SQL- and Operator-centric Data Analytics in Relational Main-Memory Databases

Linnea Passing
passing@in.tum.de

Manuel Then
then@in.tum.de

Nina Hubig
hubig@in.tum.de

Harald Lang
langh@in.tum.de

Michael Schreier
schreier@in.tum.de

Stephan Günnemann
guennemann@in.tum.de

Alfons Kemper
kemper@in.tum.de

Thomas Neumann
neumann@in.tum.de

Technical University of Munich

ABSTRACT

Data volume and complexity continue to increase, as does the need for insight into data. Today, data management and data analytics are most often conducted in separate systems: database systems and dedicated analytics systems. This separation leads to time- and resource-consuming data transfer, stale data, and complex IT architectures.

In this paper we show that relational main-memory database systems are capable of executing analytical algorithms in a fully transactional environment while still exceeding performance of state-of-the-art analytical systems rendering the division of data management and data analytics unnecessary. We classify and assess multiple ways of integrating data analytics in database systems. Based on this assessment, we extend SQL with a non-appending iteration construct that provides an important building block for analytical algorithms while retaining the high expressiveness of the original language. Furthermore, we propose the integration of analytics operators directly into the database core, where algorithms can be highly tuned for modern hardware. These operators can be parameterized with our novel user-defined *lambda* expressions. As we integrate lambda expressions into SQL instead of proposing a new proprietary query language, we ensure usability for diverse groups of users. Additionally, we carry out an extensive experimental evaluation of our approaches in HyPer, our full-fledged SQL main-memory database system, and show their superior performance in comparison to dedicated solutions.

CCS Concepts

•**Information systems** → **Online analytical processing engines;** *Main memory engines; Data mining;*

Keywords

HyPer, Computational databases

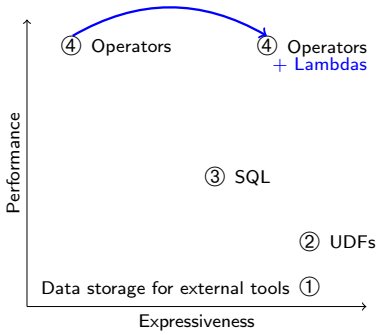


Figure 1: Overview of approaches to data analytics using RDBMS. Our system supports the novel *layer 4*, where data mining is integrated directly into the database core leading to higher performance. To maintain expressiveness, high-order functions (*lambdas*) are passed as parameters.

1. INTRODUCTION

The current data explosion in science and technology poses difficulties for data management and data analytics. Especially stand-alone data analytics applications [2, 16] are prone to have problems due to their simple data management layer. Being optimized for read-mostly or read-only analytics tasks, most stand-alone systems are unsuitable for frequently changing datasets. After each change, the whole data of interest needs to be copied to the application again, a time- and resource-consuming process.

We define *data analytics* to be algorithms and queries that process the whole dataset (or extensive subsets), and therefore are computation-intensive and long-running. This domain contains, for example, machine learning, data mining, graph analytics, and text mining. In addition to the differences between these subdomains, most algorithms boil down to a model-application approach: i.e., a two phase process where a model is created and stored first and then applied to the same or different data in a second step.

In contrast to dedicated analytical systems, classical DBMS provide an efficient and update-friendly data management layer and many more useful features to store big data reliably, such as user rights management and recovery procedures. Database systems avoid data silos as data has to be stored only once, eliminating ETL cycles (extraction, trans-

formation, and loading of data). Thus, we investigate how data analytics can be sensible integrated into RDBMS to contribute to a "one-solution-fits-it-all" system. What level of efficiency is possible when running such complex queries in a database? Can a database actually be better than single purpose standalone systems? According to Aggarwal et al. [4], seamless integration of data analytics technology into DBMS is a major challenge.

Some newer database systems, for example SAP HANA [15] and HyPer [20], are designed to efficiently handle different workloads (OLTP and OLAP) in a single system. Main-memory RDBMS, such as HyPer, are specifically well-suited for high analytics workload due to their efficient use of modern hardware, i.e., multi-core CPUs with extensive instruction sets and large amounts of main memory.

How is an analytics algorithm best integrated into an RDBMS? While existing database systems that feature data analytics include the algorithms on a very high level, we propose to add a specific set of algorithmic building blocks as deep in the system as possible. To describe and assess different approaches of integrating data analytics algorithms into an RDBMS, we distinguish four layers ranging form the least to the most deeply integrated:

(1) DBMS as data storage with external analytics algorithms—the currently most commonly used approach.

(2) User-defined functions (UDFs)—code snippets in high-level languages executed by the DBMS.

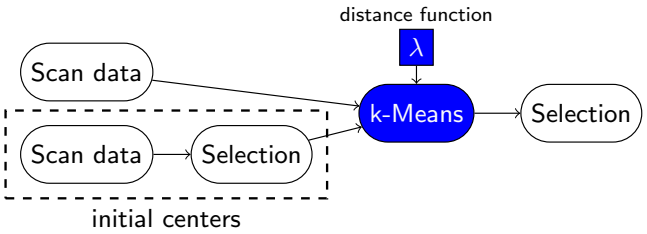
(3) SQL queries—including recursive common table expressions (CTE) and our novel iteration construct.

(4) Integration as physical operators—the deepest integration, providing the highest performance.

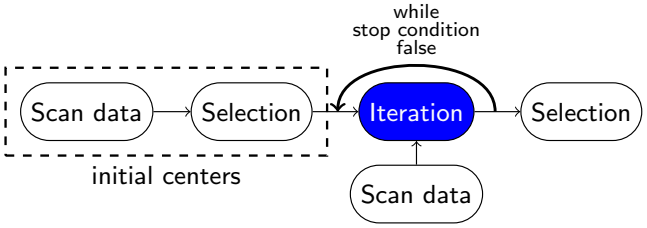
We propose user-defined code snippets as parameters to our operators to increase flexibility within *(4)*. These so-called lambda functions, containing for instance distance metrics, are able to change the semantics of a given analytical algorithm.

These four approaches trade performance versus flexibility in a different way, as depicted in Figure 1. We propose implementing several of these approaches into one system to cover the diverse needs regarding performance and expressiveness of different user groups and application domains. The novel operator integration *(4)* combines the highest performance with high flexibility but can only be implemented by the database system's architects. Approaches *(2)* and *(3)* provide environments in which expert users can implement their own algorithms. All three integrated approaches [*(2)*, *(3)*, *(4)*] avoid ETL costs, stale data, and assembling and administrating complex system environments, thereby facilitating ad-hoc data analytics.

This paper focuses on two approaches, SQL- and operator-centric approaches to data analytics in databases. Figure 2 gives a first idea of how these approaches are integrated into query plans. As depicted, both approaches handle arbitrarily pre-processed input. Both approaches result in a relation; this result can thus be post-processed within the same query. The operator-centric approach features a specialized operator that processes data as would any other relational operator such as a join. In the SQL-centric approach, analytical algorithms are expressed in SQL. k-Means is an iterative algorithm, hence Figure 2 shows an iteration as the most important part of the query.



(a) Operator-centric approach. The iterative k-Means algorithm is implemented as physical relational operator. The distance function is specified as a lambda expression.



(b) SQL-centric approach. The iterative algorithm, including initialization and stop condition, is expressed in SQL. The iteration operator can either be the standard recursive common table expression, or our optimized non-appending iteration construct.

Figure 2: Query plans for k-Means clustering.

1.1 Contributions

In this paper, we present how data analytics can efficiently be integrated into relational database systems. Our approach targets different user groups and application domains by providing multiple interfaces for defining and using analytical algorithms. High expressiveness and performance are achieved via a unique combination of existing and new concepts including[1]:

- A classification and assessment of approaches to integrate data analytics with databases.
- The iteration construct as extension to recursive common table expressions (`with recursive`) in SQL.
- Analytical operators executed within the database engine that can be parameterized using lambda expressions (anonymous user-defined functions) in SQL.
- An experimental evaluation with both dedicated analytical systems and database extensions for analytical tasks.

The remainder of this paper is organized as follows: An overview of the related work is provided in Section 2. In Section 3 we discuss what characteristics make HyPer especially suited for in-database analytics. We continue by explaining the in-database processing in Section 4. The method by which analytics are integrated into SQL is defined in Section 5 and our building blocks (operators) of the fourth layer are shown in detail in Section 6. Our operators are very flexible due to their lambda functions, which we describe in Section 7. The evaluation of our operators in HyPer is given in Section 8. We discuss the conclusions of our work in Section 9.

[1]Partially based on our prior publication [29].

2. RELATED WORK

Data analytics software can be categorized into dedicated tools and extensions to DBMS. In this section, we introduce major representatives of both classes.

2.1 Dedicated Data Analytics Tools

The programming languages and environments *R*[2], *SciPy*[3], *theano*[4] and *MATLAB*[5] are known by many data scientists and are readily available. For these reasons, they are heavily used for data analytics, and implementations of new algorithms are often integrated. In addition, these languages and environments provide data visualizations and are well-suited for exploration and interactive analytics. However, their algorithm implementations often are only single-threaded, which is a major drawback concerning currently used multi-core systems and data volumes.

The next group of existing data analytics tools are *data analytics frameworks*. Most representatives of this category are targeted at teaching and research and do not focus on performance for large datasets. Their architecture makes it easy to implement new algorithms and to compare different variants of algorithms regarding quality of results. Notable examples of data analytics frameworks include *ELKI*[6], which supports diverse index structures to speed up analytics, *RapidMiner*[7], used in industry as well as research and teaching, and *KNIME*[8], which allows users to define data flows and reports via a GUI.

Recently, Crotty et al. presented *Tupleware*, a high-performance analytical framework. Tupleware is meant for purely analytical tasks and the system does not take into account transactions.The authors endorse interactive data exploration by not relying on extensive data preparation [12] and by providing a data exploration GUI. Tupleware requires users to annotate their queries with as much semantics as possible: Queries may solely consist of simple building blocks, e.g., `loop` or `filter`, augmented with user-defined code snippets such as comparison functions. Relational operators—the building blocks of SQL queries—are fairly similar to Tupleware's building blocks but are more coarse-grained, more robust against faulty or malicious user input, and can be used in a more general fashion. They therefore do not guarantee as many invariants. SQL implementations of algorithms could be optimized in a similar fashion, although this requires major changes to relational query optimizers.

The cluster computing framework *Apache Spark* [31] supports a variety of data analytics algorithms. Analytical algorithms, contained in the Machine Learning Library (MLlib), benefit from Spark's scale-up and scale-out capabilities. *Oracle PGX*[9] is a graph analytics framework. It can run predefined as well as custom algorithms written in the *Green-Marl* DSL and is focused on a fast, parallel, and in-memory execution. *GraphLab* [22] is a machine learning framework that provides many machine learning building blocks such as regression or clustering, which facilitate building complex applications on top of them. All of these frameworks use dedicated internal data formats making it necessary to use time-consuming data loading steps. Furthermore, the synchronization of results back to the RDBMS is a complex job that often must be implemented explicitly by the user.

[2]http://www.r-project.org/
[3]http://www.scipy.org/
[4]http://deeplearning.net/software/theano/
[5]http://www.mathworks.com/products/matlab/
[6]http://elki.dbs.ifi.lmu.de/
[7]http://rapidminer.com/
[8]http://www.knime.org/
[9]http://www.oracle.com/technetwork/oracle-labs/parallel-graph-analytics/

2.2 Data Analytics in Databases

In addition to standalone systems there are database systems which contain data analytics extensions. Being faced with the issue of integrating data analytics and relational concepts, the systems mentioned below come up with different solutions: Either analytical algorithms are executed via calls to library functions, or the SQL language is extended with data analytics functions.

MADlib [17] is an example for the second level of our classification, user-defined functions. This library works on top of selected databases and heavily uses data parallel query execution if provided by the underlying database system. MADlib provides analytical algorithms as user-defined functions written in C++ and Python that are called from SQL queries. The underlying database executes those functions but cannot inspect or optimize them. While the output produced by the functions can directly be post-processed using SQL, only base relations are allowed as input to data analytics algorithms. Thus, full integration of the user-defined functions and SQL queries is neither achieved on a query optimization and execution level nor in the language and query layer.

Another example for the UDF category is the *SAP HANA Predictive Analytics Library (PAL)* [25, 15]. This library offers multi-threaded C++ implementations of analytical algorithms to run within the main-memory database system SAP HANA. It is integrated with the relational model in a sense that input parameters, input data, as well as intermediate results and the output are relational tables. The algorithms, so-called *application functions*, are called from SQL code. They are compiled into query plans and executed individually. In contrast to the afore-mentioned MADlib, PAL integrates in one ecosystem only and is therefore capable of connecting to SAP HANA's user and rights management.

Oracle Data Miner [27] is focused on *supervised* machine learning algorithms. Hence, training data is used to create a model that is then applied to test data using SQL functions. Both steps are run multi-threaded to make use of modern multi-core systems. Results of the algorithms are stored in relational tables. Interactive re-using and further processing of results within the same SQL query is not possible, but can be applied in precedent and subsequent queries.

EmptyHeaded [1] uses a datalog-like query language for graph processing. This system follows the "one-solution-fits-all" approach: Graph data is processed in a relational engine using multiway join algorithms that are more suitable for graph patterns than classical pairwise join algorithms.

LogicBlox [6] is also a relational engine that does not use SQL for queries. It relies on functional programming, as does Tupleware, but is a full relational DBMS. The functional programming language of LogicBlox, LogiQL, can be combined with declarative programming and features a relational query optimizer. LogicBlox exploits constraint solving to optimize the functional code.

SimSQL [11] is another recent relational database with analytical features. Users write algorithms from scratch

which are then translated into SQL. Several SQL extensions, such as for-each style loops over relations as well as vector and matrix data types, facilitate analytics in the database. While recognizing that its tuple-oriented approach to matrix-based problems results in low performance [10], SimSQL emphasizes its general-purpose approach. As a result of those design decisions, SimSQL is able to execute complex machine learning algorithms which many other computation platforms are not able to do [10], but lacks optimizations for standard analytical algorithms.

To conclude, while all presented database systems strive for integration of analytical and relational queries, the achieved level of integration vastly differs between systems. Most presented systems rely on black box execution of user-defined functions by the database while others transform analytical queries into relational queries to allow for query inspection and optimization by the database.

3. HYPER FOR DATA ANALYTICS

HyPer [20] is a *hybrid* main-memory RDBMS that is optimized for both transactional and analytical workloads. It offers best-in-class performance for both, even when operating simultaneously on the same data. Adding capabilities to execute data analysis algorithms is the next step towards a unified data management platform without stale data.

Several features of HyPer contribute to its suitability for data analytics. First, HyPer *generates efficient data-centric code* from user queries thus reducing the user's responsibility to write algorithms in an efficient way [24]. After transforming the query into an abstract syntax tree (AST), multiple optimization steps, and the final translation into a tree of physical operators, HyPer generates code using the LLVM compiler framework. Computation-intensive algorithms benefit from this design because function calls are omitted. As a result, users without knowledge in efficient algorithms can write fast analytical queries.

Second, *data locality* further improves performance. Data-centric execution attempts keeping data tuples in CPU registers as long as possible to avoid copying of data. If possible, a tuple is kept in registers while multiple operators are executed on it. This so-called *pipelining* is important for queries that touch tuples multiple times. For ad-hoc analytical queries pre- and post-processing steps can be combined with the data processing to generate highly efficient machine code. As many analytical algorithms are pipeline breakers, in practice we pipeline pre-processing and data materialization as well as result generation and post-processing.

Third, HyPer focuses on *scale-up* on multi-core systems rather than on *scale-out* on clusters; hence, parallelization of the operators and the generated code is a performance-critical aspect. Characteristics of modern hardware, such as non-uniform memory access (NUMA), cache hierarchies, and vector processing must be taken into account when new features are integrated into the DBMS. Avoiding data distribution onto multiple nodes is especially important when the input data cannot be chunked easily, e.g., when processing graph-structured data.

In addition to efficient integration of algorithms, other characteristics further encourage the use of HyPer for data analysis use-cases such as: the system provides a PostgreSQL-compatible SQL interface, is fully ACID-compliant and offers fast data loading [23], which is especially important for data scientists.

4. IN-DATABASE PROCESSING

Existing systems for data analysis often use their own proprietary query languages and APIs to specify algorithms (e.g., Apache Spark [31] and Apache Flink [5]). This approach has several drawbacks. For example, unusual query languages make it necessary to extensively train the domain experts that write queries. If common high-level programming languages like Java are used, many programmers are available, but they usually lack domain knowledge. Additionally, optimizing high-level code is a hard problem that compiler designers have been working on for decades, especially in combination with additional query execution logic.

Our goal is to enable data scientists to create and execute queries in a *straightforward* way, while keeping all *flexibility* for expert users. In this chapter, we assess multiple approaches to integrate data analytics into HyPer. The first layer shown in Figure 1 using the database system solely as data storage is omitted here as it does not belong to the in-database processing category. Layers two and three, UDFs and SQL queries, respectively, are already implemented in various database systems. Layer four describes our novel approach of deeply integrating complex algorithms into the database core to maximize query performance while retaining flexibility for the user.

4.1 Program Execution within the Database

Many RDBMS allow user-defined functions (UDFs) in which database users can add arbitrary functionality to the database. This eliminates the need to copy data to external systems. The code snippets are registered with the database system and are usually run by the database system as a black box, although first attempts to "open the black box" have been made [18]. If UDF code contains SQL queries, executing these queries potentially requires costly communication with the database. This is because for most UDF languages it is not possible to bind together the black box code and the code that executes the embedded SQL query thus foregoing massive optimization potential. Because of the dangers to stability and security that go along with executing foreign code in the database core, a sandbox is required to separate database code and user code.

4.2 Extensions to SQL

There is general consensus that relational data should be queried using SQL. By extending SQL to integrate new algorithms, the vast amount of SQL infrastructure (JDBC connectors and SQL editors) can be reused to work with analytical queries. Furthermore, the declarative nature of SQL makes it easy to continuously introduce new optimizations. By using this common language, one avoids the high effort of creating a new language and of teaching it to users.

Some algorithms, such as the a-priori algorithm [8] for frequent itemset mining, work well in SQL but others are difficult to express in SQL and even harder to optimize. One common difficulty is the *iterative nature* of many analytical algorithms. To express iterations in SQL, recursive common table expressions (CTE) can be used. CTEs compute a monotonically growing relation, i.e., tuples of *all* previous iterations are kept. As many iterative algorithms need to access *one* previous iteration only, memory is wasted if the optimizer does not optimize this hard-to-detect situation. This is a problem especially for main-memory databases where memory is a scarce resource.

To solve this issue, we suggest an iteration concept for SQL that does not *append* to the prior iteration but instead *replaces* it and therefore drastically reduces the memory footprint of iterative computations. As the intermediate results become smaller, less data has to be read and processed, thus, non-appending iterations also improve analytics performance. We explain the details of our iteration concept in Section 5.

4.3 Data Analytics in the Database Core

In contrast to other database systems, HyPer integrates important data analytics functionality directly into the core of the database system by implementing special highly-tuned *operators* for analytical algorithms. Because the internal structures of database systems are fairly different, such operators have to be specifically designed and implemented for each system. Differentiating factors between systems are, among others, the execution model (tuple-at-a-time vs. vectorized execution) as well as the storage model (row store vs. column store). For example, an operator in the analytical engine Tupleware, which does not support updating datasets, would look significantly different from an operator in the full-fledged database system HyPer, which needs to take care of updates and query isolation.

HyPer can arbitrarily mix relational and analytical operators leading to a seamless integration between analytics with other SQL statements into one query plan. This is especially useful because the functionality of existing relational operators can be reused for common subtasks in analytical algorithms, such as grouping or sorting. Analytical operators can focus on optimizing the algorithm's core logic such as providing efficient internal data representations, performing algorithm-dependent pre-processing steps, and speeding up computation-intensive loops. A further advantage of custom-built analytical operators is that the query optimizer knows their exact properties and can choose an optimal query plan based on this information. Having all pre- and post-processing steps in one language—and one query—greatly simplifies data analytics and allows efficient ad-hoc queries. In Section 6 we elaborate on our implementation of (physical) operators.

Of the integration layers presented in this section, special operators are integrated most deeply into the database. As a result, they provide unrivaled performance but reduce the user's flexibility. To regain flexibility, we propose lambda expressions as a way of injecting user-defined code into operators. Lambdas can, for example, be used to specify distance metrics in the k-Means algorithm or to define edge weights in PageRank.

By implementing multiple layers, we can offer data analytics functionality to diverse user groups. User-defined algorithms are attractive for data scientists wanting to implement specific algorithms in their favorite programming language without having to copy the data to another system. Persons knowledgeable in analytical algorithms and SQL might prefer to stick to their standard data querying language making extensions to SQL their best choice. Algorithm operators implemented by the database developers are targeted towards users that are familiar with the data domain but not with data analytics algorithm design.

Syntactically, UDFs, stored SQL queries and special operators cannot and should not be distinguished by the user. In this way, DBMS architects can decide on an algorithm's level of integration, which is transparent to the user.

In the following sections, we delve into the details of data analytics using SQL and using specialized analytical operators with λ-expressions. We omit the details on the first two layers—using the database solely as data storage, and running UDFs in a black box within the database—because the first layer does not incorporate any analytical algorithms on the database system side and the second layer uses the database system as a runtime environment for user-defined code. When the database is only used to provide the data, the performance is bound by data transfer performance and the data analytics software used to run the algorithms. In case the database is used to execute code in a black box, again, the runtime depends on the programming language and implementation used in these UDFs.

5. DATA ANALYTICS USING SQL

Our overall goal is to seamlessly integrate analytical algorithms and SQL queries. In the third layer, which is described in this section, SQL is used and extended to achieve this goal. Standard SQL provides most functionality necessary for implementing analytical algorithms, such as fix point recursion, aggregation, sorting, or distinction of cases. However, one vital construct is missing: a more general concept of iteration has to be added to the language. Section 5.1 introduces this general-purpose iteration construct, called *iterate operator*. Query optimization for analytical queries is discussed in Section 5.2.

Our running example is the three algorithms k-Means, Naive Bayes, and PageRank which are well-known [30, 3] examples from vector and graph analytics and used as example building blocks in other state-of-the-art analytics systems [12]. Their properties are shared by many other data mining and graph analytics algorithms. Furthermore, they represent the areas of data mining, machine learning, and graph analytics. Thus, these three algorithms are appropriate examples for this paper.

5.1 The Iterate Operator

The SQL:1999 standard contains recursive common table expressions (CTE) that are constructed using the `with recursive`. Recursive CTEs allow for computation of growing relations, e.g., transitive closures. In these queries, the CTE can be accessed from within its definition and is iteratively computed until no new tuples are created in an iteration. In other words, until a fixpoint is reached. Although it is possible to use this fixpoint recursion concept for general-purpose iterations, this is clearly a diversion from its intended use case, and can thus result in incorrect optimizer decisions.

Our iterate operator has similar capabilities as recursive CTEs: it can reference a relation within its definition allowing for iterative computations. In contrast to recursive CTEs, the iterate operator *replaces* the old intermediate relation rather than *appending* new tuples. Its final result is a table with the tuples that were computed in the last iteration only. This pattern is often used in data and graph mining algorithms, especially when some kind of metric or quality of data tuples is computed. In PageRank, for example, the initial ranks are updated in each iteration. In clustering algorithms, the assignment of data tuples to clusters has to be updated in every iteration. These algorithms have in common that they operate on fixed-size datasets, where

```
SELECT * FROM ITERATE([initialization], [step], [stop
      condition]);

-- find smallest three-digit multiple of seven
SELECT * FROM ITERATE((SELECT 7 "x"),
     (SELECT x+7 FROM iterate),
     (SELECT x FROM iterate WHERE x>=100));
```

Listing 1: Syntax of the ITERATE SQL language extension. A temporary table *iterate* is created, that in the beginning contains the result of the *initialization* subquery. Iteratively, the subquery *step* is applied to the result of the last iteration, until the boolean condition *stop condition* is fulfilled.

only certain values (ranks, assigned clusters, et cetera) are updated. This means the stop criterion has to be changed; rather than stopping when no new tuples are generated, our iterate operator stops when a user-defined predicate evaluates to true. We show the syntax of the iteration construct in Listing 1. By providing a non-appending iteration concept with a while-loop-style stop criterion, we are adding more semantics to the implementation, which has been shown to massively speed up query execution due to better optimizer decisions [12].

Although it is possible to implement the afore-mentioned algorithms using recursive CTEs, the iterate operator has two major advantages:

- Lower memory consumption: Given a dataset with n tuples, and i iterations. With recursive CTEs, the table is growing to $n * i$ tuples. Using our operator, the size of the relation remains n. For comparisons of the current and the last iteration, we need to store $2*n$ tuples and discard all prior iterations early. The iterate construct saves vast amounts of memory in comparison to recursive CTEs. Furthermore, if the stop criterion is the number of executed iterations, recursive CTEs have to carry along an iteration counter, which is a huge memory overhead because it has to be stored in every tuple.

- Lower query response times: Because of the smaller relation size, our algorithm is faster in scanning and processing the whole relation, which is necessary to re-compute the ranks, clusters, et cetera.

Lower memory requirements are particularly important in main-memory databases like HyPer, where memory is a scarce resource. This is especially true when whole algorithms are integrated into the database because they often need additional temporary data structures. Our evaluation, Section 8, shows how algorithm performance can be improved by using our iterate operator instead of recursive CTEs, while keeping the flexibility of `with recursive` statements. Both approaches share one drawback, they can both produce infinite loops. Those situations need to be detected and aborted by the database system, e.g., via counting recursion depth or iterations, respectively.

A conceptually similar idea that also features appending and non-appending iterations can be found in the work of Binnig et al. [7]. Being a language proposal for a functional extension to SQL, their paper neither discusses where which type of iteration is appropriate, nor does it list advantages and drawbacks regarding performance or memory consumption. Ewen et al. [14] also argue that many algorithms only need small parts of the data to compute the next iteration (so-called *incremental iterations*). Their work focuses on parallelizing those iterations as they are only sparsely dependent on each other. The *SciDB* engine features support for iteration processing on arrays where "update operations typically operate on neighborhoods of cells" [26]. Soroush et al.'s work enables efficient processing of this type of array iterations as well as incremental iterations.

5.2 Query Optimization and Seamless Integration with the Surrounding SQL Query

Keeping intermediate results small by performing selections as early as possible is a basic principle of query optimization. This technique, called pushing selections, is in general not possible when analytical algorithms are affected. This is because the result of an analytical algorithm is not determined by *single tuples* (as it is for example for joins), but potentially influenced by the *whole* input dataset. A similar behavior can be found in the group-by operator, where the aggregated results also depend on *all* input tuples. This naturally narrows the search space of the query optimizer and reduces optimization potentials.

One major influencing factor for query optimization is the cardinality of intermediate results. For instance, precise cardinality estimations are necessary for choosing the best join ordering in a query. It is, however, hard to estimate the output cardinality of the generic iterate operator because it can contain diverse algorithms. Some algorithms, e.g., k-Means, iterate over a given dataset and the number of tuples stays the same before and after the iterate operator. Other algorithms, e.g., reachability computations, increase the dataset with each iteration, which makes the final cardinality difficult to estimate. Cardinality estimation on recursive CTEs faces the same difficulty so that similar estimation techniques can be applied.

To conclude, the diverse nature of analytical algorithms does not offer many generic optimization opportunities. Instead, relational query optimization has to be performed almost independently on the subqueries *below* and *above* the analytical algorithm while the analytical algorithm itself might benefit from different optimization techniques, e.g., borrowed from general compiler design or constrained solving as suggested by [6]. Because of the lacking potential for standard query optimization, low-level optimizations such as vectorization and data locality, as introduced in Section 3, become more important.

6. OPERATORS

The most in-depth integration of analytical algorithms into a DBMS is by providing implementations in the form of physical operators. Physical operators like hash join or merge sort are highly optimized code fragments that are plugged together to compute the result of a query. All physical operators, including the analytical ones introduced in this paper, use tables as input and output. They can be freely combined ensuring maximal efficiency. Figure 3 shows how physical analytics operators are integrated into query translation and execution. Physical operators are performance-wise superior to the general iteration construct, introduced in Section 5.1, as these specialized operators know

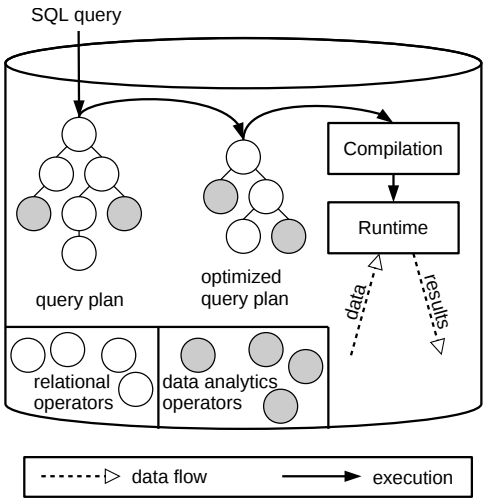


Figure 3: Query translation and execution with relational and analytical operators. A SQL query is translated to an abstract syntax tree (AST) consisting of both relational and analytical operators. The optimizer can inspect both types of operators. This approach provides highest integration and performance.

```
SELECT * FROM PAGERANK((SELECT src, dest FROM edges),
      0.85, 0.0001);
```

Listing 2: Operator integration in SQL. Arbitrary preprocessing of input data and arbitrary postprocessing of the results is possible. Additional parameters define the algorithm's behavior.

invariants of their algorithms such as the estimated output cardinality or data dependencies in complex computations. These specialized operators know best how to distribute work among threads or how to optimize the memory layout of internal data structures.

For example, the query shown in Listing 2 computes the PageRank value for every vertex of the graph induced by the *edges* relation[10]. The query is processed by a table scan operator followed by our specialized physical PageRank operator. The PageRank operator implementation defines, for example, whether parallel input (from the table scan operator) can be processed, information that is used by the optimizer to create the best plan for the given query.

In the next sections, we describe the chosen algorithms, k-Means, Naive Bayes, and PageRank, and how we implemented them in HyPer. Furthermore, we describe necessary changes to the optimizer.

6.1 The Physical k-Means Operator

k-Means is a fast, model-based iterative clustering algorithm, i.e., it divides a set of tuples into k spherical groups such that the sum of distances is minimized. It can be utilized as a building block for advanced clustering techniques. The classical k-Means algorithm by Lloyd [21] splits each iteration into two steps: assignment and update. In the assignment step, each tuple is assigned to the nearest cluster center. In the update step, the cluster centers are set to be the arithmetic mean of all tuples assigned to the cluster. The algorithm converges when no tuple changes its assigned cluster during an iteration. For practical use, the convergence criterion is often softened: Either, a maximum number of iterations is given, or the algorithm is interrupted if only a small fraction of tuples changed its assigned cluster in an iteration.

In our implementation, the k-Means operator requires two input relations, data and initial centers, that are passed via subqueries. An additional parameter defines the maximum number of iterations. Using parallelism, our implementation benefits from modern multi-core systems. Each thread locally assigns data tuples to their nearest center and to prepare the re-computation of cluster centers, each thread sums up the tuples values. The data tuples themselves are consumed and directly thrown away after processing. For the next iteration, tuples are requested again from the underlying subquery. As a result, the query optimizer can decide to compute the data relation each time, or use materialized intermediate results, whatever is faster in the given query. Data locality is ensured because all centers and intermediate data structures are copied for each thread. Thread synchronization is only needed for the very last steps, global aggregation of the local intermediate results and the final update of the cluster centers. This procedure is repeated until the solution remains stable (i.e., no tuple changed its assignment during an iteration), or until the maximum number of iterations is reached. The operator then outputs the cluster centers.

6.2 The Physical Naive Bayes Operator

Naive Bayes classification aims at classifying entities, i.e., assigning categorical labels to them. Other than k-Means or PageRank, it is a supervised algorithm and consists of two steps performed on two different datasets: First, a dataset A with known labels is used to build a model based on the Bayesian probability density function. Second, the model is applied to a related but un-labeled (thus unknown) dataset B to predict its labels. When implemented in a relational database, one challenge is storing the model as it does not match any of the relational entities, relation or index, completely.

We implemented model creation and application as two separate operators, Naive Bayes training and Naive Bayes testing, respectively. The generation of additional statistical measures is handled by two additional operators that are not limited to Naive Bayes but can be used as a building block for multiple algorithms, for example k-Means.

Similar to k-Means, the Naive Bayes training operator is a pipeline breaker. Each thread holds a hash table to manage its input data with the class as key while not storing the tuples itself. In addition, the number of tuples N is stored for each class, as well as the sum of the attribute values $\sum_{n \in N} n.a$ and the sum of the square of each attribute value $\sum_{n \in N} n.a^2$ for each class and attribute. After the whole input is consumed, the training operator computes the a-priori probability for each class as well as the mean and standard deviation for each class and attribute:

[10]Parentheses around the subquery are necessary because arbitrary queries are allowed there. The sole use of commas would have lead to an ambiguous grammar.

Let a given training set D with $|D|$ instances $d \in D$ contain a set of classes C with $|C|$ instances $c \in C$. Let $|c|$ denote the number of instances of this class c in D. Then, the a-priori probability of class c is given by:

$$PR(c) = \frac{|c| + 1}{|D| + |C|}$$

Afterwards, the results and the class labels are fed into the next operator: the testing operator.

6.3 The Physical PageRank Operator

PageRank [9] is a well-known iterative ranking algorithm for graph-structured data. Each vertex v in the graph (e.g., a website or a person), is assigned a ranking value that can be interpreted as its importance. The rank of v depends on the number and rank of incoming edges, i.e., v is important if many important vertices have edges to it. A PageRank iteration is a sparse matrix-vector multiplication. In each iteration, part of each vertex's importance flows off to the vertices it is adjacent to, and in turn each vertex receives importance from its neighbors. Similar to k-Means, PageRank converges towards a fixpoint, i.e., the vertex ranks change less than a user-defined epsilon. It is common to specify a maximum number of iterations.

The sparse matrix-vector multiplication performed in the PageRank iterations is similar to many graph algorithms in that its performance greatly benefits from efficient neighbor traversals. This means for a given vertex v it has to be efficiently possible to enumerate all of its neighbors. Our PageRank implementation ensures this by efficiently creating a temporary compressed sparse row (CSR) representation [28] that is optimized for the query at hand. We avoid storage overhead and an access indirection in this mapping by re-labeling all vertices and doing a direct mapping. After the PageRank computation we use a reverse mapping operator that translates our internal vertex ids back to the original ids.

The PageRank operator uses only the CSR graph index and no longer needs to access the base data. In each iteration we compute the vertices' new PageRank values in parallel without any synchronization. Because we have dense internal vertex ids we are able to store the current and last iteration's rank in arrays that can be directly indexed. Thus, every neighbor rank access only involves a single read. At the end of each iteration we aggregate each worker's data to determine how much the new ranks differ from the previous iterations. If the difference is less or equal to the user-defined epsilon or if the maximum iteration count is reached, the PageRank computation finishes.

7. LAMBDA EXPRESSIONS

In Section 4.3 we described the integration of specialized data analytics operators into the database core. These operators provide unrivaled performance in executing the algorithms they were designed for. However, without modification they are not flexible, i.e., they are not even applicable in the context of similar but slightly different algorithms. Consider the k-Medians algorithm. It is a variant of k-Means that uses the L1-norm (Manhattan distance) rather than the L2-norm (Euclidean distance) as distance metric. While this distance metric differs between the variants, their implementations have in common predominant parts of their code. Even though this common code could be shared, different metrics would make necessary different variants of our algorithm operators.

```
CREATE TABLE data (x FLOAT, y INTEGER, z FLOAT,
                   desc VARCHAR(500));
CREATE TABLE center (x FLOAT, y INTEGER, z FLOAT);
INSERT INTO data ...
INSERT INTO center ...

SELECT * FROM KMEANS(
  -- sub-queries project the attributes of interest
  (SELECT x,y FROM data),
  (SELECT x,y FROM center),
  -- the distance function is specified as λ-expression
  λ(a,b) (a.x-b.x)^2+(a.y-b.y)^2,
  -- termination criterion:  max.  number of iterations
  3
);
```

Listing 3: Customization of the k-Means operator using a lambda expression for the distance function.

Instead, when designing data analytics operators, we identify and aim to exploit such similarities. Our goal is to have one operator for a whole class of algorithms with variation points that can be specified by the user. To inject user-defined code into variation points of analytics operators we propose using *lambda expressions* in SQL queries.

Lambda expressions are anonymous SQL functions that can be specified inside the query. For syntactic convenience, the lambda expressions' input and output data types are automatically inferred by the database system. Also, for all variation points we provide default lambdas that are used should none be specified. Thus, non-expert users can easily fall back to basic algorithms. With lambda-enabled operators we strive not only to keep implementation and maintenance costs low, but especially to offer a wide variety of algorithm variants required by data scientists. Also, because lambda functions are specified in SQL, they benefit from existing relational optimizations.

Listing 3 shows how our k-Means operator benefits from lambdas. In the `kmeans` function call's third argument, a lambda expression is used to specify an arbitrary distance metric. The operator expects a lambda function that takes two tuple variables as input arguments and returns a (scalar) float value. At runtime, these variables are bound with the corresponding input tuples to compute the distance. Thus, by providing a k-Means operator that accepts lambda expressions we do not only cover the common k-Means and k-Medians algorithms but also allow users to design algorithms that are specific to their task and data at hand. These custom algorithms are still executed by our highly-tuned in-database operator implementation and because all code is compiled together, no virtual function calls are involved.

8. EXPERIMENTAL EVALUATION

In this section, we evaluate our implementations of k-Means, PageRank, and Naive Bayes. As introduced in Section 4, we implemented multiple versions of the algorithms, that reflect different depths of integration. We compare our solutions to other systems commonly used by data scientists. This includes middle-ware tools based on RDBMS, analytics software for distributed systems, and standalone data analysis tools.

	#tuples n	#dimensions d	k
Varying number of tuples	160 000	10	5
	800 000	10	5
	4 000 000	10	5⋆
	20 000 000	10	5
	100 000 000	10	5
	500 000 000	10	5
Varying number of dimensions	4 000 000	3	5
	4 000 000	5	5
	4 000 000	10	5⋆
	4 000 000	25	5
	4 000 000	50	5
Varying number of clusters	4 000 000	10	3
	4 000 000	10	5⋆
	4 000 000	10	10
	4 000 000	10	25
	4 000 000	10	50

⋆ same experiments, for connecting the three lines of experiments

Table 1: Datasets for k-Means experiments.

8.1 Datasets and Parameters

We use a variety of datasets to evaluate the influence of certain characteristics of the datasets and workload to the resulting performance.

8.1.1 k-Means Datasets and Parameters

k-Means is an algorithm targeted at *vector data*, i.e., tuples with a number of dimensions. This data model fits perfectly into relations. The data is characterized by the number of tuples n, the number of dimensions d used for clustering, and the data types of the dimensions. We chose to perform experiments for varied n and d while keeping the data types constant. In addition to the dataset, the algorithm itself has multiple parameters: the number of clusters k, the cluster initialization strategy, and the number of iterations i that are computed. The number of clusters k drastically influences the query performance because it defines the number of distances to be computed and compared, and is an important parameter in our evaluation. To produce comparable results with a wide range of systems, our experiments use the simplest cluster initialization strategy: random selection of k initial cluster centers. We chose to perform three iterations i, which keeps the experiment duration short while leveling out a possible overhead in the first iteration.

While modifying one parameter, we keep the other two fixed to focus on the effect on that parameter only. The resulting list of experiments is shown in Table 1. We conduct five to six experiments per parameters, which allows us to assess not only the performance but also the scaling behavior of the different systems. The dataset sizes, determined by n and d, were chosen to be processable by all evaluated systems within main memory and within a reasonable time given the vast performance differences between the systems. We create artificial, uniformly distributed, datasets because they provide an important advantage over real-world datasets in our use case. As the performance of plain k-Means with a fixed number of iterations is irrespective of data skew, our decision to use synthetic datasets does not introduce any drawbacks.

8.1.2 Naive Bayes Datasets and Parameters

The Naive Bayes experiments are conducted using the same synthetic datasets as k-Means. We vary the number of tuples N and the number of dimensions d. For the labels we chose a uniform probability density function of two labels 0 and 1. Our experiments cover the *training* phase of the algorithm only as it has a much higher complexity and thus runtime than the *testing* step.

8.1.3 PageRank Datasets and Parameters

PageRank is an algorithm targeted at *graph data*, i.e., vertices and edges with optional properties. The algorithm is parameterized with the damping factor d modeling the probability that an edge is traversed, e, the maximum change between two iterations for which the computation continues, and the maximum number of iterations i. For the damping factor d we chose the reasonable value 0.85 [9], i.e., the modeled random surfer continues browsing with a probability of 85%. To better compare different systems, we set e to 0 and run a fixed number of 45 iterations in all systems. As datasets we use the artificial LDBC graph designed to follow the properties of real-world social networks. We generated multiple LDBC graphs in different sizes up to 500,000 vertices and 46 million edges, using the SNB data generator [13], and used the resulting undirected person-knows-person graph.

8.2 Evaluated Systems

We evaluate our physical operators, denoted as *HyPer Operator*, SQL queries with our iterate operator, denoted as *HyPer Iterate*, and a pure SQL implementation using recursive CTEs, denoted as *HyPer SQL*, against diverse data analysis systems introduced in Section 2. We chose *MATLAB R2015* as a representative of the "programming languages" group. The next category is "big data analytics" platforms, in which we evaluate *Apache Spark 1.5.0* with MLlib. As contender in the "database extensions" category, we chose *MADlib 1.8* on top of the Pivotal Greenplum Database 4.3.7.1.

To ensure a fair comparison, all systems have to implement the same variant of k-Means: Lloyd's algorithm. Note that we therefore disabled the following optimizations implemented in Apache Spark MLlib. First, the MLlib implementation computes lower bounds for distances using norms reducing the number of distance computations. Second, distance computation uses previously computed norms instead of computing the Euclidean distance (if the error introduced by this method is not too big). *litekmeans*[11], a fast k-Means implementation for MATLAB, uses the same optimizations. We therefore use MATLAB's built-in k-Means implementation in our experiments.

8.3 Evaluation Machine

All experiments are carried out on a 4-socket Intel Xeon E7-4870 v2 (15×2.3 GHz per socket) server with 1 TB main memory, running Ubuntu Linux 15.10 using kernel version 4.2. Greenplum, the database used for MADlib, is only available for Red Hat-based operating systems. We therefore set

[11] http://www.cad.zju.edu.cn/home/dengcai/Data/Clustering.html

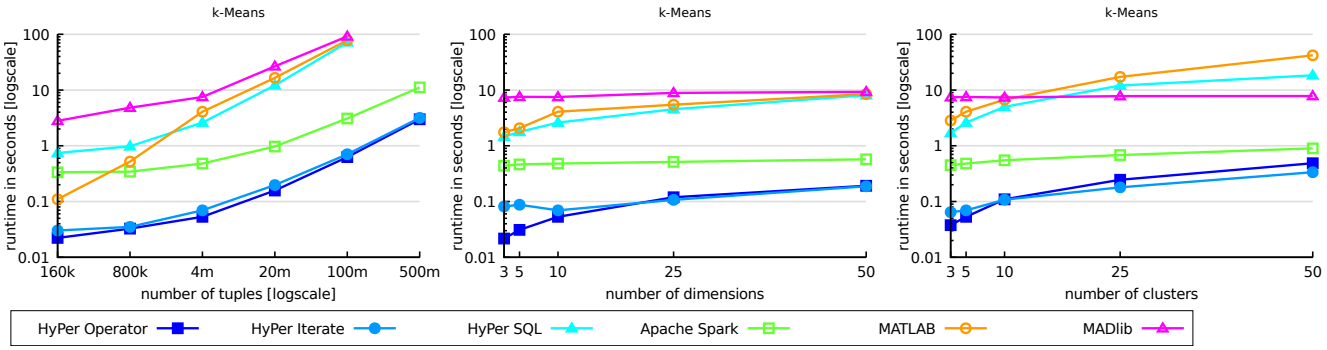


Figure 4: k-Means experiments. From left to right: varying the number of tuples N, dimensions d, and clusters k. Default parameters: 4,000,000 tuples, 10 dimensions, 5 clusters, 3 iterations.

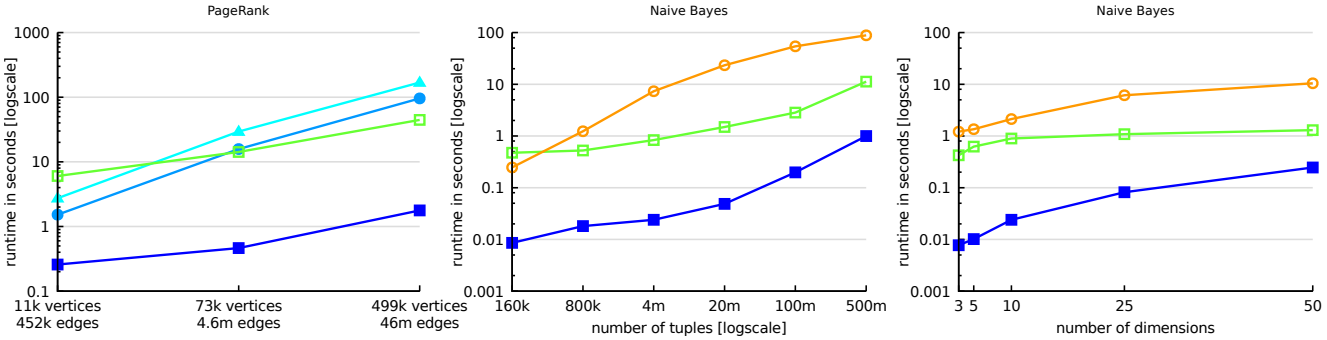


Figure 5: PageRank and Naive Bayes experiments. From left to right: PageRank using the LDBC SNB dataset, damping factor 0.85, and 45 iterations. Naive Bayes experiment varying the number of tuples N. Naive Bayes experiment varying the number of dimensions d.

up a Docker container running CentOS 7. The potential introduced overhead is considered in our discussion. As mentioned, we chose the datasets to fit into main memory, even when considering additional data structures. MATLAB does not contain parallel versions of the chosen algorithms, as mentioned in Section 2. This issue is also considered in the discussion of our results.

8.4 Results and Discussion

Figures 4 and 5 display the total measured runtimes. In general, the results match our claims regarding the four layers of integration as shown in Figure 1: Systems using UDFs (layer 2), in our experiments represented by MADlib, are slower than HyPer Iterate and HyPer SQL using SQL (layer 3). The fastest implementation, HyPer Operator, uses analytical operators (layer 4). Runtime of dedicated analytical systems, such as MATLAB and Apache Spark, heavily depends on the individual system.

8.4.1 *Recursive CTEs and HyPer Iterate*

As claimed in Section 5, using the iteration concept improves runtimes over plain SQL. While the pure SQL implementation, using recursive CTEs, has to store and process intermediate results that grow with each iteration, the iteration operator's intermediate results have constant size. In our implementations this means additional selection predicates for the pure SQL variant and more expensive aggregates due to the larger intermediate results. k-Means is more affected by this difference because it operates on larger data and is less computation-intensive than PageRank.

8.4.2 *Hyper Operator and HyPer Iterate*

The k-Means experiments show almost no difference between the HyPer Operator and the HyPer Iterate approach. k-Means is a rather simple algorithm: there is no random data access, only few branches, vectorization can be applied easily, and the data structures are straightforward. Furthermore, k-Means operates on vector data; both operator and SQL implementations use similar internal data structures. This results in very similar code being generated by the operator and the query optimizer resulting in the similar runtimes.

For PageRank, the experiments reveal a different picture: HyPer Operator runs significantly faster than HyPer Iterate because of its optimized CSR graph data structure. In contrast, HyPer Iterate has to work on relational structures, an edges table and a derived vertices table, and subsequently needs to perform many (hash) joins. As a result, its runtime is dominated by building and probing hash tables. This behavior is also found in [19] where a SQL implementation of PageRank also showed performance only comparable to stand-alone-systems. The following rule of thumb can be applied: The more similar optimized SQL code and code generated from the hand-written operator are, the smaller the runtime difference between HyPer Iterate and HyPer Operator approaches.

8.4.3 HyPer, MATLAB, MADlib, and Apache Spark

Among the contender systems, Apache Spark shows by far the best runtimes, which was expected because Spark was especially built for these kinds of algorithms. Still, Apache Spark is multiple times slower than our HyPer Operator approach for all three evaluated algorithms, as shown in Figures 4 and 5. HyPer's one-system-fits-all approach comes with some overhead of database-specific features not present in dedicated analytical systems like Apache Spark. Therefore, it is important that these features do not cause overhead when they are not used. For instance, isolation of parallel transactions should not take a significant amount of time when only one analytical query is running. Some database-specific overhead, stemming for example from memory management and user rights management, cannot be avoided. Nevertheless, HyPer shows far better runtimes than dedicated systems, while also avoiding data copying and stale data. MATLAB runs both algorithms single-threaded and therefore cannot compete, but was included because multiple heavily used data analytics tools do not support parallelism. MADlib, even taking into account the runtime impairment caused by the virtualization overhead, cannot compete with solutions that integrate data analytics deeper and produce better execution code.

Interestingly, Spark and MADlib almost seem not to be affected by the number of dimensions or clusters in the experiments. As algorithm-wise more complex computations are necessary if either of the numbers increases, we suspect those computations to be hidden behind multi-threading overhead. For example, if each thread handles one cluster, even the 50 clusters in the largest experiment still fit into the 120 hyper-threads of the evaluation machine. But k-Means with larger number of dimensions or clusters is not common, because their results are impaired by the curse of dimensionality or cannot be interpreted by humans. Regarding the scaling for larger datasets, log-scaled runtimes fail to show runtime differences appropriately. Plots with log-scaled runtimes counter-intuitively show converging lines when in fact the runtime difference between two systems is constant, which is the case for HyPer Operator/Iterate and Apache Spark in the leftmost sub-figure of Figure 4.

The results presented above support our claim that a multi-layer approach helps targeting diverse user groups. DBMS manufacturers benefit from the identical interface and syntax of UDFs, stored SQL queries, and hard-coded operators. The decision as to in which layer an algorithm should be implemented is solely affected by the implementation effort versus the gain in performance and flexibility. Laypersons can use these manufacturer-provided algorithms without having to care whether it is a UDF, an SQL query, or a physical operator. Database users with expertise, opposed to laypersons wanting to implement their own analytical algorithms can choose to implement either UDFs or SQL queries.

Briefly stated, the experiments match the expected order of runtimes: the deeper the integration of data analytics, the faster the system. Our results also support our idea of one database system being sufficient for multiple workloads. While this has been shown for combining OLTP and OLAP workloads [15, 20], our contribution was to integrate one more workload, data analytics, while keeping performance and usability on a high level.

9. CONCLUSION

We described multiple approaches of integrating data analytics into our main-memory RDBMS HyPer. Like most database systems, HyPer can be used as a data store for external tools. However, doing so exposes data transfer as a bottleneck and prevents significant query optimizations. Instead, we presented three layers of integrating data analytics directly into the database system: data analytics *in UDFs*, data analytics *in SQL*, and analytical *operators in the database core*. The layers' depth of integration and their analytics performance increases with each layer.

UDFs allow the user to implement arbitrary computations directly in the database. However, because the database runs UDFs as a black box, automated optimization potentials are limited. To prevent this lack of optimization potential, we proposed performing data analytics in SQL. As iterations are hard to express in SQL and difficult to optimize, we presented the *iteration operator* and a corresponding language extension that serves as a building block for arbitrary iterative algorithms directly in SQL. Compared to recursive common table expressions, the iteration construct significantly reduces runtime overhead, especially in terms of memory consumption, as it only materializes the intermediate results of the *previous* iteration.

For major analytical algorithms that are used frequently (e.g., k-Means, PageRank, and Naive Bayes), we proposed an even deeper integration: integrating highly-tuned analytical operators into the database core. Using our novel SQL *lambda expressions*, users can specialize analytical operators directly within their SQL queries. This adds flexibility to otherwise fixed operators and allows, for example, for applying arbitrary user-defined distance metrics in our tuned k-Means operator. Just like the iterate operator and the analytics operator, lambda expressions are part of the logical query plan and are subject to query optimization and code generation. Hence, they benefit from decades of research in database systems.

Our presented approaches enable complete integration of data analytics in SQL queries, ensuring both efficient query plans and usability. In our experiments we saw that HyPer data analytics on both graph and vector data is significantly faster than in dedicated state-of-the-art data analytics systems: 92 times faster than Apache Spark for PageRank. This is especially significant because as an ACID-compliant database, HyPer must also be able to handle concurrent transactional workloads. Thus, we showed that HyPer is suitable for integrated data management *and* data analytics on large data, with multiple interfaces targeted at different user groups.

Acknowledgment

This research was supported by the German Research Foundation (DFG), Emmy Noether grant GU 1409/2-1, and by the Technical University of Munich - Institute for Advanced Study, funded by the German Excellence Initiative and the European Union Seventh Framework Programme under grant agreement number 291763, co-funded by the European Union. Further, it was supported by the German Research Foundation (DFG), grant NE 1677/1-1. Manuel Then is a recipient of the Oracle External Research Fellowship. Linnea Passing has been sponsored in part by the German Federal Ministry of Education and Research (BMBF), grant TUM: 01IS12057.

10. REFERENCES

[1] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. EmptyHeaded: A Relational Engine for Graph Processing. In *Proc. SIGMOD 2016*, pages 431–446, 2016.

[2] E. Achtert, H. Kriegel, and A. Zimek. ELKI: A Software System for Evaluation of Subspace Clustering Algorithms. In *Proc. SSDBM 2008*, volume 5069 of *LNCS*, pages 580–585, 2008.

[3] C. C. Aggarwal and H. Wang. Graph Data Management and Mining: A Survey of Algorithms and Applications. In *Managing and Mining Graph Data*, Advances in Database Systems, pages 13–68. 2010.

[4] N. Aggarwal, A. Kumar, H. Khatter, and V. Aggarwal. Analysis the effect of data mining techniques on database. *Advances in Engineering Software*, 47(1):164–169, 2012.

[5] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The Stratosphere platform for big data analytics. *VLDBJ*, pages 939–964, 2014.

[6] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the logicblox system. In *Proc. SIGMOD 2015*, pages 1371–1382. ACM, 2015.

[7] C. Binnig, R. Rehrmann, F. Faerber, and R. Riewe. FunSQL: It is time to make SQL functional. In *Proc. DanaC 2012*, pages 41–46. ACM, 2012.

[8] F. Bodon. A fast apriori implementation. In *Proc. IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI 2003)*, 2003.

[9] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *Proc. WWW 1998*, pages 3825–3833, 1998.

[10] Z. Cai, Z. J. Gao, S. Luo, L. L. Perez, Z. Vagena, and C. Jermaine. A Comparison of Platforms for Implementing and Running Very Large Scale Machine Learning Algorithms. In *Proc. SIGMOD 2014*, pages 1371–1382, 2014.

[11] Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine. Simulation of database-valued Markov chains using SimSQL. In *Proc. SIGMOD 2013*, pages 637–648. ACM, 2013.

[12] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska. The Case for Interactive Data Exploration Accelerators (IDEAs). In *Proc. HILDA 2016*, pages 11:1–11:6, 2016.

[13] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The LDBC Social Network Benchmark: Interactive Workload. In *Proc. SIGMOD 2015*, pages 619–630, 2015.

[14] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning Fast Iterative Data Flows. *Proc. VLDB Endow.*, 5(11):1268–1279, 2012.

[15] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[16] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.*, 11(1):10–18, 2009.

[17] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib Analytics Library: Or MAD Skills, the SQL. *Proc. VLDB Endow.*, 5(12):1700–1711, 2012.

[18] F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the Black Boxes in Data Flow Optimization. *Proc. VLDB Endow.*, 5(11):1256–1267, 2012.

[19] A. Jindal, S. Madden, M. Castellanos, and M. Hsu. Graph analytics using vertica relational database. In *Proc. Big Data 2015*, pages 1191–1200, 2015.

[20] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE 2011*, pages 195–206, 2011.

[21] S. Lloyd. Least squares quantization in PCM. *IEEE Trans. Inf. Th.*, 28(2):129–137, 1982.

[22] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Framework For Parallel Machine Learning. *CoRR*, 2014.

[23] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. Instant Loading for Main Memory Databases. *Proc. VLDB Endow.*, 6(14):1702–1713, 2013.

[24] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.*, 4(9):539–550, 2011.

[25] SAP SE. *SAP HANA Predictive Analysis Library (PAL) Reference, SAP HANA Platform SPS 09*, 2014.

[26] E. Soroush, M. Balazinska, S. Krughoff, and A. Connolly. Efficient Iterative Processing in the SciDB Parallel Array Engine. In *Proc. SSDBM 2015*, pages 39:1–39:6, 2015.

[27] P. Tamayo, C. Berger, M. Campos, J. Yarmus, B. Milenova, A. Mozes, M. Taft, M. Hornick, R. Krishnan, S. Thomas, M. Kelly, D. Mukhin, B. Haberstroh, S. Stephens, and J. Myczkowski. Oracle Data Mining. In O. Maimon and L. Rokach, editors, *Data Mining and Knowledge Discovery Handbook*, pages 1315–1329. Springer, 2005.

[28] M. Then, M. Kaufmann, A. Kemper, and T. Neumann. Evaluation of Parallel Graph Loading Techniques. In *GRADES 2016*, pages 4:1–4:6, 2016.

[29] M. Then, L. Passing, N. Hubig, S. Günnemann, A. Kemper, and T. Neumann. Effiziente Integration von Data-und Graph-Mining-Algorithmen in relationale Datenbanksysteme. In *Proc. LWA 2015 Workshops*, pages 45–49, 2015.

[30] X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg. Top 10 Algorithms in Data Mining. *Knowledge and Information Systems*, 14(1):1–37, 2008.

[31] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Proc. HotCloud 2010*, 2010.