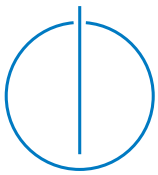
FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Efficient Data Mining Algorithms in Main Memory Databases

Martin Kapfhammer



FAKULTÄT FÜR INFORMATIK

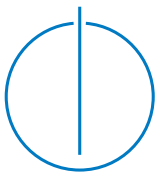
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Efficient Data Mining Algorithms in Main Memory Databases

Effiziente Data Mining Algorithmen in Hauptspeicher-Datenbanken

Author:	Martin Kapfhammer
Supervisor:	Prof. Alfons Kemper, Ph.D.
Advisor:	Linnea Passing, M.Sc.
Submission Date:	15.04.2015



I assure the single handed composition of this master's thesis in informatics only supported by declared resources.

Munich, 15.04.2015 Martin Kapfhammer

Acknowledgments

Abstract

Contents

1 Introduction

Databases are more in the center of innovation than ever. With the growing demands on storing and analyzing large amounts of data, linked with the capabilities of modern hardware, database vendors and researchers face new challenges and possibilities. One of these challenges is to execute data mining algorithms directly on the database. In this thesis, an approach is presented to integrate data mining techniques into the main memory database HyPer.

1.1 Motivation

Traditionally, databases are disk-based systems separated into two parts: One system used for Online Transactional Processing (OLTP), optimized for high rates of mission-critical, transactional write requests. As second system, a data warehouse is used for Online Analytical Processing (OLAP), executing long-running queries to gain insight into the collected data, used to make future decisions upon. Due to this contradicting requirements of critical, short write requests on the one side and long running, business-intelligence gaining read requests on the other side, traditionally two separated systems are used. The data synchronization between the two systems is ensured by an Extract-Transform-Load (ETL) process: Data is extracted from the OLTP system, transformed and loaded into the OLAP system. Since this process implicates heavy load on the database, it is usually done periodically, e.g. once a night. Obviously, this architecture reveals several drawbacks, like stale data, redundancy and the maintenance cost of two systems.
Modern hardware allows us to move away from this paradigm: Instead of using disk-based databases the data can be stored in memory. Since memory can be accessed much faster than the disk, an unprecedented throughput of OLTP transactions is possible. Using snapshots of the transactional data by exploiting the virtual memory management of the operating system, OLAP queries can run in parallel next to the OLTP transactions on up-to-date data. Such a system is HyPer [KN11], a relational main memory database guaranteeing the ACID properties, actively developed at the Chair of Database Systems at the TU München, and the main system used for our research.
The possibility of executing OLAP queries on a relational DBMS without interfering

the OLTP transaction throughput opens new possibilities for database systems: Additionally to long-running OLAP queries, data mining algorithms can be implemented and executed in HyPer. Data mining extends the possibilities of OLAP by applying more complex algorithms to discover interesting patterns and knowledge from the data. Crucial techniques are mining frequent patterns and association rules, classification, clustering and outlier detection.
Usually data mining tools have to import and integrate data from databases and other data sources by ETL processes, resulting in the same disadvantages as for data warehouses: data is not up-to-date for the crucial analyzing phase and redundant. By implementing data mining algorithms right into the database, data analysts, data scientists and business executives gain huge benefits. Instead of bringing the data to the algorithm, the algorithms are brought to the data. No ETL processes are necessary to export data to an additional system for data mining. Instead, the algorithms can be executed directly on the database system on up-to-date data. The execution of the algorithms can benefit from the modern hardware and high computational power of the database system. Additionaly, the data mining process can be combined with already existing database operators such as grouping and aggregation, resulting in a very natural environment for data analysis.

1.2 Research Questions

With more and more data generated by modern IT systems, the needs for analyzing large amounts of data are rapidly growing and data mining software gets an increasing amount of attention. So far, most data scientists use standalone data mining tools such as R [R D08], Julia [Bez+12] or the Python ecosystem SciPy [J+01]. Furthermore, the Java frameworks Weka [Hal+09] and ELKI [AKZ08] are used actively, in particular in the research community. All tools are providing an environment for statistical computing and the application of various data mining algorithms. Additionaly, for mining tera- and petabytes of data, scalable software such as Apache Hadoop [Shv+10], Apache Spark [Zah+10] and the data mining and machine learning framework Apache Mahout [Apa] are used.
While all of the presented environments are frequently used by data scientists, they demonstrate one decisive drawback: Before executing data mining algorithms, the data first has to be fetched from the database and transformed into a format readable by these tools. Therefore a first enhancement is to bring the algorithms closer to the database. Tools like MADlib [Hel+12] and RapidMiner [Rap14] are supporting specific database engines and trying to fill this gap. Algorithms can be run on the database, using extensible SQL, already existing SQL operators, stored procedures and

the possibility to connect those with high-level "glue" code. These tools provide a middleware between the database and data mining software and therefore data export becomes obsolete.
However, for complex algorithms SQL operators have to be combined with high-level constructs, often resulting in bad performance. Therefore, database vendors such as Oracle and SAP are going one step further and providing their databases with more and more statistical and data mining functionalities using the existing SQL syntax and graphical user interfaces [Ora15] [SAP14]. This leads to several advantages: A database system provides already efficient data storage and access, therefore data mining algorithms implemented on the database can benefit from these data structures. Besides, databases are optimized for modern hardware, e.g. multi core processors and cache hierarchies, which makes them presumably faster than platform-independent tools. Furthermore, data mining algorithms can profit from database features such as parallelization, scalability, recovery and backup facilities as well as from the query language SQL. SQL itself comes already with useful algorithms for data analysis such as selection, sorting and aggregation. Therefore an extension of the query language to integrate other algorithms for data mining would feel very natural to the data scientist. Regarding those advantages our research goal is to extend HyPer by data mining functionalities that can be executed directly on the database, exploiting the performance of modern database hardware for computational operations and building a general-purpose system for OLTP, OLAP and data mining queries. This thesis presents a proof-of-concept implementation of data mining in HyPer, and is part of a larger project that goes one step further and will provide an intuitive user interface around the algorithms: A functional language that is easy to learn, avoids side-effects and provides plotting options.

1.3 Approach

As proof-of-concept, the well-known k-Means algorithm will be implemented as a HyPer operator. K-Means is one of the most popular clustering algorithms and available on all presented platforms. As the algorithm is relatively simple and straightforward, a good comparability between different tools and their performance is given. Since most data mining algorithms are non-deterministic, the time per iteration will be used as the main performance criterion. This implementation of k-Means will demonstrate the possibilities of implementing and running data mining algorithms on HyPer.
HyPer provides a programming model for the implementation of operators by combining pre-compiled C++ code with dynamically generated LLVM code. Yet, the programmer has a lot of freedom in implementing complex data mining operators.

This work compares different implementation aspects of the k-Means algorithm and aims to detect best practices, patterns and building blocks for the implementation of further algorithms.
The remainder of this thesis is organized as follows: In the next section, fundamentals of the HyPer database are presented. In Chapter 3, a brief introduction of the terms knowledge discovery and data mining is given. Then, the k-Means clustering algorithm is introduced. Chapter 5 discusses other data mining tools and databases with data mining functionalities. In Chapter 6 the implementation of the k-Means clustering algorithm as HyPer operator is depicted. Chapter 7 discusses extensive experiments that demonstrate the applicability of data mining algorithms on HyPer. Chapter 8 concludes the thesis.

2 The HyPer Main Memory Database

In this chapter, the relational main memory database system HyPer [KN11] is introduced. HyPer is a database system combining both OLTP and OLAP processing and is the main research subject of this work: Extending HyPer in order to execute data mining algorithms directly in the relational database.

2.1 Motivation

As already mentioned, historically databases are disk-based systems separated into two parts: An Online Transactional Processing (OLTP) system, optimized for high rates of mission-critical, transactional write requests, and an Online Analytical Processing (OLAP) system, executing long-running queries. The data synchronization is ensured by an ETL process that extracts, transforms and loads data from the OLTP to the OLAP system, which is usually done periodically.
Obviously, this architecture reveals several drawbacks. The periodical execution of the ETL process results in stale data on the OLAP system, e.g. when implementing a nightly ETL process, data can be outdated for up to 24 hours which is problematic for real-time data analysis. Furthermore, two systems lead to higher costs: Hardware and software costs must be taken into account as well as maintenance and incident management. Additionally, implementing an ETL process can make the two systems overly complex in contrast of having one system.
Addressing these challenges the Hybrid OLTP & OLAP High-Performance Database System (HyPer) was developed, with the goal to process OLTP transactions with high performance and throughput, and, on the same system, process OLAP queries on up-to-date data.

2.2 The HyPer Architecture

In this section we give an overview about the HyPer system architecture and important design decisions. The argumentation in this section is based on [KN11].

2.2.1 High-Throughput OLTP Processing

HyPer achieves major performance gains by omitting typical disk-based database characteristics that are not necessary anymore when data resides in main memory. For example, database-specific buffer management and page structuring is not needed. Instead, data is stored in main memory optimized data structures within the virtual memory. Hence, HyPer can use the highly efficient address-translation of the operating system without any additional indirection.
Historically, OLTP transactions are suffering from slow disk access and I/O time caused by disk-based database systems. Therefore, OLTP transactions are executed in parallel to exploit the waiting time and to enhance the performance of the system. In contrast, OLTP processing on HyPer is very fast because all the data is already loaded into main memory and slow disk access is omitted and not a bottleneck anymore. Therefore, transactions do not have to wait for I/O and HyPer can implement OLTP transactions as a single-threaded approach and executes all transactions sequentially. This is possible because the execution time of an OLTP transaction is in microseconds and even without parallel execution of transactions, a high-throughput is achievable. [Müh+14] shows that 100,000 TPC-C transactions per second are possible. Another advantage of this simple model is that locking and latching of data objects becomes redundant since only one transaction is active for the entire database.
Even though HyPer achieves a high-throughput with sequential execution, these constraints can be relaxed when the database tables are partitioned. An example is multi-tenancy [Aul+08] [Aul+09]: Several clients are using the database, but do not have access to the data of the other clients. Then, the OLTP transactions can be executed in multiple threads to exploit the multi core database hardware. Still, no locking and latching is necessary since all transactions are executed sequentially within a partition.

2.2.2 Long-Running OLAP Queries

Obviously, this sequential execution of transactions is only possible if there are not any long-running transactions. Long-running transactions would be a bottleneck for the entire database and all following queries have to wait until its termination. Therefore this section discusses how HyPer implements OLAP queries on up-to-date data without negatively affecting the OLTP transaction performance.
HyPer considers OLAP queries as a new process and creates a snapshot of the virtual memory of the OLTP process for its OLAP processes. In Unix-based systems, this is done by `forking` the OLTP process and creating a `child` process for the OLAP process, exploiting the virtual memory functionality of the operating system. The OLAP process takes an exact copy of the OLTP system on process creation and is now able

to execute long running queries without interfering the OLTP transaction throughput. Thanks to modern operating systems the creation of snapshots is very efficient because the memory segments are not physically copied. Instead, operating systems apply a `copy-on-update` strategy. That means that both OLTP and OLAP processes are sharing the same physical main memory location since their virtual address translation maps to the same segments. Therefore copying memory on creation is not necessary. Only if an object gets updated by the OLTP process, a new page has to be created for the OLTP process, while the OLAP virtual memory page is still the one available when the process was created. Since this mechanism is supported by hardware, it is very fast and efficient without any implementation overhead for the HyPer database system. Experiments have shown that a `fork` can be achieved in several milliseconds, which is almost independent of the size of the OLTP system.
Since OLAP queries are read-only HyPer can execute them in parallel in multiple threads sharing the same snapshot. As in the sequential execution, locking and latching is not necessary because the used data structures are immutable. This inter-query parallelization can tremendously speed up query processing on multi core computers. Another approach is the creation of multiple OLAP session by forking the OLTP memory periodically. Therefore, for each OLAP session a new virtual memory snapshot is created and used as the current snapshot for the new session. This allows parallelization not only as inter-query parallelization but also among the different OLAP sessions.

In this chapter we have shown the advantages of a modern main memory system such as HyPer. Without the bottleneck of disk I/O, database operations can be run sequentially with up to 100,000 TPC-C transactions per second. OLAP queries can run very efficiently in parallel to the OLTP transaction system on different virtual memory snapshots with access to up-to-date data.
With these promising results for OLTP and OLAP processing, the next step is to extend HyPer by data mining functionalities. Then, HyPer provides even more possibilities for efficient, real-time data analysis on a running transactional database. In Chapter 6 we discuss these implementations of data mining algorithms, including how HyPer implements operators in general using the `consume produce` programming model and LLVM code generation.

3 Knowledge Discovery and Data Mining

In this chapter we clarify the terms and the importance of knowledge discovery and data mining. In a nutshell it is the process of discovering knowledge and patterns from data. Therefore, many tools and algorithms exist which we describe in the following sections.

3.1 Motivation

As already mentioned, data growth is exploding and new data is generated every day by every aspect of daily life, such as businesses, society, science and medicine. With this huge amount of data available, it has become one of the most valuable resources of our decade. News and Blogs are even considering data as important as oil[1] or as the new currency of our decade[2].
However, before data has any value it has to be transformed into knowledge. For example, e-commerce companies are very keen to find out not only what their customers have bought but also what they are likely to buy in the future. This knowledge is then used to generate more revenue. A very well known example is amazon.com which is advertising a variety of similar products the customer recently looked for, or products another customer with similar characteristics has purchased in the past. Therefore data mining techniques are used to boost consumerism. Additionally, this knowledge might be used to optimize storage cost, e.g. by knowing how much of certain products will be needed to distribute in the next weeks.
Other areas for data mining are telecommunication network carriers with a huge amount of data traffic generated every day, the health and medical industry and the Web in general, to name just a few examples. Again it is all about knowledge. The telecommunication network carriers are interested in customer usage behaviour, while state authorities are interested in wiretapping of potential criminals. Medical data might be used to install an ameliorated patient monitoring and Web data can be used for search engines to find the best results or for ad networks to deliver promising ads. The list of sources to generate data from and the knowledge one is interested in is

[1] http://www.forbes.com/sites/perryrotella/2012/04/02/is-data-the-new-oil/.
[2] http://www.europeanvoice.com/event/data-the-new-currency/.

endless.

3.2 Knowledge Discovery

After understanding the importance of data mining for gaining knowledge and using this knowledge for further decision making, this section defines data mining in greater detail. The term data mining itself often leads to confusion - we are not mining data but instead we are looking for knowledge and interesting patterns in a given data set or database. Therefore, the term knowledge discovery is often more appropriate.
However, the usage of knowledge discovery and data mining as synonyms is not entirely accurate. Instead, data mining is often seen as a part of the knowledge discovery process as depicted in Figure 3.1. This process starts with the preprocessing steps data cleaning, integration, selection and transformation. Afterwards, data mining and its algorithms are applied and the discovered patterns are evaluated and presented. In this work we use the term data mining as described in Figure 3.1 since we are mainly interested in the algorithms applied in the data mining phase. Nevertheless, future work on HyPer will cover the entire knowledge discovery chain and implement techniques to clean the data as well to present the results. Therefore, this section gives a profound overview over all the knowledge discovery steps; a focus on the data mining algorithms follows in Section 3.3.

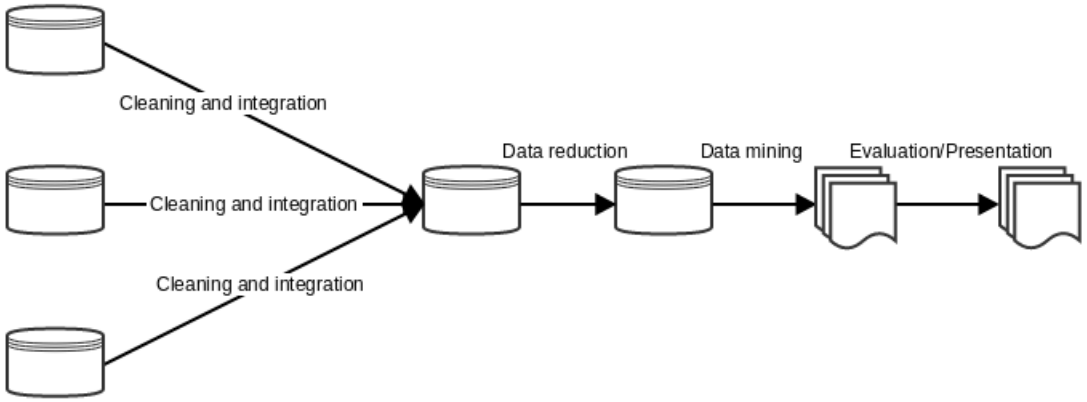


Figure 3.1: The Process of Knowledge Discovery.

Regarding all those steps, Han et al. [HKP11, p. 8] give a detailed definition of data mining:

> "Data mining is the process of discovering interesting patterns and knowledge from large amounts of data. The data sources can include databases, data warehouses, the Web, other information repositories, or data that are streamed into the system dynamically."

In the following sections we discuss each phase in detail and start with data cleaning.

3.2.1 Data Cleaning

The first step of knowledge discovery needs to ensure that the data we are interested in is complete, clean and consistent. Real world data is usually none of it. Data sets are incomplete, i.e. values are missing. If these missing values are crucial for the further data mining process, several data cleaning techniques exist to fill the gaps, e.g. by using an average value such as the mean or the median.
The same is true for inconsistent or redundant data. An example of inconsistent data are distinct values with the same meaning. Additionally, we often see random error or variance in a measured variable. This is called noisy data and must be cleaned for proper data analysis e.g. by smoothing the data set, using binning techniques.

3.2.2 Data Integration

Data mining often requires data from several sources. These sources can be databases, data warehouses, transactional data or plain data files. Also complex data might be used, such as time-related data, data streams, spatial data, multimedia data and graph data. All of these sources have to be integrated into one coherent data set which is then used for further analysis. There are several challenges regarding data integration, the most important is the entity identification problem: When combining data from different sources same data objects can have different names and types in different schemas. Therefore, data integration often requires domain knowledge of the different data sources and techniques used in the data cleaning phase to get a clean, consistent data set and to avoid redundancy.

3.2.3 Data Reduction

After cleaning and integrating data into a coherent data set, the data is ready for further mining. However, those data sets are often of huge size and contain much information, not all necessary for the data mining process. Data reduction techniques can help to reduce the information density and speed up the data analysis and mining process. Simple techniques for data reduction are projection, selection and aggregation, available in every database system. More advanced techniques are dimensionality reduction,

e.g. by a principal component analysis, numerosity reduction above aggregation, e.g. clustering or sampling, and data compression.
Other methods are data transformations which change the data by converting values into another format e.g. by using normalization. Another possibility is data discretization, e.g. by transforming numeric data into intervals.
The aim of all of the addressed methods is to reduce the size of the initial data set and therefore to make it easier to get the information the user is interested in. Some methods of the presented data reduction techniques are data mining techniques as well, e.g. clustering. Therefore the boundary between data reduction and data mining is not always easy to set.

3.2.4 Data Mining

After applying the presented preprocessing steps the data set is now ready for data mining techniques. In the data mining phase we apply algorithms for finding interesting patterns in the data set. Crucial techniques are mining frequent patterns and association rules, classification, clustering and outlier detection. Algorithms for performing those techniques are presented in Section 3.3.

3.2.5 Pattern Evaluation

After generating those patterns we have to evaluate them to figure out how valuable the acquired knowledge is. There exist several criteria for pattern evaluation. First, a pattern has to be understandable. A clustering for example does not give any knowledge if the reason for the clustering is not comprehensible. Therefore, the reasons of a data mining pattern must be clear for the user. That leads to the next criteria of evaluation: A pattern has to be valid, useful and novel. Only then, a pattern can be called interesting and can be used as knowledge, e.g. by fulfilling a hypothesis the user tries to confirm.

3.2.6 Knowledge Presentation

Knowledge presentation is the final step of knowledge discovery. Interesting patterns have to be described and visualized to the user in an appealing, understandable format. Apart from textual explanation, several data visualization techniques exist and should be applied for the respective situation.

3.3 Algorithms of Data Mining

In the following sections we discuss data mining techniques such as frequent itemset mining, classification, clustering and outlier detection. For each group the purpose of the technique is given and its most important algorithms are presented.

3.3.1 Mining Frequent Patterns and Association Rules

Frequent pattern mining aims to find the most frequent items in a data set and its association rules. A typical example is the market basket analysis: Supermarkets and e-commerce stores are interested in what their customers are buying, in particular in items that are commonly bought together. In other words, which items are frequently put in the same market basket?
This knowledge is very valuable: It is not only about finding out which items are very popular and therefore get a better a position in the market or website. It is also about knowing which items are bought together which can lead to interesting desicions: Related items could be placed next to each other to make the shopping experience for the customer more convenient. Another approach could be to put related items far away from each other, so that the customer has to walk around in the store and is more likely to buy an additional product. These are just two possible strategies that could be applied with the knowledge resulting from frequent pattern mining.
Most algorithms for frequent pattern mining expect transactional data as input, because this type of data represents a market basket the best. The most popular algorithm is the Apriori algorithm [AS94], working in an iterative manner. Apriori generates itemsets of length k and checks if these itemsets appear frequently, i.e. if their count exceeds a given threshold. All the valid itemsets are then used in the next iteration to generate itemsets of length $k+1$. The algorithm converges if no more itemsets can be generated. Since the candidate generation and the scanning of the database is expensive, there are attempts to improve the Apriori performance, e.g. [SON95][PCY95] as well as to establish other techniques. One of them is Frequent-Pattern-growth (FP-growth) [Han+04], an algorithm compressing the database into an FP-tree and therefore finding frequent itemsets without itemset generation. Furthermore, there exist algorithms for more specific cases, e.g. for finding frequent patterns in high-dimensional, spatial and multimedia data.

3.3.2 Classification

In data mining, classification uses existing data as knowledge for predicting future events. A typical example is the identification of spam emails. This is done by labeling

existing emails and categorizing them as spam or non-spam. Such a data set is called the training data set. The algorithm uses then this training knowledge to classify new, unknown emails as spam or non-spam.
Since classification needs training data it is also called *supervised learning*. That means that before the start of the data mining process, previous knowledge has to be acquired. In our example, someone has to label emails first as spam or non-spam before the algorithm can be applied.
There are many classification algorithms in practice, popular ones are Support Vector Machines (SVM) [BGV92], Decision Trees [Qui93] and Neural Networks [Ros58].
An SVM classifier starts by building a model upon a training data set. Each data tuple is represented as a vector in the SVM model space. Since the data is labeled the SVM knows which data tuples belong together and tries to find separating hyperplanes between them. These hyperplanes can be described by a small subset of the data vectors called the *support vectors*. This fact makes the SVM very efficient on high-dimensional data. Unknown data tuples are then modeled as a vector in the SVM space and belong to the same category as the other data tuples they are separated together with by the hyperplanes.

3.3.3 Clustering

While classification requires apriori knowledge, i.e. tuples must be labeled to be used as training data, clustering does not require any previous knowledge. In that sense, clustering is a very convenient method to gain knowledge about a data set without the need of knowing exactly what the result should look like. Therefore, clustering is also called *unsupervised learning*.
An example is Google News, presenting and grouping news headlines obtained from many news websites. There is no predefined set of available news topics, instead the topics can change daily. Therefore, using classification with training data is not feasible. Instead, clustering can be used to find similar topics within the latest news articles and group them together. The clustering algorithm tries to find the best grouping of news, meaning that the news in one group have a high similarity with each other, while they are very different to the news in other groups.
Clustering techniques provide a variety of algorithms. These algorithms can be categorized in the following four groups:

- Partitioning methods: Partitioning methods are the most popular clustering algorithms, trying to find clusters of spherical shape. A distance function is used to measure similarity and dissimilarity among the clusters. Popular algorithms are k-Means [Llo82] and k-Medoids [KR90].

- Hierarchical methods: Hierarchical methods are useful for data consisting of several hierarchies or levels. Clustering is applied by going up or down the hierarchy tree by merging or splitting subtrees, respectively. Similar to partitioning-based methods, hierarchical cluster algorithms tend to find spherical clusters.
- Density-based methods: For finding clusters of arbitrary shape, such as oval or "*S*-shaped" clusters, partitioning and hierarchical techniques are limited. Density-based clustering algorithms obtain much better results, using dense regions in data sets for identifying clusters instead of distances from a center point. The most popular algorithm is the DBSCAN (Density-based spatial clustering of Applications with Noise) [Est+96] algorithm. It works by finding core objects, i.e. objects within a dense neighbourhood and joining those core objects together to find dense regions, i.e. clusters.
- Grid-based methods: All the presented methods so far are data-driven, i.e. groupings are detected by the distribution of objects in the embedding space. In contrast, grid-based methods are space-driven, i.e. the object space is mapped to a finite number of cells, resulting in very fast processing times for multi-dimensional data. Popular algorithms are STING [WYM97] and CLIQUE [Agr+98]. CLIQUE is a special form of a grid-based algorithm, since it is also density-based.

3.3.4 Outlier Detection

Outlier detection techniques aim to find data objects that behave in a different way than the majority of data objects. For an e-commerce system outliers can be clients spending much more money than the average client. This information is very valuable since the company is eager to make this client particularly happy, e.g. with special customer care treatment. Also in fraud detection and medical systems, outlier detection is very important to observe a security breach or a patient problem as early as possible.
Obviously there is a strong correlation between outlier detection and clustering. Data objects that do not fit into a cluster are potential outliers. Therefore clustering techniques can be applied for outlier detection. However, their main purpose is to find clusters, whereas outlier detection algorithms are specialized in finding outliers. Often we have to optimize these algorithms to omit all dense areas to find outliers in a more efficient way.
Apart from *unsupervised learning*, *supervised learning* can be applied for outlier detection as well. First, data is labeled as normal or outlier, and upon that information future data can be classified. This data is then used as training data to classify data sets as outliers or normal data. The challenge of using classification methods for outlier detection is that outliers are very rare by definition, thus finding training instances

resembling outliers is not trivial. Therefore typical classification algorithms often have to be adapted and optimized for outlier classification as well.
Apart from clustering and classification techniques, statistical and proximity-based methods are also used for outlier detection.

The remainder of this work discusses and implements the clustering algorithm k-Means, one of the most well-known and popular data mining algorithm. It is implemented in most modern data mining tools such as R and Julia, and therefore a good comparison can be achieved. The algorithm is introduced in Chapter 4 and implemented in Chapter 6. The results are then discussed in Chapter 7.

4 The k-Means Clustering Algorithm

In this chapter the data mining algorithm k-Means is presented. K-means is a well-studied clustering algorithm, partitioning a data set into k clusters, where all data objects within a cluster are similar to each other and dissimilar to the data objects in the other clusters. The goal is to find the best clustering, minimizing the total distance between each data point and its assigned center. While the solution to that problem is NP-hard [MNV09][Alo+09], there are several heuristics, in particular the Lloyd algorithm [Llo82], which is a local search solution to this problem.

4.1 Motivation

We decided to use k-Means as proof-of-concept algorithm for data mining in the HyPer database because it is an algorithm widely used and very popular. In fact, a survey of clustering data mining techniques in 2002 states that the algorithm "is by far the most popular clustering algorithm used in scientific and industrial applications" [Ber02]. K-Means is also part of the 10 top data mining algorithms identified by the IEEE International Conference on Data Mining (ICDM) [Wu+08], next to other famous algorithms such as the SVM [BGV92] and the PageRank [Pag+99] algorithm.
Since the algorithm is popular and easy to understand, k-Means is suitable as first implementation of a data mining algorithm on HyPer. To our best knowledge, all major data mining tools are implementing k-Means, which is a basic requirement regarding the comparability among existing tools and our own implementation. Hence, the running time is a good measurement for comparing the performance; the cluster compactness, i.e. the sum of squared errors, can be used as general criterion for the formal correctness of the implementation. Parts of k-Means can be executed in parallel, and since HyPer supports parallel computation, it will be interesting to evaluate how a single-threaded execution of k-Means performs against a multi-threaded computation. In fact, we are expecting tremendous performance gains executing k-Means in parallel on a multi core machine.

4.2 The Lloyd Algorithm

In this section we discuss the Lloyd algorithm [Llo82], usually referred as k-Means in literature and in this work as well. Formally, the k-Means problem and the k-Means algorithm are described the following: For a given integer k and a data set of n data points $X \subset \mathbb{R}^d$, choose k centers C to minimize the sum of squared error function,

$$sse = \Sigma_{x \in X} min_{c \in C} ||x - c||^2.$$

After finding these center points, each data point is assigned to its closest center which forms the clustering.
Solving such a problem is NP-hard, however Lloyd proposes a local search solution usually resulting in good groupings. The algorithm starts with k arbitrary center points, typically chosen at random from the data points. Then, each data point is assigned to the closest center using the euclidean distance. After assigning each data point to its closest center, the centers get updated, i.e. the new center is the mean coordinate of all the data points assigned to this center. Then the next iteration begins, assigning the data points again, now to the updated centers. This continues until the process stabilizes and the algorithm converges.
Formally, the k-Means algorithm is described then the following:

1. Arbitrarily choose k centers $C = \{c_1, c_2, \cdots, c_k\}$ uniformly at random from X.
2. For each $i \in \{1, \cdots, k\}$, set the cluster C_i to be the set of points in X that are closer to cluster c_i than to any other cluster c_j for all $j \neq i$.
3. For each $i \in \{1, \cdots, k\}$, set c_i as the center of all points assigned to C_i:

$$c_i = \frac{1}{|C_i|} \Sigma_{x \in C_i} x.$$

4. Repeat Steps 2 and 3 until C no longer changes, i.e. the algorithm converges.

This simple algorithm terminates in practice very fast and provides mostly good results. However, the clustering can be arbitrarily bad for specific data sets. Therefore, MacQueen [Mac67] and Hartigan and Wong [HW79] propose improvements over the Lloyd algorithm, often leading to a faster convergence. Since the goal of this work is to demonstrate k-Means as a proof-of-concept implementation on HyPer, and since most tools, such as R and Julia, are implementing the Lloyd algorithm, we decided use the Lloyd algorithm in the remainder of this work.

4.3 An Example of the k-Means Algorithm

In this section the k-Means algorithm is presented by a made-up example for deeper understanding. Our example data set consists of five data points with two dimensions: $x_1(3,9), x_2(2,5), x_3(5,8), x_4(7,5), x_5(4,2)$. This data will be clustered using the k-Means algorithm with $k = 2$, i.e. we are searching for two clusters in the data set. First, the algorithm starts with an initialization phase: Randomly, we choose two instances as initial center points. In this example, these data points are $c_1(3,9)$ and $c_2(4,2)$.
Next, we begin the first iteration of the k-Means algorithm and compute the euclidean distance from each data point to c_1 and to c_2. Each data point is then assigned to the closest cluster, as shown in Table 4.1. After Iteration 1, the data points x_1 and x_3 are closer to c_1 and therefore assigned to Cluster C_1, while the data points x_2, x_4 and x_5 belong to Cluster C_2.

Table 4.1: Computations in Iteration 1.

Data Point	distance to $c_1(3,9)$	distance to $c_2(4,2)$	Cluster
$x_1(3,9)$	0.00	7.07	C_1
$x_2(2,5)$	4.12	3.61	C_2
$x_3(5,8)$	2.24	6.08	C_1
$x_4(7,5)$	5.66	4.24	C_2
$x_5(4,2)$	7.07	0.00	C_2

Now, the centers have to be recalculated as the mean of all assigned data points. The result are the updated center points $c_1(4, 8.5)$ and $c_2(4.33, 4)$. These are used to compute the distances again for each data point in Iteration 2 as shown in Table 4.2. As the center points have changed, the distances to the centers are different and data points could be assigned to new clusters. In our made-up example the data points are assigned to the same clusters as after Iteration 1, i.e. no assignment changed. Therefore the algorithm converges and we have found the result. Figure 4.1 depicts the two clusters. Cluster C_1 with the center point c_1 is marked by blue data points and is positioned on the top of the chart, while Cluster C_2 with the center point c_2 is a broader cluster from the middle of the chart to the bottom.

In this section a made-up example showed the phases of the k-Means algorithm in depth. These fundamentals help us to get a better understanding in Chapter 6, when we discuss the implementation details of k-Means as HyPer operator.

Table 4.2: Computations in Iteration 2.

Data Point	distance to $c_1(4, 8.5)$	distance to $c_2(4.33, 4)$	Cluster
$x_1(3, 9)$	1.12	5.17	C_1
$x_2(2, 5)$	4.03	2.54	C_2
$x_3(5, 8)$	1.12	4.06	C_1
$x_4(7, 5)$	4.61	2.85	C_2
$x_5(4, 2)$	6.50	2.02	C_2

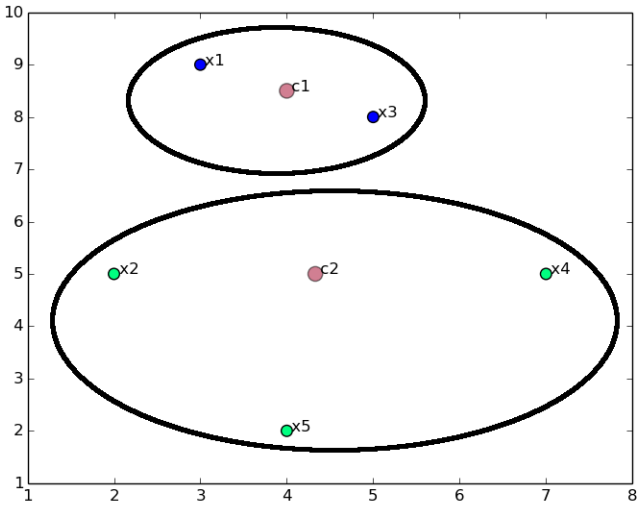


Figure 4.1: K-Means Example Clustering.

4.4 The k-Means++ Initialization Strategy

Even though the Lloyd algorithm provides good results in practice, the resulting clustering can be arbitrarily bad for some data sets. Particularly, the random choosing of the initial cluster centers can lead to a bad grouping which cannot be changed during the clustering process. Therefore, [AV07] proposes a variation to the k-Means initialization strategy, called k-Means++. The centers are still chosen randomly from the data points, but the data points are weighted according to their distance from the closest, already chosen center. Hence, the probability of choosing data points as center that are far away from each other increases, leading to enhancements in speed and accuracy of the clustering.
Formally, k-Means++ can be described the following: Let $D(x)$ be the shortest distance

from a data point to the closest, already picked center.

1. Take the first center c_1, chosen uniformly at random from X.
2. The next center c_i is chosen from $x \in X$ with the probability

$$\frac{D(x)^2}{\Sigma_{x \in X} D(x)^2}.$$

   This weighting is called the D^2 weighting.
3. Repeat Step 2 until all k centers are selected.
4. Continue as in the standard k-Means algorithm.

Many clustering tools implement a k-Means++ initialization strategy for k-Means such as Julia and Weka, therefore we will also provide this enhancement for the k-Means operator implementation in HyPer in Section 6.4.3.

5 Related Systems

In this chapter we present state-of-the-art data mining frameworks and languages, their main characteristics and areas of application. We try to give a broad overview starting with research tools and dynamic languages, big data solutions, software systems that are strongly integrated with a database, and finally in database processing.

5.1 Research Tools

In this section we discuss two data mining tools - Weka [Hal+09] and ELKI [AKZ08] - that are strongly originated in the research community and developed at Universities. Additionally, Weka is often used for teaching data mining techniques. Nevertheless, both tools, in particular the more mature Weka are used in practice by industrial scientists too.
Weka is a data mining software "workbench" developed at the University of Waikato that gained big attention in the machine learning and data mining community over the past decade. It provides and integrates a large collection of algorithms for all aspects of data mining such as data preprocessing, classification, regression, clustering, association rule mining and visualization into one framework. Weka is written in Java and provides a stand-alone GUI with the "Weka Explorer", but can also be executed directly within Java programs.
Weka's default file format are arff files, which is an ASCII text file that describes instances and its attributes. In contrast to csv files, arff files can be read incrementally by Weka, because the header of arff files already determines whether a column is numeric or nominal. In contrast, all instances have to be inspected when using csv files. Therefore, arff files provide a performance improvement and provide high accuracy about the data type of attributes.
ELKI (Environment for Developing KDD-Applications Supported by Index Structures) is a relatively new data mining framework developed at the Ludwigs-Maximilians-Universität München in 2009 with an even stronger focus on research. It's main objective is to enable a fair comparison of data mining algorithms based on experimental evaluation. In the strongly evolving field of data mining, many algorithms are released every year and are never be seen again once the paper is published. Often, the comparison of algorithms is based on a experimental evaluation, which is not always

open and reproducible. Moreover, the code is not always published, documented and clear to use. ELKI provides an implementation framework for new data mining algorithms, leading to a better comparability among them and therefore to a fairer evaluation of the newly proposed algorithm.
As Weka, ELKI is written in Java and can be either used via graphical user interface or within Java programs.A major strength of the framework is that it is able to read arbitrary data types and the support of any distance or similarity measure. Therefore, all kinds of complex data types can be integrated into existing algorithms, merely by providing a distance function for the given data type. ELKI also encourages the use of index-structures to achieve performance gains when working with high-dimensional data sets.

5.2 Dynamic Languages for Statistical Computing

While Weka and ELKI both provide an excellent suite for data mining, data scientists often prefer a more dynamic language over often bulky Java code. In particular for exploratory data analysis, dynamic scripting languages are very popular to get a first hands-on-feeling for the given data. Special purpose languages exist, with a syntax and built-in data structures that make data analysis tasks fast and concise. This section presents the R [R D08] language which is popular for decades, the previously announced Julia [Bez+12] language and the Python ecosystem SciPy [J+01]. R and Julia are both special purpose languages designed for data analysis. While Python is a general purpose programming language, the SciPy ecosystem provides excellent tools for data analysis and data mining with similar syntax and data structures.
R is a mature language for statistical computing and data analysis with a rich set of graphical techniques that makes it often researcher's number one tool for creating graphics for publications. It is designed as a true computer language, and most of the R libraries are written in R itself. However, C, C++ and Fortran code can be linked and called at run time and is often used for computationally intensive tasks. It is also possible to manipulate R objects directly via C code.
R depicts itself as an environment for statistical techniques: It highly extensible via packages and provides many packages for the most common algorithms, available via the CRAN network. RStudio provides an IDE for R programmers, and software products like JGR and R Commander provide a GUI for R programs.
Thanks to the underlying C, C++ and Fortran libraries, R can do basic computations like matrix math very efficient and fast. However, the R language interpreter is rather slow, which discourages writing large libraries or complex abstractions in the R languages itself. Instead, often C, C++ or Fortran extensions are used.

Julia is a modern dynamic programming language for scientific computing, designed to address some of these concerns. As R, Julia is a programming language written itself mainly in Julia, using C and Fortran for computationally expensive tasks. In contrast to R, Julia uses an LLVM-based just-in-time (JIT) compilation for interpreting its own language. This results in stunning performance results. While the running time for basic implementations does not differ, since R and using C++, basic language features differ tremendously. The same compilation technique is used in HyPer, therefore it will be interesting to compare both techniques.
For data analysis, external packages are available allowing the execution of state-of-the-art data mining algorithms. Julia's weakness, however, is its libraries. R has CRAN, certainly the most impressive collection of statistical libraries available anywhere. Julia also lacks a rich development environment, like RStudio, and has only rudimentary support for plotting, which is a pretty critical part of most exploratory data analysis. Julia does, however, have a very active community
SciPy is a Python environment enhancing the Python language into a data analysis environment. The NumPy library provides a solid matrix data structure, with efficient matrix and vector operations. SciPy includes a very large collection of numerical, statistical, and optimization algorithms. Pandas provides R-style Data Frame objects (using NumPy arrays underneath to ensure fast computation), along with a wealth of tools for manipulating them.
At the same time, Python has a huge number of well-known libraries for the messier parts of analysis. For example, Beautiful Soup is best-of-breed for quickly scraping and parsing real-world HTML. Together with Python's strong community and other libraries it makes Python a compelling alternative

5.3 Big Data Platforms

The presented tools in the previous section are great when working with megabytes and gigabytes of data, which is very common. Often, data scientists spend much time on working on a small sample of a larger set, an aggregated results, or the data set is not that big at all.
However, as data size grows big data frameworks like Hadoop are receiving a lot of attention: It provides an affordable, reliable and ubiquitous way to spread computation over tens or hundreds of CPUs on commodity hardware. In this section we discuss the most important trends of the big data community for data mining. Apache Hadoop is an open-source software for reliable, scalable, distributed storage and processing of large datasets across large clusters of computers. The heart of Hadoop is the Hadoop Distributed File System (HDFS) and Hadoop MapReduce, a simple programming

model for distributed processing.
The Hadoop distributed file system (HDFS) is a distributed, scalable, and portable file-system written in Java for the Hadoop framework. It is highly fault-tolerant and provides high throughput access to application data. HDFS stores large files as blocks replicated across multiple machines.
MapReduce is a programming model originally developed by Google [DG08] for processing large amounts of data in parallel on large clusters of commodity hardware. A MapReduce job splits the input data into chunks processed by the map tasks in parallel. The output of the map tasks are then the input of the reduce tasks which merges the intermediate results. Hadoop MapReduce provides an open-source software framework for writing MapReduce jobs in a reliable, fault-tolerant manner. Several Algorithms can be performed on the MapReduce programming model, e.g. k-Means [ZMH09]. For our research, it will be interesting to see how HyPer works with Big Data compared to Apache Hadoop.
Numerous Apache Software Foundation projects are based on top of Hadoop, such as Hive [Thu+09] and [LM10]. Apache Mahout is a library of scalable machine-learning algorithms, implemented on top of Apache Hadoop and using the MapReduce paradigm. Once big data is stored on the Hadoop Distributed File System (HDFS), Mahout provides the data science tools to automatically find meaningful patterns in those big data sets. The Apache Mahout project aims to make it faster and easier to turn big data into big information. Mahout provides an implementation of various machine learning algorithms, some in local mode and some in distributed mode (for use with Hadoop). Each algorithm in the Mahout library can be invoked using the Mahout command line.
In 2014, the Mahout community decided to move its code base onto modern data processing systems that offer a richer programming model and more efficient execution than Hadoop MapReduce: Apache Spark. Apache Spark is a data analytics cluster computing framework, working on top of the Hadoop Distributed File System (HDFS). In contrast to Hadoop's MapReduce, Spark comes with a richer programming model, leading to tremendous performance gains for some applications. Spark also provides in-memory cluster computing, making it well-suited to data mining algorithms.
Spark is a fast and general processing engine compatible with Hadoop data. It can run in Hadoop clusters or Spark's standalone mode, and The intention is to enhance the Hadoop Stack.it can process data in HDFS, Cassandra, Hive, and any Hadoop InputFormat. It is designed to perform both batch processing (similar to MapReduce) and new workloads like streaming, interactive queries, and machine learning.

5.4 Middleware Tools

While all of the presented environments are frequently used by data scientists, they demonstrate one decisive drawback: Before executing data mining algorithms, the data first has to be fetched from the database and transformed into a format readable by these tools. Therefore a first enhancement is to bring the algorithms closer to the database. In this section we present MADlib and RapidMiner.
MADlib is an open-source library providing a suite of SQL-based algorithms for machine learning, data mining and statistics, that can run with scale within a database engine. Therefore, it is not need to import/export data with an ETL process to other tools. The analytic methods can be installed and executed within a relational data base engine as long the engine supports extensible SQL. So far, MADLib works with Postgres, Pivotal Greenplum and Pivotal HAWQ.
MADlib's main functionality is written in declarative SQL statements which organize the date movement to and from disk on the current or on network machines. Loops running on a single CPU can benefit from SQL extensibility and call high performance math libraries in user defined functions. Higher level tasks can be implemented in Python code, that drives the algorithms and invokes data-rich computations that are computed in the database.
RapidMiner is another tool for predictive analysis with a strong focus on visual development. It's main goals are low entry barriers and quick speed up. It's main audience is not only Data Scientists and IT Specialists, but also the business department and stakeholders. Therefore RapidMiner sees itself as collaboration tool. The main feature for end users of RapidMiner is the graphical user interface to design data analysis routines. Therefore data mining queries can be created with a few clicks only.
RapidMiner provides three different analytic engines to deal with different data volumes, data variety and velocity of data. In general, all analytical algorithms and computations are in-memory. Since random access is often necessary for data mining algorithms this is often the fastest approach for medium size data sets. In-database mining brings the algorithms into the database, therefore the loading phase to get the data is abolished. This solution is offered for different database systems utilizing database functionality. And finally, in-hadoop compuations allows to combine the user interface of RapidMiner with the workflows of Hadoop Clusters. Therefore terabytes and petabytes of data can be analyzed with RapidMiner.
For our use case, the in-database processing of RapidMiner seems the closest related to our approach. However, the in-database engine is designed for performance, but for high data locality. That means that no ETL process is necessary if algorithms are executed on the database. However, RapidMiner's performance is best if the data is first loaded and executed in the in-memory engine.

5.5 Knowledge Discovery in Databases

However, for complex algorithms SQL operators have to be combined with high-level constructs, often resulting in bad performance. Therefore, database vendors such as Oracle and SAP are going one step further and providing their databases with more and more statistical and data mining functionalities using the existing SQL syntax and graphical user interfaces.
SAP HANA Predictive Analysis Library (PAL): SAP HANA is SAP's main memory database. Its functionality is very similar to the one that HyPer proves: It combines transactional processing (OLTP) and analytical processing (OLAP) on one system. Therefore that database and the datawarehouse are combined into one database.
SAP PAL [SAP14] is an extension for HANA to implement data mining algorithms in the areas of association, clustering and classification. The idea is similar to HyPer's: Instead of importing/exporting data to other, external tools, the best place to perform data mining algorithms is right on the database. Therefore, PAL's complex analytic computations are performed directly on the database with very high performance. The transfer time of large tables from the database to the application becomes redundant and calculations are much more inexpensive.
Oracle: As one of the market leaders for traditional, disk-based databases, Oracle provides a lot of possibilities for data mining and knowledge discovery on their database. By default, Oracle provides its databases with the Oracle Analytical SQL Features and Functions and the Oracle Statistical Functions. Oracle Analytical SQL Features are a suite to improve the already existing SQL syntax by wider range of analytical features such as rankings, windowing and reporting aggregates. Oracle Statistical Functions enhance the normal toolset by statistics such as descriptive statistics, hypothesis testing, correlations analysis, cross tabs and the analysis of variance (ANOVA).
On top of this rank the Oracle Advanced Analytics Options. It provides techniques for state-of-the-art data mining technologies by implementing in-database algorithms and R algorithms, accessible via SQL and the R language. Oracle Data Mining (ODM) provides datamining algorithms directly on the the database for improvements in performance. Also, a importing/exporting of data to other tools becomes redundant. Users can either use the Oracle Data Miner, a work flow based GUI, or the SQL API. Oracle R Enterprise (ORE) integrates the already presented R language for statistical computing and graphics with the database. Therefore, R algorithms can be executed directly on the database without an ETL process. Base R and popular R packages can be executed in-database, while every R package can be executed with embedded R while the database manages the data loading. This allows data scientists to write their own R packages and extension and bring the code to the database.

In this chapter we provided a broad overview over data mining tools and languages in general. Some of those tools will be used in Chapter 7 to compare our k-Means HyPer implementations with existing tools.

6 Implementation of k-Means in HyPer

In this chapter we present the implementation of the k-Means algorithm as a HyPer operator. First we discuss the general implementation of operators in HyPer, the used programming model and the LLVM code generation. Then, the constraints and requirements of the k-Means implementation are introduced. Next, the implementation details are depicted including data materialization, a C++ driven version, and an LLVM driven version. Additionaly to the random initialization strategy the k-Means++ initialization strategy is implemented as described in Section 4.4. Finally, a parallel realization of the k-Means operator is discussed.

6.1 HyPer Operator Fundamentals

In this section we discuss the general implementation of operators in HyPer and the involved code generation. With this programming model in mind we can then discuss how to implement the k-Means operator in HyPer. The argumentation and results of this section are based on [Neu11] and [Neu14].

6.1.1 The Consume Produce Programming Model

As all data resides in memory, main memory databases' query performance is much more dependent on the CPU cost of the query processing than on I/O cost as in traditional systems. Therefore, query processing had to be reinvented to achieve optimal performance. Before a query is executed, most database systems translate a query into an algebraic expression and start evaluating and executing this algebraic plan. Traditionally, this plan is executed using the iterator model [Lor74]: Each physical operator produces a tuple stream and iterates to the next tuple by calling the `next` function of the operator. This iterator model works well for I/O dominated, traditional databases, where CPU consumption is not a limiting factor. However, for main memory databases, this is not perfect: First, `next` is called many times, once for every single tuple for each intermediate and final result. This `next` call is usually a virtual call or a call via function pointer which makes it more expensive than a regular call and reduces branch prediction of modern CPUs. Moreover, the iterator model results often in poor code locality and complex bookkeeping. This can be seen from the functionality of the

table scan: As tuples are generated one after the other, the table scan has to remember where in the compressed stream the current tuple was and has to jump back when asked for the next one.
In order to resolve these issues, Neumann [Neu11] proposes a new query compilation strategy for main memory databases: Instead of the operator centric approach of the iterator model, processing is data centric. Therefore data can be kept in the CPU registers as long as possible, while the boundaries between the operators get more and more blurred. That means that each code fragment performs all actions on the given data, until the result has to be materialized, i.e. data is taken out of the registers. A code structure like this generates almost optimal assembly code, since all the relevant instructions for the given data are generated, and therefore the data can be kept in the CPU registers.
HyPer uses a simple programming model for its operators in order to make compilation as efficient as possible, while writing code remains understandable and maintainable for the developer: Each operator implements two functions, a `consume` and a `produce` function. The `produce` function computes the result tuples of an operator, which are then pushed to the next operator by calling its `consume` function. The next operator works the same way, after getting data in by a `consume` call of the predecessor, it produces result tuples by calling its own `produce` function. To wrap it up, each operator gets its own `consume` function called by its predecessor, calls its own `produce` function to compute the results and calls then the `consume` function of its successor. This process is shown in the sequence diagram in Figure 6.1.

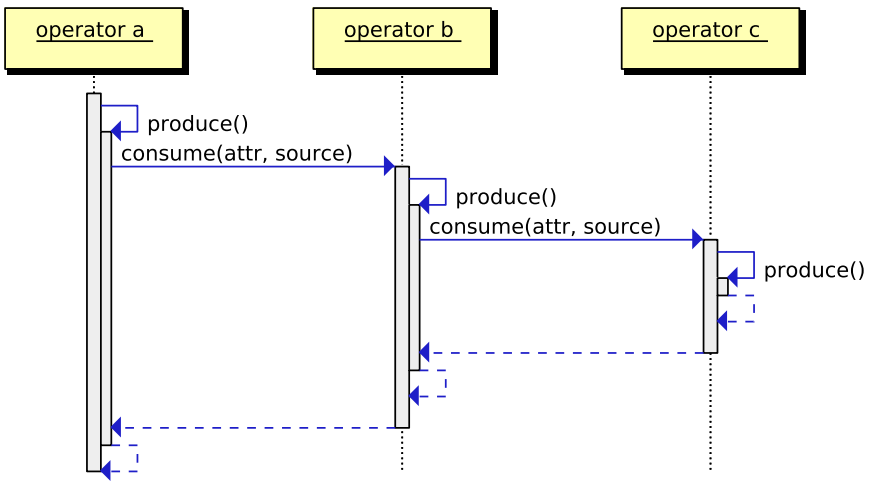


Figure 6.1: Consume Produce Sequence Diagram.

Therefore, this programming model pushes data towards the next operator, instead

of pulling the data. This results in a much better code and data locality. Tuples are pushed from one operator to the next, therefore operators benefit from keeping the data in the CPU registers, which allows very cheap and efficient computation. Only when materializing data memory has to be accessed. Additionally, small code fragments are used to handle large amounts of data in tight loops leading to good code locality and therefore high performance.
This makes the execution of algorithms very fast. However, what happens when data has to be materialized? Whenever we talk about taking tuples out of the CPU registers, i.e. materializing tuples in main memory, the operator pipeline breaks, therefore we call those operators *pipeline breakers*. If an operator materializes all incoming tuples before continuing processing, we call it a *full pipeline breaker*. An example is a join operator: One side of the two join relations has to be materialized in main memory, while the other relation can be scanned and probe the materialized data for join partners. As the join operator takes data out of the registers it is a *pipeline breaker*.
It is important to note that the `consume produce` functions are just an abstraction layer for the programmer. This abstraction layer is used by the code generation to compile assembly code. Within the generated assembly code there is no `consume produce` present anymore.

6.1.2 LLVM Code Compilation

In this section we discuss the query compilation in greater detail. As in traditional systems, queries are compiled by parsing the query, translating it into algebra and optimizing it. In contrast to a traditional system, the physical algebra is not executed, but compiled by the code generation into an imperative program. For generating this imperative program the `consume produce` model is used, as already mentioned in the previous section.
Adapting the structure of the `consume produce` model, queries are compiled into native machine code using the LLVM compiler framework [LA04]. LLVM can generate portable assembler code which can be executed directly using an optimizing JIT compiler. With LLVM HyPer uses a very robust assembly code generation, e.g. pitfalls like register allocation are hidden by LLVM. Therefore the assembly code generation is very convenient compared to other compiler frameworks. Furthermore, only the LLVM JIT compiler translates the code into machine dependent code, leading to portable code across computer architectures. Since the LLVM assembler is strongly typed, many bugs can be caught in contrast to a textual C++ code generation. Furthermore, LLVM produces highly optimized, extremely fast machine code and outperforms in some cases even hand-written code as the assembly language allows code optimization, hardware improvements and other tricks, that are hard to do in a high-level language

as C++. All this requires usually only a few milliseconds of compilation time, which is a crucial criterion for database operators with a strong focus on performance.
Additionally, LLVM code is perfectly able to interact with C++, the main language of the HyPer database. Even though LLVM code is robust and convenient to write compared to common assembler code, it is still more painful than writing code in a high-level language such as C++. An interaction of the two languages allows us to implement complex structures in C++ and to reuse database logic such as index structures alongside with LLVM code.
Therefore algorithms can be written in C++ and connected together by LLVM code, where the C++ code is pre-compiled and the LLVM code is compiled at runtime dynamically. An example is the Sort operator: The `compare` function, comparing two tuples by the rules of the sort query is dynamically generated in LLVM code, depending on the schema of the database. As actual sort function, the built-in C++ `sort` can be used. This is a great example of the mixed execution model using both C++ and LLVM code. Even though C++ and LLVM can both be used implementing an operator, LLVM code is dominant and C++ code should be seen as convenience. For performance reasons it is important that the code executed for most of the tuples is generated code, even though calling C++ from time to time is acceptable. As already mentioned, staying in LLVM allows us to keep the data in the CPU registers and is therefore the preferable way of executing a query. Calling an external function spills all registers to memory, which can be a bottleneck when doing this for all tuples in a big data set.
In conclusion, we have presented the HyPer `consume produce` programming model which allows operators to push data towards the next operator in an efficient manner. Furthermore, the LLVM compiler framework with its benefits is introduced used for code generation and interacting with C++ code at runtime. This division and cooperation of LLVM and C++ code regarding programmer friendliness and execution time is one of the dominant patterns in Section 6.4, where two alternative implementations of the k-Means operator are implemented and discussed.

6.2 Requirements and Constraints

Before discussing the technical implementation details of the k-Means operator, we introduce the requirements and constraints. Obviously, the k-Means algorithm must be implemented as a HyPer operator. That means, the `consume produce` programming model and the code generation with C++ and LLVM have to be used. The advantage of implementing k-Means as an operator is that it can be used in combination with existing SQL operators, useful for data mining as well, such as grouping and aggregation.
Regarding the functional aspects, the operator must be able to be executed in serial and

in parallel, to make use of the computing power of modern database hardware.
As input parameter, the user can specify the number of maximum iterations of the algorithm. If this parameter is omitted, the algorithm runs until convergence. For big data sets this is often a performance problem when only a few data points are changing but all the distances have to be computed all over again. Often the result is already accurate enough after a specific number of iterations. Apart from an advantage regarding the running time it is also useful to specify the number of iterations for proper testing.
Further parameters are the initialization strategy and a verbose option. As initialization strategy, the user can select between random initialization and the k-Means++ initialization, as described in Section 4.4. As default output, an additional column is added to each data row presenting the cluster identifier of the data tuple as an integer. If the verbose option is active, statistics of the run are printed to the console. This information contains the number of iterations, the final center coordinates, the number of assigned data points per center and the squared error sum.
The number of iterations is important for comparing the running time of different k-Means algorithms in a fair manner. It is possible that one run converges after three iterations, while another one converges after seven iterations. The number of iterations is non-deterministic since we are using a random initialization strategy. For fair evaluation, the time per iterations is therefore a good quality measurement and will be used in the evaluation in Chapter 7.

6.3 Data Materialization

In this section we discuss the data materialization of the k-Means operator. As already mentioned, materialization is the process of taking incoming tuples out of the CPU registers and write them into memory. Since this decreases the performance of the database, HyPer tries to avoid it, and instead keeps the data in the pipeline as long as possible, i.e. until a *pipeline breaker* occurs. Unfortunately, k-Means is a *pipeline breaker*, e.g. center tuples have to be compared with all data tuples to find the minimum distance between them. Therefore, we have to put all tuples into memory. This is depicted in Figure 6.2: Each incoming tuple will be written into main memory by the `consume` function.

This is implemented by using a combination of LLVM and C++ code. In HyPer, LLVM code resides in the `compile time system (cts)` while C++ resides in the `runtime system (rts)`. Since the `compile time system` is the entry point of an operator, the `consume` function is called and generates code for each tuple. This generated code is materializing the incoming tuples. A pointer to the materialized chunk of memory is

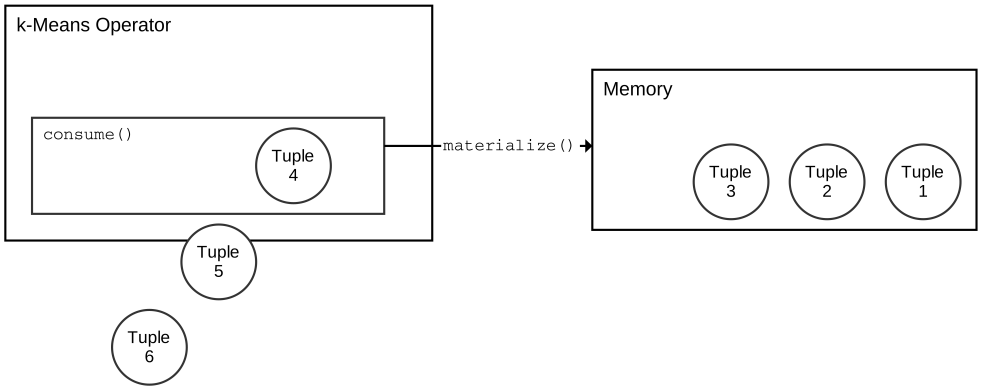


Figure 6.2: Data Materialization of Incoming Tuples.

then stored in a C++ vector in the `runtime system`. Figure 6.3 depicts this process: Each tuple is materialized into memory by the `cts` and can be referenced by a pointer stored in a vector in the `rts`. This vector can later be used to loop through the entire data set: First by looping through the vector in the `rts`, getting the pointers to the memory locations, which can then be used to load the tuple from memory into the CPU registers in the `cts`.

For k-Means it is not enough to store only the data tuples. We also need to reserve memory space for the centers. We do this by materializing the first `k` tuples of the data set two times: Once as storage for data tuples and once as storage for center tuples. Therefore we have two vectors in the `runtime system`: One for data tuples of length `n`, and one for centers of length `k`. This data materialization for k-Means is depicted in Figure 6.4.

At this point of the process, all data tuples and centers are materialized and an additional field has been added to all of them: A cluster identifier of type integer. For the center tuples, this field stores the center identifier, which goes from 0 to $k - 1$. For data tuples, this field specifies to which center and therefore to which cluster a data tuple belongs to. Initially, all data tuples are assigned to the center with identifier 0.

Another difference is the cast of data types between data and center tuples. While the data types of data tuples are specified by the table schema, the data types of the center tuples are determined differently since the center is the mean of all the data tuples assigned to that center. Therefore, an integer data value is stored as a numeric data type as center value. Numeric is an exact-value data type for integer and decimal types, and therefore able to store the mean of integer data tuples. For each data type there exists a rule to cast the type from data to center tuple.

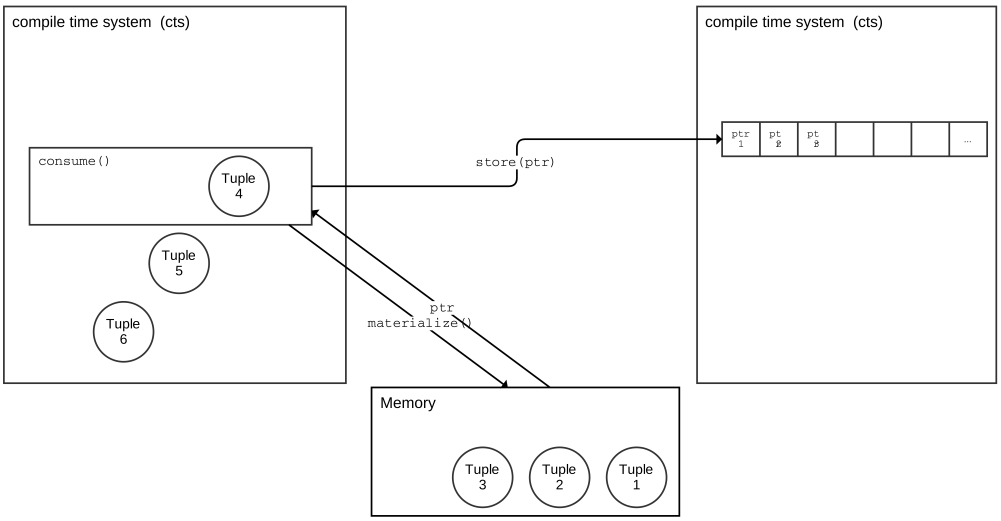


Figure 6.3: Data Materialization using the Runtime and Compile Time System.

6.4 Serial Implementation

After acquiring a basic understanding of HyPer's operator model, in particular about the interaction of dynamically generated LLVM code and pre-compiled C++ code, we can discuss the actual implementation of a serial k-Means algorithm. The most interesting part of implementing a data mining operator like k-Means is the decision about how to implement the algorithm, i.e. which parts should reside in the `compile time system` and which parts in the `runtime system`.
This question is not trivial because there are no strict rules and several possibilities. For example, a dynamic generation of code works best for comparing tuples with each other as in the sort operator, or the computation of a distance in the k-Means operator. This code has to handle different data types depending on the table schema and therefore generated code is preferable. For other parts, like the implementation of a sort function or the combination of loops in k-Means, it is not so obvious where to put the code.
In this work we present two different ways of implementing a serial k-Means operator in HyPer, first by implementing the algorithm in C++ with only a few generated LLVM functions, e.g. to compute the distance. Then, a system is presented implementing k-Means almost entirely in LLVM. Only small parts, like initializing the random centers are implemented in C++. This implementation benefits of generated, compact code and we expect performance gains over the C++ driven implementation.

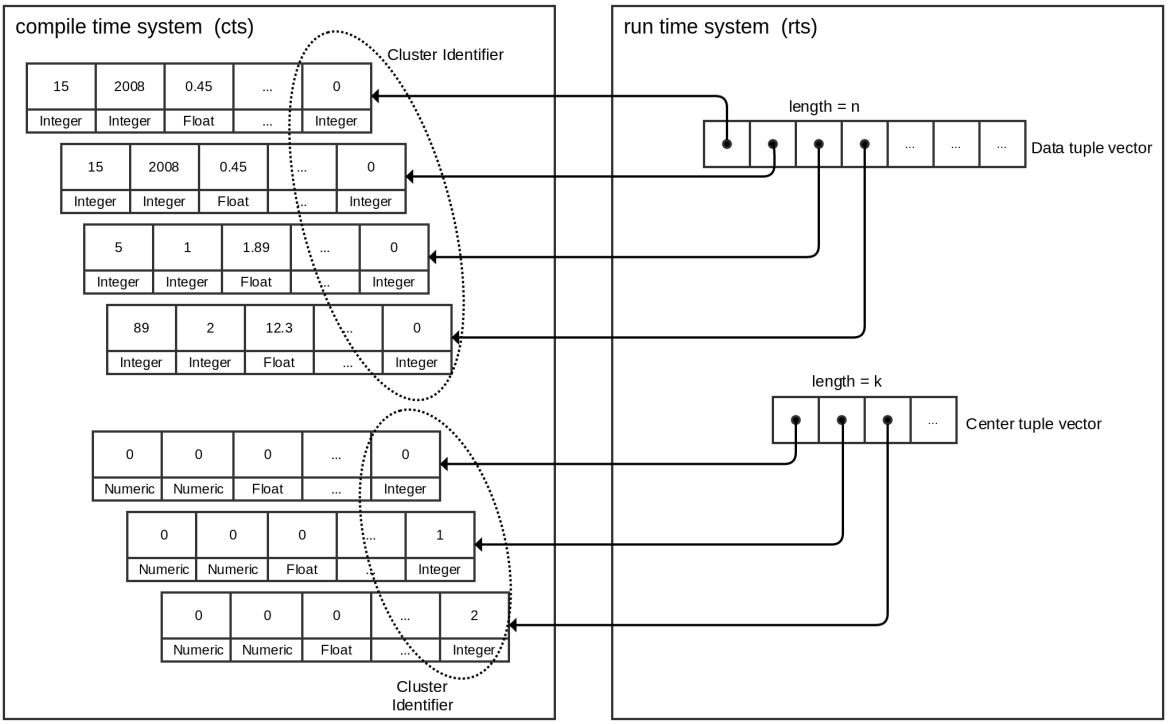


Figure 6.4: The k-Means Data Materialization in Detail.

6.4.1 A C++ driven Implementation

As first implementation approach we present a C++ driven version. The term C++ driven is maybe misleading, since all operators start their main computation after executing their `consume` function, which is an LLVM generated method. Even though the operator starts working in the `compile time system`, in this implementation, `cts` calls the k-Means operator of the `runtime system` and gives the full control to the C++ code, until the k-Means algorithm terminates.

Figure 6.5 shows this interaction in a sequence diagram: The algorithm is invoked in the `compile time system` and calls the k-Means function of the `runtime system`. The entire execution stays now in the runtime system, with calls back to LLVM from time to time.

First, the centers are selected. Let's assume we are using a random initialization strategy. `K` times, a random pointer of the data vector is selected, and its values are copied to the corresponding center tuple. Therefore, the `centerPtr` and the randomly selected

`dataPtr` are parameters of a generated `setCenter` function. This function loads both tuples from memory using its corresponding pointers. Next, the data values are casted if necessary, since we have different data types among center tuples and data tuples, and then copied to the center tuple. Afterwards, the center tuple is written back to memory. This process is repeated until all k centers are determined.

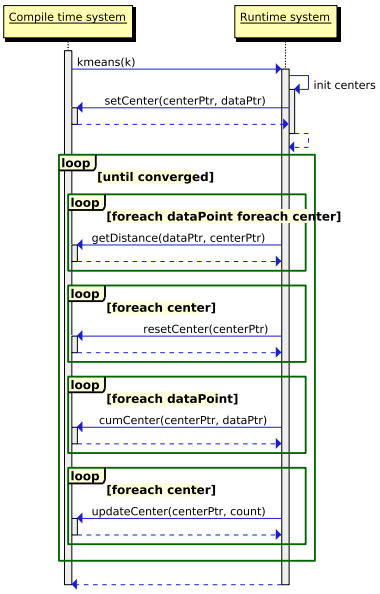


Figure 6.5: The k-Means Algorithm - C++ driven.

After selecting the initial set of centers, the `runtime system` starts its outer loop, running until k-Means converges. In the outer loop, the `runtime system` calls the `compile time system` to compute the distances between all data points and center points to find the closest center for each data tuple. Therefore, `getDistance` is invoked for each `dataPtr - centerPtr` combination. The closest center for each data point is then stored in an `unordered_map` in C++ for efficient lookup.
After finding the closest center for each data tuple, the centers have to be updated. Since the materialization of centers is done in LLVM code, the `runtime system` has to call the `compile time system` again, in fact several times: First, a `resetCenter` function is called for all center pointers to set the center values of the center tuples to zero. Then, the new mean can be computed.
The computation of the mean is done in two steps. First, the data points are accumulated for each center, and then divided by the number of data points belonging to each center. This simple mean computation leads to several `compile time system` calls for our algorithm. For each data point, its values are added to the center tuple it belongs to.

Therefore, the `cumCenter` function is called, adding the values of the data point to the center point and also updating the cluster identifier of the data tuple. The `runtime system` keeps track of how many tuples are added to each center. When finished, each center is called again to compute the actual mean of the center dividing by the count of data tuples assigned to the center using the `updateCenter` function. This concludes the first iteration of the algorithm. The next iteration starts then again with the computation of the closest center for the data points. This process continues until the algorithm converges.
Using this implementation approach the main control of the algorithm remains in the `runtime system`. However, this kind of implementation leads to many calls between the `compile time` and the `runtime system`, in total $(n+2) \cdot k + n$ calls per iteration, as Table 6.1 shows. The next section presents an implementation that prevents the algorithm from too many calls between the two systems, which might affect the performance of the operator in a beneficial way.

Table 6.1: Generated Function Calls per Iteration.

Generated Function	Calls per Iteration
`getDistance`	$k \cdot n$
`resetCenter`	k
`cumCenter`	n
`updateCenter`	k
Total	$k \cdot n + k + n + k = (n+2) \cdot k + n$

6.4.2 An LLVM driven Implementation

As already stated, LLVM code generation encourages the interaction between LLVM and C++ code which is exploited a lot in the C++ driven implementation of k-Means: The algorithm is implemented in C++ and benefits from high level data structures such as `unordered_maps` and the convenience of the C++ syntax, allowing a quick implementation.
However, the approach presented in the previous section leads to many calls between the `compile time` and the `runtime system`, as shown in Table 6.1, which could possibly affect the performance of the operator. Therefore, a second approach has been explored, implementing the k-Means algorithm almost entirely in LLVM code. Only the random initialization of the center points remains in C++. When thinking about such an implementation, we have to be aware that we cannot use C++ high-level data structures anymore, but have to work with what LLVM gives us.

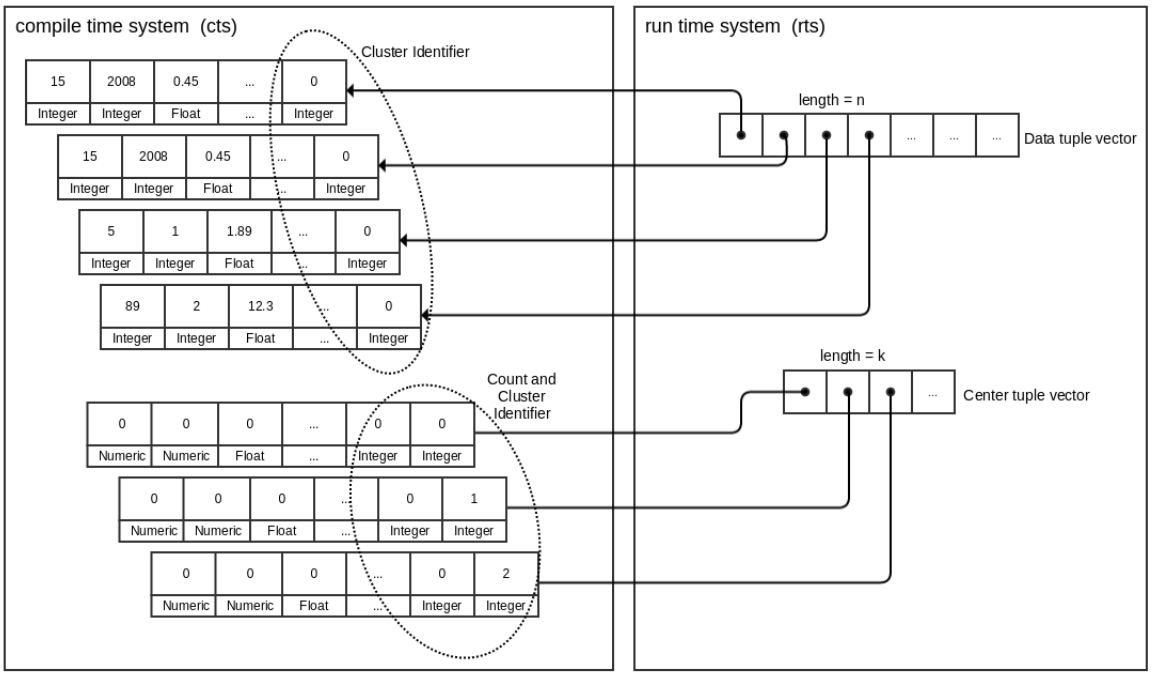


Figure 6.6: The k-Means Data Materialization - LLVM Implementation.

The only data structure we have used so far in LLVM was a structure to materialize the data and center tuples. In order to keep things simple, we exploit this data structure even further to use it with the LLVM k-Means implementation, without using any other data structures in addition. So the question is how to extend the existing data structure of the `compile time system` to emulate the C++ code we want to omit? This is done by extending the center data structure when materializing the centers in the `consume` function. As Figure 6.6 shows, an additional field has been added to the center tuple to keep track of the count, determining how many data points are close to this center. With this small modification we can implement our algorithm in LLVM with the use of only one data structure.
The algorithm is depicted in a sequence diagram in Figure 6.7 and shows the indirection of the calls compared to the sequence diagram in Figure 6.5: This time, the LLVM code executes the algorithm, calling C++ code from time to time. There are also no loops around the calls between the LLVM and C++ code, therefore the number of calls is very low.
As in the previous implementation, the algorithm starts with selecting the random

centers. This process does not change, but afterwards the control of the algorithm goes back to the `compile time system`. The only information the `compile time system` requires from the `runtime system` are the first pointer and the last pointer of the data vector and of the center vector, respectively. Then, the `compile time system` is ready to execute the k-Means algorithm without any further interaction with the `runtime system`.
First, the minimum distances are computed between center and data tuples. The distance function was already implemented in LLVM code for the C++ driven approach and can be reused, only the loops around the `getDistance` function have to be implemented in LLVM.
The next step is to update the center tuples. The `resetCenter` function can be kept the same, as well as the `cumCenter` function. Again, the loops around the functions are now implemented in generated LLVM code. When computing the mean of a center, we have to keep track of the count. For that purpose we are using the additional field of each center tuple created on data materialization. Whenever we are adding data tuples to the corresponding center, we increment the count. When invoking the `updateCenter` function, this count is used to compute the mean using a division.

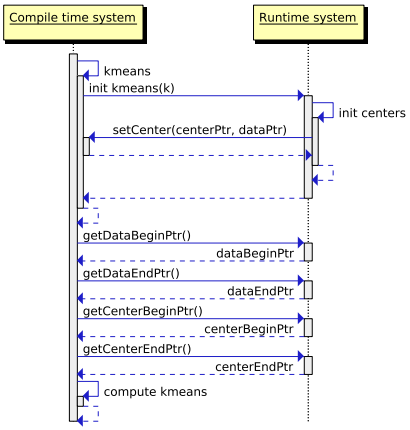


Figure 6.7: The k-Means Algorithm - LLVM driven.

Even though the differences between the two presented approaches do not seem to be huge since the overall programming concept remains the same, the difference in the code is significant. In particular using LLVM over high-level C++ constructs adds an overhead in the number of code lines. Therefore, the LLVM code in the second approach is harder to understand and to maintain. On the other hand, we are shrinking the number of calls between the C++ and LLVM system from $(n+2) \cdot k + n$ down to 4. This leads to performance improvements as we see in Chapter 7, where the LLVM

driven approach shows better results for all carried out experiments.

6.4.3 Initialization Strategy

In the so far discussed implementation details of the k-Means algorithm we assumed to use a simple random initialization strategy. Such a strategy can be implemented in the `runtime system`, since only C++ code is used for finding a random subset of length `k` of the data tuples, which can be implemented using the Fisher-Yates shuffle algorithm [FF48].
However, in Section 4.4 we figured out that one of the most popular variations of k-Means is k-Means++. The k-Means++ algorithm uses an extended initialization strategy by choosing data points as center with higher probability the further away they are from the already chosen set of center points. Often, this leads to improvements in both speed and quality of the clustering.
For implementation this means that we have to compute the distance from the chosen center points to the data points in the initialization phase too. This can be done very conveniently for the C++ version, as the `getDistance` function is implemented already. Therefore, the main initialization routine can be written in C++, only the `getDistance` function is used as generated function.
For the LLVM version, we still keep the center initialization in the `runtime system` to benefit of high level C++ program structures. To realize the k-Means++ algorithm, we have to add an explicitly generated `getDistance` function to the LLVM code: Even though the LLVM version computes the distance as well, there is no generated function anymore, since the distance generation is part of the generated program. Once we have added this function, k-Means++ can be implemented the same way as in the C++ driven approach.

6.5 Parallel k-Means

After presenting single-threaded implementations of the k-Means algorithm, in this section we show an approach to implement k-Means in HyPer in a parallel way to make use of all cores. Keeping in mind that HyPer is a high-performance database system written in C++, it makes already excessive use of multi-threaded programs, therefore it is only logical to add a parallel version of k-Means too. In the following we use the serial C++ version and modify it to allow parallelism. The C++ version is used over the LLVM version due to higher maintainability and readability.
The main bottleneck of the serial implementation is to compute the closest center for each data point: For each data point we have to calculate the distance to each center point which has to be performed in each iteration. This means $k \cdot n$ independent

distance computations, which can benefit a lot from parallelism and a computation in independent threads.
When executing a HyPer operator in parallel, the `consume` function is called for chunks of input tuples instead of the entire data set. Consider a system with four threads as shown in Figure 6.8: Each thread consumes one fourth of the data set and materializes the input tuples. Also in the `runtime system`, there are now vectors for each thread, storing the pointers to the input tuples.

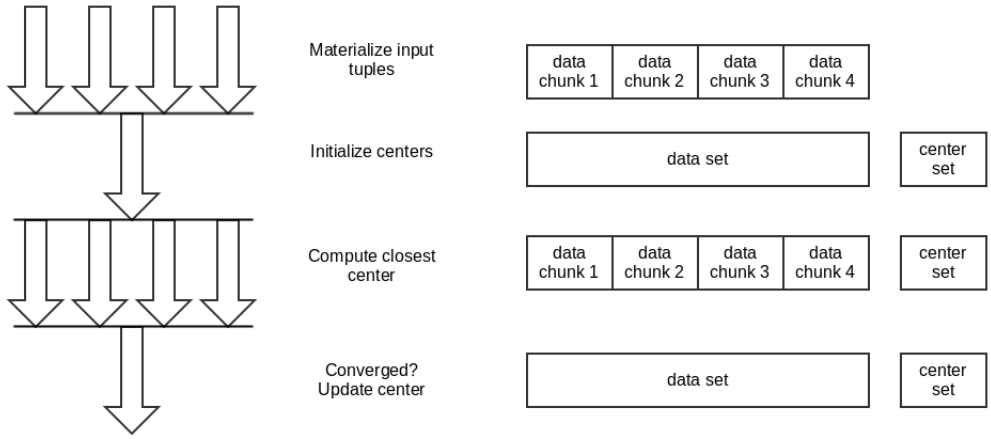


Figure 6.8: The k-Means Algorithm as Parallel Operator.

After consuming the data the k-Means algorithm continues with choosing the initial center set. For that, we have to break with the parallelism: For a random initialization of the clusters, center tuples are selected from the entire data set randomly. This is even more important for the k-Means++ initialization strategy computing centers regarding a distance function. While we could select random data points for the the center just from one data chunk, this is not possible for the k-Means++ initialization strategy anymore. Therefore, we create a global vector storing pointers to all materialization tuples. Now we can select the global center points using one of our initialization strategies and continue the parallel program sequence afterwards.
After this serial initialization we can continue the parallel execution of the algorithm and compute the distance from each data point to each center point. This expensive computation can be done independently in each data chunk on a separate thread and we benefit from the potentials of parallelism. Ideally, this leads to a running time advantage of one fourth of the serial execution, however, due to the overhead of process creation and management we will not be quite able to attain such high improvements,

as Chapter 7 shows.
After this parallel computation, we jump back to a serial execution mode of the program and check for each thread if the program has converged. Only if all threads have converged, we can terminate the algorithm and output the result of the clustering. Otherwise, we update the center points by the newly assigned data points. In the first version of this parallel implementation, this step is done in serial as already presented in Section 7.3. However, here is potential for parallelizing the program, which can be done in future implementations as discussed in Chapter 8.
In the evaluation chapter we compare the parallel k-Means algorithm with the serial implementations. Even though only one part of the algorithm is parallelized and with the overhead by introducing the parallelism in mind, we expect huge performance gains since the parallelized section contains many independent computations ideally for multi-threaded execution.

7 Evaluation

In this chapter we provide an extensive experimental evaluation of the HyPer k-Means operator and compare the different HyPer implementations to each other as well as to state-of-the-art clustering tools. All experiments have been performed on a workstation equipped with sixteen 2.93 GHz CPUs and 64 MB of main memory.

7.1 Data Sets

To provide an objective comparison we measure the execution time per iteration. Therefore the different technologies can be compared in a fair way and random initialization and internal improvments for faster convergence do not affect the outcome. For the experiments a real world data set is used and four synthetic data sets have been generated[1]. The synthetic data sets represent a high dimensional, a medium size, a medium sized high dimensional and a large size data set as shown in Table 7.1. All of them are generated on a uniform random distribution. The data sets are selected in a way that most real world use cases are covered.

Table 7.1: The Data Sets.

	Instances	Dimensions	Size (byte)	Size (Gigabyte)
3D Network	434,874	4	20,673,913	0.019
High Dimensional	50,000	50	10,000,000	0.009
Medium Size	15M	4	240,000,000	0.22
Medium Size HD	15M	50	3,000,000,000	2.79
Large Size	150M	10	6,000,000,000	5.59

- 3D Network: The first data set contains the 3D spatial road network data of a road network in North Jutland, Denmark [KYJ13]. It is a real world data set available at the UCI Machine Learning Repository [FA10] and consists of 434,874 data points in four dimensions: The id of the road segment, the latitude, the longitude and

[1]The generation script can be found in attachment ?

the altitude of the segment. The id is a large integer, the other dimensions are floating point numbers. The size of the data set is 19.72 MB, therefore it is a rather small data set.

- High Dimensional: The second data set is a synthetic data set. It consists of 50,000 data points in 50 dimensions. All dimensions are floating point numbers. The size of the data set is 9.50 MB, hence an even smaller but high dimensional data set.
- Medium Size: This data set consists of 15 million data points in four dimensions. It is also a synthetic data set, all four dimensions containing floating point numbers. The size of the data set is 228.88 MB.
- Medium Size High Dimensional (HD): The medium size high dimensional data set is generated the same way as the medium size data set. Instead of four dimensions, this time 50 dimension have been generated. This leads to a growth from 0.23 GB to 2.79 GB.
- Large Size: The large data set consists of 150 million data points in ten dimension. It is also synthetically generated by a random uniform distribution of floating point numbers. The size of the data set is 5.59 GB.

7.2 Competitor Systems

As already described in chapter 5, there exist many tools for data mining. In this section we look at a subset of the tools implementing the k-Means clustering algorithm we are using to compare with our own k-Means HyPer operator. To make the results comparable in a fair manner, we use only tools that give enough information about the clustering process, e.g. the number of iterations, total cost and most importantly, the applied algorithm. Not all tools are implementing the Lloyd algorithm, e.g. the default version of R is the Hartigan-Wong algorithm, an improvement over the standard Lloyd algorithm.
Since k-Means is a non-deterministic algorithm, the number of iterations is the most important criterion to make the results comparable. Running time of k-Means depends almost entirely on the number of iterations the algorithm has to make, which is largely influenced by the non-deterministic initialization. Therefore the running time should be relative to the number of iterations.
Unfortunately, neither ELKI nor Scipy's k-Means implementation provide the number of iterations as output. Therefore, a fair comparison is not possible. Fortunately, Weka, R and Julia provide good configuration possibilities as well as a result set that contains

information about the number of iterations. While Weka is written in Java, R and Julia are both written in their own high level language. Even though, critical code parts are implemented in C, C++ and Fortran for better performance results. Without the overhead of an entire database system behind, it will be interesting to compare HyPer with Weka, R and Julia.
The usability and user experience among R and Julia is very similar: Both can be run as an interactive program or as script using a functional language. Various plotting functionalities are provided, and both tools can import our data in the csv format. HyPer can be run as interactive console program or as script, too, and provides efficient csv loading [Müh+13]. Weka can be run from the command line, too, directly in Java programs or with its graphical user interface. However, internally Weka uses its own arff format for the algorithms. Using the Weka GUI to convert the original data (in our case in csv format) into the arff format feels fairly natural, as it happens in a preprocessing step when loading the data. However, when using the command line or Java programs to read a csv file, the file has to be preprocessed to the arff format first by an additional command. This takes time and is not as convenient as with the other tools.
Regarding the performance, Weka shows very disappointing results compared to Julia, R and HyPer, as depicted in Table 7.2. The table shows the median, the 90th and the 95th percentile of the running time per iteration after 100 runs for $k = 3, 10$ and 20. On the network data set, Weka is slower by a factor of 7 compared to the Julia implementation, which is the slowest on that data set. On the high dimensional data set, the HyPer C++ implementation shows the worst results, however, Weka is slower by a factor of around 6. As last test, the medium size data set was run against Weka, and it is slower by a factor of 10 for $k = 3$, 8 for $k = 10$ and 7 for $k = 20$ compared to Julia, the slowest one regarding this data set. Because of this tremendous differences in running time, we decided not to include Weka in the following performance section and only compare the performance to R and Julia.

Table 7.2: Weka Results - Time per Iteration.

	Network 3D			High Dimensional			Medium Size		
k	3	10	20	3	10	20	3	10	20
50	1.39	1.54	1.93	1.08	1.39	1.90	54.01	58.25	71.11
90	1.48	1.60	2.02	1.16	1.44	1.98	59.16	62.54	87.16
95	1.49	1.61	2.06	1.19	1.46	2.01	59.79	64.31	88.54

7.3 Serial Implementation

In this section we compare our two serial HyPer k-Means implementations with each other: The C++ driven and the LLVM driven approach, as described in Section 6.4. We are comparing the compilation and the execution time. The compilation time is the time for generating LLVM code, while the execution time is the time for executing the k-Means algorithm after the LLVM code was generated. Since the compilation time can be seen as a one-time setup time, we expect the execution time to be much larger.
For each data set, the two HyPer implementations are tested for $k = 3, 10$ and 20. For each k the algorithm is executed 100 times with a maximum number of iterations of 10. As result, the median time per iterations in seconds is stated. For better comparability the compilation time is also relative to the number of iterations, even though it stays constant as the number of iterations grows.
Table 7.3 shows the result of the two serial implementations for the network data set. As we observe the compilation time is independent of the cluster number k. Also the compilation time does not differ among the C++ implementation and the LLVM version. In contrast, the execution time increases as the cluster number k increases. This is true for both implementations.

Table 7.3: 3D Network - Time per Iteration.

k	HyPer C++ compilation[s]	execution[s]	HyPer LLVM compilation[s]	execution[s]
3	0.0050	0.0680	0.0046	0.0171
10	0.0044	0.0947	0.0046	0.0502
20	0.0045	0.1391	0.0046	0.0901

For better visualization, Figure 7.1 depicts the same result as stacked bar charts. Again we observe that the compilation time is low and constant, while the execution time differs among the implementations. For $k = 3$, the execution time of the C++ version is almost four times the execution time of the LLVM version. For $k = 10$ it is two times the execution time and for $k = 20$ it is 1.5 times, respectively. On the other hand that means that for a larger cluster number k, the LLVM execution time grows faster than the execution time of the C++ version. Actually, from $k = 3$ to $k = 10$, the LLVM version grows by a factor of 2.9, while the C++ version grows by 1.4. From $k = 3$ to $k = 20$ the growth is even more significant, from factor 2 for the C++ version to 5.3 for the LLVM version.

This gives us the first interesting results. Although the C++ version is implementing k-Means using LLVM only for generated functions, the compilation time does not differ

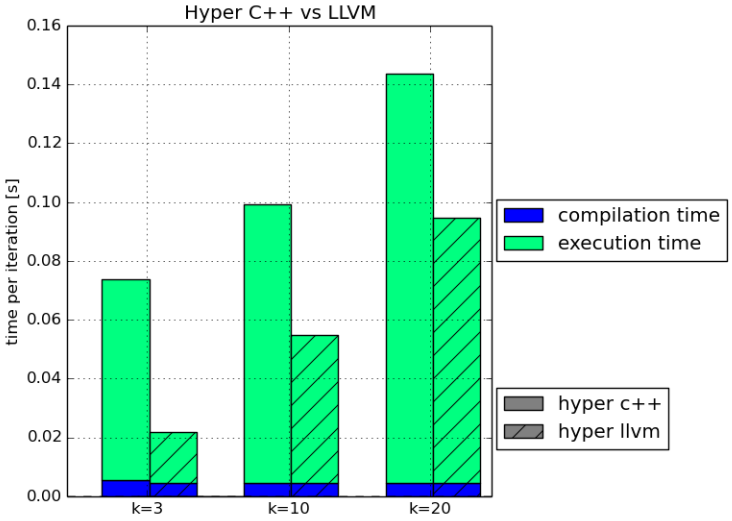


Figure 7.1: 3D Network - Time per Iteration.

to the LLVM version, where not only the functions but also the entire algorithm is written in LLVM. Nevertheless, the functions for computing the distance and updating the centers are generated in LLVM for both implementations which is an explanation for the similar compilation times: In the C++ version this code is generated as functions callable from the `runtime system`, while the LLVM version embeds the code directly into the LLVM program structure. Therefore, the difference between the two seems to be insignificant.
Regarding the execution time, the LLVM version is much faster than the C++ version. One reason is that the C++ implementation has many function calls between the `compile time` and the `runtime system`. For the LLVM system these calls are not necessary, therefore the data can remain in the CPU registers. The second advantage is that the entire algorithm is compiled in LLVM code resulting in very efficient code, optimized on a lower level than even possible with C++ code.

Table 7.4: High Dimensional - Time per Iteration.

k	HyPer C++ compilation[s]	execution[s]	HyPer LLVM compilation[s]	execution[s]
3	0.1278	0.0522	0.0933	0.0327
10	0.1287	0.1171	0.0933	0.0706
20	0.1299	0.2070	0.0933	0.1252

Table 7.4 shows the result of the same experiment with the high dimensional data set. In contrary to the network data set the compilation time is slightly different between the C++ and the LLVM implementation. Furthermore for both versions the compilation time is larger than the execution time for $k = 3$ and almost equal for $k = 10$. Only for $k = 20$ the execution time is larger than the compilation time, as depicted in Figure 7.2.

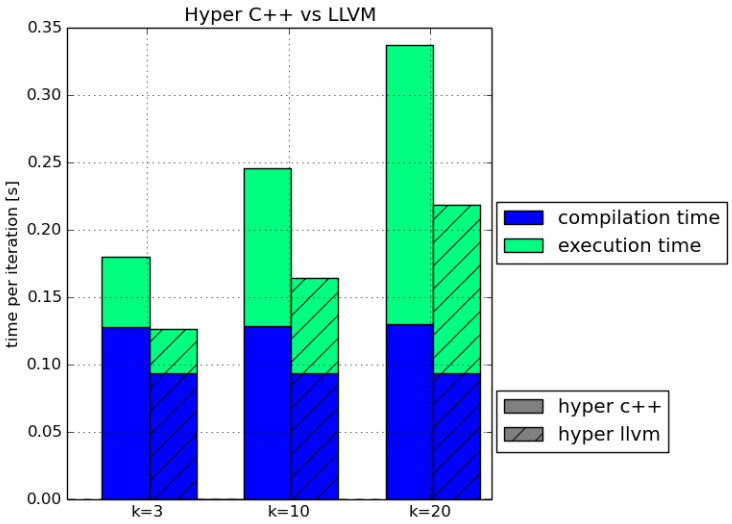


Figure 7.2: High Dimensional - Time per Iteration.

Again, the compilation time stays constant for different values of k. The increase of compilation time is induced by the high dimensionality: For each dimension additional code has to be generated. Therefore a data set with four dimension is faster to compile than a data set with 50 dimensions. The difference in the running time is similar to the network data set even though the increase in performance between the two systems is not as significant. For $k = 3$, LLVM is faster by a factor 1.6 and for $k = 10$ and $k = 20$ by facor 1.7. This time the increase in execution time is strongly correlated. From $k = 3$ to $k = 10$, the LLVM version and the C++ version grow both by a factor of 2.2. From $k = 3$ to $k = 10$, the LLVM version grows by a factor of 4, while the C++ version grows by 3.8.
Interestingly, the execution time for the network data set is lower than for the high dimensional data set, even though the network set has a size of 0.019 GB, while the high dimensional data set has only 0.009 GB. Even if we omit the compilation time which stays constant for a growing number of iterations and only look at the execution time the network data set is only for the C++ implemenation and $k = 3$ slower than the high dimensional data set. Apparently, our implementation performs much better for low dimensional data sets.

Table 7.5 and Table 7.6 show the same experiment for the medium size and the medium size high dimensional data set. Both consist of 15 million instances, the first of four dimensions and the second of 50 dimensions. The compilation time of the medium size data set is similar to the network data set since both consist of four dimensions. The same is true for the medium size high dimensional data set and the previously used high dimensional data set: Both consist of 50 dimensions and the compilation time is quite similar. This shows us that the compilation time is independent on the data size and only affected by the number of dimensions. However, as the number of instances is growing, the compilation time is not a significant factor for the overall running time anymore, as Figure 7.3 shows: The compilation time is not even visible plotting the data as stacked bar charts. Again, the LLVM version shows a much better performance compared to the C++ version. For the medium size data set the overall running time is faster by a factor of 2.5, 1.6 and 1.4 for $k = 3, 10$ and 20, and for the medium size high dimensional data set by a factor of 1.6, 1.5 and 1.5, respectively.

Table 7.5: Medium Size - Time per Iteration.

	HyPer C++		HyPer LLVM	
k	compilation[s]	execution[s]	compilation[s]	execution[s]
3	0.0041	2.3779	0.0047	0.9191
10	0.0041	3.2856	0.0047	2.1001
20	0.0041	4.6765	0.0047	3.4518

Table 7.6: Medium Size High Dimensional - Time per Iteration.

	HyPer C++		HyPer LLVM	
k	compilation[s]	execution[s]	compilation[s]	execution[s]
3	0.1127	16.2787	0.0935	10.3302
10	0.1127	34.0051	0.0935	22.0209
20	0.1126	59.2904	0.0935	38.7046

Obviously the execution time for the high dimensional data set is much larger compared to the low dimensional data set: This time the high dimensional data set has the same number of instances but 50 dimensions. For the C++ implementation the execution time is increased by a factor of 6.8 for $k = 3$, 10.3 for $k = 10$ and 12.6 for $k = 20$. For the LLVM implementation it is 11.2, 10.5, 11.2, respectively. Since the high dimensional data (2.79 GB) set is larger than the medium size data set (0.22 GB) by a factor of 12.7, the values are strongly correlated to the growth of the data set.

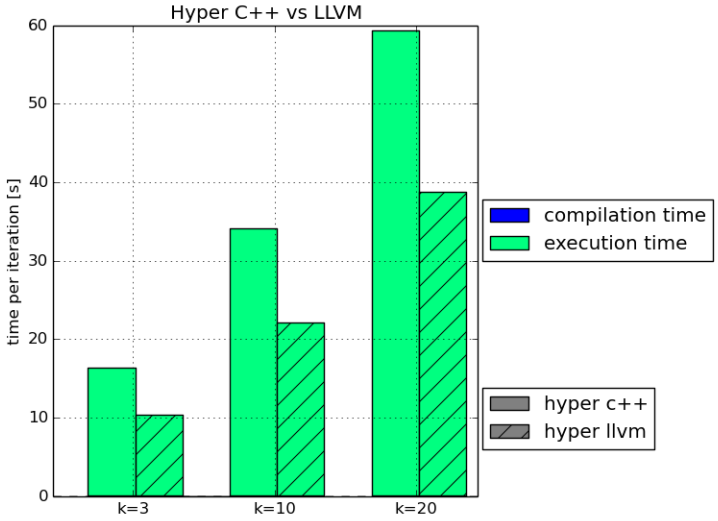


Figure 7.3: Medium Size High Dimensional - Time per Iteration.

Finally, Table 7.7 shows the same experiment for the large size data set. The data set consists of ten dimensions and therefore the compilation time increases compared to the data sets with four dimensions by a factor of around two. However, since the data set has a size of 5.59 GB we can omit the compilation time since it does not affect the overall running time. Comparing the two versions regarding the overall running time, we observe that the LLVM version provides again much better results and is faster by a factor of 2.0, 1.5 and 1.4 for $k = 3, 10$ and 20 compared to the C++ implementation. This results in a difference of around 20 seconds per iteration between the two implementations.

Table 7.7: Large Size - Time per Iteration.

k	HyPer C++ compilation[s]	execution[s]	HyPer LLVM compilation[s]	execution[s]
3	0.0088	37.2804	0.0097	18.3734
10	0.0088	62.0034	0.0097	40.4783
20	0.0191	92.5868	0.0097	67.2364

In conclusion, we observed that the compilation time is not affected by the number of instances but the number of dimensions, where high dimensional data sets slow down the compilation phase. Interestingly, there is no significant difference regarding

the compilation time between the two HyPer implementations.
Usually, the execution time outnumbers the compilation time by several factors. The only exception is the small, high dimensional data set as shown in the experiment. Here, the compilation time is slower than the execution time for small k. Furthermore the execution time grows by number of instances and is much faster for the LLVM implementation. With equality regarding the compilation and a performance increase regarding the execution time the LLVM version was for all experiments the best choice for the implementation of the k-Means algorithm.

7.4 Performance Comparison with Competitor Systems

In this section we compare our serial HyPer implementations with R and Julia's implementation of the k-Means algorithm. As algorithm, the Lloyd algorithm with a maximum iteration number of ten is chosen. Apart from the large data set, the algorithm is executed 100 times for $k = 3, 10$ and 20, respectively. For the large data set, the algorithm is executed ten times. The result is then presented as the execution time per iteration. For each algorithm the median, the 90th percentile and the 95th percentile are given for every k.

Table 7.8: 3D Network - Time per Iteration.

	Julia			R			HyPer C++			HyPer LLVM		
k	3	10	20	3	10	20	3	10	20	3	10	20
50	0.21	0.22	0.29	0.03	0.04	0.08	0.08	0.10	0.14	0.02	0.06	0.10
50	0.27	0.30	0.32	0.06	0.06	0.10	0.09	0.12	0.20	0.03	0.06	0.10
50	0.31	0.35	0.35	0.08	0.07	0.11	0.10	0.13	0.22	0.03	0.06	0.10

First, we test against the real-world network data set. The result is presented in Table 7.8. As we already know, the HyPer LLVM implementation outnumbers the C++ version. For the median, the LLVM implementation is 3.4, 1.8 and 1.5 times faster for $k = 3, 10$ and 20. Julia is the slowest, our LLVM implementation is 9.5, 4.0 and 3.1 times faster, respectively. Only the R implementation can compete with our HyPer LLVM operator and is with factors of 0.7, 0.8 and 0.8 for $k = 3, 10$ and 20 even a bit faster. However, all tested programs differ in a few hundred milliseconds therefore the differences are not very significant.
Figure 7.4 shows the results as a boxplot. We observe that the C++ version, the LLVM version and R are closely together, while Julia performs poorly. Also regarding the

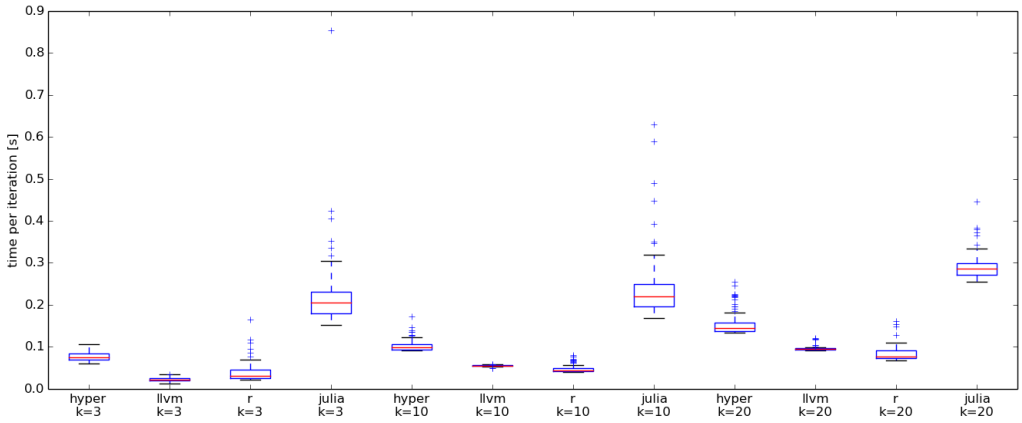


Figure 7.4: 3D Network - Time per Iteration.

variance of the 100 runs, Julia shows the strongest variations and many outliers.

Table 7.9: High Dimensional - Time per Iteration.

	Julia			R			HyPer C++			HyPer LLVM		
k	3	10	20	3	10	20	3	10	20	3	10	20
50	0.04	0.05	0.07	0.03	0.05	0.08	0.18	0.24	0.35	0.13	0.16	0.22
90	0.04	0.05	0.07	0.04	0.06	0.10	0.21	0.34	0.40	0.13	0.16	0.22
95	0.04	0.05	0.07	0.05	0.07	0.11	0.22	0.35	0.44	0.13	0.17	0.22

Running the same experiment on the high dimensional data set we see a different result, as depicted in Table 7.9. As already shown, the HyPer C++ and the LLVM version are slower compared to the network data set even though the data set is smaller by size. R takes almost the same time for both data sets. Interestingly, Julia's k-Means algorithm is now as fast as the R implementation, for $k = 10$ and 20 it is even faster. This is also depicted by the boxplot in Figure 7.5: The HyPer C++ version shows the highest variance with many outliers, while Julia's variance is very small this time.
This result gives us interesting knowledge about the different implementations. Both HyPer operators perform poorly when dimensions increase. An increase in dimensions and a decrease in instances does not affect R significantly. On the other hand, Julia shows much better results as dimensions increases and seems to be optimized for high dimensions: It is 5.4, 4.4 and 4.2 times faster for $k = 3, 10$ and 20 on the high dimensional set compared to the network data set, and the variance is much lower. A

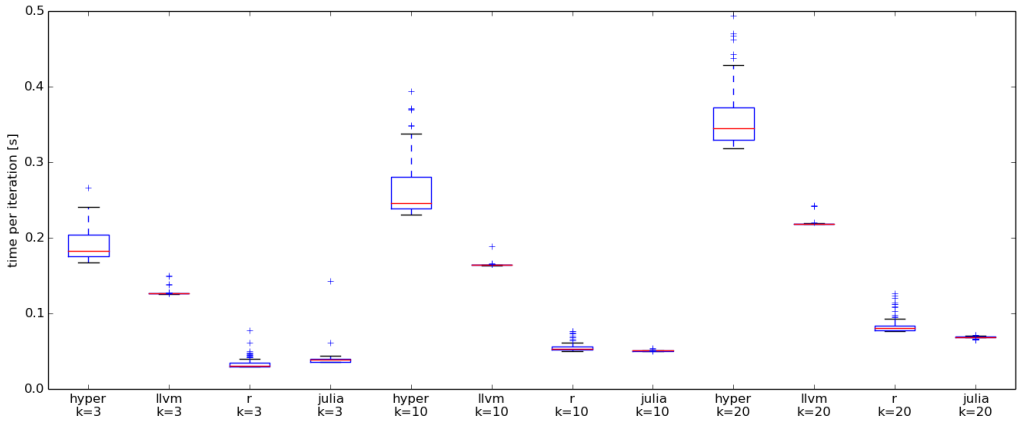


Figure 7.5: High Dimensional - Time per Iteration.

conversation with Prof. Timothy E. Holy, Ph.D. substantiated this hypothesis[2].

Table 7.10: Medium Size - Time per Iteration.

	Julia			R			HyPer C++			HyPer LLVM		
k	3	10	20	3	10	20	3	10	20	3	10	20
50	5.42	7.61	10.62	0.77	1.61	2.50	2.38	3.29	4.68	0.92	2.10	3.46
90	5.43	7.65	10.71	0.79	1.63	2.55	2.57	3.44	4.77	0.94	2.14	3.51
95	5.44	7.70	10.91	0.80	1.63	2.56	2.64	3.45	4.84	0.94	2.14	3.52

We perform the same experiment on the medium size and the medium size high dimensional data set. Table 7.10 shows the results of the medium size data set. Again, the LLVM version is faster than the C++ version by a factor of 2.5, 1.6 and 1.4 for $k = 3, 10$ and 20. Since the dataset has only four dimensions Julia performs poorly and is by a factor of 5.9, 3.6 and 3.1 slower compared to the LLVM version. As for the network data set R demonstrates the best performance and is faster than LLVM by a factor of 0.8, 0.8 and 0.7 for $k = 3, 10$ and 20. Again we present the result as a boxplot in Figure 7.6. The variance is small for all used tools, only Julia has a few outliers for $k = 20$.

For comparison Table 7.11 and Figure 7.7 depict the result for the medium size high dimensional data set. As before, our own implementation does not perform ideally on high dimensions. For $k = 3$, the LLVM version is slower than R by a factor of 0.9 but faster than Julia for a factor of 1.1. However, as k grows, R and Julia show slightly

[2]Julia User Group: `http://bit.ly/1anXGMF`, 25.01.2015.

Table 7.11: Medium Size HD - Time per Iteration in seconds.

		Julia			R			HyPer C++			HyPer LLVM		
	k	3	10	20	3	10	20	3	10	20	3	10	20
percentile	50	11.46	15.69	21.67	9.43	15.48	23.77	16.39	34.12	59.40	10.42	22.11	38.80
	90	11.51	15.81	21.69	9.50	15.50	23.79	16.77	36.11	59.53	10.44	22.12	38.84
	95	11.65	15.83	21.70	9.64	15.58	23.79	17.29	42.15	59.57	10.45	22.13	38.84

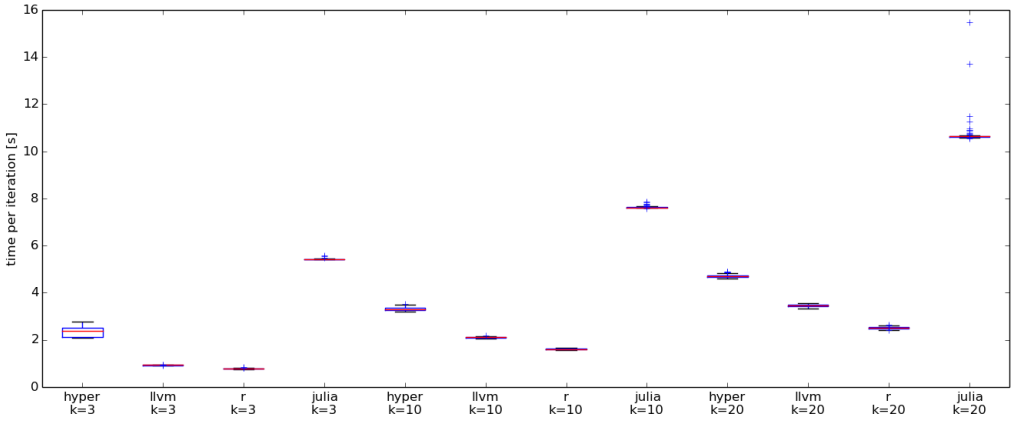


Figure 7.6: Medium Size - Time per Iteration.

better results. Both show a speed up by a factor of 0.7 for $k = 10$, and 0.6 for $k = 20$, compared to the LLVM version.
Obviously the high dimensional data set takes more time per iteration having the same number of instances with a lot more dimensions. However, Julia shows a slow down by a factor of only 2 for all k's, although the dimensions increase from 4 to 50. R shows a slow down by a factor of 12 for $k = 3$ and 10 for $k = 10$ and 20, the C++ version by a factor of 7, 10, 13 for $k = 3, 10$ and 20 and the LLVM version by a factor of 11 for all k's. Julia performs again the best for high dimensional data. R and HyPer's LLVM operator behave very similar this time. An explanation is that the compilation time is not a limiting factor anymore for the LLVM implementation: For the high dimensional data set with 50,000 instances the compilation time was higher than the execution time. This time the data set consists of 15 millions tuples therefore the poor compilation time does not affect the overall performance anymore.
As final test we perform the same experiment on the large data set containing 150 million instances, as shown in Table 7.12. This time the HyPer LLVM implementation

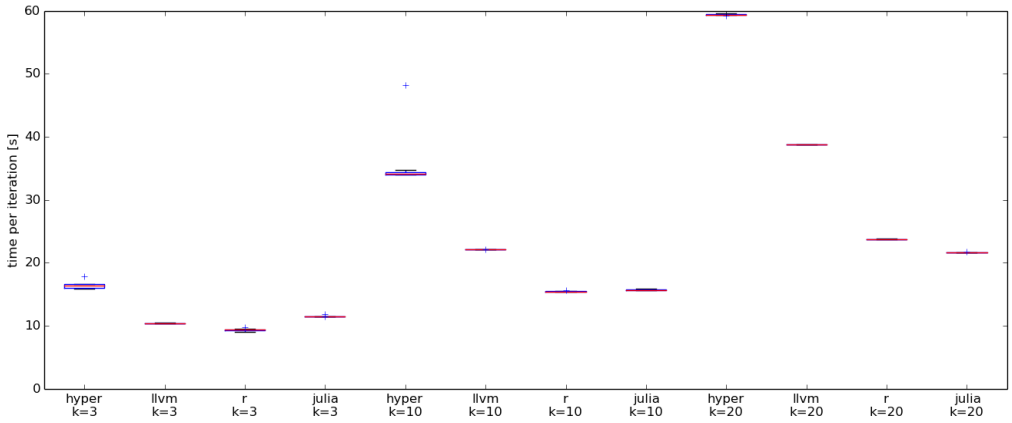


Figure 7.7: Medium Size HD - Time per Iteration.

outnumbers all the other tools, the C++ version by a factor of 2.0, 1.5 and 1.4 for $k = 3, 10$ and 20, Julia by a factor of 3.6, 2.2, 2.0, respectively, and even R by a factor of 1.6, 1.2 and 1.1, respectively. For growing k, the R implementation comes very close to the HyPer LLVM version. This result is also depicted in the boxplot in Figure 7.8.
In conclusion, the experiments show that the HyPer k-Means operator can compete with state-of-the-art tools for clustering. Particularly the LLVM implementation demonstrates a good performance and shows similar results to the R implementation of the k-Means algorithm. However, for high dimensional data sets both HyPer operators do not perform ideally and are outnumbered by Julia's implementation optimized for high dimensions. Here is potential for future optimization as we demonstrate in Section 7.5 with a parallel implementation. Yet, if we take into account that the k-Means algorithm runs on top of an entire database with all its overhead, these results are supporting further development and the implementation of other data mining algorithms as HyPer operators.

Table 7.12: Large Size - Time per Iteration.

	Julia			R			HyPer C++			HyPer LLVM		
k	3	10	20	3	10	20	3	10	20	3	10	20
50	65.62	90.37	132.39	29.47	48.21	72.40	37.29	62.01	92.60	18.38	40.49	67.25
90	65.72	92.01	135.87	30.80	54.73	77.27	37.50	62.41	92.86	18.44	40.72	67.57
95	65.78	92.27	137.21	33.08	55.02	77.97	37.52	62.72	92.95	18.46	40.72	67.63

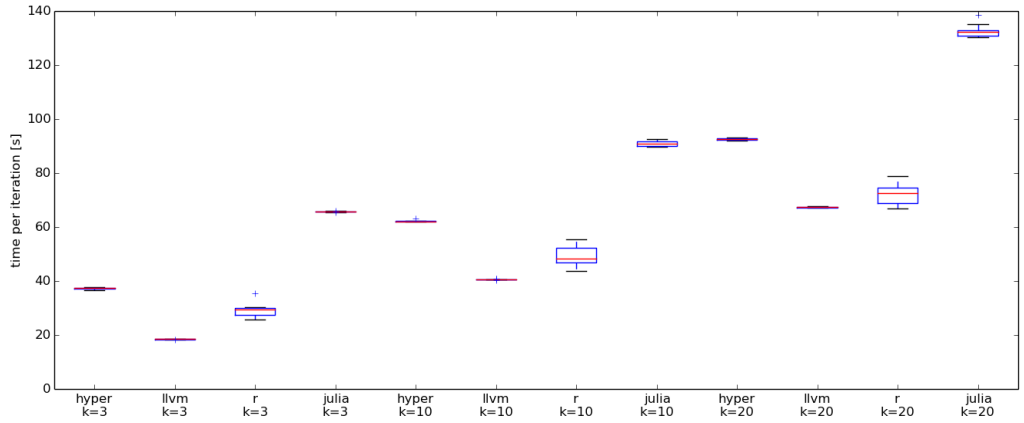


Figure 7.8: Large Size - Time per Iteration.

7.5 Parallel Implementation

So far we only looked at serial executions of the HyPer k-Means operator: In Section 7.3 we figured out that the LLVM implementation outperforms the C++ version on all used data sets. In Section 7.4 we compared the performance against Julia and R and showed that HyPer is able to compete with state-of-the-art technologies for data mining. However, as data sets grow serial execution takes more and more time, making real-time data mining almost impossible. Furthermore, we do not exploit the possibilities of modern database hardware and the advantages of multi-threaded execution of algorithms. To address these challanges we presented our parallel implementation of the HyPer k-Means operator in Section 6.5. In this section we compare this parallel implementation with the LLVM version and the R implementation. Both tools demonstrate the best results on the medium size, the medium size high dimensional and the large data set.

Table 7.13: Medium Size - Time per Iteration.

	R			HyPer LLVM			HyPer Parallel		
k	3	10	20	3	10	20	3	10	20
50	0.77	1.61	2.50	0.92	2.10	3.46	1.35	1.52	1.75
90	0.79	1.63	2.55	0.94	2.14	3.51	1.38	1.53	1.84
95	0.80	1.63	2.56	0.94	2.14	3.52	1.40	1.53	1.85

Table 7.13 shows the result as the time per iteration for $k = 3, 10$ and 20. As before,

the median, the 90th and 95th percentile are presented. For $k = 3$, we see that both the R and the LLVM version are faster than the parallel version. However, as k grows, the parallel version outperforms R and the LLVM version. This circumstance is also shown in the bar chart in Figure 7.9.
The reason is that for the parallel execution the running time is almost independent of k, therefore the parallel version outperforms R and LLVM. For R, the time per iteration grows by a factor of 2.1 from $k = 3$ to $k = 10$, and by a factor of 1.6 from $k = 10$ to $k = 20$. For LLVM we observe an increase by a factor of 2.3 and 1.6, respectively. In contrast, the parallel implementation grows only by factor of 1.1 and 1.2, respectively. In other words, R and the LLVM version grow by seconds as k grows, while the parallel version grows insignificantly by around 200 milliseconds.

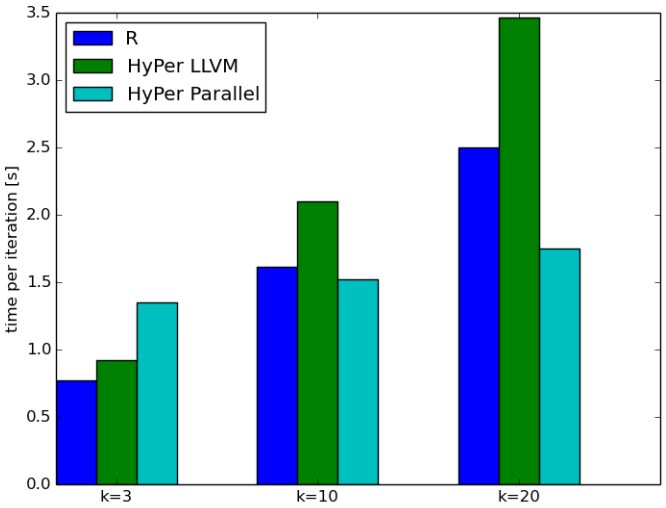


Figure 7.9: Medium Size - Time per Iteration.

Table 7.14 shows the same experiment on the medium high dimensional data set. For this data set the parallel version outperforms the other two implementations for all k's, even for $k = 3$. Figure 7.10 depicts that the parallel version is again almost independent of k, in contrast to R and LLVM. R grows by a factor of 1.6 from $k = 3$ to $k = 10$, and 1.5 from $k = 10$ to $k = 20$, for LLVM we observe an increase by a factor of 2.1 and 1.8, respectively and for the parallel version a factor of 1.4 and 1.4, respectively. Even though the parallel version is not as independent by the cluster number k as for the low dimensional data set, the increase in time is much lower. Additionaly, a performance gain is achieved by the ability to handle high dimensional data better: From four dimensions to 50 dimensions, the parallel execution time is only affected by a factor of 4.3, 5.3 and 6.5 for $k = 3, 10$ and 20. R and LLVM are affected by a much

higher factor: 12.3, 9.6 and 9.5 for R, 11.3, 10.5 and 11.2 for the LLVM implementation. Only the for high dimensions optimized Julia shows with factors of 2.1, 2.1 and 2.0 better results.

Table 7.14: Medium Size HD - Time per Iteration.

	R			HyPer LLVM			HyPer Parallel		
k	3	10	20	3	10	20	3	10	20
50	9.43	15.48	23.77	10.42	22.11	38.80	5.84	8.04	11.30
90	9.50	15.50	23.79	10.44	22.12	38.84	5.88	8.26	11.35
95	9.64	15.58	23.79	10.45	22.13	38.84	5.88	8.29	11.37

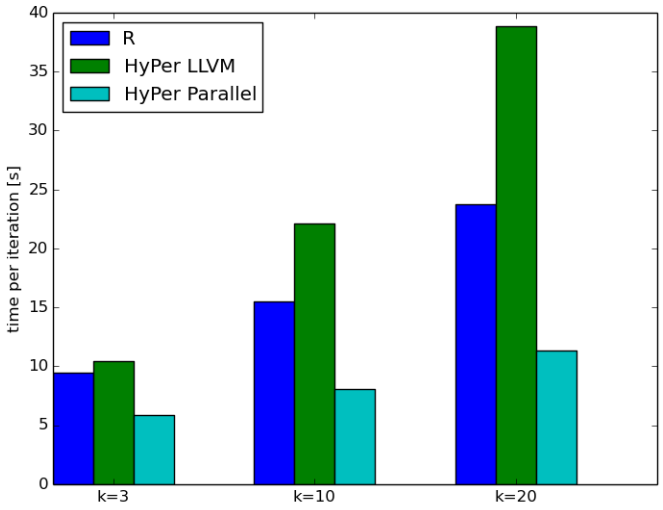


Figure 7.10: Medium Size HD - Time per Iteration.

Finally, Table 7.15 shows the result of the same experiment on the large size data set. Again, the parallel version is the fastest. For $k = 3$, the parallel version is faster than R by a factor of 1.8 and for LLVM by a factor of 1.1. For $k = 10$ the factor is even larger, 2.4 for R and 2.0 for LLVM, and 3.0 and 2.8 for $k = 20$, respectively. Since k affects the performance of the parallel version only slightly, the factor is increasing as k increases.

To conclude, the parallel version is much better than the LLVM and R implementation. Except for the medium size data set and $k = 3$ it demonstrates always better results. However, since we are using 16 cores even a higher speed up would be possible.

As explanation, we have to take into account the overhead of the parallel process creation. Furthermore, we are using the slower C++ version of the HyPer k-Means operator as foundation for our parallel implementation. Even though parts of the algorithm are parallelized, we still have all the downsides of many function calls between the `runtime` and the `compile time system`. Additionaly, we are using high-level C++ constructs over the more efficient LLVM code generation. And finally, we parallize only the computation of the distances, not the cluster update. Therefore, we can even expect better results when we enhance the addressed shortcomings in future versions.

Table 7.15: Large Size - Time per Iteration.

	R			HyPer LLVM			HyPer Parallel		
k	3	10	20	3	10	20	3	10	20
50	29.47	48.21	72.40	18.38	40.49	67.25	16.71	19.79	24.03
90	30.80	54.73	77.27	18.44	40.72	67.57	16.94	20.13	24.33
95	33.08	55.02	77.97	18.46	40.72	67.63	17.01	20.17	24.36

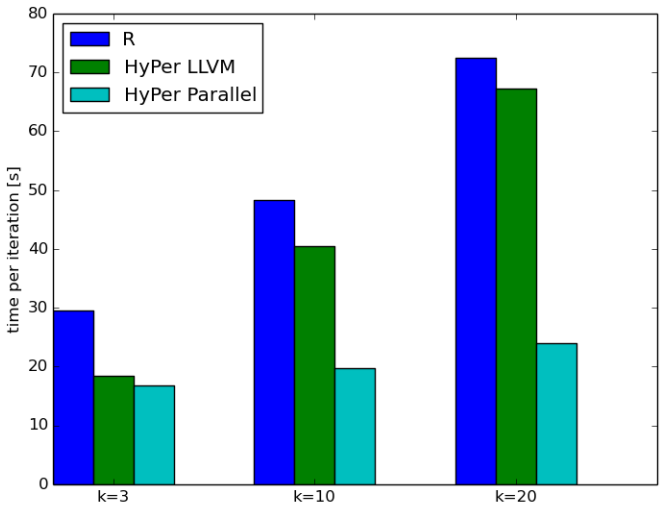


Figure 7.11: Large Size - Time per Iteration.

8 Conclusions and Future Work

In this work, we have presented a proof of concept implementation of the k-Means clustering algorithm as HyPer operator. This chapter provides conclusions of the presented work and addresses open challenges.

8.1 Conclusions

With main memory databases we are able to relax traditional database constraints and benefit from unprecedented performance gains. The main memory system HyPer can execute OLTP and OLAP queries on one systems without the need of an additional data warehouse. Such a system provides the opportunities to execute data mining algorithms directly on the database to benefit from data locality and high performance. After discussing existing systems for data mining, we presented several approaches to implement the k-Means algorithm as a HyPer operator: A C++ driven approach, a LLVM driven approach and a parallel implementation.
As result we make the observation that the LLVM version is faster than the C++ version on all data tests, and that our operator is able to compete with existing data mining solutions. In fact, the LLVM approach and the R implementation are usually providing the best results. Only for high dimensional data sets, the HyPer operator presents poor results and is outnumbered by Julia and R's implementation. However, the HyPer operator also provides the opportunity for parallel execution. As data sets grow, the parallel implementation is faster than all the other tools, also on high dimensional data sets.
In conclusion, we showed that the k-Means algorithm implemented as HyPer operator can compete and even outperform existing solutions, therefore data mining using the HyPer main memory database is strongly practicable.

8.2 Future Work

Many open challenges remain for data mining in HyPer. In this section we discuss future work and begin with general improvements for the k-Means algorithm.
Our k-Means implementation can take up to three input parameters: The cluster

number k, the maximum number of iterations and a verbose option to output statistics about the algorithm, such as the center coordinates and the cluster compactness. For the future we want to enhance the capabilities of k-Means and make our algorithm more flexible.
One possibility is to allow the user to choose a distance function. Our k-Means implements the euclidean distance, however, for some problems other distances are more appropriate. Therefore we should offer to choose among the manhattan, euclidean, minkowski and chebyshev distance as well as the cosine similarity.
Applying normalization before data clustering can improve the final result. This can be either implemented as input parameter for the k-Means algorithm or even better as standalone operator since other data mining algorithms can benefit from a normalization too.
For big data sets the execution time of the k-Means algorithm can be quite high. The reasons is that the iterations has to take many iterations until it converges. An enhancement is an earlier termination: Usually, k-Means terminates if the clusters do not change anymore. However, for large data sets this could be a problem if a few data points are moving from cluster to cluster while the majority of data points remains constant. Nevertheless, an additional iteration of the algorithm is initiated. Therefore an additional input parameter can specify a tolerable change of data points. If the number of changing data points is underneath this threshold, the algorithm terminates even though there is a chance to find a slightly better clustering.
Heterogeneous data is another problem for our algorithm. No all data sets consist only of numerical data. Often it is a mixture between numerical, binary and categorical attributes. An improvement is to first convert the binary and categorical attributes to numerical attributes, and then improve the distance and cost function to keep the different dimensions comparable, as proposed in k-Means Mixed [AD07].
Instead of implementing a new cost function the existing distance function can be improved by assigning a weight to each dimension. This is not only useful for mixed data sets but also for numerical attributes. Apart from dimensions instance can be also weighted differently since some are more significant than others.
K-Means is usually implemented as the Lloyd [Llo82] algorithm, resulting in reasonable results. However, the execution time can be improved implementing the Hartigan-Wong algorithm [HW79]. In contrast to the Lloyd algorithm it updates the centroids at any time a point is moved and makes time-saving choices for finding the closest cluster.
In our implementation the output of k-Means is a table with an additional column specifying the cluster number. Since this table is returned by the consume function, it can be used in further SQL statements. The statistical information however, such as the final center coordinates or the number of iterations is only printed to the console. Therefore this information is lost and cannot be used in further SQL statements. An

improvement is to generate additional tables storing the statistical information of a clustering. In this tables more information can be stored in an accessible way, e.g. the final center coordinates, the distance from each data point to its closest center and all statistical information. Up to now HyPer is only able to push tuples to the next consume operator and cannot generate additional tables. Since this information is very valid to analyze the k-Means result an extension of the existing operator model is suggested. Almost all data mining algorithms produce several additional intermediate and final results, thus this is an important feature for computational data mining in databases.

HyPer is a database system that makes already extensive use of index structures. An option to improve the k-Means algorithm is by using nearest neighbor data structures when clustering high dimensional data. This could boost the performance even more.

As shown in Chapter 7 the LLVM implementation is much faster than the C++ version. Therefore, the center initialization using the k-Means++ method could benefit from a pure LLVM implementation instead of a C++ implementation with slow `getDistance` calls for the center selection process.

Not only the k-Means++ implementation could benefit from an LLVM implementation, but also the parallel execution. As we have shown, for small k and medium size data sets the parallel execution is still slower than the LLVM implementation. One reason is that the C++ version was used as basis of the parallel version leading to many calls from the `runtime` to the `compile time system`. A parallel implementation in LLVM code could tremendously boost the performance of the algorithm.

Also the parallelization itself can be improved: Only the process of computing the distances is parallelized so far. While this is already a big performance improvement the algorithm would even benefit more from a parallel execution of the initialization process and of the updating of the cluster centers.

Obviously more data mining algorithms have to be implemented in HyPer to enable a full data mining experience. Therefore, the existing k-Means algorithm can be used as a building block. A very similar partitional clustering algorithm is the k-Medoids algorithm [KR90]. In contrary to k-Means it chooses data points as centers instead of the mean of the mass of all assigned data points. This makes the algorithm more robust for handling outliers. As implementation, the existing k-Means algorithm can be reused, only the way of computing center points after data point assignment has to change according to the requirements of k-Medoids.

So far we assign each object to one distinct cluster. However, objects may belong to several clusters. A next step could be to enhance the k-Means algorithm with a focus on fuzzy clustering. As a long term goal, we also plan to include algorithms for mining frequent patterns and association rules, classification and outlier detection.

For finding a ready market in the data science community it will be necessary to

provide a functional language for executing the k-Means algorithm since not all data scientist might be fluent in the SQL language. However, this is a challenge that goes beyond the implementation of a k-Means algorithm as a language has to be found that adapts all the available SQL expressions and operators of HyPer.
Moreover, data scientists do not only want to use a functional language, they also have to visualize the result of data mining algorithms. Therefore the HyPer system and its functional language must be able to connect with a 2D plotting library to produce publication quality figures in a variety of forms. For the k-Means clustering algorithms this means a possibility to plot the data points, e.g. as a scatter plot. An example for a 2D plotting library based on a functional language is the Python library matplotlib [Hun07].

Acronyms

ACID Acid.

DBMS Technische Universität München.

ELKI Technische Universität München.

ETL Technische Universität München.

GUI Technische Universität München.

HyPer Technische Universität München.

IT Technische Universität München.

LLVM Technische Universität München.

OLAP Technische Universität München.

OLTP Technische Universität München.

PAL Technische Universität München.

SAP Technische Universität München.

SciPy Technische Universität München.

TU Technische Universität München.

TUM Technische Universität München.

Weka Technische Universität München.

List of Figures

List of Tables

Bibliography

[AD07] A. Ahmad and L. Dey. "A k-mean clustering algorithm for mixed numeric and categorical data." In: *Data Knowl. Eng.* 63.2 (2007), pp. 503–527.

[Agr+98] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. "Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications." In: *SIGMOD Rec.* 27.2 (June 1998), pp. 94–105. ISSN: 0163-5808. DOI: 10.1145/276305.276314.

[AKZ08] E. Achtert, H. Kriegel, and A. Zimek. "ELKI: A Software System for Evaluation of Subspace Clustering Algorithms." In: *Scientific and Statistical Database Management, 20th International Conference, SSDBM 2008, Hong Kong, China, July 9-11, 2008, Proceedings*. 2008, pp. 580–585. DOI: 10.1007/978-3-540-69497-7_41.

[Alo+09] D. Aloise, A. Deshpande, P. Hansen, and P. Popat. "NP-hardness of Euclidean Sum-of-squares Clustering." In: *Mach. Learn.* 75.2 (May 2009), pp. 245–248. ISSN: 0885-6125. DOI: 10.1007/s10994-009-5103-0.

[Apa] Apache Software Foundation. *Apache Mahout:: Scalable machine-learning and data-mining library*.

[AS94] R. Agrawal and R. Srikant. "Fast Algorithms for Mining Association Rules in Large Databases." In: *20th International Conference on Very Large Data Bases*. Morgan Kaufmann, Los Altos, CA, 1994, pp. 478–499.

[Aul+08] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. "Multi-tenant Databases for Software As a Service: Schema-mapping Techniques." In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. Vancouver, Canada: ACM, 2008, pp. 1195–1206. ISBN: 978-1-60558-102-6. DOI: 10.1145/1376616.1376736.

[Aul+09] S. Aulbach, D. Jacobs, A. Kemper, and M. Seibold. "A Comparison of Flexible Schemas for Software As a Service." In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. SIGMOD '09. Providence, Rhode Island, USA: ACM, 2009, pp. 881–888. ISBN: 978-1-60558-551-2. DOI: 10.1145/1559845.1559941.

[AV07] D. Arthur and S. Vassilvitskii. *k-means++: the advantages of carefull seeding*. 2007, pp. 1027–1035.

[Ber02] P. Berkhin. *Survey of clustering data mining techniques*. Technical report. Accrue Software, San Jose, CA, 2002.

[Bez+12] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. "Julia: A Fast Dynamic Language for Technical Computing." In: *CoRR* abs/1209.5145 (2012).

[BGV92] B. E. Boser, I. M. Guyon, and V. N. Vapnik. "A Training Algorithm for Optimal Margin Classifiers." In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. COLT '92. Pittsburgh, Pennsylvania, USA: ACM, 1992, pp. 144–152. ISBN: 0-89791-497-X. DOI: 10.1145/130385.130401.

[DG08] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492.

[Est+96] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise." In: *Second International Conference on Knowledge Discovery and Data Mining*. Ed. by E. Simoudis, J. Han, and U. M. Fayyad. AAAI Press, 1996, pp. 226–231.

[FA10] A. Frank and A. Asuncion. *UCI machine learning repository*. 2010.

[FF48] R. A. ; Y. Fisher and Frank. *[1938]*. Ed. by S. tables for biological, agricultural, and medical research (3rd ed.) London: Oliver & Boyd, 1948, pp. 26–27.

[Hal+09] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. "The WEKA Data Mining Software: An Update." In: *SIGKDD Explor. Newsl.* 11.1 (Nov. 2009), pp. 10–18. ISSN: 1931-0145. DOI: 10.1145/1656274.1656278.

[Han+04] J. Han, J. Pei, Y. Yin, and R. Mao. "Mining Frequent Patterns Without Candidate Generation: A Frequent-Pattern Tree Approach." In: *Data Min. Knowl. Discov.* 8.1 (Jan. 2004), pp. 53–87. ISSN: 1384-5810. DOI: 10.1023/B:DAMI.0000005258.31418.83.

[Hel+12] 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." In: *Proc. VLDB Endow.* 5.12 (Aug. 2012), pp. 1700–1711. ISSN: 2150-8097. DOI: 10.14778/2367502.2367510.

[HKP11] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 0123814790, 9780123814791.

[Hun07] J. D. Hunter. “Matplotlib: A 2D graphics environment.” In: *Computing In Science & Engineering* 9.3 (2007), pp. 90–95.

[HW79] J. A. Hartigan and M. A. Wong. “Algorithm AS 136: A k-means clustering algorithm.” In: *In: Applied Statistics 28* 1 (1979), pp. 100–108.

[J+01] E. Jones, T. Oliphant, P. Peterson, et al. *SciPy: Open source scientific tools for Python*. [Online; accessed 2015-02-28]. 2001–.

[KN11] A. Kemper and T. Neumann. “HyPer: A hybrid OLTP amp;OLAP main memory database system based on virtual memory snapshots.” In: *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. Apr. 2011, pp. 195–206. DOI: 10.1109/ICDE.2011.5767867.

[KR90] L. Kaufman and P. J. Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. New York: John Wiley and Sons, 1990.

[KYJ13] M. Kaul, B. Yang, and C. Jensen. “Building Accurate 3D Spatial Networks to Enable Next Generation Intelligent Transportation Systems.” In: *Mobile Data Management (MDM), 2013 IEEE 14th International Conference on*. Vol. 1. June 2013, pp. 137–146. DOI: 10.1109/MDM.2013.24.

[LA04] C. Lattner and V. S. Adve. *LLVM: A compilationframework for lifelong program analysis & transformation*. InACM International Symposium onCode Generation and Optimization (CGO), pages75-88, 2004.

[Llo82] S. P. Lloyd. “Least squares quantization in PCM.” In: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 129–136. DOI: 10.1109/TIT.1982.1056489.

[LM10] A. Lakshman and P. Malik. “Cassandra: A Decentralized Structured Storage System.” In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40. ISSN: 0163-5980. DOI: 10.1145/1773912.1773922.

[Lor74] R. A. Lorie. *XRM - an extended (n-ary) relationalmemory*. IBM Research Report, 1974.

[Mac67] J. B. MacQueen. *Some Methods for classification and Analysis of Multivariate Observations*. In: Berkeley Symposium on Mathematical Statistics and Probability, 1967.

[MNV09] M. Mahajan, P. Nimbhorkar, and K. Varadarajan. “The Planar k-Means Problem is NP-Hard.” In: *Proceedings of the 3rd International Workshop on Algorithms and Computation*. WALCOM ’09. Kolkata, India: Springer-Verlag, 2009, pp. 274–285. ISBN: 978-3-642-00201-4. DOI: 10.1007/978-3-642-00202-1_24.

[Müh+13] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. "Instant Loading for Main Memory Databases." In: *Proc. VLDB Endow.* 6.14 (Sept. 2013), pp. 1702–1713. ISSN: 2150-8097. DOI: 10.14778/2556549.2556555.

[Müh+14] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. "One DBMS for All: The Brawny Few and the Wimpy Crowd." In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: ACM, 2014, pp. 697–700. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2594527.

[Neu11] T. Neumann. "Efficiently Compiling Efficient Query Plans for Modern Hardware." In: *Proc. VLDB Endow.* 4.9 (June 2011), pp. 539–550. ISSN: 2150-8097. DOI: 10.14778/2002938.2002940.

[Neu14] T. Neumann. "Viktor Leis: Compiling Database Queries into Machine Code." In: *IEEE Data Eng. Bull* 37.1 (2014).

[Ora15] Oracle. *Oracle Advanced Analytics Option*. Mar. 2015. URL: http://www.oracle.com/technetwork/database/options/advanced-analytics/overview/index.html.

[Pag+99] L. Page, S. Brin, R. Motwani, and T. Winograd. "The PageRank Citation Ranking: Bringing Order to the Web." In: 1999-66 (Nov. 1999). Previous number = SIDL-WP-1999-0120.

[PCY95] J. S. Park, M.-S. Chen, and P. S. Yu. "An Effective Hash-based Algorithm for Mining Association Rules." In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. SIGMOD '95. San Jose, California, USA: ACM, 1995, pp. 175–186. ISBN: 0-89791-731-6. DOI: 10.1145/223784.223813.

[Qui93] J. R. Quinlan. *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN: 1-55860-238-0.

[R D08] R Development Core Team. *R: A Language and Environment for Statistical Computing*. ISBN 3-900051-07-0. R Foundation for Statistical Computing. Vienna, Austria, 2008.

[Rap14] RapidMiner. *RapidMiner Studio 6.2 and Server 2.2*. 2014.

[Ros58] F. Rosenblatt. "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain." In: *Psychological Review* 65.6 (1958), pp. 386–408.

[SAP14] SAP. *SAP HANA Predictive Analysis Library*. Nov. 2014.

[Shv+10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. "The Hadoop Distributed File System." In: *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. ISBN: 978-1-4244-7152-2. DOI: 10.1109/MSST.2010.5496972.

[SON95] A. Savasere, E. Omiecinski, and S. B. Navathe. "An Efficient Algorithm for Mining Association Rules in Large Databases." In: *Proceedings of the 21th International Conference on Very Large Data Bases*. VLDB '95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 432–444. ISBN: 1-55860-379-4.

[Thu+09] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. "Hive: A Warehousing Solution over a Map-reduce Framework." In: *Proc. VLDB Endow.* 2.2 (Aug. 2009), pp. 1626–1629. ISSN: 2150-8097. DOI: 10.14778/1687553.1687609.

[Wu+08] X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. McLachlan, A. Ng, B. Liu, P. Yu, Z.-H. Zhou, M. Steinbach, D. Hand, and D. Steinberg. "Top 10 algorithms in data mining." English. In: *Knowledge and Information Systems* 14.1 (2008), pp. 1–37. ISSN: 0219-1377. DOI: 10.1007/s10115-007-0114-2.

[WYM97] W. Wang, J. Yang, and R. R. Muntz. "STING: A Statistical Information Grid Approach to Spatial Data Mining." In: *Proceedings of the 23rd International Conference on Very Large Data Bases*. VLDB '97. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, pp. 186–195. ISBN: 1-55860-470-7.

[Zah+10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. "Spark: Cluster Computing with Working Sets." In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. Boston, MA: USENIX Association, 2010, pp. 10–10.

[ZMH09] W. Zhao, H. Ma, and Q. He. "Parallel K-Means Clustering Based on MapReduce." English. In: *Cloud Computing*. Ed. by M. Jaatun, G. Zhao, and C. Rong. Vol. 5931. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 674–679. ISBN: 978-3-642-10664-4. DOI: 10.1007/978-3-642-10665-1_71.