Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management

Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauck,
Matthias Uflacker, Hasso Plattner
Hasso Plattner Institute
Potsdam, Germany
firstname.lastname@hpi.de

ABSTRACT

Research in data management profits when the performance evaluation is based not only on individual components in isolation, but uses an actual DBMS end-to-end. Facilitating the integration and benchmarking of new concepts within a DBMS requires a simple setup process, well-documented code, and the possibility to execute both standard and custom benchmarks without tedious preparation. Fulfilling these requirements also makes it easy to reproduce the results later on.

The relational open-source database Hyrise (VLDB, 2010) was presented to make the case for hybrid row- and column-format data storage. Since then, it has evolved from being a single-purpose research DBMS towards becoming a platform for various projects, including research in the areas of indexing, data partitioning, and non-volatile memory. With a growing diversity of topics, we have found that the original code base grew to a point where new experimentation became unnecessarily difficult. Over the last two years, we have re-written Hyrise from scratch and built an extensible multi-purpose research DBMS that can serve as an easy-to-extend platform for a variety of experiments and prototyping in database research.

In this paper, we discuss how our learnings from the previous version of Hyrise have influenced our re-write. We describe the new architecture of Hyrise and highlight the main components. Afterwards, we show how our extensible plugin architecture facilitates research on diverse DBMS-related aspects without compromising the architectural tidiness of the code. In a first performance evaluation, we show that the execution time of most TPC-H queries is competitive to that of other research databases.

1 INTRODUCTION

Hyrise was first presented in 2010 [19] to introduce the concept of hybrid row- and column-based data layouts for in-memory databases. Since then, several other research efforts have used Hyrise as a basis for orthogonal research topics. This includes work on data tiering [7], secondary indexes [16], multi-version concurrency control [42], different replication schemes [43], and non-volatile memories for instant database recovery [44].

Over the years, the uncontrolled growth of code and functionality has become an impediment for future experiments. We have identified four major factors leading to this situation:

- Data layout abstractions were resolved at runtime and incurred costs that sometimes had a disproportional overhead.
- Prototypical components have been implemented to work in isolation, but did not interact well with other components.
- The lack of SQL support required query plans to be written by hand and made executing standard benchmarks tedious.
- Accumulated technical debt made it difficult to understand the code base and to integrate new features.

For these reasons, we have completely re-written Hyrise and incorporated the lessons learned. We redesigned the architecture to provide a stable and easy to use basis for holistic evaluations of new data management concepts. Hyrise now allows researchers to embed new concepts in a proper DBMS and evaluate performance end to end, instead of implementing and benchmarking them in isolation. At the same time, we allow most components to be selectively enabled or disabled. This way, researchers can exclude unrelated components and perform isolated measurements. For example, when developing a new join implementation, they can bypass the network layer or disable concurrency control.

In this paper, we describe the new architecture of Hyrise and how our prior learnings have led to a maintainable and comprehensible database for researching concepts in relational in-memory data management (Section 2). Furthermore, we present a plugin concept that allows testing different optimizations without having to modify the core DBMS (Section 3). We compare Hyrise to other database engines, show which approaches are similar, and highlight key differences (Section 4). Finally, we evaluate the new version and show that its performance is competitive (Section 5).

1.1 Motivation and Lessons Learned

The redesign of Hyrise reflects our past experiences in developing, maintaining, and using a DBMS for research purposes. We motivate three important design decisions.

Decoupling of Operators and Storage Layouts. The previous version of Hyrise was designed with a high level of flexibility in the storage layout model: each table could consist of an arbitrary number of containers, which could either hold data (in uncompressed or compressed, mutable or immutable forms) or other containers with varying horizontal and vertical spans. In consequence, each operator had to be implemented in a way where it could deal with all possible combinations of storage containers. This made the process of adding new operators cumbersome and led to a system where some operators made undocumented assumptions about the data layout (e.g., that all partitions used the same encoding type). Instead of relying on operators to properly process data structures with varying memory layouts, Hyrise now follows an iterator-based approach. By accessing data through iterators, the implementation of new operators is decoupled from the implementation of new data storage concepts without compromising the flexibility. Operators can implement custom specializations for specific iterators, but execution falls back to the default iterator if no implementation exists. The iterator abstraction is explained in Section 2.3.

Benchmarking. Just as modern software development processes require that fundamental development steps, such as setting up the environment, building the code, and running tests should require only one step each, we believe that the same is true for benchmarks. This might seem obvious, but in our experience, also with other databases, both productive and research, this is almost never the case. In Hyrise, benchmarks are now single binaries that generate their data, run the queries, and print the results. As of writing, the TPC-H benchmark is included in the code base; the alternative data generator JCC-H [8], as well as the TPC-C and Join-Order [30] benchmarks are work-in-progress. Custom benchmarks can be easily added by creating table and query files, which are automatically executed by a generic benchmark runner (cf. Section 2.10). To facilitate reproducibility, all benchmark results contain the parameters relevant to their execution, including the Git commit hash, information about the utilized scheduler, thread count, and more. Benchmarks are executed by the CI process for each commit on the master branch to aid in the identification of performance regressions.

Memory Management & Metaprogramming. When the development of Hyrise started 10 years ago, C++11 was not yet finalized. This meant that memory management had to be done manually using *new/malloc* and *delete/free*, resulting in spurious segmentation faults and considerable debugging efforts. In the new version, Hyrise almost exclusively uses shared or unique smart pointers, which guarantee that objects are only deleted right after there is no remaining pointer to them. The overhead of reference counting for shared pointers seems to be well justified by the noticeable reduction in time spent on debugging memory management issues. Only when these pointers become an actual bottleneck, we look into whether the memory management in that particular component can be made more explicit.

Hyrise employs template metaprogramming to decouple the supported data types, storage layers, encoding types, and operators while at the same time preserving a high degree of compile-time type safety. We use `Boost.Hana` to automatically generate code for the different supported data types and for the resolution of our iterators. Mostly because of this, but also because of other C++17 features that are used all over the code base, Hyrise can only be compiled with current versions of GCC and Clang.

1.2 Building a Database with Students

We believe that the best way to learn about database systems is to program them yourself. Therefore, in addition to our PhD research projects, we use Hyrise in a graduate-level database class in which students develop, integrate, and test new features, such as additional join implementations, optimizer rules, or index structures. Not only does this hands-on experience help their understanding of databases and improve their programming skills; it also raises their interest to research database topics in greater depth, for example as part of their Master's theses. As a result, a considerable fraction of the Hyrise code has been developed together with students as part of their class assignments or thesis work.

Developing a database with students also entails one of the main challenges in the process, which is a relatively short developer turnover time. Even those Master's students who decide to specialize in the field rarely spend more than two years on the project. We address this challenge with strict code reviews, a high degree of test coverage (currently at >85%), and by encouraging new students to highlight existing components that are difficult to understand. On the technical side, we use automatic formatting (clang-format), multiple linting tools (cpplint and clang-tidy), sanitizers (ASan, UBSan, TSan, and Memcheck), and enforce most compiler warnings (`-Wall -Wextra -pedantic -Werror`).

The main criterion for building a maintainable research DBMS is that students without prior database knowledge can be brought up to speed and contribute code in six weeks. The fact that we were able to prove this repeatedly in our database seminars shows us that we are on the right track.

2 SYSTEM ARCHITECTURE

In this section, we give an overview of the new architecture of Hyrise. In places where it improves clarity, we refer to the original Hyrise architecture as *Hyrise1* and to that of the rewritten DBMS *Hyrise2*. After a general description of the high-level architecture, which follows the visualization in Figure 1 and focuses on query execution, we describe major components in their respective subsections, which are also given in the figure.

We decided to make as much functionality and as many components optional as reasonably possible. This decision is based on a learning from Hyrise1, where we found it difficult to isolate the root of performance issues because of the number of involved components. In Hyrise2, even core concepts, such as optimization, concurrency control, or scheduling, can be disabled. Without an optimizer, queries get executed close to how they are defined in SQL; for example, joins are only identified if `JOIN ... ON ...` is used. If MVCC is turned off, all tables are read-only, do not store information with regards to transactions or concurrency, and validation operators are not inserted into the query plan. If the scheduler is turned off, tasks are immediately executed in the same thread (while still guaranteeing progress). Similarly, Hyrise2 can be benchmarked with and without the just-in-time compiler, the network interface, or logging.

2.1 General Overview

Figure 1 shows the core components of Hyrise. We discuss them by following the process of answering a user-provided SQL query. Users have three options to submit queries to Hyrise. As a first option, we provide a command line interface, which can not only be used to submit queries, but also offers convenience functions for generating TPC-C or TPC-H benchmark tables, visualizing query plans, and toggling optional Hyrise components. As a second option of interacting with the database, Hyrise has an integrated TCP/IP server implementing the wire protocol of PostgreSQL. Users can send queries using PostgreSQL's interactive terminal *psql* or existing drivers for PostgreSQL. More details on how this interface is designed and implemented can be found in Section 2.5. Lastly, the third option is the SQL-C++ interface, which is used by our benchmark binaries (cf. Section 2.10) and also enables hand-written optimization of query plans. Currently, the benchmark binaries use the SQL-C++ interface. We are working on a benchmark implementation that utilizes the network interface.

All of the three entry points hand the user's SQL string to Hyrise's SQL Pipeline (cf. Section 2.6), which consists of multiple steps that transform the provided SQL string to an efficient query plan. The *SQL Parser* translates the SQL string to an abstract syntax tree expressed as C++ data structures. Next, the *SQL to LQP Translator* changes the abstraction level by creating a logical query plan (*LQP*), which is a directed acyclic graph (DAG) whose nodes loosely resemble the operations of the relational algebra. Finally, the *Optimizer* applies a series of rules to the LQP, which

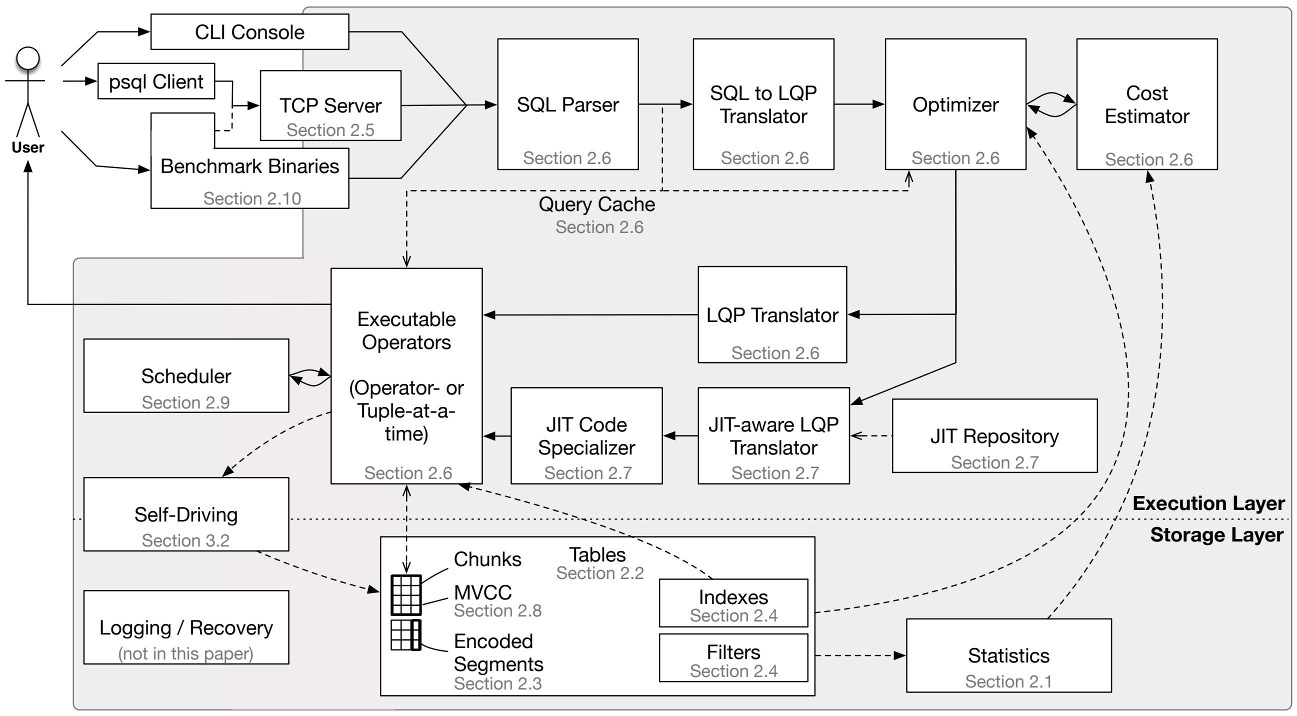


Figure 1: Hyrise Architecture Overview.

transform it into a more efficient, semantically equivalent version of the query. Some rules work by simply applying logical properties of the query plan (e.g., single-table predicates should almost always be pushed below joins), while others utilize information about the referenced tables that is only known at runtime. This information is collected from auxiliary data structures, such as general *statistics*, *indexes*, and *filters* (cf. Section 2.4). At the time of writing, statistics rely on histograms (equal height, equal width, equal distinct count) and other simpler metrics, e.g., the number of distinct values. Filters are probabilistic data structures that support approximate membership queries and allow the pruning of chunks (i.e., logically ruling out the necessity to access a chunk for a given predicate), whereas indexes return qualifying positions for a certain predicate directly without scanning through the data.

Hyrise implements a query plan cache, which can store both logical and physical query plans. Thereby, translation and optimization can be skipped to avoid doing these steps repeatedly for the same queries. While the cache does not yet auto-parameterize incoming queries, users can use prepared statements for queries with varying parameters. Both implicitly cached queries and prepared statements use the same caching data structures and shortcuts shown in the figure.

After optimization, the LQP is passed to the *LQP Translator*, which translates logical operators (such as predicates, joins, and sorts) to physical operators. Physical operators are concrete implementations of the logical concepts and more than one implementation might exist for a logical operator. For example, we implement joins as either sort-merge joins (cf. [2]), hash joins (cf. [4]), or nested-loop joins. Based on the optimizer's hints, one of these implementations is chosen for each logical operator. The result of the LQP Translator is another DAG, which we call Physical Query Plan (*PQP*). If the just-in-time compiler (cf. Section 2.7) is enabled, the LQP is not passed to the LQP translator, but to a *JIT-aware LQP Translator*, which creates JIT operators whose code is specialized using runtime information.

In both cases, the resulting PQP is then handed to the scheduler, which takes care of executing the translated operators. Once all operators have been executed, the resulting table is returned to the user.

Having looked at the entities that make executing a query possible, we now go over the data structures that are accessed for this. During execution, the primary table data as well as secondary data structures such as *indexes* are accessed by the operators. Tables are horizontally partitioned into *chunks*. Within a chunk, we have vertical partitions called *segments* where the segments across all chunks constitute the columns. Data within a segment might be encoded (cf. Section 2.3), for example using dictionary or run-length encoding. Additionally, chunks hold the data needed for multi-version concurrency control (MVCC, cf. Section 2.8). For more information on our storage layout refer to Section 2.2.

There are two more components shown in Figure 1: Self-Driving and Logging / Recovery. We envision future database systems to be self-driving [26], meaning that they autonomously adjust their configurations. Therefore, a *self-driving* component that assesses the database's current workload and tunes the configuration accordingly is part of Hyrise as well. This component is explained in more detail in Section 3.2. *Logging and recovery* are currently work-in-progress and are not described in this paper.

2.2 Storage Layout

Hyrise1 offers hybrid layouts, providing a maximum of flexibility with regards to storage layouts. This flexibility causes increased system complexity as well as runtime overhead by introduced abstractions. While the underlying system architecture of Hyrise2 supports hybrid layouts, we focus on columnar-oriented data for now. Figure 2 depicts the storage layout for an example table.

Storage layouts for HTAP database must support efficient read and write operations. This is often achieved by separating the data into read- and write-optimized partitions. Data is always added to write-optimized partitions. Update and delete operations invalidate entries in read-optimized partitions. From time to time, data has to be moved from write- to read-optimized partitions.

This transformation may happen by merging data of write-optimized partitions into read-optimized partitions [15, 27]. Merging may require re-encoding of already compressed data. Furthermore, the merge algorithm introduces implementation-specific complexity. For example, modifications to currently merged data have to be handled. In addition, the same data is encoded repeatedly during consecutive merge processes. Last, merge costs increase with the size of involved partitions.

To avoid a merge process in Hyrise2, tables are implicitly divided into *Chunks*, horizontal partitions of a certain size. Optimal chunk sizes are both data- and workload-dependent. A self-driving database system would decide on these autonomously. So far, our experiments showed suitable sizes to be between roughly fifty thousand and a few million records. The optimal chunk size is largely independent of the width of the table, both in terms of data sizes and number of columns in our default setup, where a column-based layout and dictionary encoding are used.

There are two types of chunks, mutable and immutable chunks. Initially, chunks are mutable and append-only containers. Data is added in a plain, unencoded fashion. When a chunk's capacity is reached it becomes immutable. Once this happens, encodings (cf. Section 2.3) can be asynchronously applied. Chunks encapsulate fractions of all of the table's columns, so-called segments.

There are a couple of advantages of the chunk-based approach. First, by implicitly partitioning the data, both multiprocessing (one core processes one chunk) and data placement, e.g., in NUMA environments, are simplified. Chunks can easily be distributed over multiple NUMA nodes, thereby leveraging multiple memory busses and CPUs for simultaneous processing.

Furthermore, auxiliary data structures like indexes and filters can be created on a per-chunk basis. Thus, these data structures are only created for those chunks where they yield a certain benefit. It also offers the flexibility to create different structures, for example, different index types for different chunks. The same can be applied to encodings: Some segments of a column might stay unencoded, others dictionary-encoded, and further segments run length-encoded.

Chunks are implicitly prunable entities. Thereby, in some cases, they can be excluded early from query processing without having to process the contained data. This can be achieved by using approximate membership query properties of filters (cf. Section 2.4) or characteristics of certain encoding types.

Partitioning the data into chunks has some drawbacks that need to be mitigated. First, it introduces the memory cost of storing per-chunk metadata. If, however, chunk sizes are chosen to be hundreds of thousands or millions of rows, this is not an issue as we show in Section 5.2. Second, for some encoding types, chunks may introduce redundant storage of information. For example, chunks encoded using the dictionary encoding store values that occur in all chunks over and over again in every chunk-local dictionary. On the other hand, if the overlap of values across chunks is low, the size of the dictionary can be kept low, and the number of needed bits for the attribute vector is reduced. Thus, choosing the optimal chunk size is a tradeoff between memory overhead and flexibility.

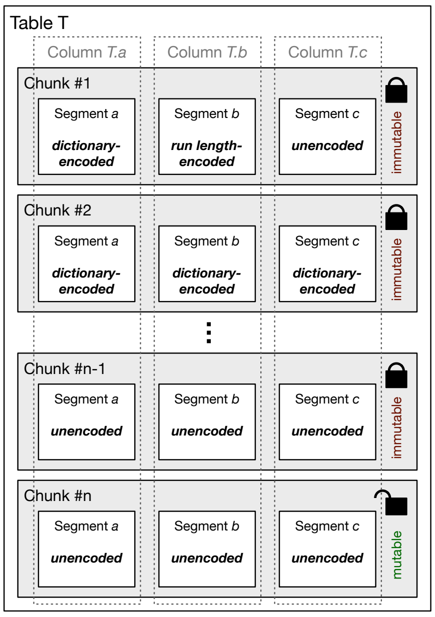


Figure 2: Depiction of Hyrise's storage layout for an exemplary table T with n chunks and three attributes.

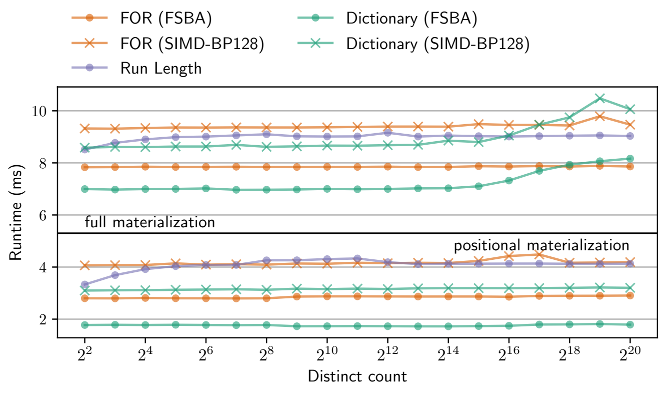
2.3 Segment Encoding

To our best knowledge, all modern columnar and memory-resident databases employ some form of column encoding. This is to (i) compress data and reduce memory consumption, (ii) better utilize the available memory bandwidth by increasing the entropy, and (iii) increase performance since operations on integer-encoded columns can be vectorized and processed by modern CPUs more efficiently. This effect is even stronger when relational operators can operate on encoded data without prior decoding.

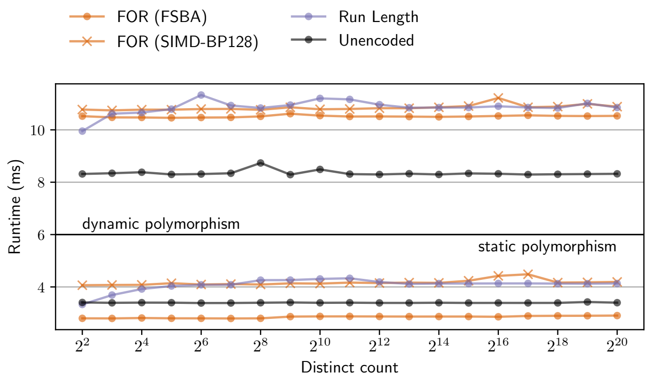
Hyrise supports both logical (i.e., mapping input data to an integer representation) and physical (i.e., further compressing integer codes) encoding schemes (cf. [13]). The implemented logical schemes include frame of reference, run length, and order-preserving dictionary encoding. The physical ones include fixed-size byte alignment and SIMD-BP128 for null suppression. Logical and physical encoding schemes can be arbitrarily combined so that existing logical schemes can profit from a new physical encoding without modification.

Hyrise1 includes several encoding and compression schemes, but as mentioned above, no abstraction layer separated the data layout and the execution engine. This caused maintainability and performance issues and led us to formulate the following requirements for an encoding framework in Hyrise2:

- The encoding framework should be an abstraction layer where operators do not need to be implemented for each added encoding type.
- Still, implementing specialized access methods for certain encodings should be possible. For example, scans on dictionary-encoded columns should search for the integer value id, without having to decompress the data.
- Performance should be on par with manually optimized encoding schemes. This means that the compiler should be able to statically resolve the abstractions without having to resort to virtual method calls in hot loops.



(a) Full vs. positional accessing.



(b) Effect of polymorphism.

Figure 3: Runtime comparison for an aggregation accessing 25% of 1M integer values. Left: impact of decoding the full vector upfront vs. positional decoding. Right: impact of static polymorphism via C++ templating vs. dynamic polymorphism as done in the previous version of Hyrise.

- Abstractions should be analyzable by the compiler to the point where auto-vectorization (i.e., the automatic generation of SIMD code) is possible.
- The framework should seamlessly cooperate with our long-term plans on just-in-time query plan compilation.
- Instead of defining encodings per column, it should be possible to have different encodings in different segments.

We fulfill these requirements by heavily relying on static polymorphism, i.e., C++ templates. For each implementation of an encoding type, an *iterable* is implemented. These iterables inherit from a common base class using the curiously recurring template pattern (CRTP). The base class provides the `with_iterators` interface to operators, but instead of implementing this method using virtual inheritance, the CRTP is used to statically call a protected `_on_with_iterators` method. Operators pass a functor (i.e., a lambda or closure) to `with_iterator` as shown in Listing 1. Optionally, `with_iterators` takes a position list, which is used to selectively iterate over the values that, for example, are the result of a previous scan operation.

```
segment_iterable.with_iterators(
  [&](auto left_it, auto left_end) {
    for (; left_it != left_end; ++left_it) {
      const auto left = *left_it;

      if (!left.is_null() && predicate(left.value())) {
        matches.emplace_back(left.chunk_offset());
      }}});
```

Listing 1: Implementation of a vectorizable[1] scan operator using the iterators provided by the encoding framework

Not only the iterators, but also the functors (in the example, the predicate) are resolved at compile time. This allows us to define an adaptable and flexible encoding framework that avoids virtual method calls. Using iterators allows us to hide implementation details of encoded columns, which eases the maintainability of operators. In case query compilation is used, the iterators also provide efficient accesses for tuple-at-a-time processing without incurring virtual method calls. A downside of this approach is that the number of template instantiations grows exponentially. For a scan on a single column, we instantiate $|DataTypes| * |EncodingSchemes| * |Comparators|$ templates, resulting in a compile time of up to five minutes for the most complex operators. To prevent that cost from slowing down our development, the static resolution only takes place for *Release* builds; *Debug* builds use conventional virtual method calls.

The same iterator model can also be used to implement hybrid data layouts. A row-oriented segment type can provide iterators for each included attribute. Because the iterators are resolved at compile-time, accesses to attributes within one tuple would result in contiguous memory accesses.

Figure 3 shows two micro-benchmarks evaluating the performance of our encoding schemes and the encoding framework. As Hyrise is optimized for HTAP workloads, which include frequent positional accesses, random access iterators play an important role. Figure 3a shows the overhead of decoding the encoded vector beforehand (cf. [25]) compared to using positional random access iterators. For most encodings, positional accesses are 2−3× faster, even when decoding large position lists. For typical OLTP queries with short position lists, the advantage is even more pronounced.

The performance advantage of static polymorphism over dynamic polymorphism, i.e., virtual method calls, is shown in Figure 3b. In this benchmark, we aggregated a set of randomly chosen positions (25% of 1M integer values). No matter the encoding type, the cost of static polymorphism is significantly lower, with the biggest improvement being a factor of 3×.

2.4 Indexes and Filters

During execution, two types of secondary data structures are used to reduce the amount of data accessed: (i) secondary indexes and (ii) filters, which are lightweight probabilistic data structures for chunk pruning. Moreover, Hyrise uses these data structures during query optimization for cardinality estimation.

Secondary Indexes. Indexes in Hyrise yield qualifying positions for one or more predicate(s). Three secondary index structures are implemented: (i) adaptive radix trees (ART, cf. [31]), (ii) B-trees[2], and (iii) the group-key index [16]. The group-key index has been developed particularly for Hyrise. It builds on compressed position lists and exploits order-preserving dictionaries.

[1]To enable the compiler's auto-vectorization, it is helpful to first iterate over a constant number of rows at a time. This is not shown in the example for the purpose of brevity.

[2]Google C++ B-tree: https://code.google.com/archive/p/cpp-btree/

Filters. Filters are space-efficient (i.e., significantly smaller than a secondary index) auxiliary data structures, which allow the pruning of chunks (similar to partition elimination) for a given predicate. Hyrise supports min-max filters, counting quotient filters [37], and pruning-optimized histograms, which are comparable to adaptive range filters [3]. The latter two are not only capable of pruning but also support selectivity estimation.

Both indexes and filters are created on a per-chunk basis on immutable chunks and not globally for the whole table so that no maintenance cost is caused by inserts, updates, or deletes. This simplifies the code base and avoids computational overhead in forms of logging or heavy updates of large data structures. However, it is conceptually possible to add, e.g., a B-tree index to mutable chunks when required in OLTP scenarios. As filters and indexes are chunk-local, they can be selectively created for chunks where the estimated performance improvement outweighs the necessary memory space.

An important difference to previous work on filters is that, in Hyrise, they are integrated with the query optimizer, instead of being only used in the execution phase. Doing so enables optimizations that can only be used at query planning time: First, chunk pruning can be propagated through conjunctive predicate chains down to the plan node that initially represents the input table (cf. [6]). That plan node is configured to skip chunks that would later be excluded by one of the predicates. As a result, the number of accessed rows is reduced from the start and not only at the location of the respective predicate. Second, in case chunk pruning has a significant impact on the selectivity of a predicate, this knowledge can be exploited for operator-reordering, which would not be possible when pruning is done later in the execution phase. Similarly, we plan to use indexes not only in the execution phase but also for estimating cardinalities (cf. [32]).

2.5 Networking

Hyrise implements the wire protocol of PostgreSQL [40]. Reusing existing wire protocols is common [41] for new systems for several reasons. First, existing clients and drivers can be reused. This is advantageous for database products as well as for research platforms as it offers the possibility to access Hyrise from many programming languages and makes it accessible to many users. Second, tools such as *Wireshark* can be used to investigate how the sent and received PostgreSQL messages are encapsulated in network packets. As Hyrise is a research platform, we only implement the features needed for receiving SQL queries and returning results, but do not implement features such as user authentication or SSL. This keeps our implementation lean. The wire protocol uses TCP/IP, and our server is implemented using the asynchronous network features of *Boost.Asio.*

2.6 SQL

Figure 4 depicts the different steps in our SQL Pipeline. A dedicated *SQLPipeline* class is the main entry point to everything related to query execution. It takes an SQL string as a parameter and returns one or more tables. Optionally, all intermediary artifacts can be inspected by the developer in their text or graph forms. This was designed based on our experience with other databases where, in some cases, it is difficult to even understand the path that an SQL query takes through the system and which steps are involved. In the following paragraphs, we describe the steps of the SQL pipeline.

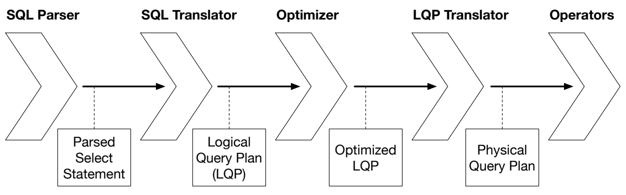


Figure 4: The different steps of the SQL Pipeline, leading from an SQL string to executable operators

Parsing. The SQL parser transforms the SQL query string into data structures that can be accessed and modified programmatically. When we started to add SQL support to Hyrise, we found no easy-to-use component that would transform an SQL string into an Abstract Syntax Tree (AST) that uses C(++) structs. While open-source databases come with their own parser (e.g., PostgreSQL), we found these to be too interwoven with the rest of the database for our purposes. Thus, we built a standalone C++ SQL parser based on Flex and Bison and released it as open source software[3].

SQL-to-LQP Translation. Having parsed the SQL query string into an Abstract Syntax Tree, which still resembles the structure of the SQL Query, we now translate the AST into a *Logical Query Plan (LQP)*, which is based on the relational algebra. Each edge in the LQP represents a table, either a user-defined table that is stored in the central *Storage Manager* or an intermediary table generated by the previous operator. It could also be a user-defined SQL view, which we have stored as its LQP and can embed into the query plan at this point. Operators do not need to perform expensive materializations of intermediary results, but can also pass positional references to the next operator. Having these references avoids expensive materializations. These operators are represented as nodes, which hold information about their input relations and attributes and parameters for their execution. LQP nodes are, however, not executable operators but only form the basis for logical query optimization. They get translated into physical operators only after all optimization steps have been performed. An example of a Logical Query Plan is shown in Figure 5.

Most LQP nodes, such as Predicates, Joins, or Aggregations, are one-to-one representations of their equivalent in the relational algebra. One node type that stands out is the *Projection*, which is our workhorse for most non-trivial column operations. This includes more complex logical operators (nested AND/ORs), string manipulation, but also the execution of subselects: In the initial LQP, all subselects are expressed as sub-LQPs attached to a Projection node close to the point of their first use. As such, subselects are executed as if they were stand-alone queries. For non-correlated subselects, this is done only once. For correlated subselects, the query plan contains placeholders that are replaced with the correlated attributes during the execution. Obviously, this is quite inefficient, which is why the optimizer later rewrites the LQP into a more efficient, join-based version.

Optimization. All optimizations are achieved by rules that are executed on the Logical Query Plan (LQP). Rules are maintained by the optimizer and are part of an optimization pipeline that contains single-pass and multiple-pass rules. While some rules need

[3]Hyrise SQL Parser: https://github.com/hyrise/sql-parser

to be executed only once (e.g., the substitution of constant expressions), others can be re-executed if the LQP has been changed by a different rule or if they themselves improve the resulting LQP in multiple passes. A rule takes an LQP as a modifiable input and returns whether it has modified that LQP. The information on whether a rule modified the LQP is used by the optimizer to decide whether iterative rules should be executed again. At the end of every rule stands a valid LQP. Thus, the optimization process can be skipped or stopped after a certain time, e.g., if the expected runtime of the query is determined to be too low to warrant further optimization efforts.

Out of the eight currently implemented rules, we use three examples to highlight how these rules operate on the LQP: the *Predicate Pushdown Rule*, the *Join-Ordering Rule*, and the *Chunk Pruning Rule*. The Predicate Pushdown Rule is a rule that is almost always applicable: For every LQP, it makes sense to execute cheap filtering predicates as early as possible, that is, before more expensive joins or aggregations. Currently, it is applied to all trivial predicates. Correctly estimating the cost of more complex predicates (such as nested predicates or LIKE expressions) is work in progress. The Join-Ordering Rule is an example, which relies on the statistics component to gather the estimated selectivities of the join predicates and on the cost estimator to estimate the cost of the different joins. These joins are then ordered using DpCcp [34] in what is considered to be the most effective order. Finally, our third example rule, the Chunk Pruning Rule, uses the LQP and the filter information discussed in Section 2.4 to identify chunks that will not contribute to the final result. While the LQP is not structurally changed by this rule, the table nodes at the bottom are augmented with the information of which chunks do not need to be passed to the first operator (cf. Section 2.4).

LQP-to-PQP Translation. Finally, the LQP has to be translated into a PQP. As described above, there are potentially multiple physical implementations of a logical operator. The optimizer has already left hints in the LQP nodes. An example of such a hint is when a logical predicate node contains the information that a secondary index can and should be used. Starting from the bottom, each LQP node is now translated into one of the available physical operators. Because all decisions have already been made by the optimizer, nothing of great interest happens here. This is different for the JIT-Aware LQP Translator, which we describe in Section 2.7.

Prepared Statements and Query Plan Caching. One goal in the design of the previous steps was to keep the SQL Pipeline lean, which is why the cost of query planning is comparably low. Still, commonly reoccurring queries can profit from previously generated query plans. We treat SQL Prepared Statements and Query Plan Caching similarly. In both cases, we store a mapping from an SQL query string to a Physical Query Plan. The only difference is that for Prepared Statements, entries in this mapping are manually maintained, while the query plan cache is limited and automatic eviction takes place. For Prepared Statements, we store placeholders instead of actual values. Before the execution of such a statement, these placeholders are replaced with actual values.

Regular SQL queries are currently cached with all parameters left in place. Automatically replacing these with placeholders would increase the number of cache hits but at the same time introduce further complexity. For example, for skewed workloads, where the reuse of a previously generated query plans may not be safe [5], caching PQPs might not always result in the best

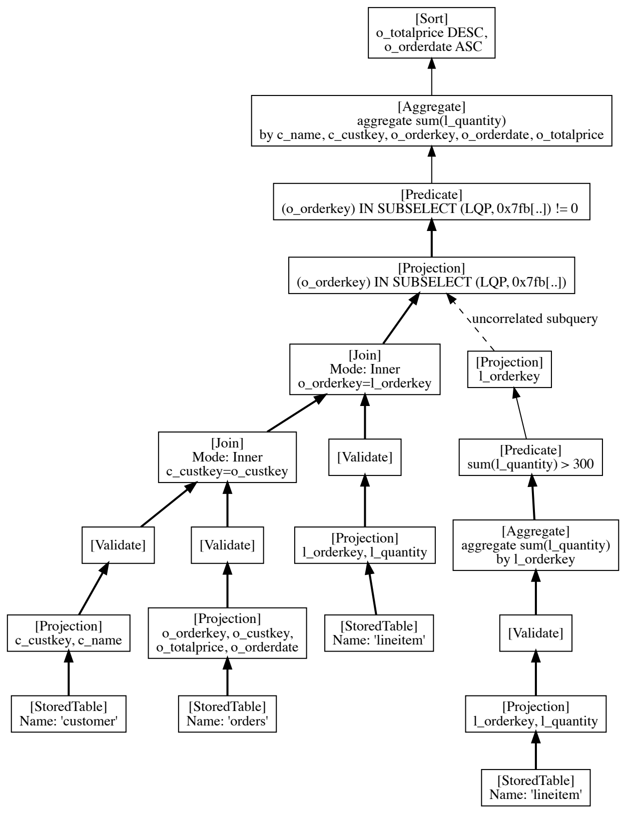


Figure 5: A sample visualization of a Logical Query Plan, here of TPC-H query 18

execution strategy. For these cases, we can easily switch the cache to only cache optimized LQPs, which are then re-optimized once the actual parameters are bound to the placeholders. Because the optimizer has already operated on the cached plan, we can save on optimization time compared to starting the optimization from scratch. Deciding when a PQP is safe to cache and reuse without re-optimization is subject to future work.

2.7 Just-in-Time Compilation

Hyrise's JIT engine is based on three components: The *JIT Repository*, which holds JITtable operators; the *JIT-Aware LQP Translator*, which translates Logical Query Plans into a chain of JITtable operators; and the JIT Code Specializer, which optimizes the operator code and fuses multiple operators into a single loop.

It is based on a code specialization approach, where, during development, generalized C++ code is written for each operator. This code still contains the virtual method calls for calling the next operator, the switches for different column and encoding types, or checks for null values. At runtime, when this information is available, the JIT compiler replaces these abstractions with their concrete values. This is done using LLVM's ORC (On Request Compilation) interface. For the examples, this means that virtual method calls to the next operator can be inlined, type switches can be removed and replaced with the known type, and checks for null values can be removed if the column is known to be non-nullable. The result of this optimization step is a single binary that represents all logical operators between two pipeline breakers. Because of this, we can optimize across operator boundaries and profit from keeping values in the CPU caches (or even registers) as well as from (auto-)vectorization of the code. We believe that this specialization approach is a better fit for our goals as it does not require developers to write code-generating code. The downside of this approach is a high complexity of the specializer component.

We maintain two execution engines, one with and one without support for just-in-time compilation, for three reasons. First, having two engines allows us to compare their performance and identify bottlenecks. Second, despite all effort to make the development of JIT operators as simple as possible, we consider it to still be more complicated as developing traditional operators. Not only do developers have to understand the JIT model, but they also need to familiarize themselves with our specialization engine in order to be able to debug or profile the resulting binary code. Especially for students who work on the project for just a semester, this becomes a limiting factor. Third, most of the research projects evaluated on Hyrise are orthogonal to the optimizations achieved with JIT. The filter-based access pruning methods presented in Section 2.4, for example, can prune the same number of chunks no matter if the remaining chunks are evaluated using traditional or JITted operators. As such, the relative performance impact is comparable, even if the absolute performance of the JITted queries is better.

In some cases, we can achieve a 22x performance improvement over the traditional, operator-based approach, for example when complex expressions have to be calculated. At the current stage, not all operators are implemented as JIT operators (most notably the different joins) and the JIT-aware LQP Translator automatically falls back to non-JITtable implementations. Also, the encoding-specific optimizations have not made it into the JIT component yet, so table scans on dictionary-encoded segments have to decompress all values. Because of this, the JIT component has to be explicitly enabled.

2.8 Concurrency

As described in Section 2.2, chunks become immutable when they reach their maximum capacity. This makes it easier to efficiently encode them without having to consider future modifications. Updates to these chunks are thus implemented in an insert-only fashion as invalidations and reinsertions. While a table's last chunk is mutable and would theoretically support in-place updates, we keep the architecture simple by following the insert-only approach for that chunk as well. For each chunk, we store three vectors: the (1) *begin* and (2) *end commit ids* of the transactions that inserted or invalidated this row, as well as the *transaction id* of a transaction that currently has this row in its set of modified rows.

When a transaction starts, it is assigned a unique (not necessarily contiguous) transaction id by the transaction manager. Also, it stores the commit id of the last transaction that was committed successfully. We previously called this "last commit id of a transaction" [42] or $lcid_T$, but have since renamed it to snapshot commit id, which we believe communicates its purpose better.

A transaction can identify rows that are visible by comparing the begin commit id with its snapshot id: If it is higher, the row has been inserted after the transaction has started and should not be visible. Similarly, a lower end commit id means that the row has already been deleted by the time the transaction started. Invalidations use the transaction id field for two purposes: First, when checking the visibility of a row, seeing its own transaction id in a row signals to the transaction that it has already modified (i.e., inserted or invalidated) the row. This keeps us from having to maintain a separate list of updated rows. Second, as modifying the transaction id is an atomic compare-and-swap operation, it identifies concurrency conflicts. If two transactions concurrently try to set the transaction id of a single row, only one can succeed and the other has to abort. For more detail on our implementation of MVCC, please refer to our description of transaction processing in Hyrise1 [42].

2.9 Scheduling

Most databases do not leave the task of scheduling their work items to the operating system's scheduler [18]. We, too, have instead implemented a cooperative task-based scheduler that tries to keep the OS scheduling out of the equation. Our unit of work is a *task*, which can be an operator, a subroutine within an operator, a maintenance job, or any other subroutine. The easiest type of task has been modeled after `std::thread` to take a function object or a lambda. This keeps the entry threshold for new developers low. All tasks are stored in a task queue, which is using a lock-free `tbb::concurrent_queue`. Tasks can have dependencies on other tasks and only tasks with fulfilled dependencies get emplaced in the queue by the scheduler. Once a task finishes, it iterates over its list of successors and asks them to check if they are now ready to be scheduled.

Hyrise spawns one active worker thread per CPU core. It polls the queue for new tasks to be executed. Once it has begun executing a task, it continues to do so uninterruptedly until it finishes. A task can also spawn subtasks, which are then emplaced in the scheduling queue and executed in parallel.

On NUMA systems, we use one queue per node. Tasks can specify a preferred node, for example when they should be scheduled close to the data that they are processing. When the queue on one node runs dry, workers on that node perform work stealing and attempt to help other nodes with finishing their queue. To prevent high contention on the queues, workers back off for a fixed time interval (currently 10 milliseconds) if the work stealing was unsuccessful.

In line with our goal to keep the system modular, the scheduler can be entirely disabled so that tasks are executed in the main thread. When `schedule` is called on a task, it is either directly executed or, if it has predecessors, their predecessors are executed first. As a result, when measuring the multi-threaded scalability of our system, there are differences between the measurements for one core with and without scheduler. This allows us to inspect the cost of the scheduler.

2.10 Benchmark Runner

Running standard benchmarks on different databases is often a tedious task. Benchmark tables have to be generated and, in many cases, the generated CSV files need some modifications before they can be loaded. For many research databases, some features of SQL are not supported and queries need to be reformulated accordingly. Finally, many databases have special configuration parameters that users need to be aware of. In our case, this includes the default chunk size and encoding options. Our goal is to provide both developers and new users with a binary that is a one-stop solution for executing such benchmarks.

Currently, Hyrise supports the TPC-H benchmark with the binary `hyriseBenchmarkTPCH`. Support for the alternative data generator JCC-H, as well as the benchmarks TPC-C and Join Order Benchmark are in development. These binaries return a JSON file with the total number of executed queries per second, as well as the individual run times of each query and additional information about the setup. Configuration parameters like default chunk size, encoding, or the number of used threads can optionally be set either via the command line or in a JSON file. Our idea is that by providing a git commit id and optionally a JSON file, results can easily be communicated in a reproducible way. Finally, users can provide their own table and queries in .csv and .sql files, which are then automatically executed. We use this to experiment with enterprise-specific workloads and real-world data that, unfortunately, cannot be published. A tool with a similar goal is OLTP-Bench [14], which we plan to support in the future.

3 PLUGINS

For Hyrise to be a multi-purpose research vehicle, it is important not to clutter the code base with limited-purpose extensions. Often, research concepts and their implementations tackle very distinctive challenges that are not necessary for normal database operation. We believe that these specialized implementations should not necessarily become part of the database core to avoid complicating the process of understanding Hyrise and its source code. Not merging them into the main code base also avoids behavior that is unexpected by other researchers, for example when self-tuning components automatically create new indexes. We see plugins as a solution that allows us to extend the system's functionality beyond that of typical database systems. In this section, we present the plugin interface offered by Hyrise and our plans for its use by a self-driving database.

3.1 Interface

Plugins are dynamic libraries, which can be loaded and unloaded by the user during database runtime. They can access all of Hyrise's components using the respective public interface. This means that, except for the boilerplate code needed to initialize them, the development of plugins is almost indistinguishable from that of the database core. Also, most code can be moved from the core into plugins without modification. The main limitation is that while plugins can call public methods, they cannot modify the code of existing components. For example, no new encoding type can be added via a plugin, as all encoding types have to be known during the compilation of Hyrise.

Plugins are implemented as Singletons to ensure that there is only a single instance in the system. The *Plugin Manager* is responsible for administrative work, such as loading and unloading of plugins via `libdl`. We provide a blueprint for plugins[4] that can be used as a starting point for developing new plugins.

3.2 Self-Driving Database

One prime use case for plugins in Hyrise is the area of self-driving database systems. In contrast to traditional databases, these systems do not rely on human database administrators (DBAs) anymore, but adjust their configuration and tune their physical database design autonomously. Such behavior can be achieved by employing workload-driven optimization and machine learning models. To achieve this efficiently, self-driving components need to access and manipulate database-internal data structures and processes. At the same time, it should still be possible to run Hyrise independently of such a plugin and its dependencies and it should not be necessary for other developers to consider such external components. In addition, components of such a self-driving system should be easily exchangeable to enable experiments with different strategies.

The basic architecture and conceptual ideas for a *Generalized Self-Driving Framework* [26] are currently under development in Hyrise. A couple of typical database configuration and physical design aspects should be adjusted autonomously, e.g., the selection of indexes, data placement in NUMA and in replicated systems, and an automatic selection of efficient encoding and compression schemes per chunk. These are either already part of Hyrise and are going to be transferred to the plugin-based self-driving system or are under development.

[4]Exemplary plugin for Hyrise: https://github.com/Bensk1/example-plugin

4 RELATED WORK

The field of database systems for analytical applications has seen vast progress over the last decade [1]. Amongst the first academic systems were MonetDB [9] and C-Store [46], which are disk- and OLAP-optimized systems. While the first column stores focused solely on analytical workloads, the rise of main memory-optimized databases also led to the use of column stores for mixed workload processing (OLxP or HTAP). Amongst these HTAP-optimized column stores are SAP HANA [15] and Hyper [23]. Other comparable database systems include academic systems like Quickstep [38] and Peloton [39]. In the remainder of this section, we discuss major design decisions of Hyrise and how other systems approached them.

The storage layouts of early column-oriented systems like MonetDB and C-Store closely resemble the *decomposition storage model* (DSM [12]), storing attributes in one large consecutive allocation. In addition to the base attributes, both systems further include additional sort orders (cf. C-Store's projections [46] and MonetDB's cracking [22]). A similar layout is used in SAP HANA with the difference that it tries to minimize the memory footprint and avoids redundant data storage. More recent systems use a different approach that splits columns into several smaller units. With the rising importance of NUMA systems, such a form of an automatic horizontal partitioning in fixed-size blocks eases the equal distribution of data over several nodes [10]. This storage paradigm is used, e.g., in Hyrise, HyPer, and Quickstep. Peloton's *tiles* are a hybrid storage layout and resemble the variable-width containers of the first version of Hyrise.

One of the main challenges for columnar databases is handling a steady stream of modifications in HTAP environments. C-Store introduced the separation into read- and write-optimized stores. A similar concept is used in SAP HANA and has been used in Hyrise1, namely separating each table in a read-optimized main partition and a write-optimized delta partition [27, 29]. In contrast, HyPer and Quickstep have chosen a model that is closer to the chunk concept in Hyrise. While Quickstep is read-optimized, HyPer uses uncompressed small blocks at the beginning and shifts to larger and more compressed blocks over time [17].

Storing aggregated and space-efficient data structures for early pruning of data is done by many database systems. The simplest form is to store the min/max values of each column (e.g., Netezza's zone maps or SAP HANA's synopses [36]) or small materialized aggregates [33]. HyPer uses so-called positional small materialized aggregates, which include scan ranges for multiple value ranges [28]. More sophisticated approaches comparable to Hyrise's filters are adaptive range filters [3] or SuRF [47].

In the area of modern in-memory databases, Hyper has been amongst the first systems that generated data-centric code using

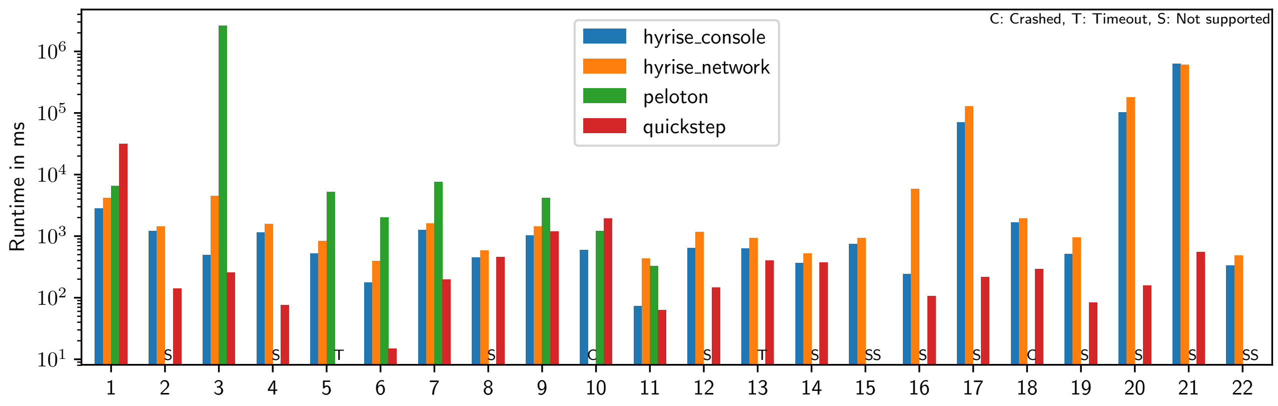


Figure 6: Runtime when executing the TPC-H queries on fresh database instances of Peloton, Quickstep, and Hyrise (SF=1)

LLVM IR [35]. Since then, other database systems, both research and productive, have started to compile queries at runtime, most generating either LLVM IR or C code that is then compiled. Hyrise differs from these approaches in that it does not generate new code but specializes existing code that is stored in LLVM IR and fuses it across operator boundaries. This approach has previously been used by other projects, such as DexterDB [21] or Sharygin et al.'s modifications to PostgreSQL [45]. Hyrise supports both compiled and vectorized queries, allowing for similar evaluations as done by Kersten et al. [24].

The idea of database systems that administrate themselves is almost as old as that of databases itself. Early work [20] dates back to the 1970s where certain aspects of the physical database design were tuned. In later years, vendors of commercial database systems integrated advisors into their products to support human DBAs [11]. Recently, self-driving databases [39] received fresh attention. New systems have a more holistic approach where systems do not rely on human interaction anymore and aim for administrating all aspects of databases instead of certain parts only. With its plugin architecture, Hyrise enables the integration and interplay of self-driving components with reusable components and clearly specified interfaces.

5 EVALUATION

Only recently has the SQL and expression subsystem reached the point where the syntax of all TPC-H queries was supported. Since then, we have constantly been working on improving the optimizer so that it generates better query plans. Although Hyrise still misses some LQP optimizations, which currently limit the performance of a few SQL queries, we can show that Hyrise is already in the same ballpark as other research databases.

5.1 Single Query Comparison

In this section, we compare Hyrise's performance with two other open-source research databases, namely Quickstep [38] and Peloton [39]. All databases were built from source in their release mode[5] using gcc 8.2. Our benchmark system has four Intel Xeon E7-4880 v2 CPUs, a total of 60 cores with 2.5 GHz (up to 3.1 in Turbo Boost Mode), and 2 TB of DRAM. We start the databases using either their network interface and `psql` (Peloton and Hyrise) or by sending queries to the command-line interface using `expect` (Quickstep and Hyrise).

[5]Git-Hashes for Hyrise: 9a60098b, Peloton: 3bc6d461, Quickstep: 5cbaa7ef

The TPC-H CREATE statements have been slightly modified to reflect the level of data type support in the databases. DECIMAL has been replaced by FLOAT and DATE has been replaced by CHAR(10). While Quickstep seems to offer a date type, using it in comparisons gives us an error. Within the queries, slight modifications have been made to compensate for the lack of date functions. No indexes were created. This evaluation aims at comparing the database performance that researchers can expect when looking at a system for the first time. As such, no additional settings were made. Most importantly, the databases are running with their default number of threads: 1 for Hyrise[6], 120 for Quickstep, and 4 for Peloton. We expect that there are probably better settings for these databases or more advanced code branches that would lead to better results, but limit the evaluation to what is publicly available and what would be a reasonable attempt at generating first numbers.

Results can be found in Figure 6. Considering the mentioned limitations, we can see that for most queries, Hyrise's performance is within an order of magnitude of the other databases. It also shows that there is still optimization potential both within the optimizer and the query execution, but also especially within our network component.

5.2 Chunk Size Evaluation

We discussed Hyrise's storage layout, including the chunk concept, in Section 2.2. In this subsection, we evaluate the performance impact of chunks. The test setup (hardware, compiler, scale factor) remains unchanged from the previous section. Figure 7 depicts the throughput for selected TPC-H queries for chunk capacities varying from one thousand to ten million records per chunk. The throughput is shown relative to a setup without any form of chunking. In the figure, TPC-H queries for which the performance is only marginally impacted by the chunk capacity are combined into "Avg. of other queries". This experiment demonstrates that the chosen chunk capacity influences the throughput to a large extent. Compared to a chunk capacity of 10 000 000 rows, which effectively results in a single chunk for the given scale factor, a chunk capacity of 100 000 improves the throughput by a factor of 26 for TPC-H query 21. At the same time, chunks containing only 1 000 records diminish the throughput by 97% for TPC-H query 22.

[6]In the default configuration, the scheduler is currently disabled as we are investigating a performance regression.

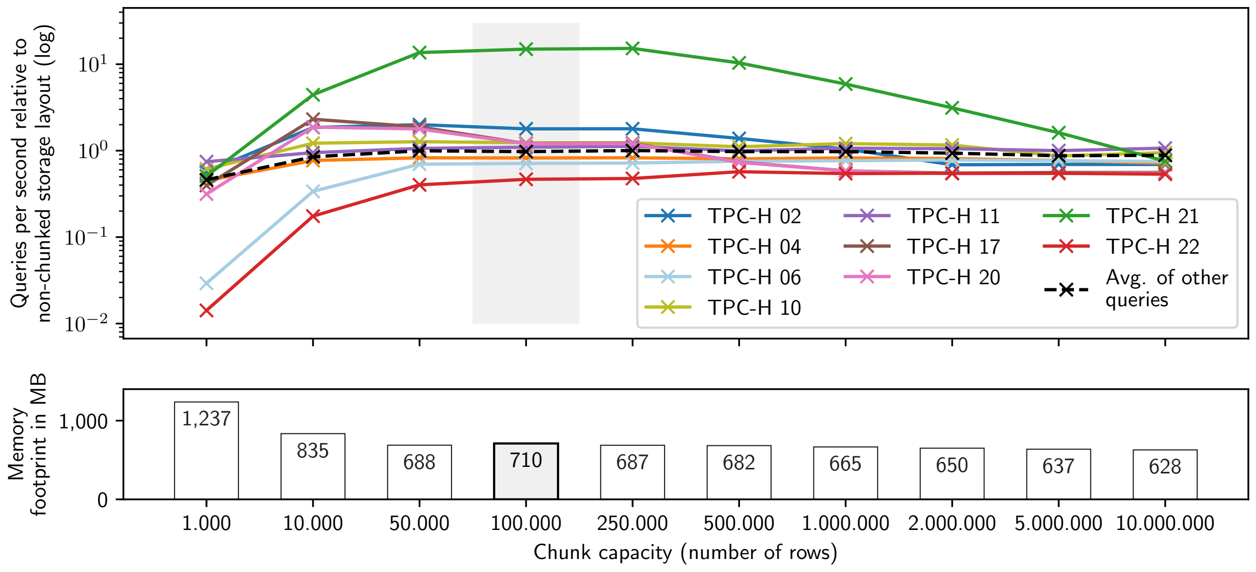


Figure 7: Performance impact and memory consumption of varying chunk capacities (the highlighted chunk capacity of 100 000 is Hyrise's default setting).

As stated earlier, chunks can influence the performance in different ways. First, chunks provide inherent partitions that can be used to distribute the workload. In the results above, however, multithreading was disabled and we see a second effect: Because chunks enable pruning, they can sometimes help in avoiding the access to large parts of the data. Whether pruning is possible depends on the underlying data. If pruning cannot be applied, queries might see drawbacks when too small chunks are used. Queries 6 and 22 with a chunk capacity of 1 000 are an example for this: pruning cannot be applied and many small chunks introduce a significant overhead. Overall, the best throughput is achieved with chunk sizes around 100 000. Compared to a non-chunked layout in Hyrise, the performance increases by 146%.

In addition, the chunk size affects a table's memory footprint as stated in Section 2.2. The lower half of Figure 7 shows the memory footprint for different chunk sizes of all TPC-H tables for a scale factor of 1 when applying dictionary encoding. The configuration that is best for throughput consumes roughly 10% more memory than the most space-efficient configuration. Fine-tuning this parameter on a per-table basis instead of setting it globally is subject to future work. Depending on the encoding scheme, a smaller chunk size can also reduce the footprint in edge cases, for example when a lower number of dictionary entries enables the use of fewer bits for their encoded representation.

6 SUMMARY

In this paper, we have presented how our research database Hyrise was re-engineered and rewritten to be a platform for future research projects. Our new architecture was built around the goals of being easy to understand and extend, enabling end-to-end benchmarking, and delivering high performance even in the presence of multiple abstraction layers. We have described which prior experiences with Hyrise1 have influenced the development, both from an architectural, and from a processual perspective.

Most components that are relevant to our query execution have been explained, including storage and encoding, auxiliary data structures, networking, the SQL pipeline, JIT compilation, multi-threading, and benchmarking. Furthermore, we have described our plugin interface and the self-driving plugin as an example for its use. Finally, we have evaluated chunks as the main storage concept of Hyrise, have shown how pruning can reduce the number of accessed rows during execution, and have compared Hyrise to other research databases.

Future work has been discussed for the separate components. From a high-level perspective, we will focus on implementing the missing LQP optimizations that are current bottlenecks to our performance, will implement more benchmarks, and improve the performance of the system by making better use of modern hardware components.

We invite the reader to experiment with Hyrise by following our first steps guide[7], which not only contains instructions on how to setup Hyrise and run first benchmarks within minutes, but also examples on how to run queries, and visualize them.

ACKNOWLEDGEMENTS

Hyrise is partly developed during Master's seminars and theses. The presented improvements would not have been possible without the contributions and ideas of over 60 students in the last two years. We would like to thank all of them for their dedication. The work of Moritz Eyssen has not only brought Hyrise its current degree of SQL support but, through his immensely helpful feedback on pull requests, also greatly contributed to the overall quality of the code base.

REFERENCES

[1] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2012. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases* 5, 3 (2012), 197–280.
[2] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *PVLDB* 5, 10 (2012), 1064–1075.

[7] Hyrise – First steps: https://github.com/hyrise/hyrise/wiki/FirstSteps

[3] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson. 2013. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. *PVLDB* 6, 14 (2013), 1714–1725.

[4] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proc. ICDE*. 362–373.

[5] Harold R Berenson, Peter A Carlin, Nigel R Ellis, Cesar A Galindo-Legaria, Goetz Graefe, Ajay Kalhan, Craig C Peeper, and Samuel H Smith. 2002. Auto-parameterization of database queries. US Patent 6,356,887.

[6] Martin Boissier. 2018. Reducing the Footprint of Main Memory HTAP Systems: Removing, Compressing, Tiering, and Ignoring Data. In *Proc. VLDB PhD Workshop*.

[7] Martin Boissier, Rainer Schlosser, and Matthias Uflacker. 2018. Hybrid Data Layouts for Tiered HTAP Databases with Pareto-Optimal Data Placements. In *Proc. ICDE*. 209–220.

[8] Peter A. Boncz, Angelos-Christos Anatiotis, and Steffen Kläbe. 2017. JCC-H: Adding Join Crossing Correlations with Skew to TPC-H. In *Proc. TPCTC at VLDB*. 103–119.

[9] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. CIDR*. 225–237.

[10] Craig Chasseur and Jignesh M. Patel. 2013. Design and Evaluation of Storage Organizations for Read-Optimized Main Memory Databases. *PVLDB* 6, 13 (2013), 1474–1485.

[11] Surajit Chaudhuri and Vivek R. Narasayya. 2007. Self-Tuning Database Systems: A Decade of Progress. In *Proc. VLDB*. 3–14.

[12] George P. Copeland and Setrag Khoshafian. 1985. A Decomposition Storage Model. In *Proc. SIGMOD*. 268–279.

[13] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. 2017. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *Proc. EDBT*. 72–83.

[14] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288.

[15] Franz Färber, Norman May, Wolfgang Lehner, et al. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.

[16] Martin Faust, David Schwalb, Jens Krüger, and Hasso Plattner. 2012. Fast Lookups for In-Memory Column Stores: Group-Key Indices, Lookup and Maintenance. In *Proc. ADMS at VLDB*. 13–22.

[17] Florian Funke, Alfons Kemper, and Thomas Neumann. 2012. Compacting Transactional Data in Hybrid OLTP & OLAP Databases. *PVLDB* 5, 11 (2012), 1424–1435.

[18] Jana Giceva, Gerd Zellweger, Gustavo Alonso, and Timothy Roscoe. 2016. Customized OS support for data-processing. In *DaMoN*. 2:1–2:6.

[19] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. 2010. HYRISE - A Main Memory Hybrid Storage Engine. *PVLDB* 4, 2 (2010), 105–116.

[20] Michael Hammer. 1977. Self-adaptive automatic data base design. In *Proc. AFIPS*. 123–129.

[21] Carl-Philip Hänsch, Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. 2015. Plan Operator Specialization using Reflective Compiler Techniques. In *Proc. BTW*. 363–382.

[22] Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. 2011. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB* 4, 9 (2011), 585–597.

[23] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proc. ICDE*. 195–206.

[24] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *PVLDB* 11, 13 (2018), 2209–2222.

[25] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *Proc. CIDR*.

[26] Jan Kossmann. 2018. Self-Driving: From General Purpose to Specialized DBMSs. In *Proc. VLDB PhD Workshop*.

[27] Jens Krüger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. 2011. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *PVLDB* 5, 1 (2011), 61–72.

[28] Harald Lang, Tobias Mühlbauer, Florian Funke, et al. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *Proc. SIGMOD*. 311–326.

[29] Juchang Lee, Michael Muehle, Norman May, Franz Faerber, Vishal Sikka, Hasso Plattner, Jens Krüger, and Martin Grund. 2013. High-Performance Transaction Processing in SAP HANA. *IEEE Data Eng. Bull.* 36, 2 (2013), 28–33.

[30] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215.

[31] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013*. 38–49.

[32] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling. In *Proc. CIDR*.

[33] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB*. 476–487.

[34] Guido Moerkotte and Thomas Neumann. 2006. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In *Proc. VLDB*. 930–941.

[35] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011), 539–550.

[36] Anisoara Nica, Reza Sherkat, Mihnea Andrei, Xun Chen, Martin Heidel, Christian Bensberg, and Heiko Gerwens. 2017. Statisticum: Data Statistics Management in SAP HANA. *PVLDB* 10, 12 (2017), 1658–1669.

[37] Prashant Pandey, Michael A. Bender, Rob Johnson, and Robert Patro. 2017. A General-Purpose Counting Filter: Making Every Bit Count. In *Proc. SIGMOD*. 775–787.

[38] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: A Data Platform Based on the Scaling-Up Approach. *PVLDB* 11, 6 (2018), 663–676.

[39] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *Proc. CIDR*.

[40] PostgreSQL 10 Documentation. 2018. Frontend/Backend Protocol. https://www.postgresql.org/docs/10/static/protocol.html

[41] Mark Raasveldt and Hannes Mühleisen. 2017. Don't Hold My Data Hostage - A Case For Client Protocol Redesign. *PVLDB* 10, 10 (2017), 1022–1033.

[42] David Schwalb, Martin Faust, Johannes Wust, Martin Grund, and Hasso Plattner. 2014. Efficient Transaction Processing for Hyrise in Mixed Workload Environments. In *IMDM at VLDB*. 112–125.

[43] David Schwalb, Jan Kossmann, Martin Faust, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. 2015. Hyrise-R: Scale-out and Hot-Standby through Lazy Master Replication for Enterprise Applications. In *Proc. IMDM at VLDB*. 7:1–7:7.

[44] David Schwalb, Girish Kumar, Markus Dreseler, Anusha S., Martin Faust, Adolf Hohl, Tim Berning, Gaurav Makkar, Hasso Plattner, and Parag Deshmukh. 2016. Hyrise-NV: Instant Recovery for In-Memory Databases Using Non-Volatile Memory. In *Proc. DASFAA, Part II*. 267–282.

[45] E. Yu. Sharygin, Ruben Buchatskiy, Roman Zhuykov, and A. R. Sher. 2017. Query compilation in PostgreSQL by specialization of the DBMS source code. *Programming and Computer Software* 43, 6 (2017), 353–365.

[46] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-Store: A Column-oriented DBMS. In *Proc. VLDB*. 553–564.

[47] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proc. SIGMOD*. 323–336.