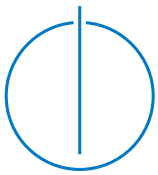
FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Efficient Data Mining Algorithms in Main Memory Databases

Martin Kapfhammer



FAKULTÄT FÜR INFORMATIK

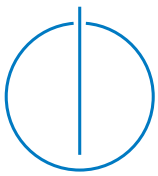
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Efficient Data Mining Algorithms in Main Memory Databases

Effiziente Data Mining Algorithmen in Hauptspeicher-Datenbanken

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

1.1 Motivation

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.
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 business 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 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. over 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 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, 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 compelex 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, Julia or the Python ecosystem SciPy. Also the Java frameworks Weka and ELKI are used actively, in particular in the research community. All tools are providing an environement 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, Apache Sparx and the data mining and machine learning framework Apache Mahout are used.
While all of the presented environments are frequently used by data scientists, they demonstrate one decisive drawback: Before executing data mining algorithms on the data sets, 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 and RapidMiner 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 run high-level "glue" code on the database. 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 by high-level constructs, often resulting in a bad performance. Therefore, database vendors such as Oracle and SAP are providing their databases with more and more statistical and

data mining functionalities using existing SQL syntax. 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. multicore 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. To go one step further, a intuitive user interface for great user experience has to be provided: 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. Since 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 the main criteria. Therefore, the implementation of k-Means should demonstrate the possibilities of implementing and running data mining algorithms in 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 paper is organized as follows: In the next section, fundamentals of the HyPer database are presented. In chapter 3, the terms knowledge discovery and data mining are defined. Then, the k-Means clustering algorithm is introduced. Chapter 5 dicusses 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 in HyPer. Chapter 8 concludes the thesis.

2 HyPer

2.1 Motivation

In this chapter, the relational main memory database system HyPer 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.
As already mentioned, historically databases are disk-based systems separated into two parts: An OLTP system, optimized for high rates of mission-critical, transactional write requests, and an 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. Since this process implicates heavy load on the database, it is usually done periodically, e.g. over night.
Obviously, this architecture reveals several drawbacks. The periodical execution of the ETL process results in stale data on the OLAP system, i.e. when implementing a nightly ETL process, data can be outdated for up to 24 hours which can be problematic for real-time data analysis. Furthermore, two systems lead to a higher amount of cost. Hardware and software costs must be taken into account as well as maintenance cost and incident management. Additionally, implementing an ETL process can make a system overly complex in contrast of having one system.
Addressing these challenges the HyPer main memory system was developed, with the goals to process OLTP transactions with high performance and throughput, and, on the same system, process OLAP queries on up-to-date data.

2.2 Architecture

In this section we give an overview about the HyPer system architecture and important design decisions. Major performance gains of HyPer are realized by omitting typical disk-based database characteristics that are not necessary any more 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.
On such a system, OLTP processing 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 never have to wait for I/O and can be executed very quickly. Thus, HyPer implements 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. Another advantage of this simple model is that locking and latching of data objects become redundant since only one transaction is active for the entire database. Even though there are developments to relax these constraints the basic OLTP processing is sequentially.
Obviously, the sequential execution of transactions is only possible if there aren't any long-running transactions. Long-running transactions would be a bottleneck for the entire database and must be handled in a different way. Therefore we will now look at how HyPer deals with OLAP queries.
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, 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 when an object is updated by the OLTP process, a new page gets 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 also shown that a `fork` can be achieved in several milliseconds, 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 multicore 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 a high throughput. OLAP queries can run very efficiently in parallel to the OLTP transaction system on different virtual memory snapshots. Due to these possibilties we explore now how to not only execute OLAP queries on a running transactional database but also more complex data mining algorithms.

3 Knowledge Discovery and Data Mining

3.1 Motivation

In this chapter we clarify the terms and importance of knowledge discovery and data mining. Shortly it is the process of gaining knowledge and insight from data. As already mentioned, data growth is exploding and petabytes of new data are generated every day by every aspect of daily life, such as businesses, society, science and medicine. With this huge amount of data available, data has become one of the most valuable resources of our decade. Quotes that compare data with the importance of oil or see it as a new currency become more frequent.
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 bought but also what they are likely to buy in the future. Therefore, they are using data mining techniques to present products to their customers in which they might be interested. This products can be similar to products the customer has already bought in the past, or it is a product another customer with similar characteristics has bought in the past. A very well know example is amazon.com which is offering a variety of similar products the customer recently looked for. In that case, data mining is 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 their customers usage behaviour, while state authorities are interested in wiretapping of potential criminals. Medical data might be used to install a ameliorated patient monitoring and web data can be used for a search engine to find the best results or deliver the promising ads. The list of sources to generate data and the knowledge one is interested is 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 the data mining steps 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.

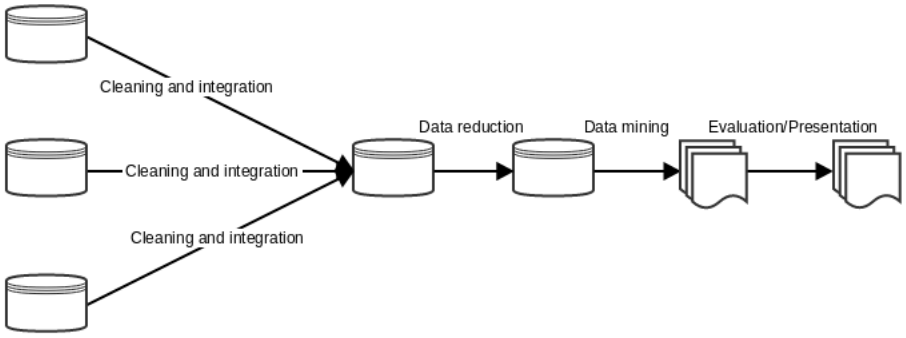


Figure 3.1: The Process of Knowledge Discovery.

Usually, data mining is seen as a part of the knowledge discovery process as depicted in Figure 3.1. The knowledge discovery process starts with the preprocessing steps data cleaning, integration, selection and transformation. The actual knowledge gain is then acquired in the data mining step afterwards the knowledge is 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, in this chapter we give a profound overview over all the knowledge discovery steps. Regarding all those steps, Han et al. give a very accurate 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."

3.2.1 Data Cleaning

In the following sections we look at each phase in detail and start with 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. Are these missing values crucial for the further data mining process, several data cleaning techniques exist to fill the gaps in the data, e.g. by using an average value such as the mean or 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 requires often 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 these sources have to be integrated into one coherent data set which can be 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 data cleaning 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 should be 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 transformation which changes 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 to get the information the user is interested. 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 draw.

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. These algorithms are part of 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 criterias 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 then 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. That are just two possible strategies that could be applied with the knowledge of frequent pattern mining.
Most algorithms for frequent pattern mining expecting transactional data as input, because this type of data represents a market basket the best. The most popular algorithms is the Apriori algorithms, working in an iterative manner. Apriori generates itemsets of length k and checks if these itemsets appear frequently, that means 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 algorithms converges if no more itemsets can be generated.
Since the candidate generation and the scanning of the database is expensive, there is a lot of research to improve the Apriori performance as well as establish other techniques. One of them is FP-growth (frequent pattern growth), an algorithm compressing the database into an FP-tree and therefore finds frequent itemsets without candidate 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: Emails should be classified into two categories: spam and non-spam. This is done by labelling existing emails and categorizing them as spam or non-spam. These data set is then used as the training data set for the classification algorithm. The algorithm uses this knowledge to classify new emails as spam or non-spam.
Since classification needs training data it is also called supervised learning. That means that before starting 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 exist many classification algorithms in practice, the most popular ones are Naive bayes, Support Vector Machines (SVM), Decision Trees and Neural Networks.
A very popular classifier is the Support Vector Machine. An SVM requires training data and builds a model upon. Each data tuple is represented as a vector in the SVM model space. Since the data is labelled the SVM knows which data tuples belong together and tries to find separating hyperplanes between them. These hyperplanes can be represented by a small subset of the data vectors called the support vectors. This fact

makes the SVM very efficient using high-dimensional data. New data tuples are then represented 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 labelled 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 set of available news topics, instead the topics can change daily. Therefore, setting up training data and use classification 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 categories:

- 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 and k-Medoids.

- 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* shape 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 algorithms is the DBSCAN (Density-based spatial clustering of Applications with Noise) algorithm, finding core objects, i.e. objects within a dense neighbourhood and joining those core objects together to find dense regions.

- Grid-based methods: All the presented methods so far are data-driven, i.e. groupings are found 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 and CLIQUE. 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 securtiy 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. These algorithms are often optimized to omit all dense areas and close data points to find outliers in a more effective way.
Apart from unsupervised learning, supervised learning can be applied for outlier detection as well. First, data is labelled 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.
Apart from clustering and classification techniques, statistical and proximity-based methods are also used for outlier detection.

4 The k-Means Clustering Algorithms

In this chapter the data mining algorithm k-Means is presented. K-means is a well-studied clustering algorithms, 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 squared distance between each data point and its assigned center. While the solution to that problem is NP-hard, there are several heuristics, in particular the Lloyd algorithm, which is a local search solution to this problem.

4.1 Motivation

We decided to use the k-Means algorithms 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 algoritm "is by far the most popular clustering algorithm used in scientific and industrial applications". K-means is also part of the 10 top data mining algorithms identified by the IEEE International Conference on Data Mining (ICDM) in December 2006, next to other famous algorithms such as SVM and the PageRank algorithm.
Since the algorithm is very popular and easy to understand, makes it suitable for the first implementation of a data mining algorithm on HyPer. To our best knowledge, all major data mining tools are implementing k-Means, which makes it neat to compare regarding the running time. The cluster compactness, i.e. the sum of squared errors is a good quality measurement of the clustering, and allows very good comparability. Parts of K-means can be executed in parallel, and since HyPer supports parallel computation, it will be another interesting research to evaluate how single-threaded execution of K-means vs a multi-threaded computation. We are expecting tremendous performance gains executing HyPer on a multicore machine.

4.2 The Lloyd Algorithms

In the following section we discuss the Lloyed algorithm, 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,

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

By finding these center points, each data point is assigned to a center point and we have found our clustering.
Solving such a problem is NP-hard, however Lloyd cite proposed 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 a distance function. Usually, the euclidean distance is used, other implementations allow to choose between several distance functions. After assigning each data point to its closest center, the centers gets updated, i.e. the center is the mean coordinate of all the data points assigned to this center. Then the process begins again, assigning the data points again, now to the updated centers. This continues until the process stabilizes and the algorithm converges.
The k-Means algorithms is described then the follwing:

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 result.

4.3 Example

In this section the k-Means algorithm is presented by an example. Our example data set consists of five data points with two dimensions: $x_0(3,9), x_1(2,5), x_2(5,8), x_3(7,5), x_4(4,2)$. This data will be clustered using the k-Means algorithm, with $k = 2$, i.e. we are searching for two clusters. First, the algorithm starts with an initialization phase: Randomly, we take two instances as initial center points of our two clusters. In this example, these data points are $c_0(3,9)$ and $c_1(4,2)$.

Next, we start the first iteration of the k-Means algorithms and compute the euclidean distance from each data point to c_0 and to c_1. Each data point is then assigned to the closest cluster, as shown in Table 4.1.

Table 4.1: Computations in Iteration 1.

Data Point	$c_0(3,9)$	$c_1(4,2)$	Cluster
$x_0(3,9)$	0.00	7.07	C_0
$x_1(2,5)$	4.12	3.61	C_1
$x_2(5,8)$	2.24	6.08	C_0
$x_3(7,5)$	5.66	4.24	C_1
$x_4(4,2)$	7.07	0.00	C_1

Now, the centers have to be recalculated as the mean of all assigned data points. The result are the updated center points $c_0(4, 8.5)$ and $c_1(4.33, 4)$. These are used to compute the distances again for each data point as shown in Table 4.2. As the data points are then assigned to the closest cluster. As the center points have changed, the distances to the centers changes and data points could be assigned to new clusters. In our example the data points are assigned to the same clusters as after Iteration 2. Therefore the algorithm converges and we have found the result. Figure 4.1 depicts the two clusters. Cluster C_0 with the center point c_0 is marked by blue data points and is positioned on the top of the chart, while Cluster C_1 with the center point c_1 is a broader cluster from the middle of the chart to the bottom.

Table 4.2: Computations in Iteration 2.

Data Point	$c_0(4,8.5)$	$c_1(4.33,4)$	Cluster
$x_0(3,9)$	1.12	5.17	C_0
$x_1(2,5)$	4.03	2.54	C_1
$x_2(5,8)$	1.12	4.06	C_0
$x_3(7,5)$	4.61	2.85	C_1
$x_4(4,2)$	6.50	2.02	C_1

4.4 The k-Means++ Initialization Strategy

Even though the Lloyd algorithm provides good results in practice, a clustering can be arbitrarily bad for some data sets. Particularly, the random choosing of the initial cluster

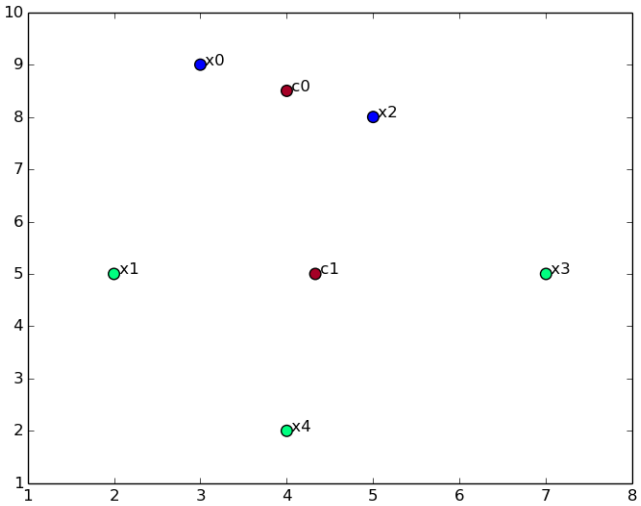


Figure 4.1: k-Means Example Clustering.

centers can lead to a bad grouping which cannot be changed during the clustering algorithms. Therefore, [**kmeans++**] 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 $D(x)$ the shortest distance from a data point the closest, already picked center.

1. Take the first center $c1$, chosen uniformly at random from X.
2. The next center c_i is choosen from $x \in X$ with the propability

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

3. Repeat Step 2 until all k centers are selected.
4. Continue as in the standard k-Means algorithm.

The authors call the weighting in Step 2 the D^2 weighting.
Most clustering tools implement a k-Means++ initialization strategy for k-means, therefore we will also provide this for the k-Means algorithm on HyPer.

5 Related Work

Nowadays data mining landscape can be separated in stand-alone machine learning frameworks such as WEKA and ELKI, general-purpose scientific languages such as R, Julia or the Python library SciPy, and Big Data platforms such as Apache Hadoop and Spark.

5.1 Research Tools

- Weka: Weka is a data mining software providing a large collection of machine learning algorithms for all aspects of data mining like data preprocessing, classification, regression, clustering, association rule mining and visualization. It is actively developed by the University of Waikato and gained big attention in the machine learning community over the past decade. Weka is written in Java and can be used as a stand-alone GUI tool or executed directly within Java programs.
- ELKI: ELKI (Environment for Developing KDD-Applications Supported by Index Structures) is a relatively new data mining framework developed by the Ludwigs-Maximilians-Universität München to implement and evaluate algorithms in the field of data mining. ELKI provides the most important algorithms for data mining, is written in Java, and provides a GUI to be easily used by data scientists. ELKI sees itself as 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. ELKI also encourages the use of index-structures to achieve performance gains when working with high-dimensional data sets.

5.2 Tools for Statistical Computing

- R: R is a language for statistical computing and data analysis, providing a variety of data mining libraries for all aspects of data mining and machine learning. It comes with rich graphical techniques and is therefore often researchers number one tool to create graphics for publications. Most of R libraries are written in R itself, however, C, C++ and Fortran code can be called at run time and is often

used for computationally-intensive tasks. Unlike ELKI and Weka, R does not provide a graphical interface, however, there exist several projects, e.g. JGR or R Commander that aim to provide a R GUI.

- SciPy: SciPy is a python environment providing several libraries to perform data mining such as NumPy, pandas and Matplotlib. As R, SciPy comes with algorithms for aggregation, clustering, classification and regression, all embedded in the Python language. Due to the elegance of the Python syntax, its popularity is growing, not only among data scientists, but also for prototyping new algorithms.
- Julia: Julia is a relatively new dynamic programming language for scientific computing with a main purpose on high performance. As R, Julia is a programming language itself written mainly in Julia, as well as C and Fortran to gain better performance. For data analysis, external packages are available allowing the execution of state-of-the-art data mining algorithms. Interestingly, Julia uses LLVM-based just-in-time (JIT) compilation, and therefore is often matched with the performance of C. The same compilation technique is used in HyPer, therefore it will be interesting to compare both techniques.

5.3 Big Data Platforms

General about Big Data

- Apache Hadoop: Apache Hadoop is an open-source software for reliable, scalable, distributed computing of large datasets across clusters of computers. The heart of Hadoop is the Hadoop Distributed File System (HDFS) and Hadoop MapReduce, a simple programming model for distributed processing. Since MapReduce programs can be run on up to thousands of machines, it is ideal for Big Data. Several Algorithms can be performed on the MapReduce programming model, e.g. k-Means. For our research, it will be interesting to see how HyPer works with Big Data compared to Apache Hadoop.
- 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. Regarding the main-memory capabilities of Spark, it will be interesting to compare Spark with HyPer.
- Apache Mahout:

5.4 Knowledge Discovery in Databases

- 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 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 possibilites 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 improvemnts 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.

5.5 Middleware Tools

- MADlib: 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.
  More details in the paper.

- RapidMiner: 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.

6 Implementation of k-Means in HyPer

In this section 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 advantages over existing solutions. Then, the functionality of the k-Means implementation is introduced. Next, the technical details are shown: The data materialization, two different serial implementation approaches and a parallel version. Finally, an integration of the popular k-Means++ initialization strategy is presented.

6.1 HyPer Operator Fundamentals

6.1.1 The Consume Produce Programming Model

In this section we talk about the implementation of operators in HyPer in general. With this understanding we can then show how k-Means can be implemented to be used in this programming model.
One of the main observations when working with main memory databases where all data resides in main memory is that query performance is much more dependent on the CPU costs of the query processing than on I/O cost as in traditional systems. Therefore, query processing for HyPer has 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 [AV07a]: 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 was not a limiting factor. However, for main memory databases, this is not perfect: First, next is called up to a million times, since it is called 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. And finally, the iterator model results often in a poor code locality and complex bookkeeping. This can be seen by the functionality of a 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 [AV07b] [AV07c] proposes a new query compilation strategy for main memory databases: Instead of the operator centric approach of the iterator model, processing is now data centric. Therefore data can be kept in the CPU registers as long as possible, while the boundaries between the operators are 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.

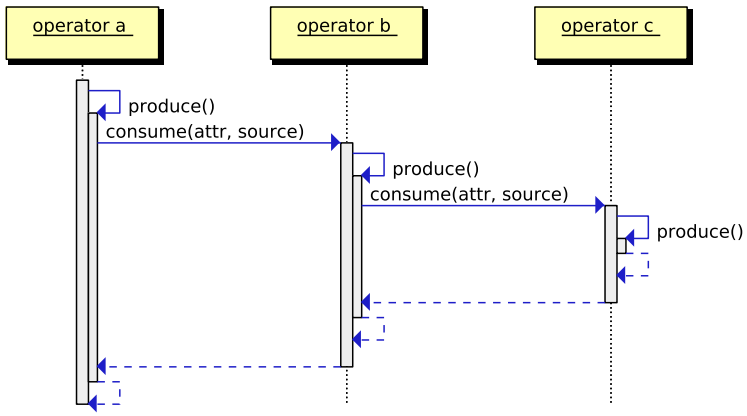


Figure 6.1: Consume Produce Sequence Diagram.

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. Thus, most computation is done using the CPU registers, 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.
Therefore, computation is 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 it a pipeline breaker. Materializes an operator all incoming tuples before continuing processing, we call it a full pipeline breaker. An example is a join operator: One side of the 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 we call it 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 assembly code there is not `consume produce` present anymore.

6.1.2 LLVM Code Compilation

After an understanding of the HyPer operator programming model, next we discuss the query compilation in further detail. As in traditional systems, queries are compiled by parsing the query, translated into algebra and optimized. In contrast to a traditional system, the algebra is now not translated into physical algebra and executed, but compiled by the code generation into an imperative program. For this imperative program the `consume produce` model is used.
For compiling the algebraic expressions into machine code, the first approach was to generate C++ code and load the result as a shared library at runtime. This seems logical because HyPer is already written in C++ and the shared library could access the existing data structures of the database system easily. On the other hand, compiling optimized C++ code is very slow and the compilation time can already take several seconds for a complex query, which is too slow for a database system. Additionally, the C++ compiler does not offer total control over the generated code, e.g. overflow flags are not available.
Therefore, queries are compiled into native machine code using the Low Level Virtual Machine (LLVM) compiler framework [AV07d]. 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 portable 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 the original 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.
Additionally, LLVM code is perfectly able to interact with C++, the main language of the HyPer database which is a big advantage. 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 like C++. This enables us to implement the operators using both C++ and LLVM code, and reuse database logic such as index structures, that are already implemented in C++.
Therefore complex data structures or 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. This results also in a low query compilation time. 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++ textttsort 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 gains, it is important that the code that is executed for most of the tuples is pure LLVM 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 a million times, which is quite likely when using big amounts of data.
As conclusion, we have shown that the HyPer programming model for operators implements a `consume produce` model to push data towards the next operator. Furthermore, the LLVM compiler framework is used for code generation and can be combined with C++ code at runtime. This division between LLVM and C++ code regarding programmer friendliness and execution time is one of the dominant patterns of the following sections.

6.2 Requirements and Constraints

Before we look at the technical implementation details in greater details, we discuss the requirements and constraints of our k-Means operator.

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 other operators useful for data mining, such as grouping and aggregation functionalities.
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 workstations.
As input parameters, 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 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. The verbose option let the algorithm to compute and to print additional information about the k-Means algorithm.
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 about 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.

6.3 Data Materialization

In this section we discuss the data materialization for the k-Means operator. As already stated, materialization is the process where we take our incoming tuples out of the CPU registers and write them into memory. Since this decreases the performance of the database, HyPer tries to avoid this process whenever possible and keeping the data in the pipeline until a pipeline breaker occurs. Unfortunately, k-Means is a pipeline breaker: E.g., center tuples have to be compared with all the data tuples to find the minimum distance between them. Therefore, we have to put all the incoming tuples in memory. That means, when the predecessing operator calls the `consume` function

of the k-Means operator, each incoming tuple will be written into main memory, as Figure 6.2 shows.

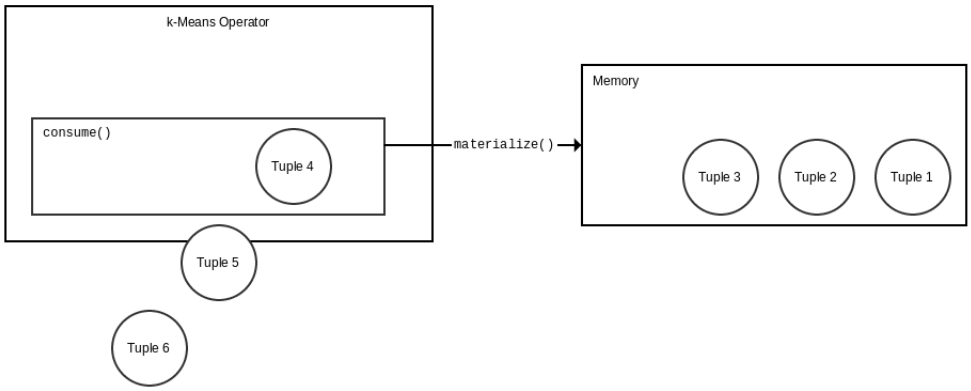


Figure 6.2: Data Materialization of Incoming Tuples.

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 code is materializing the incoming tuples. A pointer to the materialized chunk of memory is then stored in a C++ vector in the `runtime system`. Figure 6.3 shows 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 C++, getting the pointers to the memory locations, which allows to load the tuple from memory and back to 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.

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 is 0 to `k`. For data tuples, this field specifies to which center a data tuple belongs to. Initially, all data tuples are assigned to the center with identifier 0.

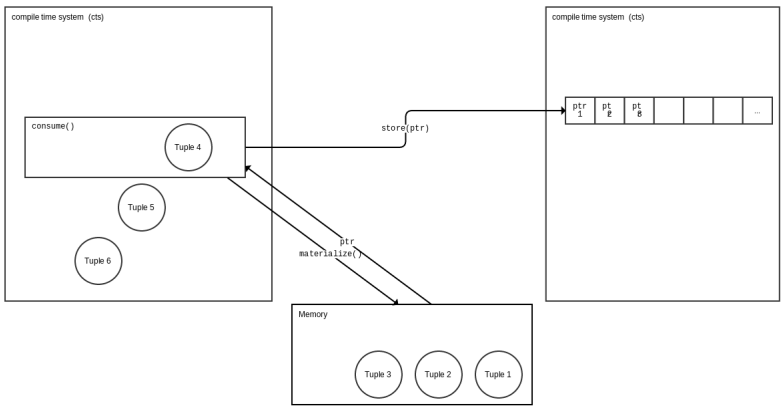


Figure 6.3: Data Materialization using the Runtime and Compile Time System.

Another difference is the change of data types between data and center tuples. While the data types of data tuples are specified in 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. Hence, the mean of an integer can be stored in the center field. For each data type there exists a rule to converse the type from data tuple to center tuple.

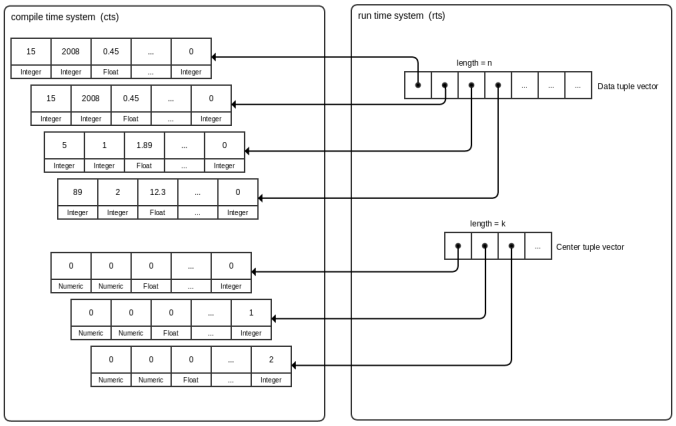


Figure 6.4: k-Means Data Materialization in Detail.

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, e.g. which parts should reside in the `compile time system` and which parts in the `run time system`.
This question is not trivial because there are no strict rules and several possibilities. 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 LLVM 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++ and using only a few generated LLVM functions, e.g. to compute the distance. Secondly, a system is presented implementing k-Means almost entirely in LLVM. Only small parts, like initializing the random centers are implemented in C++.

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 are doing their main computation after executing their `consume` function, which is a 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 picked. 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 then written to the corresponding center tuple. Therefore, a `centerPtr` and a random `dataPtr` are parameters of a generated `setCenter` function. There, the data tuple and the center tuple are loaded from memory using the two pointers. Data values are casted if necessary, since we have different data types among center tuples and data tuples, and then stored in the center tuple. Afterwards, the center tuple is written back to memory.

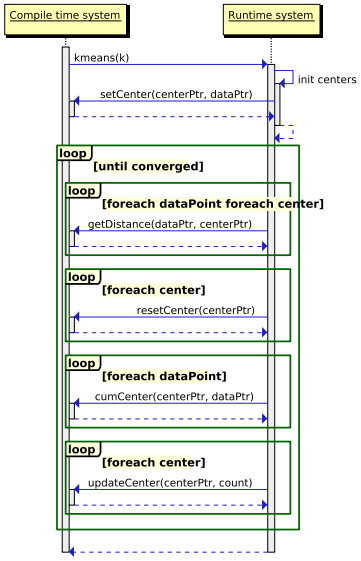


Figure 6.5: The k-Means Algorithm - C++ driven.

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 all center points to find the closest center for each data tuple. Therefore, `getDistance` is called for each `dataPtr - centerPtr` combination. The closest center for each data point is stored in a `unordered_map` in C++.
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 even 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.
This 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 `code time system` calls for our algorithm. For each data point, the values are added to the center tuple it belongs to. Therefore, the `cumCenter` function is called, adding the data point to the center point and also updates the cluster identifier of the data tuple. The `runtime system` keeps count about how many tuples are added to each center. When finished, each center is called again with this count to compute the actual mean of the center using the `updateCenter` function. This process continues until the algorithm converges.
As we see, 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 $(n+2) \cdot k + n$ per iteration, as Table 6.1 shows. The next sections presents an implementation that prevents the algorithms from too many calls between the two systems.

Table 6.1: Generated Function's 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 A 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 like `unordered_maps` and the convenience of the C++ syntax, allowing a quick implementation.
However, this 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++ data structures any more, but have to work with what LLVM gives us.

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, storing the count, i.e. 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.

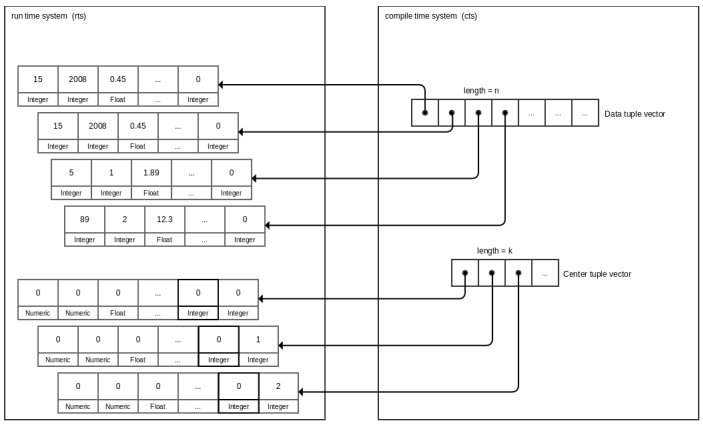


Figure 6.6: k-Means Data Materialization for the LLVM Implementation.

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 algorithms, 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.
The initial center setting process does not change, but after this the control of the code goes back to the `compile time system`. The only thing the `compile time system` requires from the `runtime system` are the first pointers and the last pointers 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 are now implemented in pure 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 each center tuple gets on data materialization. When adding data tuples to the corresponding center, we increment the count. At the end, when invoking the `updateCenter` function, this count is used to compute the mean.

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

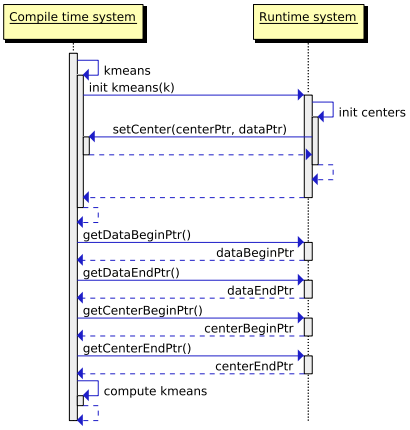


Figure 6.7: The k-Means Algorithm - LLVM driven.

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. An evaluation chapter will show how this affects the performance of the two serial implementations.

6.4.3 Initialization strategy

So far we discussed the implementation details of our k-Means algorithm without paying attention to the used initialization strategy. As shown in chapter 4, kmeans uses a random initialization strategy by default. A random initialzation strategy is easy to implement in the `runtime system`, since only C++ code is used finding a random subset of length `k` of the data tuples. This can be implemented using the Fisher-Yates shuffle algorithm [AV07e].
When discussing k-Means 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 accuracy of the clustering.
For implementation we have to compute the distance from the chosen center points to the data points already in the initialization phase. 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 initialzation in the `runtime system` to benefit of high level C++ program structures. To execute the k-Means++ algorithm, we have to add a 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 function is just part of LLVM code. 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 looking at 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 because of 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 done in each iteration. This means $k \cdot n$ distance computations, which can benefit a lot from parallelism.
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. In the runtime system, there are vectors for each thread, storing the pointers to the input tuples.

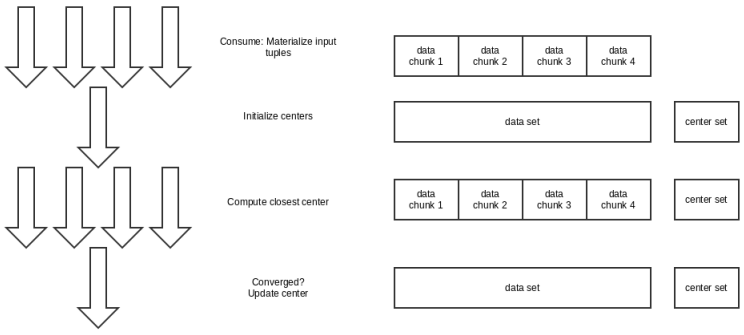


Figure 6.8: The k-Means Algorithm as Parallel Operator.

After consuming the data the k-Means algorithm continues with finding the initial center set. For that, we have to break the parallelism. For random initialization of the clusters, center tuples are selected from the entire data set. 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 by one of our initialization strategies and continue the parallel program sequence afterwards.
After the serial initialization we jump back to parallel execution: The algorithm computes the distance from each data point to each center point. For each data chunk we compute the distances on a separate thread. With omitting the overhead for process creation and management, this could lead to a running time advantage of one fourth of the serial execution.
Afterwards, we jump back to a serial execution of the program and check for each thread if the program converged. Only if all threads converged, we can terminate the algorithm and output the result of the clustering. Otherwise, we update the global centers by the newly assigned data points. In the first version of this implementation, this step is done in serial, but here is also potential to parallelize the program.
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 of by introducing the parallelism, we expect huge performance gains.

7 Evaluation

In this section we provide an extensive experimental evaluation of the HyPer k-Means operator. We compare the different HyPer implementations to each other as well as to state-of-the-art technologies of clustering data. 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 the execution time per iteration is measured. Therefore the different technologies can be compared fairly 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. 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 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: Data Sets.

	Instances	Dimensions	Size (byte)	Size (Gigabyte)
3D Network	434874	4	20673913	0.019
High Dimensional	50000	50	10000000	0.009
Medium Size	15M	4	240000000	0.22
Medium Size HD	15M	50	3000000000	2.79
Large Size	150M	10	6000000000	5.59

- 3D Network: The first data set contains the 3D spatial road network data of a road network in North Jutland, Denmark [**3dnet**]. It is a real world data set available at the UCI Machine Learning Repository [**uci**] and consists of 434874 data points in four dimensions: An id for the road segment, the latitude, the longitude and 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 20,673,913 bytes, therefore it is a rather small data set.

- High Dimensional: The second dataset is a synthetic data set. It consists of 50000 data points in 50 dimensions. All dimensions are floating point numbers. The size of the data set is 10000000 bytes, hence also a rather small data set, but this time high dimensional.
- Medium Size: The data set consists of 15 million data points in four dimensions. It is also a synthetic data set generated, all four dimensions containing floating point numbers. The size of the data set is 240000000 bytes and presents the medium size data set in this experiment section.
- Medium Size HD: The Medium Size HD 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 0.22 GB to 2.79 GB.
- Large Size: Finally, 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 Used Tools

As already described in chapter 5, there exist many tools for data mining. In this chapter we compare a selection of tools implementing the k-Means clustering algorithm with our 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. number of iterations, total cost and most important, 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 Lloyed algorithm.
Since k-Means is a non-deterministic algorithm, the number of iterations is another important criteria to make the results comparable. Running time of k-Means depends almost entirely on the number of iterations the algorithm has to make, which depends on the 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 result. Therefore, a fair comparison is not possible. Weka, R and Julia provide good configuration possibilities as well as a result set that contains information about the number of iterations. Weka is written in Java, while R and Julia are both written in a high level language. Even though, critical code parts are written in C, C++

and Fortran for better performance results, without the overhead of an entire database system behind. Therefore it will be interesting to compare Hyper with the Java written Weka and the optimized R and Julia.
weka, big data? why not others? weka arrf format -> loading time, results etc.

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 the two serial HyPer k-Means implementations: The C++ driven and the LLVM driven approach. For comparison we consider the compilation and the execution time. The compilation time is the time for generating LLVM code, while the execution time is the execution of the k-Means algorithm. 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 3,10 and 20 clusters. For each cluster number the algorithms was 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 does not change for a higher number of iterations.
Table 7.3 shows the result of the two serial implementations for the network data set. The compilation time is independent of the cluster number k and 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. For better visualization, Figure 7.1 depicts the same result as stacked bar charts.

While the compilation time is low and constant, the execution time differs among implementation. In particular 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

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

grows by a factor of 2.9, while the C++ 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.

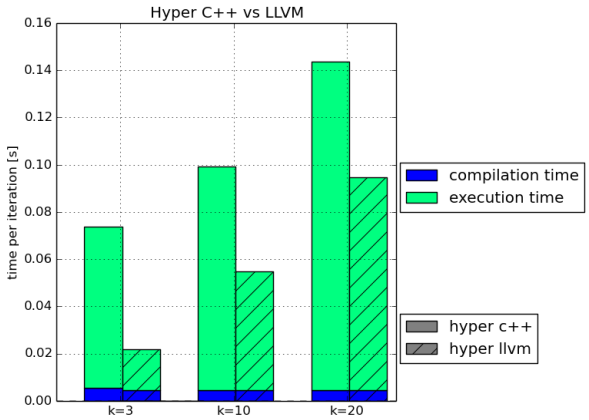


Figure 7.1: 3D Network - Time per Iteration.

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 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 C++ these functions are generated as functions callable from C++, while in the LLVM version, the code is directly embedded into an LLVM program structure. 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 and the data remains in the CPU registers. The second advantage is that the entire algorithm is compiled in LLVM code resulting in a 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.

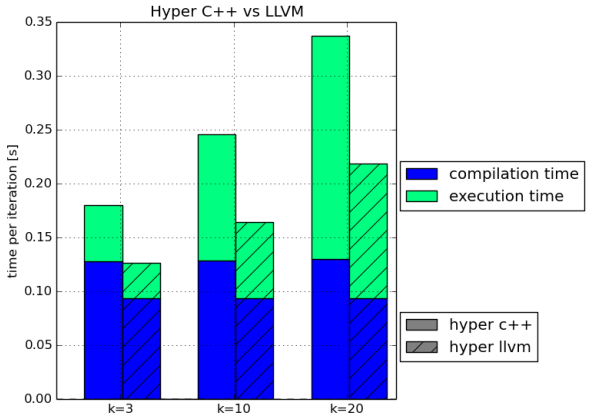


Figure 7.2: High Dimensional - Time per Iteration.

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 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 by a factor of 2.2. From $k = 3$ to $k = 10$, the LLVM version grows by a factor of 4, while the C++ grows by 3.8.

Interestingly, ten iterations terminate much quicker for the network data set than for the high dimensional data set, even though the network set is 2.1 times as large as the high dimensional data set. Even if we omit the compilation time which stays constant for a growing number of iterations, the execution time of the network data set is only for the C++ implemenation and $k = 3$ slower than the high dimensional data set. That shows us that our implementation performs much better for low dimensional data sets.

Table 7.5 and Table 7.6 show the same experiment for larger data sets. 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 smaller high dimensional data set. This proves that the compilation time is independent on the data size and only affect by the number of dimensions. However, as the number of instances is growing, the compilation time is not a significant factor for the overall execution time, as Figure 7.3 shows: The compilation time is not even visible plotting the data as barcharts as before.

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 again. 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 set is actually larger by a factor of 12.7, the values are strongly correlated to the growth of the data set.

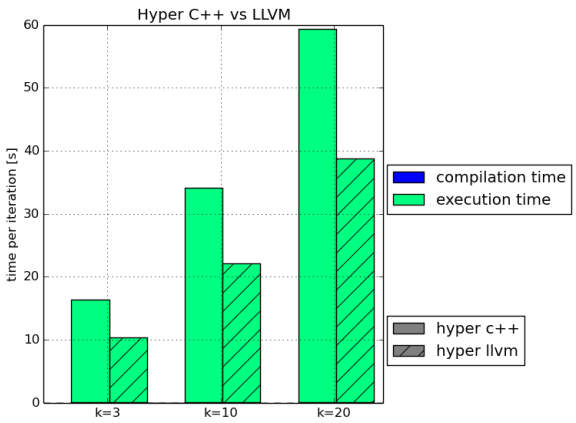


Figure 7.3: Medium Size High Dimensional - Time per Iteration.

Table 7.7 shows the same experiment for the large 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 2. Since the data set has a size of 5.59 GB the difference in the execution time is very significant. Again, the LLVM implementation is faster by around 20 seconds independent of the cluster number k.

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 the compilation time is not affected if the number of instances gets large but only if the number of dimensions grows. The compilation time is similar between the LLVM version and the C++ version. 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 can be slower than the execution time.
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 decrease in execution time the LLVM version was for all experiments the best choice regarding the running time of the k-Means algorithm.

7.4 Performance Test

In this section we compare our serial HyPer implementations with Weka, 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

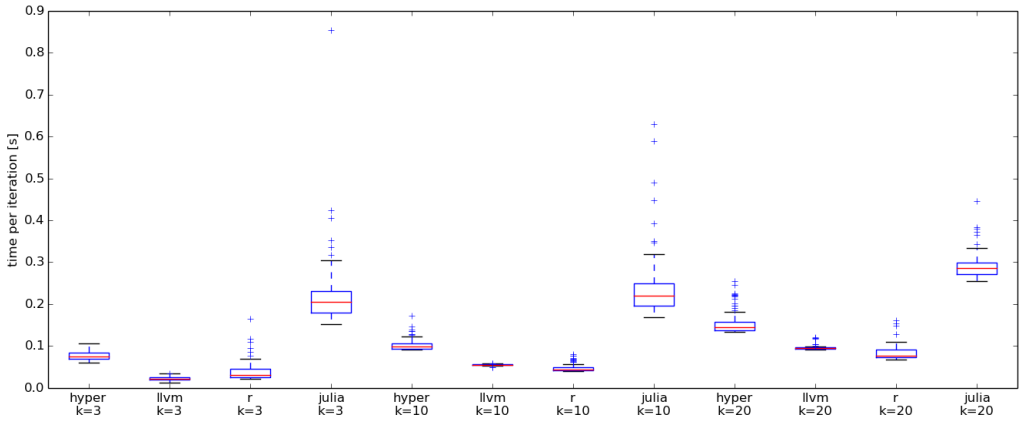


Figure 7.4: 3D Network - Time per Iteration.

The result is presented in Table 7.8. We already know that 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 for $k = 3, 10, 20$ times faster. Only the R implementation can compete with our HyPer operator and is even a bit faster. However, all

tested programs differ in time per iterations in a few hundred milliseconds therefore the differences are not very significant. Figure 7.4 shows the results as a boxplot.

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
50	0.04	0.05	0.07	0.04	0.06	0.10	0.21	0.34	0.40	0.13	0.16	0.22
50	0.04	0.05	0.07	0.05	0.07	0.11	0.22	0.35	0.44	0.13	0.17	0.22

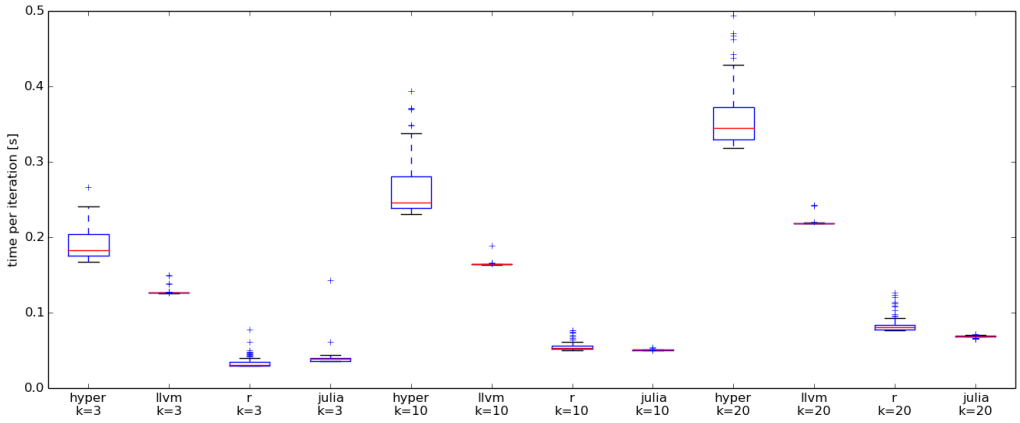


Figure 7.5: High Dimensional - Time per Iteration.

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 is now as fast as the R implementation, for $k = 10$ and 20 it is even faster. Figure 7.5 shows this results as a boxplot.
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 faster for $k = 3, 10$ and 20 times on the high dimensional set compared to the network data.

We run the same experiment on the medium size and the medium size high dimen-

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
50	5.43	7.65	10.71	0.79	1.63	2.55	2.57	3.44	4.77	0.94	2.14	3.51
50	5.44	7.70	10.91	0.80	1.63	2.56	2.64	3.45	4.84	0.94	2.14	3.52

Table 7.11: Medium Size HD - Time per Iteration.

	Julia			R			HyPer C++			HyPer LLVM		
k	3	10	20	3	10	20	3	10	20	3	10	20
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
50	11.51	15.81	21.69	9.50	15.50	23.79	16.77	36.11	59.53	10.44	22.12	38.84
50	11.65	15.83	21.70	9.64	15.58	23.79	17.29	42.15	59.57	10.45	22.13	38.84

sional 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 for $k = 3$ and 10 and 0.7 for $k = 20$. **??** shows the results as a boxplot.
For comparison Table 7.11 and **??** depict the result for the medium size high dimensional data set. As before, our own implementation does not perform ideally on high dimensions. For small $k = 3$, 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 better results. Both show a speed up by a factor of 0.7 for $k = 10$, and 0.6 for $k = 20$.
Obviously the high dimensional data set takes more time per iteration having a constant number of instances. Julia shows a slow down by a factor of 2 for all k's, R 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 perform 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 50000 instances the compilation time is 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.

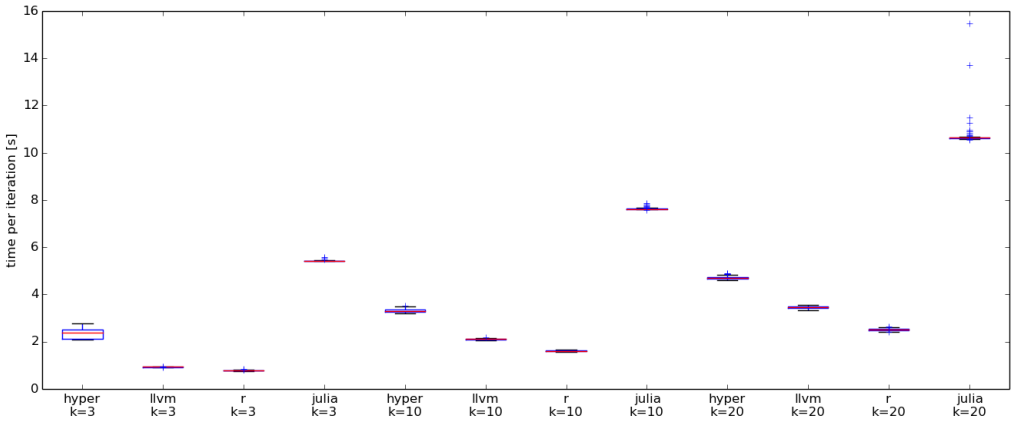


Figure 7.6: Medium Size - Time per Iteration.

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 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. This is also shown by a boxplot in Figure 7.8. Julia's performance is quite compared to the other tools since the data set consist only of ten dimensions. For growing k, the R implementation comes very close to the HyPer LLVM version.
In conclusion the experiments show that HyPer's 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. For high dimensional data sets the LLVM implementation does not perform ideally and is outnumbered by Julia's implementation optimized for high dimensions. If we take into account that the k-Means algorithm runs on top of an entire database with all its overhead, these results are affirmative for future development and the implementations of other operators on HyPer.

7.5 Parallel Implementation

So far we only looked at the serial execution 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.

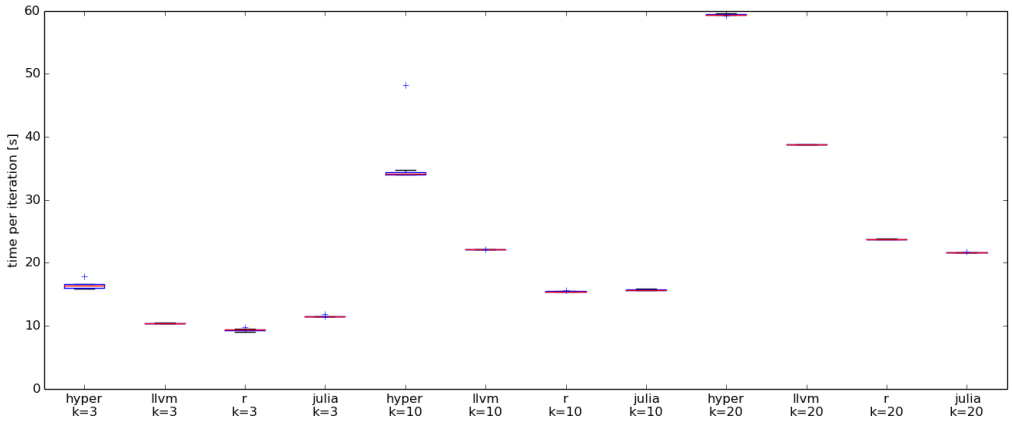


Figure 7.7: Medium Size HD - Time per Iteration.

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
50	65.72	92.01	135.87	30.80	54.73	77.27	37.50	62.41	92.86	18.44	40.72	67.57
50	65.78	92.27	137.21	33.08	55.02	77.97	37.52	62.72	92.95	18.46	40.72	67.63

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 modern hardware of database systems and the advantages of multi-threaded execution. In this section we compare the parallel implementation of the HyPer k-Means operator 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 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 **??**.

The reason is that for the parallel execution the running time is almost independent of k, threfore the parallel version outperforms R and LLVM: For R, the time per iterations grows by a factor of 2.1 from $k = 3$ to $k = 10$, and by a factor of 1.6 from $k = 10$ to $k = 20$, and for LLVM by a factor of 2.3 and 1.6, respectively. In contrast, the

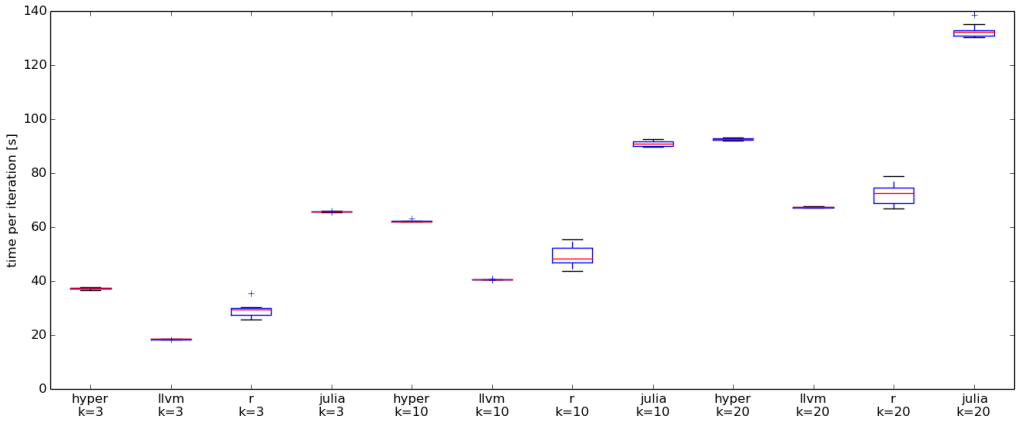


Figure 7.8: Large Size - Time per Iteration.

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

parallel implementation grows only by factor of 1.1 and 1.2, respectively. Therefore, R and the LLVM version grow by seconds as k grows, while the parallel version grows insignificantly by around 200 milliseconds.

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 $k = 3$. **??** depicts that the parallel version in 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 $k = 20$, and for LLVM 2.1 and 1.8, respectively. The parallel version starts at low 5.84 seconds and grows by a factor 1.4 from $k = 10$ and $k = 20$.
So this time, the parallel version not as independent by the cluster number k as for the low dimensional data set of same size. Instead, the performance gain results in the ability to handle high dimensional data better: From four dimensions to 50 dimension, 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

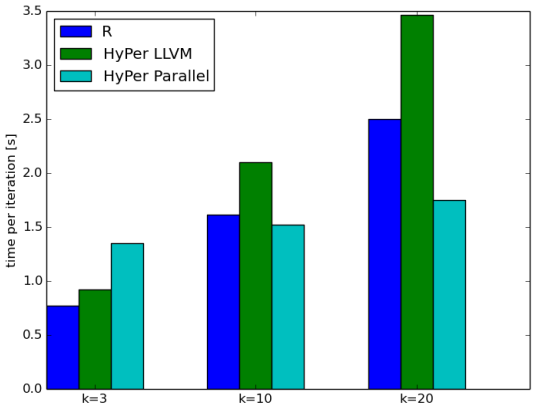


Figure 7.9: Medium Size - Time per Iteration.

Julia shows with a factor 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

Finally, Table 7.15 shows the results of the same experiment on the large data set. Again, the parallel 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, except for the medium size data set and $k = 3$ the parallel version is much better than the LLVM and R algorithm. 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. 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 and of the high-level C++ constructs. And finally, we parallize only the computation of the distances, not the cluster update.

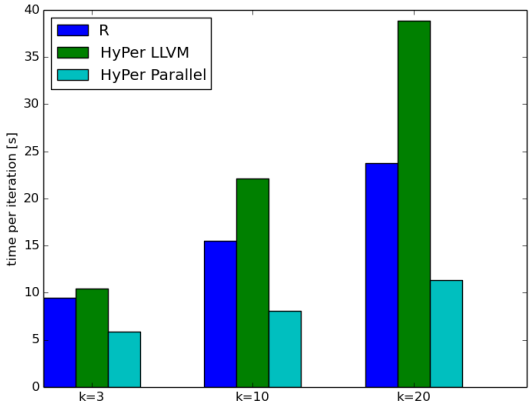


Figure 7.10: Medium Size HD - Time per Iteration.

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

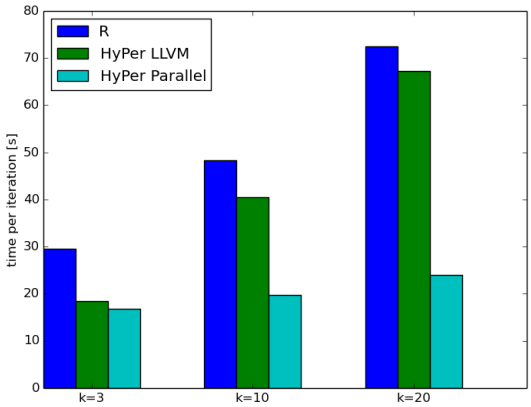


Figure 7.11: Large Size - Time per Iteration.

8 Conclusions and Future Work

8.1 Conclusions

8.2 Future Work

Many open challenges remain in mining data in HyPer. In this section we discuss this 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 execution. For the future we want to enhance the behaviour of k-Means and make our algorithm more flexible. That can be done by allowing the user to choose a distance function. For now, k-Means implements a Euclidean distance, however, for some problems other distances are more appropriate. Therefore we should offer to use 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 algorithms 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 algorithms has to be initiated. Therefore an additional input parameter can specify a tolerable change of data points. Is the change of data points underneath this border, the algorithm terminates even though there is a slight chance to find a better clustering.
Heterogenous data is another problem for our algorithms. No all data sets consist only of numeric data. Often it is a mixture between numeric, binary and categorical attributes. An improvement is to first convert the binary and categorical to numeric attributes, and then improve the distance and cost function to keep the different dimensions comparable, as proposed in k-Means Mixed.

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 only. Assigning weights is not only possible for dimensions - also instance can be weighted differently since some are more significant than others.
K-Means is usually implemented as the Lloyd algorithm, resulting in reasonable results. However, the execution time can be improved implementing the Hartigan-Wong algorithm. In contrary 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.

So far, the result of k-Means is a table with an additional column specifying the cluster number. Since this table is returned using the consume function, it can be used on 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. A way forward 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 a big step for computational data mining on databases.
HyPer is a database system that makes already extensive use of index structures. Therefore an option is to improve the k-Means algorithms, e.g. by using nearest neighbour data structures when clustering high dimensional data.
As shown in the evaluation chapter 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 call for the center selection process.
We have also shown the advantages of parallel execution of the k-Means algorithms. However, 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 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.

For finding a ready market in the data scientist 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 SQL scripting languages. 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.
The next point goes in the same direction: Data scientists do not only want to use the output as a database table or in a data structure of the functional language but also in a visual way. 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.
To fulfil the aim to provide further algorithms for data mining, it is appropriate to use the existing k-Means algorithm as building block. A very similar partitional clustering algorithm is the k-Medoids algorithm. 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. The Expectation–maximization (EM) algorithm is very similar to the k-Means algorithm and has a focus on fuzzy clustering. As a long term goal, we want to include also algorithm for mining frequent patterns and association rules, classification and outlier detection. information.

Glossary

computer is a machine that....

Acronyms

TUM Technische Universität München.

List of Figures

List of Tables

Bibliography

[AV07a] D. Arthur and S. Vassilvitskii. *k-means++: the advantages of carefull seeding*. 2007, pp. 1027–1035.

[AV07b] D. Arthur and S. Vassilvitskii. *k-means++: the advantages of carefull seeding*. 2007, pp. 1027–1035.

[AV07c] D. Arthur and S. Vassilvitskii. *k-means++: the advantages of carefull seeding*. 2007, pp. 1027–1035.

[AV07d] D. Arthur and S. Vassilvitskii. *k-means++: the advantages of carefull seeding*. 2007, pp. 1027–1035.

[AV07e] D. Arthur and S. Vassilvitskii. *k-means++: the advantages of carefull seeding*. 2007, pp. 1027–1035.