Msc Projects at CWI and Databricks 2018-2019

CWI (www.cwi.nl) is the Dutch national research institute for computer science and mathematics, located in Science Park Amsterdam. The Database Architectures (DA) research group at CWI is specialized in software architecture for large-scale data systems (including database systems), and is quite well known in both industry and academia for its work on column stores and vectorized execution. The group has a lot of expertise in the interaction between computer architecture and database architecture, i.e. how to use modern and future hardware for data management software; think of integrating persistent RAM (e.g. 3D XPoint), RDMA networks, GPUs, FPGAs or even tensor processors in data management software. The DA group builds large software systems and has been involved in >5 spin-off companies in the past decade.

One option is to do an internal MSc project at CWI with **Hannes Muehleisen**, **Stefan Manegold** or **Peter Boncz**. In this case, you will be working from CWI at Science Park, and you would also be entitled to an internship grant of EUR ~550/m if your average grade in the MSc program is 8 or higher. Hannes and Peter work part-time for VU so would be your primary MSc advisor there, and can act as co-advisor for VU students in projects with Stefan (who works for Leiden University). We are open to students from other universities as well -- you then need a co-advisor from that university.
Our projects are high-tech and the DA group has a high reputation with data companies, so a MSc project here can either be a stepping stone to a PhD track or a job in a tech company. We will however be picky in selecting MSc students; the first application step is to **send us your CV** and **grade list**; the next would be to come over to CWI and talk about topics. Here is the list:

C1. In-Database Machine Learning on TensorFlow
C2. LeapFrogTrieJoin on Compressed Column Stores
C3. Self-Learning Whitebox Compression
C4. Piggy-Backing Integrity Checking on Scans
C5. Snapshotting Virtual Memory for In-Database Model Execution
C6. Automatically Detecting Database Performance Regressions
C7. Adaptive Partial Multi-dimensional indexing
C8. Progressive Database Query Evaluation
C9. Practical Hash-functions in Database Query Operators
CA. Evolving Graphs on Packed Memory Arrays

In the back of the document, you also find a list of projects at **Databricks**, with whom CWI collaborates. Databricks is a University of Berkeley spin-off that creates the world's most popular data science software stack: Spark. Databricks Amsterdam hosts the Spark performance teams (IO, query processing & benchmarking). Doing an internship there means you will be employed (and paid) in their office near RAI station. In addition to sending us your CV and grades, you will have to pass a Databricks interviewing process (US tech company style), as they seek MSc interns who could become Databricks engineers afterwards. Note that Databricks projects D6 and D8 are with **Dick Epema** of TU Delft. Here the list of Databricks topics:

D1. A Profiler for Compiled Spark Query Plans
D2. Adaptive Joins in Spark
D3. Native Vectorized Execution in Spark
D4. Native Regex Join in Spark
D5. Self-Learning Data Layouts for Databricks Delta
D6. Learning Minimal Test Cases from Spark Workloads
D7. Compressed Execution in Spark
D8. Building Benchmarks from Modern AI and Big Data Application Patterns

C1: In-Database Machine Learning on TensorFlow

Machine learning usually assumes a feature matrix representation of training and test samples, but when considering learning on enterprise data that stems from tables in a warehouse, usually a query is needed to produce this data from multiple queries, using joins and aggregations.

There has been recent work on structure-aware machine learning (e.g. AC/DC, DEEM2018) that shows that machine learning on such structured data could be done much more efficiently if we do not first compute the matrix, but rather learn on the not-yet-joined base tables, e.g. *push the learning through the database query.*

This novel class of algorithms has so far been demonstrated as a CPU algorithm. However, we know that deep learning, in general, performs up to 100x better on GPUs. In order to profit from GPUs (and potentially even tensor-processing hardware, TPUs) it is attractive to use deep learning frameworks such as TensorFlow to generate low-level GPU or TPU code from a higher-level description. Hence, the challenge of this project is to implement an AC/DC like algorithm on TensorFlow, and investigate its performance characteristics wrt CPU implementations.

You will be advised by Peter Boncz of CWI (Peter.Boncz@cwi.nl).

C2: LeapFrogTrieJoin on Compressed Column Stores

Normal database systems execute joins queries one join-at-a-time, i.e. with binary joins (a join between *two* input tables). Recently, theoretical work has shown that complex queries with multiple joins can be executed potentially orders of magnitude faster with so-called “worst-case-optimal-join” (WCOJ) algorithms, that execute a full join graph (multiple joins) in one go. The most well-known such algorithm is the LeapFrogTrieJoin (LFTJ).

Due to its theoretical pedigree, LFTJ has not been studiesd in depth from a low-level systems angle. We are interested in creating a machine-efficient LFTJ algorithm, that leverages many-core CPUs, columnar and compressed data storage and SIMD.

You will be advised by Peter Boncz of CWI (Peter.Boncz@cwi.nl).

C3: Self-Learning Whitebox Compression

Columnar stores on data formats such as Parquet or ORC apply column compression using techniques such as RLE (run-length encoding), Frame-Of-Reference (FOR), Dictionary and Delta-coding. It is well known that it is sometimes beneficial to combine these, e.g. first delta-code so you get differences, and then apply FOR on the differences. Still, FOR and DELTA are black-box compression methods, but in such case they are *combined* in sequence.

The idea of white-box compression is to decompose compression methods into even more basic operator expressions. Thus, one could store a *decompression-expression* together with a column chunk in a data block. To decode the data-block, one evaluates this expression on a vectorized query processor (..efficiently). This allows powerful (de-)compression schemes.

Taking that idea further, one can think of allowing many more kinds of functions in the expressions. Think of a column containing strings like “Customer0102389”; these could be (de-)compressed with an expression like sprintf(“Customer%07d”,PFOR(DELTA($1))). The first goal of the project is to realize basic white-box compression that can represent the classical compression methods RLE, FOR, DICT and DELTA. The stretch goal of the project is to develop a learning algorithm that given a dataset can automatically learn appropriate generic (de-)compression expressions.

You will be advised by Peter Boncz of CWI (Peter.Boncz@cwi.nl).

C4: Piggy-Backing Integrity Checking on Scans

Blind trust in the correct operation of computer hardware (CPU, RAM, Disk) is often misplaced. These devices break, and sometimes they do so subtly, e.g. flipping single bits. If those failures are not detected, database query results can be are arbitrarily wrong. We propose to “piggy-back” checksum calculation as base data on disk or in memory is read to answer a query. Then, this checksum can be compared to a known checksum for the data, and if there is a mismatch, an error can be raised. This

requires of course an efficient maintenance of those check sums, too, which is another dimension of the project. The project will also entail a real-world implementation of piggyback checksumming in a database system.
You will be advised by Hannes Mühleisen of CWI (hannes.muehleisen@cwi.nl).

C5: Snapshotting Virtual Memory for In-Database Model Execution
User defined functions (UDF) written in languages such as R or Python are a powerful feature to extend data management systems. However, loading model background data or other configuration is often prohibitively slow if all that is desired is for example making a prediction using the model. In this project, we propose to capture the entirety of used virtual memory used by a third-party UDF so they can be efficiently re-executed from the trained state as often as desired. Major challenges will be the capture, efficient storage and re-execution of memory images.
You will be advised by Hannes Mühleisen of CWI (hannes.muehleisen@cwi.nl).

C6: Automatically Detecting Database Performance Regressions
Measuring database performance is an important quality assurance tool for development. Over time, bugs are fixed and performance is hopefully improved. However, manually running a set of benchmarks every time a change is made is prohibitively expensive when done manually. Along the lines of Continuous Integration (CI), we propose a system to perform Continuous Benchmarking. For every version control check-in, a set of benchmarks is to be run and results (e.g. runtime, disk IO etc.) recorded. However, this creates a huge amount of numbers that are difficult to interpret. The main challenge of this thesis is to automatically and statistically accurately detect performance regressions or improvements according to a certain confidence level and the standard deviation of previous runs on same hardware, configuration, dataset size etc. The topic will also encompass a usable real-world implementation and integration of the new system.
You will be advised by Hannes Mühleisen of CWI (hannes.muehleisen@cwi.nl).

C7: Adaptive Partial Multi-dimensional indexing
The performance of database query evaluation largely depends on the availability of indexes (or data access structures in general) that support quick access to only the relevant portion of the data. Indexing all data is prohibitively expensive in terms of both storage required to store the indexes and time required to build the indexes and maintain them under updates. Selecting a limited set of indexes, restricted by storage space and creation-/maintenance-time, that yield the best performance for all queries of a given workload has been proven to be an NP-hard problem.
Adaptive partial indexing techniques like Database Cracking have emerged in recent years and proven useful and successful in scenarios where a-priori construction of complete indexes is unfeasible due to lack of resources of lack of a-priori workload knowledge, e.g., in exploratory data analysis. While intensely studied for single-attribute indexes / single-dimensional data, adaptive indexing techniques for multi-dimensional data have not been proposed or studied, yet. On "obstacle" is the fact that existing adaptive indexing techniques, just like classical single-dimensional indexing techniques assume a total sorting order, which is not given in multi-dimensional data. A first attempt towards adaptive partial indexing for multi-dimensional data could be to adopt the classical technique of reducing multi-dimensional data to a single dimension using space-filling curves (e.g., Hilbert-curve or Z-ordering), and apply, say, database cracking on top of that. In addition to exploring the opportunities and challenges of this route, other projects could explore the space of devising adaptive partial indexing techniques inspired by multi-dimensional tree structures (R-Trees, kd-trees) or gridding techniques.
You will be advised by Stefan Manegold of CWI (Stefan.Manegold@cwi.nl).

C8: Progressive Database Query Evaluation

Interactive exploratory data analysis requires that database management system (DBMS) provides the application (or end-user) almost instantaneously, say in less than one second, regardless of the data volume, with an initial approximated indicative answer for a given query, and continuously updates and refines this result until it converges to the complete and correct results, or the user has sufficient information to reviser the query, or abandon it entirely in favor of a new different query. DBMS fail to meet this requirement as they are built under the assumption that they need to provide the (only) the entire and correct query results “as quick as possible”, which might take minutes or even hours with complex queries over huge amounts of data. The challenge of this project is to investigate whether and how one can design and implement a middleware infrastructure on top of a DBMS, and/or (partly) modify the query execution engine of an open-source DBMS, in order to provide (or at least “simulate”/”mimic”) such progressive query evaluation functionality (efficiently and effectively).
You will be advised by Stefan Manegold of CWI (Stefan.Manegold@cwi.nl).

C9: Practical Hash-functions in Database Query Operators
Hashing-based techniques are an important alternative to sorting-based technique to efficiently implement complex and expensive database operations such grouping and joins, in particular for in-memory processing. The efficiency of the respective technique, and their implementations, largely depend in the quality and the efficiency of the actual hash function(s) used. In this context, “quality” assesses how well a hash functions randomizes and spread any given subset of the input domain into a (restricted) result domain, while “efficiency” assesses how much time or how may CPU cycles is/are required to evaluate the hash function, i.e., to calculate one hash value. In general, there is a tread-off: the better the quality, the lower the efficiency, and vice versa. Existing work focuses on assessing and maximizing the quality for general purpose hash functions, at the expense of neglecting their efficiency. The challenge of this problem is to investigate the trade-off between hash-function quality and efficiency specifically in the context of their deployment database query operations. The question is how to design specific hash functions that yield “good-enough” quality for the requirements of database query operations, while still being efficient to evaluate (i.e., calculate a hash value in just a few CPU cycles). Ideally, this problem can be addressed both empirically and theoretically.
You will be advised by Stefan Manegold of CWI (Stefan.Manegold@cwi.nl).

CA. Evolving Graphs on Packed Memory Arrays
The Packed Memory Array (PMA) is a mostly theoretical data structure that consists of normal arrays with “gaps” in the middle. They allow sequential accesses to memory when scanning, while improving the complexity, compared to standard arrays, of any insertion or deletion from O(N) to $O(\log^2 N)$.
The purpose of this project is to analyse, optimise and practically evaluate the performance and the suitability of PMAs in the context of graph algorithms (shortest paths, page rank, few others). Intuitively, the value of PMAs for graphs is that one can refer to nodes directly by position, so that traversing edges is a fast operation -- note that in an updatable database system, each edge traversal would require a join (requiring a pre-built index or on-the-fly hash table).
Specifically, we are interesting in how the mentioned (analytical) graph algorithms work while the graph is being updated, i.e. the graph is evolving all the time, because of a stream of updates.
You will be advised by Peter Boncz of CWI (Peter.Boncz@cwi.nl).

D1: A Profiler for Compiled Spark Query Plans
The Tungsten project in Spark introduced on-the-fly compilation of Spark code into Java code, which significantly enhanced its query execution performance. This advanced compilation method fuses all non-materializing operators in a pipeline into a single for-loop. A negative consequence, however, is that it has become more difficult to understand the performance of Spark scripts, which both affects end-users of Spark as well as the developers (who are interested in which operators to improve).

In this project, we would like you to add profiling and debugging information to the generated Java code, such that a profiler can relate lines of Java code to the relational operators in the Spark query plan, as generated by its Catalyst optimizer. Thus, it should become possible to relate (an estimate) of execution time to each individual operator (e.g. scans, aggregations, joins) in the plan. A stretch goal is to also allow debugging of such generated code, where we would like to be able to step through query plans and see not only which operator generated a Java statement, but also which line of a Spark query generated the line.
You will be advised by Stefan Manegold of CWI (Stefan.Manegold@cwi.nl) and a Databricks engineer.

D2: Adaptive Joins in Spark
Spark SQL implements different distributed join algorithms (broadcast, partitioned hash, partitioned sort and merge, nested loop). Additionally there are optimizations to handle difficult data cases: skew join optimization and range join optimization. However, the mechanism that chooses between all these is not reliable.
Choosing the join algorithm (e.g. when to do broadcast join) is done based on estimated statistics which can be extremely inaccurate or totally missing. Deciding when to use skew join or range join is done separately, based on user supplied hints which require that the user is able to diagnose the problem and determine some parameters (what is the skewed column and values, what is the range bin size).
The solution to this is to make the decision automatic by looking at actual data characteristics at runtime. Possible areas for research: query re-planning (a.k.a adaptive query planning), adaptive operators (shuffle, join).
You will be advised by Peter Boncz of CWI (Peter.Boncz@cwi.nl) and a Databricks engineer.

D3: Native Vectorized Execution in Spark
The Tungsten project in Spark introduced on-the-fly compilation of Spark code into Java code, which significantly enhanced its query execution performance. This advanced compilation method fuses all non-materializing operators in a pipeline into a single for-loop. The generated code operates on a per-row basis.
In this project, we would like you to implement a "vectorized", column-oriented execution mode for a subset of execution operators in Apache Spark, and to evaluate its performance against existing alternatives. This vectorized execution should be implemented in templated C++ with a clean separation between control logic in Java/Scala and execution primitives in C++ called through JNI that manipulate small vectors in off-heap memory. The C++ types chosen to represent values should be as small as possible, and be informed by the MinMax information that Parquet scans can deliver. The scope could be limited to vectorize expressions that come from Parquet scans.
Big differences in performance will surface especially in operations on Hash Tables, specifically aggregation and Join. Re-implementing these (probably aggregation first) in vectorized fashion will be stretch goals of this project.
Initially, we would rely on in the C++ templated primitive implementation on the compiler to SIMD-ize expressions. For certain hot-spots, one may also precompiled implementations based on intel intrinsics.
You will be advised by Peter Boncz of CWI (Peter.Boncz@cwi.nl) and a Databricks engineer.

D4: Native Regex Join in Spark
Several customers are struggling with important queries that do multi-pattern matching, for example, A join B ON (A.col LIKE “%B.col%”) where B contains tens of thousands of rows and A has billions of rows. Other similar cases include queries with a very long list of pattern-disjunctions in the WHERE clause. These queries are ripe for optimization with >10x performance gains that will delight our customers.

The goal is to recognize such bulk-pattern-matching queries and optimize them using an efficient library, for example, Intel's HyperScan. A first step could be to do workload analysis, and implement a specialized regex join that follows an broadcast index-nested loop strategy.
You will be advised by Peter Boncz of CWI (Peter.Boncz@cwi.nl) and a Databricks engineer.

D5: Self-Learning Data Layouts for Databricks Delta
Small Materialized Aggregates are a typical approach for optimizing highly selective queries in the big data world. They work best in conjunction with Clustered Indexes. Databricks Delta already supports both of these features, but indexing is currently manual: users have to explicitly call the OPTIMIZE table command and specify which columns to Z-order by.
We'd like to take this burden off the users' shoulders and have the system automatically and transparently adjust the data layout of Delta tables in order to maintain top query performance for the tables' specific usage patterns. This can be achieved by keeping track of workload statistics and devising some rules / heuristics for when and how to act on changes in the workload.
An additional idea is to take advantage of the DBIO cache and trigger data rewrites in the background when data is already brought in by queries anyway.
You will be advised by Peter Boncz of CWI (Peter.Boncz@cwi.nl) and a Databricks engineer.

D6: Learning Minimal Test Cases from Spark Workloads
Correctness is mandatory in our Databricks ecosystem, and our way to test it for customers is through random query generation. Whereas fuzzing techniques provide an automatic way to test our framework against a reference database implementation such as PostgreSQL, we are currently facing the following challenges. First, testing correctness while achieving full coverage of our Spark framework is difficult because Spark is not compliant with PostgreSQL specification. Thus, we want to be able to compare different Spark versions in such a way that the correctness guarantees still hold. To this end, we will explore techniques such as fuzzing the Spark configuration space or toggling query optimizer rules and experimentally evaluate their coverage. Second, complex queries are usually desired in order to have a good coverage for capturing non-trivial errors. Currently root causing such errors requires manual intervention, which is difficult because random queries can be very complex and incomprehensible. In this project, we want to extend the coverage of the random query generation and to automate detection and reporting of errors through query minimization techniques.
You will be advised by Dick Epema of TU Delft (d.h.j.epema@tudelft.nl) and a Databricks engineer.

D7: Compressed Execution in Spark
Lacking a schema, Apache Spark queries often operate without much extra knowledge on the data that can be exploited in a query. However, as most (volume of) data that goes into queries stems from Parquet files on Amazon S3, the existing MinMax information for these (both at the file and rowgroup) level, could be exploited to operate on **compressed** forms of the data.
In the case of decimals, which internally are represented as integers, the system should be able to operate on them in their smallest form (8, 16, 32, 64 or 128 bits). This also includes estimating bounds for decimals and integers after aggregation, under reasonable assumptions. Specifically, we should avoid the usage of BigInt wherever possible.
In the case of strings, many of these in the Parquet files are dictionary encoded. Ideally, we should operate on their integer codes rather than as strings. This allows to compare strings by codes and also avoid effort on hashing. The proposal is to represent dictionary-compressed strings as 64-bits values. Many of these would be pointing to a *dynamically unified dictionary,* but the non frequently repeating strings could also point to Java-based buffers, all hidden by a CompressedString class

interface. The unified dictionary would contain the strings, but also precomputed hash numbers and string lengths.
The final goal is devising a compact hash table format that exploits these data types (bit-packing data closely together) and avoids expensive data types (BigInt and String), mapping these to integer manipulations (pointer comparisons, hardware-supported integer operations) instead. It would save both CPU cache misses as the hash table is much smaller, and save significant computational effort.
You will be advised by Peter Boncz of CWI (Peter.Boncz@cwi.nl) and a Databricks engineer.

D8: Building Benchmarks from Modern AI and Big Data Application Patterns
Benchmarking big data systems in datacenters is challenging with the existing standard SQL suites such as TPC-DS because they are not representative for big data workloads in the cloud. In particular, they do not cover newer data science and learning jobs, unstructured data, and cloud deployments i.e., server-less. Database benchmarks currently also fail to capture the new usage trends of data intensive jobs, in particular the specific SQL patterns of joins, aggregations, and window functions. In this project we seek to design and experimentally evaluate a new benchmarking suite that is representative for our production Spark data science workloads. Using modern machine learning techniques that take into account execution patterns from production systems, we want to be able to derive models to automate building benchmarks and test workloads.
You will be advised by Dick Epema of TU Delft (d.h.j.epema@tudelft.nl) and a Databricks engineer.