemented by using a single flag. Since programs containing mostly unreachable instructions are hardly useful, and high-level language compilers are unlikely to generate them, our translator only handles the cases required by the WebAssembly specification to achieve higher translation speeds for useful programs.

4.3 Parallel Execution

The specification of WebAssembly currently does not allow for parallel execution of WebAssembly functions. However, an extension to WebAssembly called the "Threads Proposal" is available which is supported by many compilers and WebAssembly runtimes (see also Section 3.1.7). The Threads Proposal defines several new instructions, most importantly atomic memory instructions. Since Umbra is designed to execute all queries in parallel, Umbra IR also has several different atomic instructions. Both WebAssembly and Umbra IR require their atomic operations to have sequential consistency, so most atomic memory instructions in WebAssembly can be translated directly to exactly one atomic instruction in Umbra IR.

The Threads Proposal also contains less common atomic instructions. For example, WebAssembly programs can use an atomic xor instruction. Because the hardware architectures that Umbra runs on do not have atomic xor instructions, Umbra IR does not define them. To still be able to translate WebAssembly programs containing this instruction, our WebAssembly translator generates a function call to a runtime function that implements the atomic xor operation using a loop with an atomic compare-exchange operation.

To allow WebAssembly to efficiently implement semaphores that do not only rely on spin locks, the Threads Proposal also defines a notify and a wait instruction. Their semantics maps closely to the functions with similar names used by implementations of condition variables in different programming languages, such as pthread_cond_wait and pthread_cond_signal/pthread_cond_broadcast for POSIX systems. Therefore, our translator translates the notify and wait instructions to function calls to runtime functions that use a condition variable implementation.

Since every memory access in a WebAssembly program semantically requires a bounds check with the current memory size, a WebAssembly runtime using the Threads Proposal must ensure that the bounds checks work as expected even if the memory is grown in one thread while other threads concurrently access the memory. Fortunately, the Threads Proposal does not require the bounds checks to have atomic sequential consistency with respect to the memory size. This choice was made deliberately as otherwise every memory access, even if it is non-atomic, would require an atomic sequentially consistent bounds check. Instead, almost no requirements are put on the memory grow instruction for parallel execution. In particular, all bounds checks can be implemented with weak consistency which means that no atomic instructions are required on architectures that guarantee atomicity of individual loads and stores, such as x86.

Finally, even though the LLVM project and its Clang compiler support emitting the new instructions specified in the Threads Proposal, a WebAssembly runtime still needs to take additional steps before functions of the same WebAssembly module can be executed in parallel. Programs written in C or C++ require dynamic global initialization for global variables which must run before any other function of the program is called. Naturally, the global initialization must be complete before any other threads start executing any C or C++ function that was compiled to WebAssembly. The Clang compiler generates the special functions called __wasm_call_ctors and __wasm_call_dtors which must be called for global initialization and destruction, respectively. Similarly, when C++ programs use thread_local global variables, they must be initialized in every thread. The Clang compiler generates a special function to initialize thread-local storage called __wasm_init_tls. Our implementation calls all of these special functions correctly and ensures that no other WebAssembly functions are executed before the initialization is complete.

The Clang compiler additionally creates a WebAssembly global variable called __stack_pointer. Since C and C++ programs can take the addresses of stack variables and pass pointers to them to other functions but the WebAssembly stack cannot be accessed by memory instructions, the compiler instead generates code to store the stack variables at the location stored in __stack_pointer. Obviously,

every thread requires a separate stack. So, our implementation allocates a single WebAssembly page for every thread dedicated to its stack and sets the __stack_pointer variable accordingly. Note that global WebAssembly variables are not shared between threads, so every thread will correctly read its own stack pointer address.

4.4 Safety

We proposed using WebAssembly as an intermediate language so that it is possible to safely execute arbitrary, potentially untrusted user code. With its specification, WebAssembly lies some ground rules for the behavior of its execution. However, it does not prescribe how the execution of WebAssembly programs must be implemented by a WebAssembly runtime. As long as the execution of a WebAssembly program semantically follows the behavior described in the specification, a runtime has no additional restrictions to implement the execution. In particular, since WebAssembly programs cannot inspect the current state of the execution, such as the stack trace of the function that is currently being executed, or the current instruction pointer, a runtime may even modify programs to optimize the execution. Still, it must make sure that any potential trap may not be accidentally removed by an optimization.

In the remainder of this section, we discuss how our translator ensures that the execution of a translated WebAssembly program follows the semantics of the specification. In particular, we explain how the execution of the programs generated by the translator maintains the safety guarantees of the WebAssembly language.

4.4.1 Safety of Numeric Instructions

For most numeric WebAssembly instructions, Umbra IR has an equivalent instruction. For these instructions, a simple one-to-one mapping between WebAssembly and Umbra IR instructions can be used. Since both WebAssembly and Umbra IR use twos-complement integer arithmetic, both languages handle overflows in addition and subtraction identically, so no special instructions are required. Similarly, both languages use IEEE 754 floating-point numbers, so most floating-point instructions can also be translated directly.

Some numeric operations can sometimes have unexpected results or even lead to program crashes. To prevent safety issues, the WebAssembly specification precisely defines every numeric instruction and which outcomes are allowed for all possible inputs. When a numeric operation has no unexpected behavior, such as integer additions and subtractions as described above, all pos-

sible combinations of input values must always lead to a deterministic output value.

Integer Division. Integer division has two edge cases that are handled differently by different processor architectures: division by zero and overflow. In both cases the WebAssembly specification requires an implementation to always raise a trap. For a translator this means that it must generate additional code that checks whether the divisor is zero. Also, overflow in integer division when using twos-complement arithmetic only happens if the dividend is equal to the smallest possible negative integer and the divisor is -1. Thus, this special case can also be detected by generating additional code that checks for these two values in the dividend and the divisor. WebAssembly also has an integer reminder instruction for which the same handling of both special cases applies.

Integer Multiplication. Like division, integer multiplication can also lead to overflow, for example when multiplying the smallest possible negative integer by -1. However, unlike for division, this overflow is not the only possible overflow when multiplying two numbers in twos-complement. Naturally, the magnitude of the result of a multiplication is always larger than the magnitude of the factors. As a result, modern hardware handles overflow while multiplying by just truncating the result to the number of bits of the result type. Because this cannot lead to unexpected execution behavior, WebAssembly also defines integer multiplication such that the potentially overflowing result must always be truncated. Thus, for the translator this means no special handling for integer multiplication is required.

Bitwise Shifts. Conceptually, bitwise shifts simply multiply or divide by a power of two. However, especially higher-level languages require the shift amount to not exceed the number of bits in the shifted value. For example, shifting a 32-bit integer by more than 32 is considered undefined behavior in C and C++. Similarly, shifts by negative amounts are also often disallowed. To avoid ambiguities and unexpected results, WebAssembly specifies that the shift amount in a shift instruction is always treated as an unsigned integer modulo 2^n where n is the number of bits of the result type. Therefore, a translator must generate an additional bitwise-and instruction that ensures that the shift amount is less than 2^n.

Bit Counting. Modern CPUs usually offer special instructions to efficiently count bits in an integer: “count leading zeroes”, “count trailing zeroes” and “popcount”. Since these instructions are useful to efficiently implement many

algorithms, they are also available in WebAssembly as clz, ctz, and popcnt, respectively. The two instructions clz and ctz count the number of consecutive zero bits until the first one bit starting with the most-significant and least-significant bit, respectively. The popcnt calculates the number of bits that are set in an integer. Some CPU architectures have different results for these instructions when the input is zero. In WebAssembly, on other hand, all three instructions each only allow exactly one result when the input is zero. Thus, again, the translator must generate additional code that explicitly handles this case to ensure compatibility with the semantics of WebAssembly.

Float Functions. Several common operations on floating-point values such as rounding a number or computing the square root are available as WebAssembly instructions. These functions have several edge cases, especially when their operands are NaN or infinity values, for which the outcomes are precisely defined in WebAssembly. Our translator does not generate additional code to handle all of the numerous edge cases. Instead, it generates only a function call instruction which calls a runtime function provided by the translator. The runtime functions for every instruction can be implemented in a higher-level language, such as C++ for our translator to Umbra IR. While in general a function call is much more expensive than a single instruction, these float functions are complex enough to make any function call overhead hardly noticeable. Also, implementing them in a higher-level language as opposed to an assembly-like language like Umbra IR reduces the complexity of ensuring that all edge cases are handled correctly.

Float Conversions. To convert floating-point values to integers, WebAssembly offers two sets of conversion instructions: truncating and saturating. Both sets of instructions make sure that all conversions that would lead to an overflow in the resulting integer are handled. In the truncating instructions, all overflows raise a trap. The saturating instructions result in the smallest negative or largest positive integer when converting negative and positive infinity, respectively, and in zero when the argument is NaN. These conversion are usually available as single CPU instructions, however, CPUs often handle overflow differently. Thus, our translator implements the conversion in separate runtime functions as well, and only generates a call to one of the runtime functions when translating a float conversion instruction.

4.4.2 Memory Safety

The source for most critical safety issues and one of the main motivations for the creation of WebAssembly are invalid memory accesses. To ensure memory safety, the WebAssembly specification defines a special memory model with several restrictions (see also Section 3.1.2). One important component of the memory model is the contiguous memory area, also called the WebAssembly memory. WebAssembly programs access the WebAssembly memory using 32-bit addresses. The WebAssembly memory can be grown dynamically by WebAssembly programs but its size cannot be reduced. The second component of memory safety in WebAssembly is that the specification defines that all memory accesses which lie outside of the current memory size must raise a trap.

All constraints on the WebAssembly memory and how it can be used were chosen to allow for efficient implementation of the WebAssembly memory while maintaining safety guarantees. Conceptually, for every WebAssembly load and store instruction, an implementation has to check whether the referenced address is valid. Since the WebAssembly memory is only one contiguous area of memory, this can be implemented by a single comparison of the WebAssembly memory address with the current memory size, called *bounds check*.

Since WebAssembly memory pages are a multiple of common page sizes found in real CPU architectures, bounds checks can also be avoided entirely: The implementation must reserve 8 GiB of contiguous virtual address space. Then, only the first n pages must be made accessible, where n is the number of pages required for the current WebAssembly memory size, while all remaining pages must be configured to allow no reads or writes. All WebAssembly memory instructions can then be implemented by adding the WebAssembly memory address to the first address of the virtual address space. For all invalid memory accesses, the CPU will then generate a page fault which subsequently leads to the operating system terminating the process.

This solution using virtual memory addresses is suggested by Rossberg et al. to implement WebAssembly memory efficiently. As it requires no additional bounds checks, this approach incurs no overhead compared to native machine code. Also, on modern 64-bit architectures allocating 8 GiB of virtual memory is easily possible without requiring any special operating system support or memory layout of the process.

Because pre-allocating virtual memory for the WebAssembly execution is easy to implement and allows for efficient bounds checking, this approach is used by many current implementations of WebAssembly. Since after an illegal memory access the WebAssembly program must be terminated anyway, letting the operating system kill the program is a sensible solution.

Our motivation for using WebAssembly is to safely integrate UDOs into a database system. In particular, this means that the WebAssembly program is not the only code that is being executed in the process. Since we intend to inline user code directly into code generated by the database system from a SQL query, it is not possible to easily distinguish between the execution of user code and database-generated code. When the implementation of the WebAssembly memory in the query execution relies on the operating system to kill the process when an illegal memory access occurs, the entire database system could be affected. Of course, this goes directly against our goal to safely execute arbitrary user code in the database system.

Thus, our translator actually generates a bounds check for every WebAssembly memory instruction when translating it. Of course, doing so naively not only negatively affects the execution speed, but also can potentially heavily slow down the translation of WebAssembly programs.

Conceptually, a bounds check can be implemented by an integer comparison and a conditional branch instruction that jumps to a trap-block when the memory address is invalid. The trap-block then contains the code that raises a trap to terminate the execution. Therefore, every naive translation of a memory instruction leads to at least one additional basic block and one conditional branch instruction in the generated Umbra IR program. If we assume that the number of memory instructions in a WebAssembly program is linear in the amount of total instructions, the bounds checks result in a number of new blocks and branch instructions linear in the amount of total instructions as well. The large amount of new blocks and branch instructions slows down all analysis and optimization on the generated program. Since many optimization passes need to reason about the control-flow graph and the dominator relation of instructions, they require traversing all blocks and branch instructions.

To avoid generating many blocks and branch instructions when translating WebAssembly memory instructions, we instead extend Umbra IR by adding a dedicated boundscheck instruction. This instruction has two operands: The WebAssembly memory address representing the end of a memory access, and the current size of the memory. During execution, this instruction semantically compares the two operands and raises a trap if the first operand is larger than the second.

Now, when the translator translates a memory instruction, it first generates a load instruction to determine the current memory size. The current memory size is stored in a global variable which is accessible from all functions in all threads which are executing the same WebAssembly program. Then, it generates the code to calculate the WebAssembly memory address representing the end of the memory access. Since the number of bytes that are accessed by a memory instruction is known statically, e.g., a load of an 32-bit integer reads exactly

WebAssembly	Umbra IR

`i32.load 42`	`%wasm_addr_base = zext i64 %load.i32_arg`
	`%wasm_addr = add i64 %wasm_addr_base, 42`
	`%wasm_addr_end = add i64 %wasm_addr_base, 4`
	`%mem_size = load int64 @mem_size_global`
	`boundscheck i64 %wasm_addr_end, %mem_size`
	`%real_addr = getelementptr int8`
	`  %wasm_memory, %wasm_addr`
	`%load.i32_result = load int32 %real_addr`

Figure 4.4: Translation of the WebAssembly memory load instruction i32.load. To ensure memory safety, the translator generates a special boundscheck instruction.

4 bytes, together with the fixed offset specified as an immediate in WebAssembly memory instructions, the translator only needs to generate add instructions with constant operands.

Figure 4.4 shows the generated code including a bounds check when translating a WebAssembly memory load instruction. The WebAssembly instruction uses a fixed offset of 42, so this constant is added to the operand of the instruction which in this example is stored in the register load.i32_arg. To avoid any overflows when calculating the memory offsets, our translator first converts the operand to a 64 bit integer using a zero-extension instruction. After calculating the correct WebAssembly address, the end-address for the bounds check is calculated. In this example a 32 bit integer is loaded, so the constant 4 is added to the memory address. The boundscheck instruction then takes the end-address and the memory size as operands. Since the memory size can change during the execution of a program, it must be loaded from the global variable mem_size that contains the current memory size. When the bounds check does not terminate the execution, the real memory address is calculated by using a getelementptr instruction which adds the WebAssembly address to the base pointer of the WebAssembly memory which is stored in the register wasm_memory. Finally, the load instruction uses the real memory address to perform the actual memory load.

By generating one boundscheck instruction for every WebAssembly memory instruction the translator ensures the memory safety of the execution. Untrusted programs have no way to access memory outside of the address range reserved for the WebAssembly module. When an Umbra backend compiles an Umbra IR program containing boundscheck instructions, it can translate them to a condi-

```
mov 0x28(%rsp), %r14d # load the WebAssembly memory address into %r14d
lea 0x2e(%r14), %rax # Calculate the end of the memory access (4 + 42)
cmp (%rbx), %rax # Compare the end address with the memory size
ja  trap_block # Conditional jump to the trap block
mov 0x2a(%r13, %r14, 1), %r15d # Load 4 bytes into %r15d at address
# %r13 (base address) + %r14 (WebAssembly address) + 0x2a (42)
```

Listing 4.5: Translation of the Umbra IR code shown in Figure 4.4 into x86 machine code. The current memory size is stored in memory at the address rbx. The real memory address of the WebAssembly memory is stored in r13.

tional branch which consists of an integer comparison and a conditional jump instruction which jumps to a "trap block" if the bounds check is unsuccessful. The trap block then contains the machine code that causes the execution to terminate which could be a single function call to a function that throws an exception, for example.

Listing 4.5 shows how the Umbra IR generated for the WebAssembly load instruction in Figure 4.4 could be compiled to x86 machine code. The machine code first fetches the WebAssembly address, which in this example is stored on the CPU stack at offset 0x28. It stores the result in the register r14d which represents the lower 32 bits of the r14 register. This mov instruction also implicitly zero-extends the 32-bit value so that the same value is available as 64-bit integer in the r14 register. The next instruction calculates the end-offset for the bounds check. Instead of using an add instruction, small offset calculations can be encoded more efficiently on x86 using the lea instruction. Here, it adds 0x2e to the register r14 and stores the result in rax. The offset 0x2e is calculated by taking the fixed offset of the WebAssembly immediate (42) and adding the size of the memory access (4). The register rax is then compared to the value at the memory address which is stored in the register rbx. Here, rbx contains the memory address of the global variable that contains the current size of the WebAssembly memory. If the comparison determines that the memory access is out of bounds, the execution jumps to the trap block using the conditional jump instruction ja. Finally, the last instruction performs the actual memory load and stores the result in register r15d. The real memory address is calculated by using a x86 memory operand in the mov instruction. Semantically, the memory operand 0x2a(%r13, %r14, 1) is equivalent to %r13 + %r14 + 0x2a. The register r14 contains the WebAssembly address as calculated earlier and r13 contains the real memory address of the WebAssembly memory.

4.4.3 Optimization of Bounds Checks

As can be seen in Listing 4.5, multiple Umbra IR instructions are often combined into one x86 instruction. For example, zero extension usually is an implicit result of many instructions, and multiple additions followed by a memory instruction can be combined into a single mov instruction with a memory operand. Still, every memory access in WebAssembly leads to multiple instructions in the machine code. In the example, a single WebAssembly memory load instruction leads to two x86 instructions that load memory: one to read the memory size and another for the actual memory load.

Naturally, executing multiple machine instructions for each WebAssembly memory instruction leads to additional runtime overhead. Every bounds check is implemented as an integer comparison followed by a conditional branch instruction. For modern CPUs with sophisticated out-of-order execution, branch instructions can potentially lead to pipeline stalls which significantly delays and slows down the execution. A bounds check for a WebAssembly memory access is usually expected to succeed as it can only fail due to a programming error, which means that the branch predictor of the CPU will be able to predict branches and improved pipelining most of the time. Still, our experiments show that the bounds checks often reduce the performance of a program by more than 50%.

The bounds checks are crucial to ensure the memory safety of the execution, so we cannot remove them. However, memory accesses frequently occur in predictable patterns or loops with fixed upper bounds. When a WebAssembly function contains two adjacent memory accesses and a static analyzer can determine that one of the memory accesses always uses an address which is larger than the address of the other memory access, the bounds checks for both memory access can be combined into one. Since the WebAssembly memory has no holes and can only grow but never shrink, a bounds check for an address addr_a that is executed after the bounds check for an address addr_b always succeeds if $\text{addr}_b > \text{addr}_a$.

Listing 4.6 shows an example for optimizing adjacent memory accesses. The function init written in C accesses two fields of the struct Input. When this function is compiled to WebAssembly, the memory accesses could be translated into the two WebAssembly instructions i32.store 0 and i32.store 4. Both pop an i32 value and a memory address from the stack and add their immediate offset, 0 and 4, respectively, to the address. The unoptimized Umbra IR shows that both memory access have separate bounds checks.

Since through static analysis we can determine that wasm_addr2 is always greater than wasm_addr1, even if the exact value of wasm_addr1 is unknown, the bounds check for wasm_addr2 also implicitly includes the bounds check

```c
struct Input { int a; int b; };
void init(struct Input* in) {
    in->a = 123;
    in->b = 456;
}
```

(a) Original C code

```
(func $init (param i32)
    local.get 0
    i32.const 123
    i32.store 0
    local.get 0
    i32.const 456
    i32.store 4
)
```

(b) Generated WebAssembly

```
%wasm_addr1 = ; [...]
%wasm_addr1_end = add i64 %wasm_addr1, 4
boundscheck i64 %wasm_addr1_end, %mem_size
store int32 i32 123, %wasm_addr1
%wasm_addr2 = add i64 %wasm_addr1, 4
%wasm_addr2_end = add i64 %wasm_addr2, 4
boundscheck i64 %wasm_addr2_end, %mem_size
store int32 i32 456, %wasm_addr2
```

(c) Umbra IR with regular bounds checks

```
%wasm_addr1 = ; [...]
%wasm_addr2 = add i64 %wasm_addr1, 4
%wasm_addr2_end = add i64 %wasm_addr2, 4
boundscheck i64 %wasm_addr2_end, %mem_size
store int32 i32 123, %wasm_addr1
store int32 i32 456, %wasm_addr2
```

(d) Umbra IR with optimized bounds checks

Listing 4.6: Optimization of bounds checks for adjacent memory accesses. The optimizer determines that wasm_addr2 is always larger than wasm_addr1, so it combines both bounds checks into one.

for wasm_addr1. So, our optimizer combines them into one bounds check and eliminates the other.

Note that our static analyzer knows that WebAssembly memory addresses cannot be larger than 33 bits (32 bit memory address plus 32 bit immediate). Thus, adding a constant offset to wasm_addr1 to calculate wasm_addr2 using 64 bit integers cannot lead to integer overflow. The condition wasm_addr2 > wasm_addr1 always holds.

Similarly, memory accesses inside a loop often directly depend on the loop variable. Also, the upper bound of the loop variable can often be determined

```
void init(int* a, int n) {
    for (int i = 0; i < n; ++i) {
        // bounds check at address a + 4 * 2 * i + 4
        a[2 * i] = 123;
        // bounds check at address a + 4 * (2 * i + 1) + 4
        a[2 * i + 1] = 456;
    }
}
```

(a) C code with comments annotating where regular bounds checks are inserted.

```
void init(int* a, int n) {
    // bounds check at address a + 4 * (2 * (n - 1) + 1) + 4
    for (int i = 0; i < n; ++i) {
        a[2 * i] = 123;
        a[2 * i + 1] = 456;
    }
}
```

(b) C code with comments annotating how the bounds checks are optimized.

Listing 4.7: Optimization of bounds checks in loops. The optimizer can statically derive the upper bound of the loop and sees that all memory accesses depend on the loop variable.

statically. Listing 4.7 shows a C function that contains a loop for which the bounds checks can be optimized. The regular bounds checks as annotated in Listing 4.7 (a) are inserted for every memory access. Our optimizer detects that all memory accesses inside the loop depend on the loop variable i and the upper bound for the value of i inside the loop is n due to the loop condition. Therefore, the address of the highest memory access is $a + 4 \cdot (2 \cdot (n - 1) + 1)$, so the last bounds check must check the end of this memory access at address $a + 4 \cdot (2 \cdot (n - 1) + 1) + 4$.

The optimized bounds check is inserted before the loop and the bounds checks inside the loop are eliminated, as shown in Listing 4.7 (b). Since the number of executed bounds checks is now constant and independent of the value of n, the optimized program will execute significantly less bounds checks.

Both types of bounds check optimizations can eliminate many bounds checks from common WebAssembly programs, thereby increasing their performance while maintaining the memory safety. However, our optimizer must not modify the behavior of the program; it needs to ensure that the new optimized bounds checks always succeed if the bounds checks in the unoptimized program would have succeeded.

There are two cases that prevent the optimization of bounds checks: function calls and conditional branches. If a function contains a call to another function, the optimizer has to assume that the called function may grow the memory. If the memory is potentially grown between two bounds checks, the bounds checks obviously cannot be combined. Also, a bounds check must not be moved outside of a conditional branch. When the conditional branch is not taken at runtime but its bounds checks are still executed, the execution of the program could be terminated even without invalid memory accesses.

4.4.4 Resource Exhaustion

When WebAssembly programs are executed, they obviously use real resources that they share with the rest of the system, such as the available main-memory and the CPU. The resource utilization of potentially untrusted programs must be taken into account for the safety of the execution.

Since WebAssembly programs cannot allocate memory arbitrarily, keeping track of the memory utilization of a WebAssembly programs can be implemented efficiently. The WebAssembly instruction memory.grow is the only instruction for which an implementation potentially has to allocate new memory. Also, this instruction is not expected to be executed often, so it can be implemented as a function call into the runtime system. The function of the runtime system that handles this instruction can then decide in every call whether to allow the WebAssembly program to allocate more memory.

When WebAssembly functions are translated and compiled to machine-code functions, the resulting machine-code frequently uses the CPU stack. The CPU stack is used to store temporary variables if the limited number of hardware registers is not sufficient. Also, the CPU stack often needs to store return addresses of function calls so that a called function knows where to continue the execution when it returns. In particular when functions call themselves recursively, keeping track of the call stack is necessary.

WebAssembly functions are allowed to call each other arbitrarily and every function call can potentially require several bytes of memory from the CPU stack. However, the CPU stack usually has a very limited size and it is separate from the region of memory allocated for the WebAssembly memory. When machine code tries to access the CPU stack in excess of its capacity, the program is usually terminated by the operating system similar to an illegal memory access. The CPU stack is also not accessed explicitly by the generated Umbra IR as only the compilation backend generates real machine-code that may use the CPU stack. So, it is not possible to add explicit bounds checks for accesses of the CPU stack when translating WebAssembly programs.

Since the CPU stack is a limited resource and exceeding its capacity may lead to program crashes, the translator must prevent the translated WebAssembly programs to use an unbounded amount of CPU stack space. Conceptually, this can be implemented by keeping track of the current depth of the call stack. At the beginning of the translation of every WebAssembly function, the translator needs to generate instructions that load the current depth of the call stack, increase it, and then raise a trap if the depth exceeds a pre-defined limit. As an approximation for the amount of stack space a compiled WebAssembly function will potentially use, our translator counts the number of local variables that are defined in the function and adds this number to the current call stack depth. At the same time, it uses a relatively low limit of 1000 for the stack depth. Every local variable requires up to eight bytes of CPU stack space, so a limit of 1000 bounds the CPU stack utilization to approximately 8 KB. Our implementation was tested on Linux where programs usually receive several MiB of stack space, so limiting the WebAssembly execution to use 8 KB of stack space prevents program crashes. Often, the compiled machine code will actually use much less stack space as local variables can often be optimized to be stored only in registers.

4.4.5 Open Safety and Security Issues

WebAssembly is designed to avoid most common safety and security issues. The specification focuses largely on the execution of untrusted code in web browsers and how to achieve memory safety and the safety of individual instructions. However, using WebAssembly to execute UDOs in a database system poses additional security requirements which are not covered by WebAssembly.

Loss of Control

A database system usually executes queries from many different users. Often, multiple queries from the same or different users are executed in parallel. To fairly distribute the available hardware resources between all queries and users, the query execution engine must schedule all currently active queries onto the available CPU cores, main memory, and hard disk capacity. Umbra uses morsel-driven parallelism to efficiently distribute queries onto multiple CPU cores. A morsel can be processed by any worker in the system. Umbra's query execution engine and scheduler cooperate to dynamically adjust the morsel sizes to generate morsels that have a predictable runtime of only few milliseconds [WKN21].

Crucially, Umbra's scheduler is entirely cooperative, i.e., it relies on the execution of a morsel to finish within only a few milliseconds, but cannot

preempt the execution of a morsel. If a morsel is executed that contains a call to a function of a WebAssembly UDO, the query engine has no control over how long the execution of the WebAssembly function takes. In the worst-case, a WebAssembly program could contain an infinite loop. Since the scheduler currently cannot forcibly terminate the execution of a morsel, an infinite loop in WebAssembly effectively blocks at least one CPU core from doing any useful work.

WebAssembly does not disallow infinite loops. In general, a static analyzer also cannot detect whether executing a function always or dependent on the inputs leads to an infinite loop. Similar to bounds checks that ensure memory safety, our WebAssembly translator could add cancellation checks at the beginning of each loop. The cancellation checks must track how much CPU time was spent on executing the WebAssembly function so far and terminate the execution if the time exceeds a pre-defined limit.

The cancellation checks add significant runtime overhead and in many cases can prevent useful optimizations, such as loop vectorization. Preliminary experiments showed that for our implementation cancellation checks can increase the execution time of WebAssembly UDOs by more than 30%. Therefore, our translator does not generate any cancellation checks and the loss of control can only be solved by user intervention in our system.

Side-Channel Attacks

In our system, WebAssembly UDOs run in the same process as other queries and other parts of the system. A WebAssembly function cannot access arbitrary memory locations but is restricted to its own separate memory. However, the machine code that Umbra's compilation engine generates contains both instructions generated by relational operators and also instructions generated for the code of WebAssembly UDOs. Since the machine instructions generated for a WebAssembly program have no spatial or temporal boundaries to other instructions, i.e., the memory locations and the execution of all instructions are interleaved, malicious WebAssembly programs can use side-channel attacks.

For example, Spectre [Koc+19] is a famous side-channel attack that relies on speculative execution of modern CPUs and precise timing. WebAssembly UDOs are compiled to native machine code and directly executed on the CPU, so it is easily possible for WebAssembly programs to launch a Spectre-style attack. Spectre allows programs to read the contents of a memory location without any direct memory load instruction from that location. A WebAssembly UDO could therefore theoretically read memory outside of the WebAssembly memory. Since it runs in the main process of the database system, it could read confidential data that enables further security exploits.

Side-channel attacks for JIT-compiled programs such as the WebAssembly UDOs are hard to mitigate. Our WebAssembly translator could add a synchronizing instruction such as lfence on x86 after every conditional branch to prevent speculative execution. However, disabling branch prediction significantly reduces performance [Koc+19]. So, executing WebAssembly UDOs in Umbra still leads to potential security vulnerabilities and should therefore be disallowed for completely untrusted users.

4.5 Integration into the UDO Query Compiler

We translate WebAssembly programs to Umbra IR to safely execute UDOs in a compiling database system such as Umbra. Untrusted, potentially unsafe user code must first be compiled to WebAssembly and can then be translated to Umbra IR. The translator ensures that the generated Umbra IR program cannot lead to program crashes or illegal memory accesses. We call all UDOs that use WebAssembly as an intermediate language *WebAssembly UDOs*, independent of the high-level language that the original program is written in.

Conceptually, the user code is first processed by the UDO User Compiler (see Section 2.2.2) and then integrated into the execution environment of the database system by the UDO Query Compiler (see Section 2.2.3). Ideally, users should not need to change their code when it is safely executed using WebAssembly. Thus, the UDO User Compiler for WebAssembly UDOs should also accept different high-level languages that can be compiled to WebAssembly. C++ can be compiled to WebAssembly using the Clang compiler, so the UDO User Compiler can use the exact same API for the user code as described in Section 2.2.2.

The UDO User Compiler compiles the user code such that it can be processed by the UDO Query Compiler. As shown in Figure 2.1, the UDO User Compiler then passes the compiled program to the UDO Query Compiler. In regular UDOs, the compiled program is usually an object file and in our implementation also the program compiled to LLVM IR. For WebAssembly UDOs, the UDO User Compiler creates binary WebAssembly module files, instead. These module files are entirely self-contained and have no external dependencies. They can contain one WebAssembly function for each of the UDO functions accept, extraWork and process and have one function import that represents the emit function.

When a SQL query that contains a call to a WebAssembly UDO is executed, the UDO Query Compiler uses the WebAssembly translator to generate the Umbra IR program for the UDO. As for any UDO, the relational operator UDO_a

generates code that calls the UDO functions. For WebAssembly UDOs, the functions are directly available as Umbra IR functions, so UDO_a which also needs to generate Umbra IR, can directly create call instructions to the UDO functions.

4.5.1 Allocation and Initialization of the UDO State

The WebAssembly modules used as WebAssembly UDOs are programs created, for example, by the Clang compiler that compiles the user code for a C++ UDO to WebAssembly. Therefore, the UDO Query Compiler for WebAssembly UDOs must offer the same functionality as the UDO Query Compiler for C++ UDOs. In particular, in C++ every UDO function is implemented as member function of a class which means that the state of the UDO is allocated at the beginning of the query execution and is then passed to the UDO functions as the implicit first this parameter. Since WebAssembly functions can only access the WebAssembly memory, the state for a WebAssembly UDO must be allocated inside of the WebAssembly memory.

To allocate the state of a WebAssembly UDO, the UDO Query Compiler generates code that semantically executes the WebAssembly instruction memory.grow. It grows the WebAssembly memory by at least one page which will eventually be used as the state of the UDO. Growing the memory from code that does not originate from the WebAssembly program could potentially confuse a memory allocator implementation in the WebAssembly program as the memory size changes outside of the control of the allocator. We analyzed several memory allocators used by WebAssembly modules and saw that, in practice, all memory allocators handle external modifications of the memory size correctly.

To implement C++ constructors and destructors that initialize and clean up the UDO state, and similar functions in other programming languages, the UDO User Compiler for WebAssembly UDOs creates the two functions exports wasmudo_init and wasmudo_destroy in the generated WebAssembly module. If either is defined in a WebAssembly module, the UDO Query Compiler generates code to call them at the beginning or the end of the query execution, respectively.

4.5.2 Passing SQL Values to WebAssembly

As for UDOs written in C++, as described in Section 2.3.5, it is necessary to convert SQL values to a format that can be processed by WebAssembly. WebAssembly supports 32 bit and 64 bit integers natively and they use the same native twos-complement layout as SQL integers in Umbra IR, so the UDO Query Compiler can pass SQL integer values directly to the accept function as long as they are not NULL. Also, floating-point numbers are passed as f64 values to the

WebAssembly functions. Similarly, integers and floating-point numbers are passed back from the WebAssembly code to the query engine by taking the WebAssembly values of types i32, i64, or f64 and using them directly as SQL values.

4.5.3 String Handling

WebAssembly has no native type for strings but WebAssembly UDOs should also be able to use SQL attributes of types text or varchar. Also, the WebAssembly memory is entirely separate from the memory of the query engine, so it is not possible for a WebAssembly function to use a pointer to data that exists outside of the WebAssembly module.

Eventually, all string data that a WebAssembly function wants to access must to be copied to the WebAssembly module. Especially copying larger strings or binary blobs is expensive, so the UDO Query Compiler should not eagerly copy all strings that are passed to a WebAssembly UDO.

Instead, our implementation passes strings as values of type externref. Conceptually, these values reference data that is external to the WebAssembly module, which our UDO Query Compiler uses to reference the string data. To then access the actual string contents, our UDO User Compiler for WebAssembly UDOs provides the two functions wasmudo_string_length and wasmudo_extract_string. The function wasmudo_string_length takes an externref as argument and returns the length of the string. The function wasmudo_extract_string takes the following arguments: The externref for the string, an offset off, a WebAssembly address addr, and a size s. The function then copies s bytes to the memory area starting at addr from the string starting at offset off. The off and s parameters allow WebAssembly UDOs to only extract parts of input strings and to read only parts of a string in smaller chunks, if necessary.

When a WebAssembly UDO wants to output a tuple containing a text or varchar attribute, it must also pass an externref value to the emit function. By design, no WebAssembly instruction to create externref values exists, so our UDO User Compiler provides the additional function wasmudo_create_string. This function takes a WebAssembly address and a size and then creates a string with the given size by copying the data starting from the given WebAssembly address. It returns the new string as an externref value which can then be passed to emit. Our UDO Query Compiler implements this function by copying the data from the WebAssembly module into the internal string representation used by Umbra and then creating an externref value that references this internal string.

Of course, the three functions that access or create strings must ensure memory safety. The implementation of these functions in our UDO Query

Compiler ensures that all WebAssembly addresses are valid before it uses them. Also, the internal strings referenced by externref values are only deallocated when they are not referenced by any externref anymore. Since there are no WebAssembly instructions to duplicate externref values or to store them in memory, an internal string is always only referenced by exactly one externref. Therefore, it is trivial for the UDO Query Compiler to track the lifetime of an externref and deallocate internal strings correctly.

4.6 Evaluation

Our goal is to allow compiling query engines to execute untrusted UDOs which could contain malicious code. Thus, we first compile a UDO to WebAssembly as an intermediate step, and then translate the generated WebAssembly module to Umbra IR. To ensure the safety of the execution, the translator must insert runtime checks into the generated Umbra IR, such as bounds checks that ensure the memory safety.

We evaluated the impact of the runtime checks by measuring the execution time of different algorithms executed in Umbra using WebAssembly UDOs. As a comparison, we test equivalent UDOs written in C++. In fact, since the Clang compiler can compile C++ to WebAssembly, our benchmarks use the exact same user code for the WebAssembly and the C++ UDOs. Thus, potential differences in the execution time of both approaches directly show the overhead caused by runtime checks inserted by the WebAssembly translator.

In all experiments, we run the WebAssembly UDOs in three different configurations of the WebAssembly translator. The configurations affect how the WebAssembly translator generates the bounds checks for memory access, which can be one of “none”, “optimized”, or “full”. When full bounds checks are enabled, the translator creates one bounds check for every WebAssembly memory instruction that it translates to Umbra IR. Conversely, in the “none” configuration, the translator creates no bounds checks at all. Disabling all bounds checks can obviously lead to safety violations but allows us to establish a baseline of the overhead caused by compiling a C++ program to WebAssembly. Finally, when the bounds checks are optimized, the translator also generates all bounds checks, to ensure memory safety. We additionally enable the optimizations described in Section 4.4.3 to reduce the impact of the bounds checks on the execution.

As for our benchmarks for UDOs that do not use WebAssembly, we ran all our benchmarks on a NUMA machine with two Intel® Xeon® E5-2680 CPUs with 14 cores and 28 hyper-threads each and 128 GiB of DRAM per node. To

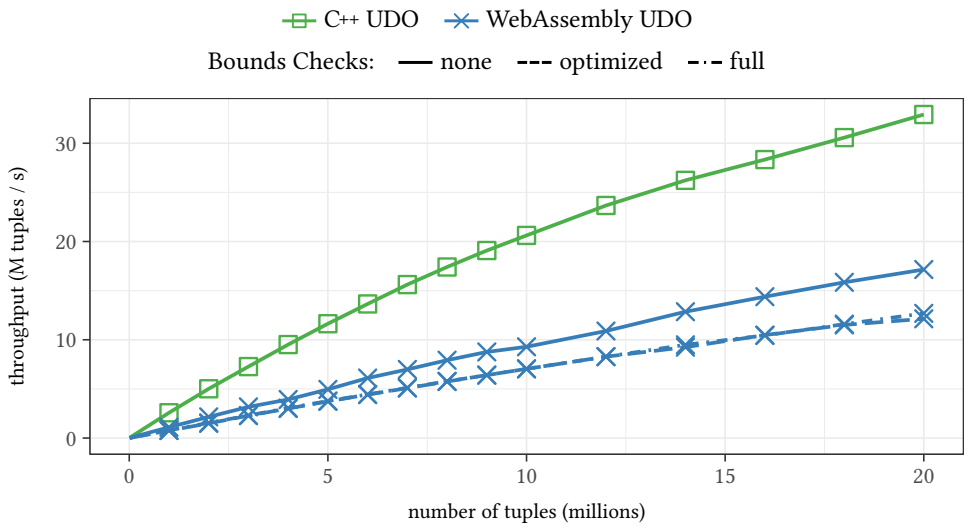


Figure 4.5: End-to-end throughput of k-Means using a WebAssembly UDO. The WebAssembly translator adds a significant compilation overhead which decreases the total throughput.

compile C++ programs to WebAssembly, we used Clang 17 and wasi-sdk 20[1]. We ran the same algorithms using the same UDOs as presented in Section 2.5.

4.6.1 Complex Iterative Algorithm: k-Means

In our first experiment, we tested the widely-used clustering algorithm k-Means. As described in Section 2.5.1, our implementation clusters two-dimensional points from a randomly generated data set with eight clusters and iterates exactly 10 times over all points. The UDO has no other exit conditions to make different runs of the benchmark directly comparable.

We ran the benchmark using a C++ UDO that provides no security. We also compiled the same C++ code to a WebAssembly module and then executed it as a WebAssembly UDO. Thus, the only difference in the execution of both approaches is the WebAssembly translator which adds additional runtime checks to ensure the safety of the execution of the WebAssembly module.

Running a WebAssembly UDO has one significant disadvantage over C++ UDOs; currently, memory addresses in WebAssembly are 32-bit integers, so a WebAssembly UDO can effectively only use 4 GiB of memory. To run a clustering algorithm such as k-Means, the implementation must scan over the input data multiple times which means that the entire input must be materialized

[1] `https://github.com/WebAssembly/wasi-sdk/releases/tag/wasi-sdk-20`

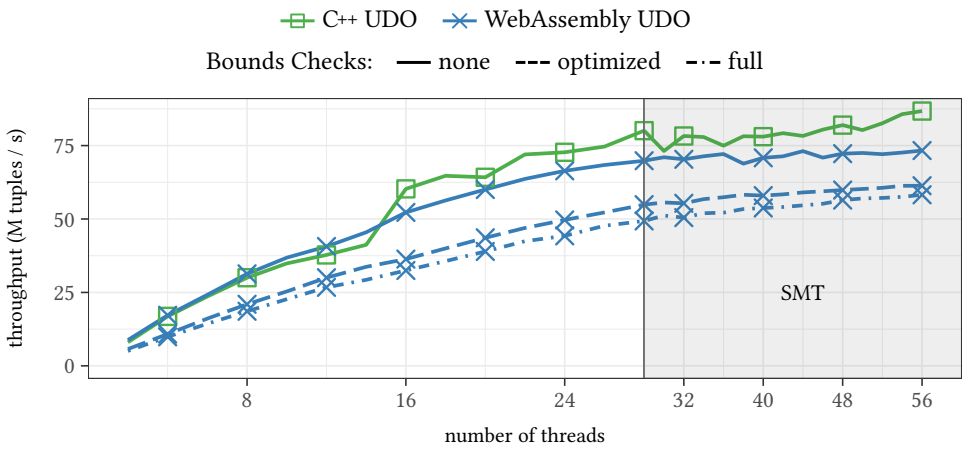


Figure 4.6: Scalability of runtime throughput of k-Means using a WebAssembly UDO. The WebAssembly UDO scales similarly to the C++ UDO, and reaches a comparable throughput when bounds checks are disabled.

and stored. For our implementation, each tuple of the input requires 32 B of memory, so the WebAssembly UDO implementation can process only up to $\frac{4\,\text{GiB}}{32\,\text{B}} = 2^{27} \approx 134 \cdot 10^6$ tuples. The real limit before the execution terminates with an out-of-memory error is even lower. Because of memory fragmentation and allocation overhead our implementation cannot reliably store more than 20 million tuples.

Figure 4.5 shows the total throughput of running the k-Means algorithm with both UDO types with up to 20 million tuples. For the largest data set, even the WebAssembly UDO that has no bounds checks reaches only half the throughput of the C++ UDO. However, this difference in the throughput is caused entirely by the additional compilation overhead of the WebAssembly translator; when running the C++ UDO, Umbra takes approximately 375 ms to compile the query while compiling the WebAssembly UDO takes around 890 ms when the bounds checks are disabled. Executing the query for 20 million tuples has very similar runtimes for both approaches: 255 ms for the C++ UDO and 276 ms for the WebAssembly UDO without bounds checks.

In C++ UDOs, the database system can run the most expensive steps of the compilation once when executing a create function statement. In Umbra, the statement then creates an LLVM module file which is cached and can be used by all queries that contain the UDO. For a WebAssembly UDO, the create function statement only checks the WebAssembly module for syntactic errors. The translator then has to translate the WebAssembly module to Umbra IR for every query. Because Umbra IR is designed specifically for efficient ad-hoc query compilation, it is not possible to pre-translate a WebAssembly module to

Umbra IR and then insert the pre-translated IR into another Umbra IR module. All Umbra IR instructions are stored in a dense array of opcodes which does not allow to move around instructions or functions after they are generated.

Translating a WebAssembly module to Umbra IR for every query also leads to significantly more Umbra IR instructions. For C++ UDOs, the UDO Query Compiler only generates one call instruction for every UDO function. These call instructions are only replaced by the cached LLVM code in Umbra's LLVM compilation backend. In a WebAssembly UDO, the WebAssembly translator always generates Umbra IR for all WebAssembly instructions which then must also be translated to LLVM IR instructions. For the k-Means algorithm, using the WebAssembly UDO results in almost 30.000 Umbra IR instructions when bounds checks are enabled, and around 22.000 instructions without bounds checks, but only 1100 instructions for the C++ UDO.

Interestingly, the differences between the throughputs of the different WebAssembly configurations as shown in Figure 4.5 are also mostly caused by the compilation time and not the actual execution time of the query. When a query contains bounds checks, the resulting LLVM module contains a huge number of conditional branch instructions. Every memory instruction requires a bounds check and every bounds check is implemented using a conditional branch instruction. Thus, the LLVM module contains one conditional branch for every WebAssembly memory instruction. When an LLVM module contains lots of conditional branches, LLVM's register allocator and other parts of its compilation backend take significantly longer to generate machine code. For the k-Means UDO, translating the WebAssembly module to Umbra IR with bounds checks, then translating it to LLVM IR, and finally using the LLVM compiler to generate machine code takes around 1.25 s.

The pure runtime without compilation overhead for both WebAssembly configurations that use bounds checks are similar: 350 ms for the full bounds checks and 325 ms for the optimized bounds checks. The C++ code and the resulting WebAssembly do not have many memory access patterns that can be optimized easily. Thus, our optimizer cannot significantly reduce the runtime overhead of the bounds checks.

The runtime throughputs without compilation overhead are also shown in Figure 4.6. We measured the throughput of the k-Means UDOs with an input size of 20 million tuples and different numbers of threads. The plot shows that the WebAssembly UDO without bounds checks has nearly identical throughput to the C++ UDO. When only a small number of threads are used, the WebAssembly UDO without bounds checks can slightly outperform the C++ UDO which has a higher constant overhead caused by linking the runtime dependencies required to execute C++ code as described in Section 2.3.5. The WebAssembly UDOs with bounds checks have a lower throughput but scale similarly when increasing

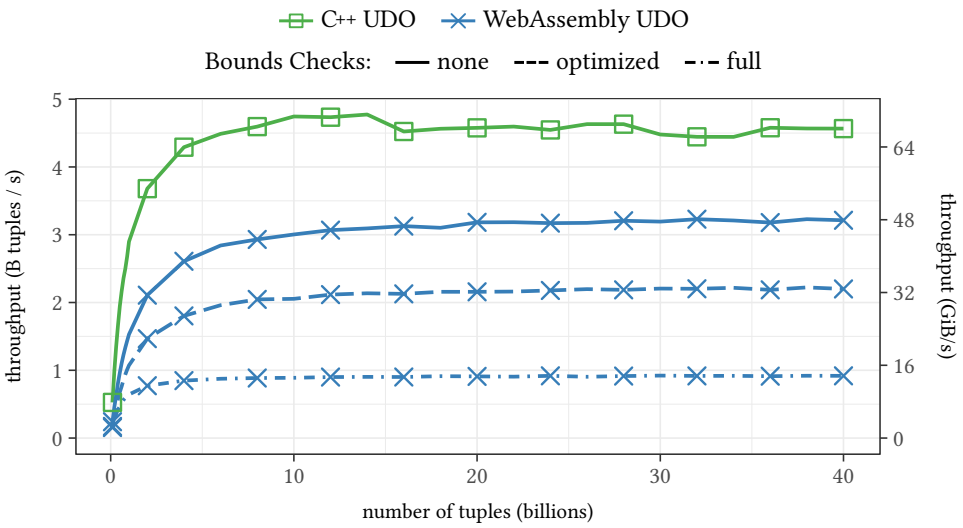


Figure 4.7: End-to-end throughput of linear regression using a WebAssembly UDO. Because the WebAssembly UDO requires significantly more memory loads than the C++ UDO, it cannot reach the same performance even without bounds checks.

the number of threads. So, while the bounds checks impact the throughput negatively, they do not affect the scalability of the query execution. In a bounds check, the current memory size is loaded using a weakly consistent atomic load which requires no expensive synchronization between the CPU cores.

4.6.2 Linear Regression

We also tested the performance of an implementation of simple linear regression with a quadratic target function using the least squares error function. As described in Section 2.5.2, our UDO implementation for linear regression solves the following problem: For given pairs of values x, y choose a, b, and c while minimizing the error term $\sum_i (a + bx_i + cx_i^2 - y_i)^2$.

The UDO implementation for the linear regression computes partial sums of the input data and in the end combines the partial sums to calculate the values for a, b, and c. Every thread only needs to store the partial sums but does not need to materialize any other state. Since the partial sums require only 64 B of memory per thread and no tuples of the input are stored, the WebAssembly UDO for this algorithm can process an arbitrary number of input tuples and is not limited to data sets smaller than the maximum addressable WebAssembly memory size of 4 GiB.

The C++ code from which the C++ UDO and the WebAssembly UDO are created contains mostly floating-point operations that calculate the partial sums and memory load and store operations that load the current value of the partial sums and update the values. The floating-point operations remain mostly unchanged when they are translated to WebAssembly, then to Umbra IR, and finally to LLVM IR, compared to the output of the Clang compiler when it generates LLVM IR directly from the C++ source code. Thus, when our WebAssembly translator generates no bounds checks, the WebAssembly UDO should reach a performance similar to the C++ UDO.

Figure 4.7 shows our measurements for the end-to-end throughput of executing the linear regression as a C++ UDO and as a WebAssembly UDO. For the larger data sets, the WebAssembly UDO can process up to 48 GiB/s of data when the bounds checks are disabled. However even without bounds checks, the WebAssembly UDO is significantly slower than the C++ UDO which reaches a throughput of almost 70 GiB/s. The machine code generated by Umbra for both UDOs is very similar and both use the same x86 instructions to compute the partial sums. The machine code of these two versions differs only in the memory operands of the generated instructions and how they are interleaved.

To load the current value of a partial sum, the machine code for the C++ UDO uses the memory operand 0x8(%rax). The register rax contains the base address of the memory location for the partial sums to which the memory operand adds the constant offset 8. The WebAssembly UDO on the other hand uses the memory operand 0x8(%r15,%rsi,1) for the same partial sum. The CPU calculates the memory address of this operand as %r15 + %rsi + 0x8. The register r15 contains the memory address of the start of the WebAssembly memory and rsi contains the WebAssembly memory address. The additional register in the memory operand requires at least one additional cycle to compute the final address which leads to noticeable overhead in the execution, but does not explain the entire difference in the throughput.

The main reason for the significantly reduced throughput in the WebAssembly UDO are the memory loads in the machine code. The core of the machine code in both UDO implementations is a loop that iterates over several thousand tuples of the input and updates the partial sum for every tuple. Our implementation calculates eight partial sums in total. The machine code for the C++ UDO contains eight load instructions that load the current values into the registers xmm0-xmm7, which are the floating-point registers of a x86 CPU. These eight load instructions are executed before the loop that iterates over the tuples. Inside the loop, the registers xmm0-xmm7 are updated to calculate the new values for the partial sums and then written back to memory.

The machine code for the WebAssembly UDO, on the other hand, interleaves the memory loads and stores. For every partial sum, it first loads the value

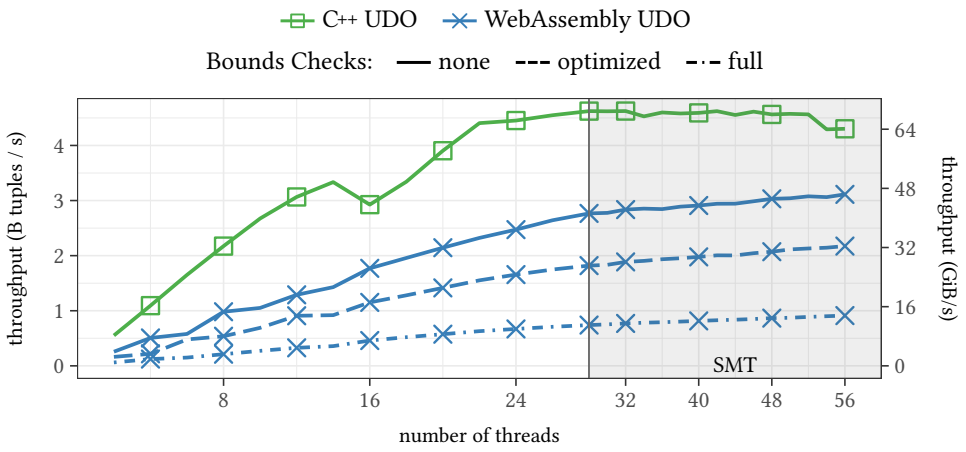


Figure 4.8: Scalability of runtime throughput of linear regression using a WebAssembly UDO. Because the WebAssembly UDO uses more memory loads, it cannot scale as well as the C++ UDO.

from memory into the register xmm0 or xmm1, calculates a new value, and stores it into memory again. Unfortunately, these interleaved loads and stores are inside of a loop which means that for every input tuple one iteration of the loop contains eight memory loads and eight memory stores. The LLVM optimizer is not able to move the memory operations out of the loop because it is missing the high-level alias information which is provided by the Clang compiler when compiling the C++ UDO. Our measurements show that executing the WebAssembly UDO leads to ten times more memory loads compared to the C++ UDO.

Naturally, the performance decreases even more when bounds checks are enabled. Figure 4.7 shows that for the "full" bounds checks, the throughput of the WebAssembly UDO is reduced to a third of the throughput without bounds checks. However, for this algorithm, our bounds check optimization can significantly improve the performance while maintaining the memory safety properties of WebAssembly. Because the core of the implementation of the linear regression consists of several sequential memory access with constant offsets, our optimizer is able combine all bounds checks for these memory accesses into one. In total, the optimizer reduces the number of bounds checks by 90%.

We also tested the scalability of the WebAssembly UDOs for linear regression. Figure 4.8 shows the runtime throughput without compilation overhead when executing the UDOs on different numbers of threads for the data set with 40 billion tuples. The measurements show that the WebAssembly UDOs do not scale as well as the C++ UDO, especially when the bounds checks are not optimized. As explained above, the WebAssembly UDOs in our implementation require

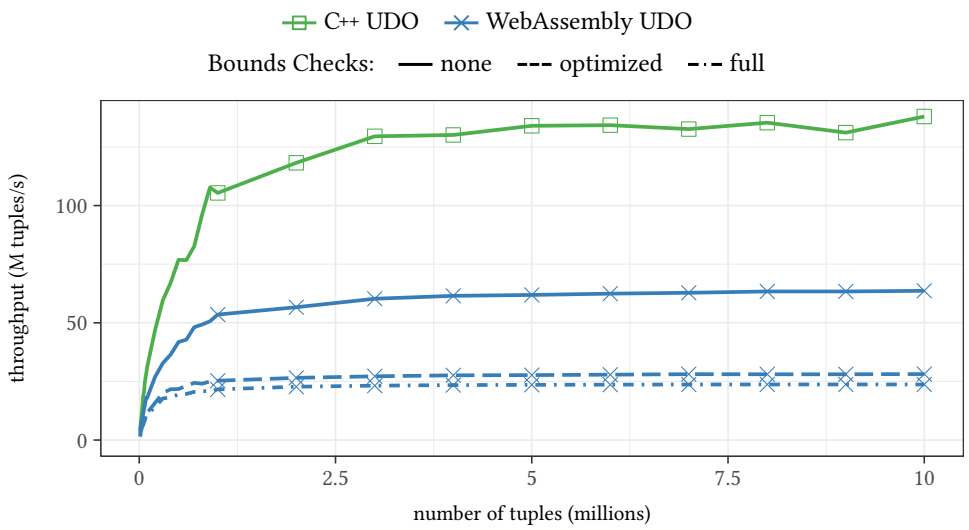


Figure 4.9: Runtime throughput of splitting comma-separated values into individual tuples using a WebAssembly UDO. Because all strings must be copied into the WebAssembly memory, the WebAssembly UDO has a significantly lower throughput than the C++ UDO.

significantly more memory loads which means that the available resources of the system are exhausted more quickly when adding more threads.

The figure also shows the positive impact of our bounds check optimizer. Figure 4.8 does not include the compilation overhead which brings the throughput of the WebAssembly UDO with optimized bounds checks slightly closer to the UDO without bounds checks. Translating and compiling the UDO with optimized bounds checks takes more than 2 s but only 1.5 s when the bounds checks are disabled. The runtime without compilation on all 56 threads is 16.4 s for the optimized bounds checks and 11.0 s without bounds checks. So, the difference of 0.5 s in the compilation time still has a small effect on the total runtime which means that for longer running queries, optimizing the bounds checks leads to even better total throughput.

4.6.3 Imperative Programming

Our last experiment shows how UDOs can easily express problems that are formulated using loops in imperative programming languages. We tested a UDO that takes a string of comma-separated integers, splits the string into the individual integers and then generates a new tuple for each integer. So, the UDO generates a variable number of output tuples for every input tuple.

In Section 2.5.3, we compared the UDO implementation to other approaches using recursive SQL statements or special unnesting functions supported by some database systems. The UDO easily outperformed all other approaches since the query engine could generate an efficient loop only when using the UDO.

For WebAssembly UDOs, this algorithm poses a different challenge. WebAssembly functions can take integer and floating-point numbers as arguments but cannot handle strings directly. Instead, our implementation passes strings to the WebAssembly module as externref values. The WebAssembly module can then use special functions that operate on the externref values to extract the actual bytes of the string, as discussed in Section 4.5.3. In any case, all strings passed to the WebAssembly module must be copied completely into WebAssembly memory before the WebAssembly UDO can access them.

Figure 4.9 shows the runtime throughput excluding compilation overhead when executing this algorithm with up to 10 million input tuples. Unfortunately, even without bounds checks, the WebAssembly UDO is significantly slower than the C++ UDO. The C++ UDO directly operates on the strings created by the table scan operator which fetches them from the table they are stored in whereas the WebAssembly UDO copies every string it receives. Also, the loop that finds all substrings of the input contains several conditional branches which prevents our bounds check optimizer from eliminating the most frequent bounds checks.

Still, when the bounds checks are optimized, the WebAssembly UDO is able to process over 25 million tuples per second. The strings in our generated input data have an average length of 65 B, so the WebAssembly UDO processes more than 1.5 GiB of string data per second. All approaches not using UDOs at all, as shown in Section 2.5.3, are slower by several orders of magnitude.

4.7 Summary

In this chapter, we presented our WebAssembly translator. The translator takes WebAssembly modules and generates equivalent Umbra IR instructions which can then be used directly by the query compilation system of Umbra. WebAssembly is a stack-based language while Umbra IR is a register-based language which uses static single assignment (SSA). We translate WebAssembly instructions directly to Umbra IR instructions by statically analyzing the state of the WebAssembly stack in a virtual stack. The virtual stack does not contain actual values but only register names which are then used as operands for the generated Umbra IR instructions. Our translator also directly generates phi instructions to efficiently translate WebAssembly control-flow instructions to SSA.

We use WebAssembly to safely execute UDOs containing untrusted code. While translating WebAssembly programs, we insert runtime checks that ensure the safety of the execution, such as bounds checks for every memory access. Naturally, these runtime checks cause a noticeable runtime overhead. Since memory bounds checks frequently occur in sequential patterns, our translator can often combine multiple bounds checks into one without affecting the semantics of the program.

Our evaluation shows that WebAssembly UDOs can be used to execute a large variety of algorithms efficiently and safely. However, the safe execution comes at a cost; the memory bounds checks add significant runtime overhead, often leading to a reduction of throughput by more than 50%. Especially when a UDO handles a large number of strings, the performance decreases by over 80% because all strings must be copied between the query execution system and the WebAssembly memory. Currently, WebAssembly memory addresses are 32-bit integers, so it is impossible to run algorithms that need to materialize their entire input on data sets larger than 4 GiB. Nevertheless, WebAssembly UDOs can still easily outperform alternative approaches that do not use UDOs by multiple orders of magnitude.

4.7.1 Future Work

The WebAssembly language is under active development. As discussed in Section 3.1.7, multi-threaded execution in WebAssembly is being proposed in the Threads Proposal. Two other new features that could significantly improve the performance of WebAssembly UDOs are currently being proposed: "Memory64" and "Multiple Memories". The Memory64 Proposal[2] adds new memory instructions that can use 64-bit memory addresses. Enabling 64-bit memory addressing in WebAssembly modules would allow WebAssembly UDOs to materialize inputs larger than 4 GiB and run complex algorithms on these larger data sets. The Multiple Memories Proposal[3] also modifies the memory instructions by adding an immediate to each memory instruction for a memory index. This allows WebAssembly modules to use multiple separate contiguous memories which can be accessed and grown independently. For WebAssembly UDOs, this feature could be used to efficiently implement passing string data between the query execution engine and the WebAssembly module. Instead of relying on externref values, the UDO Query Compiler could use a dedicated WebAssembly region that references the memory of the query engine. Then, WebAssembly

[2] `https://github.com/WebAssembly/memory64` (Accessed: 23 June 2023)
[3] `https://github.com/WebAssembly/multi-memory` (Accessed: 23 June 2023)

programs could directly access string data using regular memory instructions without costly copying of the string data.

CHAPTER 5

Conclusion

In this thesis, we presented a novel approach to integrate custom algorithms into modern database systems. *User-Defined Operators* (UDOs) allow users to write new algorithms in an imperative programming language such as C++ or Rust, which are then handled like a relational operator by an existing database system.

We identified that relational database systems are rarely used for complex analytical queries in modern data analytics. Other specialized systems offer better usability to run complex data-mining and machine-learning algorithms. Also, systems like Spark have easy-to-use programming interfaces that can be used to extend or customize existing functionality easily. Relational database systems, on the other hand, must be queried using SQL. Even though SQL was originally designed several decades ago, it has remained the dominant query language for database systems. Because SQL is declarative and, therefore, does not prescribe how a query engine must compute the result of a query, SQL gives query engines a lot of flexibility in their implementation. Modern database systems such as our compiling database system Umbra can efficiently process SQL queries on machines with hundreds of CPU cores and terabytes of main memory. Unfortunately, because SQL is declarative, writing imperative algorithms in SQL is not easily possible. Theoretically, SQL with recursive CTEs is turing-complete, i.e., any imperative algorithm can also be formulated in SQL. However, manually writing recursive CTEs is tedious, and database systems cannot execute them as fast as imperative code.

Running analytics in a relational database system still has several advantages. Critical business data is usually stored in a relational database system and then exported to be analyzed in other systems. Naturally, exporting and importing data leads to more overhead in data analytics. Also, relational database systems

offer strong transactional guarantees, which are lost when data is extracted from the system.

To enable easy-to-use data analytics in relational database systems, UDOs have a simple API that can be implemented in any imperative programming language. The API allows user code to interact with the database system by receiving tuples as inputs and generating tuples as output. Thus, User-Defined Operators act like any other relational operator in the database system, such as selections, aggregations, or joins, and can easily be combined with them. We demonstrated that UDOs can also be implemented in other database systems like Postgres. UDOs perform best when running in Umbra, where the user code can be inlined directly into the code generated by the query engine for other relational operators.

When the database system executes arbitrary code that could be malicious alongside code generated by native relational operators, the boundary between trusted and untrusted code is blurred. To execute UDOs safely, we introduced WebAssembly UDOs, which first compile the user code to WebAssembly as an intermediate step. WebAssembly is a low-level language initially designed to safely execute arbitrary code in web browsers. The language guarantees that all potential safety issues, such as invalid memory accesses, lead to a termination of the execution. We introduce a WebAssembly translator, which can translate WebAssembly programs to Umbra IR, the low-level language used internally by Umbra's compiling query engine. The translator generates code ensuring that all WebAssembly language safety guarantees are maintained.

Our evaluations showed that UDOs in Umbra can easily outperform specialized data analytics systems. Umbra employs code generation and morsel-driven parallelism to achieve excellent performance on modern hardware. When UDOs are executed in Umbra, they benefit from Umbra's fast query execution and achieve comparable throughputs to native code-generating operators. Using WebAssembly UDOs to ensure the safe execution of untrusted user code leads to significant runtime overhead. The throughput of WebAssembly UDOs is often reduced by 50% due to the runtime checks introduced by our WebAssembly translator. Nevertheless, WebAssembly UDOs outperform other data analytics systems by several orders of magnitude.

Bibliography

[19] "IEEE Standard for Floating-Point Arithmetic." In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[22a] *PL/pgSQL – SQL Procedural Language*. 2022. URL: https://www.postgresql.org/docs/14/plpgsql.html.

[22b] *Rust Programming Language*. 2022. URL: https://www.rust-lang.org/.

[22c] *Transact-SQL Reference*. 2022. URL: https://docs.microsoft.com/en-us/sql/t-sql/language-reference.

[Aba+16] Martín Abadi et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems." In: *CoRR* abs/1603.04467 (2016).

[AK09] Yanif Ahmad and Christoph Koch. "DBToaster: A SQL Compiler for High-Performance Delta Processing in Main-Memory Databases." In: *Proc. VLDB Endow.* 2.2 (2009), pp. 1566–1569.

[All70] Frances E. Allen. "Control flow analysis." In: *Symposium on Compiler Optimization*. ACM, 1970, pp. 1–19.

[BGN21] Maximilian Bandle, Jana Giceva, and Thomas Neumann. "To Partition, or Not to Partition, That is the Join Question in a Real System." In: *SIGMOD Conference*. ACM, 2021, pp. 168–180.

[Böt+20] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. "Scalable and robust latches for database systems." In: *DaMoN*. ACM, 2020, 2:1–2:8.

[BRN20] Altan Birler, Bernhard Radke, and Thomas Neumann. "Concurrent online sampling for all, for free." In: *DaMoN*. ACM, 2020, 5:1–5:8.

[BZN05] Peter A. Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution." In: *CIDR*. www.cidrdb.org, 2005, pp. 225–237.

[Car+86] Michael J. Carey, David J. DeWitt, Daniel Frank, Goetz Graefe, M. Muralikrishna, Joel E. Richardson, and Eugene J. Shekita. "The Architecture of the EXODUS Extensible DBMS." In: *OODBS*. IEEE Computer Society, 1986, pp. 52–65.

[CB74] Donald D. Chamberlin and Raymond F. Boyce. "SEQUEL: A Structured English Query Language." In: *SIGMOD Workshop, Vol. 1*. ACM, 1974, pp. 249–264.

[CH90] Michael J. Carey and Laura M. Haas. "Extensible Database Management Systems." In: *SIGMOD Rec.* 19.4 (1990), pp. 54–60.

[Cha96] Donald D. Chamberlin. *Using the New DB2: IBM's Object-Relational Database System*. Morgan Kaufmann, 1996.

[Cod70] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks." In: *Commun. ACM* 13.6 (1970), pp. 377–387.

[Coh87] Fred Cohen. "Computer viruses: Theory and experiments." In: *Comput. Secur.* 6.1 (1987), pp. 22–35.

[Cro+15] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Çetintemel, and Stan Zdonik. "An Architecture for Compiling UDF-centric Workflows." In: *Proc. VLDB Endow.* 8.12 (2015), pp. 1466–1477.

[Cyt+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph." In: *ACM Trans. Program. Lang. Syst.* 13.4 (1991), pp. 451–490.

[DG08] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." In: *Commun. ACM* 51.1 (2008), pp. 107–113.

[DHG20] Christian Duta, Denis Hirn, and Torsten Grust. "Compiling PL/SQL Away." In: *CIDR*. www.cidrdb.org, 2020.

[DLN19] Dominik Durner, Viktor Leis, and Thomas Neumann. "Experimental Study of Memory Allocation for High-Performance Query Processing." In: *ADMS@VLDB*. 2019, pp. 1–9.

[DLN21] Dominik Durner, Viktor Leis, and Thomas Neumann. "JSON Tiles: Fast Analytics on Semi-Structured Data." In: *SIGMOD Conference*. ACM, 2021, pp. 445–458.

[FN19] Michael J. Freitag and Thomas Neumann. "Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates." In: *CIDR*. www.cidrdb.org, 2019.

[FN21] Philipp Fent and Thomas Neumann. "A Practical Approach to Groupjoin and Nested Aggregates." In: *Proc. VLDB Endow.* 14.11 (2021), pp. 2383–2396.

[FPC09] Eric Friedman, Peter M. Pawlowski, and John Cieslewicz. "SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions." In: *Proc. VLDB Endow.* 2.2 (2009), pp. 1402–1413.

[Fre+20] Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. "Adopting Worst-Case Optimal Joins in Relational Database Systems." In: *Proc. VLDB Endow.* 13.11 (2020), pp. 1891–1904.

[GR21] Surabhi Gupta and Karthik Ramachandra. "Procedural Extensions of SQL: Understanding their usage in the wild." In: *Proc. VLDB Endow.* 14.8 (2021), pp. 1378–1391.

[Gra94] Goetz Graefe. "Volcano - An Extensible and Parallel Query Evaluation System." In: *IEEE Trans. Knowl. Data Eng.* 6.1 (1994), pp. 120–135.

[Gre+05] David Gregg, Andrew Beatty, Kevin Casey, Brian Davis, and Andy Nisbet. "The case for virtual register machines." In: *Sci. Comput. Program.* 57.3 (2005), pp. 319–338.

[HG21] Denis Hirn and Torsten Grust. "One WITH RECURSIVE is Worth Many GOTOs." In: *SIGMOD Conference*. ACM, 2021, pp. 723–735.

[HJ84] Donald J. Haderle and Robert D. Jackson. "IBM Database 2 Overview." In: *IBM Syst. J.* 23.2 (1984), pp. 112–125.

[HR83] Theo Härder and Andreas Reuter. "Principles of Transaction-Oriented Database Recovery." In: *ACM Comput. Surv.* 15.4 (1983), pp. 287–317.

[Ker+18] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. "Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask." In: *Proc. VLDB Endow.* 11.13 (2018), pp. 2209–2222.

[KLN18] André Kohn, Viktor Leis, and Thomas Neumann. "Adaptive Execution of Compiled Queries." In: *ICDE*. IEEE Computer Society, 2018, pp. 197–208.

[KLN21a] Timo Kersten, Viktor Leis, and Thomas Neumann. "Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra." In: *VLDB J.* 30.5 (2021), pp. 883–905.

[KLN21b] André Kohn, Viktor Leis, and Thomas Neumann. "Building Advanced SQL Analytics From Low-Level Plan Operators." In: *SIGMOD Conference*. ACM, 2021, pp. 1001–1013.

[Koc+19] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution." In: *IEEE Symposium on Security and Privacy*. IEEE, 2019, pp. 1–19.

[LA04] Chris Lattner and Vikram S. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *CGO*. IEEE Computer Society, 2004, pp. 75–88.

[Lei+14] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age." In: *SIGMOD Conference*. ACM, 2014, pp. 743–754.

[Lor74] Raymond A. Lorie. "XRM - An Extended (N-ary) Relational Memory." In: *Research Report / G / IBM / Cambridge Scientific Center* G320-2096 (1974).

[Mén+21] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. "Twine: An Embedded Trusted Runtime for WebAssembly." In: *ICDE*. IEEE, 2021, pp. 205–216.

[Mur+13] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. "Naiad: a timely dataflow system." In: *SOSP*. ACM, 2013, pp. 439–455.

[Mur+16] Derek Gordon Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martín Abadi. "Incremental, iterative data processing with timely dataflow." In: *Commun. ACM* 59.10 (2016), pp. 75–83.

[Neu11] Thomas Neumann. "Efficiently Compiling Efficient Query Plans for Modern Hardware." In: *Proc. VLDB Endow.* 4.9 (2011), pp. 539–550.

[NF20] Thomas Neumann and Michael J. Freitag. "Umbra: A Disk-Based System with In-Memory Performance." In: *CIDR*. www.cidrdb.org, 2020.

[NK15] Thomas Neumann and Alfons Kemper. "Unnesting Arbitrary Queries." In: *BTW*. Vol. P-241. LNI. GI, 2015, pp. 383–402.

[NR18] Thomas Neumann and Bernhard Radke. "Adaptive Optimization of Very Large Join Queries." In: *SIGMOD Conference*. ACM, 2018, pp. 677–692.

[Pal+18] Shoumik Palkar et al. “Evaluating End-to-End Optimization for Data Analytics Applications in Weld.” In: *Proc. VLDB Endow.* 11.9 (2018), pp. 1002–1015.

[Pas+17] Linnea Passing, Manuel Then, Nina Hubig, Harald Lang, Michael Schreier, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. “SQL- and Operator-centric Data Analytics in Relational Main-Memory Databases.” In: *EDBT*. OpenProceedings.org, 2017, pp. 84–95.

[Prö+21] Magdalena Pröbstl, Philipp Fent, Maximilian E. Schüle, Moritz Sichert, Thomas Neumann, and Alfons Kemper. “One Buffer Manager to Rule Them All: Using Distributed Memory with Cache Coherence over RDMA.” In: *ADMS@VLDB*. 2021, pp. 17–26.

[RM19] Mark Raasveldt and Hannes Mühleisen. “DuckDB: an Embeddable Analytical Database.” In: *SIGMOD Conference*. ACM, 2019, pp. 1981–1984.

[RN22] Maximilian Reif and Thomas Neumann. “A Scalable and Generic Approach to Range Joins.” In: *Proc. VLDB Endow.* 15.11 (2022), pp. 3018–3030.

[Ros+18] Andreas Rossberg, Ben L. Titzer, Andreas Haas, Derek L. Schuff, Dan Gohman, Luke Wagner, Alon Zakai, J. F. Bastien, and Michael Holman. “Bringing the web up to speed with WebAssembly.” In: *Commun. ACM* 61.12 (2018), pp. 107–115.

[Ros22] Andreas Rossberg. *WebAssembly Core Specification*. W3C Working Draft. https://www.w3.org/TR/2022/WD-wasm-core-2-20220419/. W3C, Apr. 2022.

[RSN22] Maximilian Rieger, Moritz Sichert, and Thomas Neumann. “Integrating Deep Learning Frameworks into Main-Memory Databases.” In: *4th International Workshop on Applied AI for Database Systems and Applications*. 2022.

[Sch+20] Maximilian E. Schüle, Jakob Huber, Alfons Kemper, and Thomas Neumann. “Freedom for the SQL-Lambda: Just-in-Time-Compiling User-Injected Functions in PostgreSQL.” In: *SSDBM*. ACM, 2020, 6:1–6:12.

[Sch+86] Peter M. Schwarz, Walter Chang, Johann Christoph Freytag, Guy M. Lohman, John McPherson, C. Mohan, and Hamid Pirahesh. “Extensibility in the Starburst Database System.” In: *OODBS*. IEEE Computer Society, 1986, pp. 85–92.

[Shi+08] Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. "Virtual machine showdown: Stack versus registers." In: *ACM Trans. Archit. Code Optim.* 4.4 (2008), 2:1–2:36.

[SL05] Herb Sutter and James R. Larus. "Software and the concurrency revolution." In: *ACM Queue* 3.7 (2005), pp. 54–62.

[SM21] Benedikt Spies and Markus Mock. "An Evaluation of WebAssembly in Non-Web Environments." In: *CLEI*. IEEE, 2021, pp. 1–10.

[SN22] Moritz Sichert and Thomas Neumann. "User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases." In: *Proc. VLDB Endow.* 15.5 (2022), pp. 1119–1131.

[SR86] Michael Stonebraker and Lawrence A. Rowe. "The Design of Postgres." In: *SIGMOD Conference*. ACM Press, 1986, pp. 340–355.

[TW17] Thomas N. Theis and H.-S. Philip Wong. "The End of Moore's Law: A New Beginning for Information Technology." In: *Comput. Sci. Eng.* 19.2 (2017), pp. 41–50.

[Win+20] Christian Winter, Tobias Schmidt, Thomas Neumann, and Alfons Kemper. "Meet Me Halfway: Split Maintenance of Continuous Views." In: *Proc. VLDB Endow.* 13.11 (2020), pp. 2620–2633.

[Win+22] Christian Winter, Jana Giceva, Thomas Neumann, and Alfons Kemper. "On-Demand State Separation for Cloud Data Warehousing." In: *Proc. VLDB Endow.* 15.11 (2022), pp. 2966–2979.

[Win+23] Christian Winter, Moritz Sichert, Altan Birler, Thomas Neumann, and Alfons Kemper. "Communication-Optimal Parallel Reservoir Sampling." In: *BTW*. Vol. P-331. LNI. Gesellschaft für Informatik e.V., 2023, pp. 567–578.

[WKN21] Benjamin Wagner, André Kohn, and Thomas Neumann. "Self-Tuning Query Scheduling for Analytical Workloads." In: *SIGMOD Conference*. ACM, 2021, pp. 1879–1891.

[WRP19] Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. "Weakening WebAssembly." In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019), 133:1–133:28.

[Yin+21] Lujia Yin, Yiming Zhang, Zhaoning Zhang, Yuxing Peng, and Peng Zhao. "ParaX: Boosting Deep Learning for Big Data Analytics on Many-Core CPUs." In: *Proc. VLDB Endow.* 14.6 (2021), pp. 864–877.

[Zah+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing." In: *NSDI*. USENIX Association, 2012, pp. 15–28.

[Zha+21a] Wangda Zhang, Junyoung Kim, Kenneth A. Ross, Eric Sedlar, and Lukas Stadler. "Adaptive Code Generation for Data-Intensive Analytics." In: *Proc. VLDB Endow.* 14.6 (2021), pp. 929–942.

[Zha+21b] Yuhao Zhang, Frank Mcquillan, Nandish Jayaram, Nikhil Kak, Ekta Khanna, Orhan Kislal, Domino Valdano, and Arun Kumar. "Distributed Deep Learning on Data Systems: A Comparative Analysis of Approaches." In: *Proc. VLDB Endow.* 14.10 (2021), pp. 1769–1782.

[Zou+21] Jia Zou, Amitabh Das, Pratik Barhate, Arun Iyengar, Binhang Yuan, Dimitrije Jankov, and Chris Jermaine. "Lachesis: Automated Partitioning for UDF-Centric Analytics." In: *Proc. VLDB Endow.* 14.8 (2021), pp. 1262–1275.

[ZWB12] Marcin Zukowski, Mark van de Wiel, and Peter A. Boncz. "Vectorwise: A Vectorized Analytical DBMS." In: *ICDE*. IEEE Computer Society, 2012, pp. 1349–1350.