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 them is ensured with 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, modern hardware allows memory-based databases. With all data residing in the memory of the hardware, 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 TU München, and the main system used for our research. The possibility of executing OLAP queries on a RDBMS without interfering the OLTP transaction throughput opens new possibilities for database systems: Additionally to long-running queries for complex data analysis, Data Mining algorithms can be integrated for more profound insight into the data. In this work, the well-known k-Means algorithm is implemented in HyPer as a proof-of-concept, demonstrating the benefits of data mining operations in relational main-memory databases. First, the HyPer database is presented. Next, the term data mining is defined in greater detail and related work is presented. Then, the k-Means operator is introduced and its the implementation in HyPer is shown. Finally, extensive experiments demonstrate the benefits of k-Means in HyPer .

1.2 Research Questions

Research

1.3 Approach

Approach

2 HyPer

2.1 Motivation

In this section, 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 OLTP system, executing long-running queries. The data synchronization between the two systems is ensured with 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. The periodical execution of the ETL process results in stale data in 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 problems. Furthermore, two systems lead to a higher amount of infrastructure and maintaining 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 the following we give a short overview about the HyPer system architecture. Major performance gains are realized by omitting typical disk-based database characteristics, that are not necessary any more when data resides in main memory. Therefore, 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.
OLTP processing is highly efficient because all the data is already loaded in main memory, and the very 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 quick. Thus, HyPer implements OLTP transactions as single-threaded approach and executes all transactions sequentially. This is possible because the execution time of an OLTP transaction is only around ten 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.
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 long-running queries and OLAP-style queries in general.
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 process are sharing the same physical main memory location since their virtual address translation maps to the same segments. Therefore copying the 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 about 10 ms, almost independent of the size of the OLTP system.
So far we have shown how HyPer executes long-running OLAP queries parallel to high-throughput OLTP transactions by using virtual memory snapshots. 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 an appropriate 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 data mining algorithms.

3 Knowledge Discovery and Data Mining

3.1 Motivation

Before we talk about the opportunities and advantages of running data mining algorithms in HyPer, we first clarify the term Data Mining. Data Mining ,also known as knowledge discovery from databases (KDD), 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 a very valuable resource in our decade. Data is seen as the new oil, others call data the new currency. These quotes highlight the importance of data. However, it is important to note that we have to transform data into knowledge to actually gain value. For example, online shopping companies like amazon are very keen to find out not only what customers bought in the past, but also what they are likely to buy in the future. Therefore, amazon is using data mining techniques to present their users products related to products they recently viewed or bought. Also products of other customers with similar interests are displayed, as shown in figure 1. In that case, data mining is used to boost the consumerism. Additionally, this knowledge can be used to optimize storage cost, e.g. by knowing how much of certain products will be sold in the next time. Another example is Google Trends... Other areas

3.2 Knowledge Discovery

After understanding the importance of data mining for gaining knowledge and using this knowledge for further decision making, this section defines data mining 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 from databases, or KDD is not as arbitrary as data mining.

Often, data mining is seen as a part of the knowledge discovery process, as depicted by figure xx. The knowledge discovery process starts with the preprocessing steps data cleaning, integration, selection and transformation. The actual knowledge gain is then

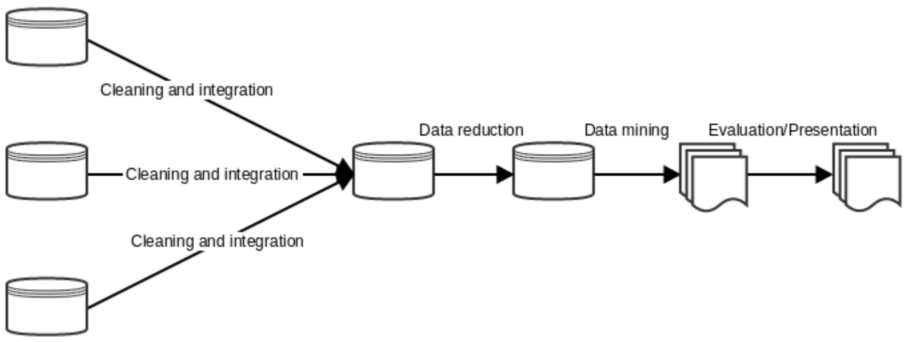

Figure 3.1: KDD.

acquired in the data mining step, afterwards the knowledge is evaluated and presented. 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."

In the following, we look at each of the knowledge discovery steps.

3.2.1 Data Cleaning

The first step of knowledge discovery needs to ensure that the data to be mined 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 for inconsistent data are distinct values with the same meaning, e.g. the chars w and f both meaning female. Additionally, we often see random error or variance in a measured variable. For proper data analysis, we have to clean this noisy data first 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 but also more complex data sets such as time-related data, data streams, spatial data, multimedia data, graph data or any data available on the Internet. All these data sources have to be integrated into one coherent data set which can be used for 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 source and to avoid redundancy.

3.2.3 Data Reduction

After cleaning and integrating data into a coherent data set, the data should be consistent, clean, complete and ready for further mining. However, those data sets are usually 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 principal component analysis, numerosity reduction above aggregation, like clustering or sampling, and data compression. Data transformation changes the values given data, e.g. by converting the data into another format, e.g. by using normalization. Data discretization is another part of data transformation, e.g. numeric data is presented in intervals. The aim of all of the addressed methods is to reduce the size of the 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 line between data reduction and data mining is not always easy to draw.

3.2.4 Data Mining

After the presented steps the data sets are ready to apply actual data mining techniques. In the following sections we will revise data mining goals, techniques and the associated algorithms for frequent pattern mining, classification, clustering and outlier detection.

3.2.5 Pattern Evaluation

So far we have learned several techniques of data mining to find patterns in the data set, such as clusters, outliers or frequent item sets. We can also use training data to predict and categorize new data. After generating those patterns we have to evaluate them to figure out how valuable the acquired knowledge is. First, a pattern has to be understandable, e.g. a certain clustering does not gain any knowledge if the reason for it is not comprehensible. Therefore, the reasons for the clustering must be clear for the user. That leads to the next step 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 data mining. 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. More ...

A section about the applicability of data mining methods in Hyper

3.3 Algorithms of Data Mining

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 online commercial stores are interested in what their customers are buying, in particular in items that are frequently bought together. In other words, which items are frequently put in the same market basket. This knowledge is very valuable: It's 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 and making decisions upon this information: Related items could be placed next to each other, to make the shopping experience for the customer more convenient. Another approach would be to put related items far aways from each other, so that the customer has to walk around more 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 frequent, that means if there count exceeds a given threshold. All the valid itemsets and then used in the next iteration to generate itemsets of length k+1. The algorithms converges after no more itemsets can be generated. Since the candidate generation and the scanning of the database is expensive, there is still active 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 more algorithms for more specific cases, e.g. for finding frequent patterns in high-dimensional, spatial and multimedia data.

3.3.2 Classification

n data mining, classification uses existing data as existing 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 as spam or non-spam email. These initial set of categorized emails is then used as the training data for the classification algorithms. Based on the knowledge the algorithms gets from the training data, new emails can be classified as spam or non-spam. Another example is to .... Since classification needs training data, where all the data tuples have to be labelled first, classification is also called supervised learning. That means that before starting the data mining process, some 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 as input 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 tuples, 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 the category as the other data tuples in their gap between 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 news headlines obtained from many news websites and presented in one place. There is no set of available news topics, instead the topics about what is important can change over time. Therefore, setting up training data and use classification is not feasible. Instead, clustering can be used, finding important topics in the latest news articles and groups them together. The algorithms tries to find the best grouping of news, meaning that the news objects in one group have a high similarity with each other, while they are very different to the news objects in other clusters. Clustering is used in all disciplines from biology, security, business intelligence and the web and comes with a wide range of algorithms. These algorithms can be categorized in the following four categories, each with advantages and disadvantages regarding the input data set. Partitioning methods: Partitioning methods are the most popular clustering methods, trying to find clusters of spherical shape. A distance function is used to measure similarity and dissimilarity among the clusters. Most popular algorithms are k-Means and k-Medoid. 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 and merging or splitting subtrees, respectively. Similar to partitioning-based methods, hierarchical cluster algorithms find spherical clusters. Density-based methods: Finding clusters of arbitrary shape, such as oval clusters or S shape clusters, partitioning and hierarchical methods are limited. Density-based clustering techniques 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, i.e. a cluster. Grid-based method: 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 grid-based algorithms, since it is both density and grid based.

3.3.4 Outlier Detection

Outlier detection are techniques to find data objects that behave in a different way than the majority of data objects. For a commercial system, outliers can be clients spending much more money than the average client. In all kind of fraud detection systems, outlier detection is very important, e.g. for medical care or security. Obviously, there is a strong correlation between outlier detection and clustering. Data objects that do not fit into a cluster are potential outliers, and therefore clustering techniques can be used for outlier detection. However, their main purpose is to find clusters, whereas outlier detection algorithms are specialized in finding outliers. Therefore, clustering techniques are often optimized to omit all data points in dense region and only search for outliers, leading to a more effective search. Apart from unsupervised learning, supervised learning can be applied for outlier detection as well. First, data is labelled as normal or outlier, and therefore 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, and therefore typical classification algorithms often have to be adapted and optimized for outlier classification. Apart from adopting clustering and classification techniques, statistical and proximity-based methods are also used for outlier detection.

4 The k-Means Clustering Algorithms

4.1 Motivation

In this work, the data mining algorithm k-Means will be implemented in HyPer using the operator model, in order to show a proof-of-concept implementation.
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 cluster. 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 Lloyed algorithm, which is a local search solution to this problem.
The algorithm is one of the most popular data mining algorithms. 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 PageRank.

4.2 The Lloyd Algorithms

In the following paragraph we explain the Lloyed algorithm, usually referred as k-Means in literature and in this work as well. 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.
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 Teilmenge of Rd, choose k

Centers c to minimize the sum of squared error function,

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

When found the centers, we know which data points belongs to the same center and we have found our clustering.
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.

4.3 Example

$x_0(3,9), x_1(2,5), x_2(5,8), x_3(7,5), x_4(4,2)$
Initial center of the Clusters: $c_0(3,9), c_1(4,2)$

Table 4.1: Computations in Iterations 1.

Distance	$c_0(3,9)$	$c_1(4,2)$	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

4.4 The k-Means++ Initialization Strategy

This simple algorithm terminates in practice very fast and provides mostly good result, even though the clustering can be arbitrarily bad for some data sets. Particularly, the random choosing of the initial cluster centers can lead to a bad clustering, which cannot

Table 4.2: Computations in Iterations 2.

Distance	$c_0(4,8.5)$	$c_1(4.33,4)$	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

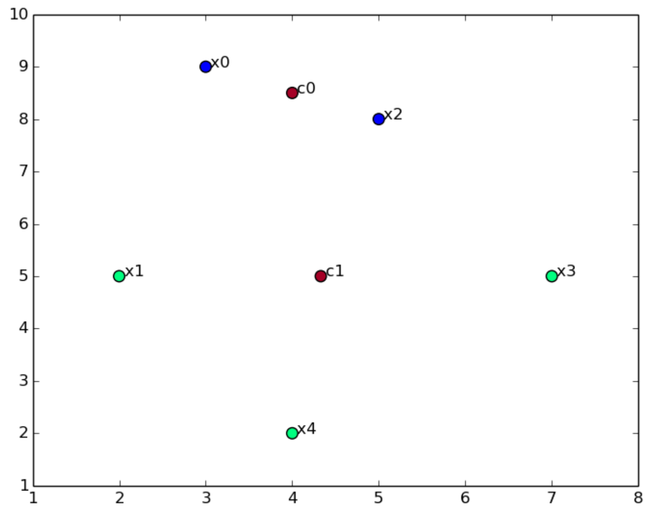

Figure 4.1: k-Means Example Clustering.

be changed during the clustering process. Therefore, propose a variation to the k-Means initialization strategy, called k-means++. The centers are still chosen randomly from the data points, but the ata points are weighted according to their distance from the closest center already chosen. Therefore, the probability that of choosing data points as center that are far away get higher, leading to improvements, both 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.

Most clustering tools implement a k-means++ initialization strategy for k-means, therefore we will also test k-means++ on HyPer.
Since the algorithm is very popular and easy to understand, makes it suitable for the first implementation of a data mining algorithm on HyPer. All data mining tools we looked at (see related work) 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. Since the algorithm is well-defined, we can expect that the data mining tools are implementing the actual k-Means algorithms, only adapting to their programming model. Another nice fact is that many data mining tools also implement k-Means++ as initialization strategy, another thing that can be compared. 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, and it will be interesting to compare these results to the results of big data computing, such as MapReduce or Apache

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

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 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

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.

5.4 Databases KDD

6 Research Approach

While all of the presented environments are frequently used by data scientists, they demonstrate one decisive drawback: Before executing data mining algorithms on the datasets, the data first has to be fetched from the database and transformed into a format readable by these tools. With HyPer, we take a different approach: Instead of pulling the data from the database, we push the algorithms to the database. 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 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 with data mining functionalities, exploiting the performance of a database for computational operations and building a general-purpose system for OLTP, OLAP and data mining queries. Such a system should outnumber above systems in both performance and usability. As proof of concept, the well-known k-Means algorithm is used and will be implemented in HyPer. K-Means is one of the most popular data mining algorithms and available on all presented platforms. Since the algorithm is relatively simple and straightforward, a good comparability between different tools is given. Apart from performance evaluation, the implementation of k-Means should also demonstrate the possibilities of implementing data mining algorithms in HyPer and help to detect patterns and building blocks for further algorithms.

7 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.

7.1 HyPer Operator Fundamentals

7.1.1 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: 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 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 7.1.

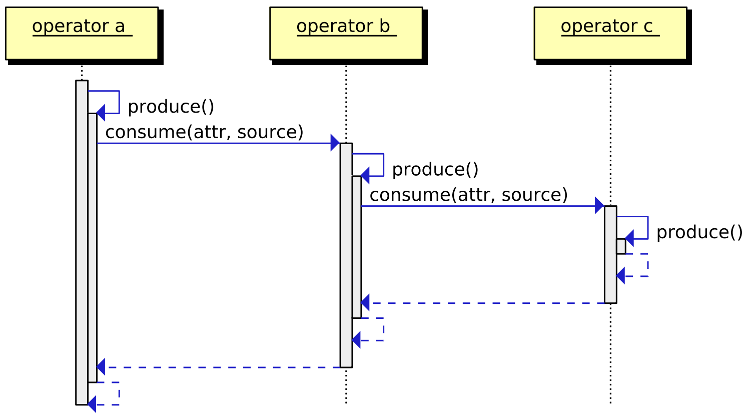

Figure 7.1: Consume Produce Sequence Diagram.

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.

7.1.2 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 [**LLVM**]. 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++ texttttsort 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.

7.2 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.

7.3 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 7.2 shows.

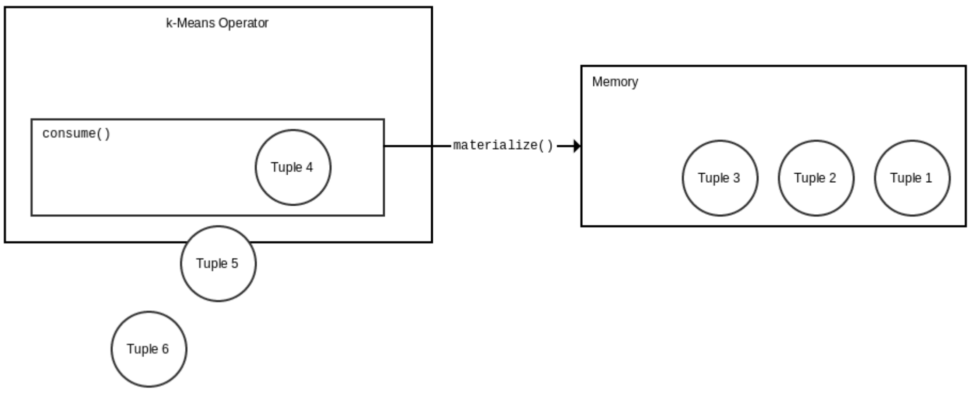

Figure 7.2: Data Materialization of Incoming Tuples.

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 7.3 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 7.4.

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.

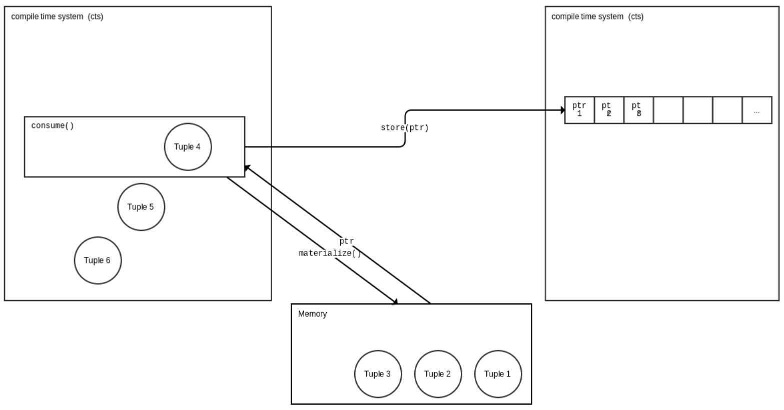

Figure 7.3: Data Materialization using the Runtime and Compile Time System.

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.

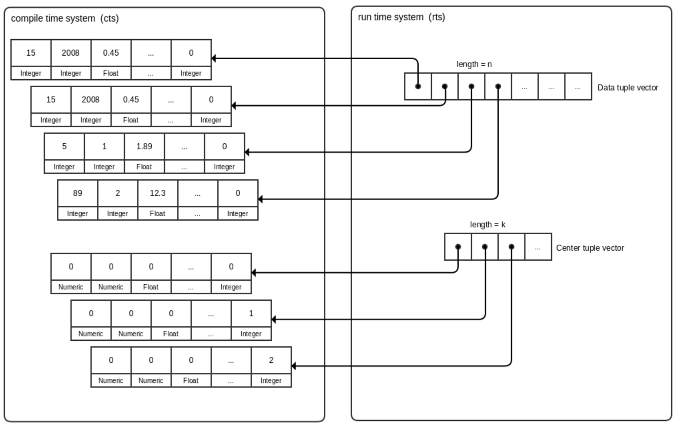
compile time system (cts)
run time system (rts)
15 2008 0.45 ... 0
Integer Integer Float ... Integer
15 2008 0.45 ... 0
Integer Integer Float ... Integer
5 1 1.89 ... 0
Integer Integer Float ... Integer
89 2 12.3 ... 0
Integer Integer Float ... Integer
0 0 0 ... 0
Numeric Numeric Float ... Integer
0 0 0 ... 1
Numeric Numeric Float ... Integer
0 0 0 ... 2
Numeric Numeric Float ... Integer
length = n
Data tuple vector
length = k
Center tuple vector

Figure 7.4: k-Means Data Materialization in Detail.

7.4 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++.

7.4.1 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 7.5 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.

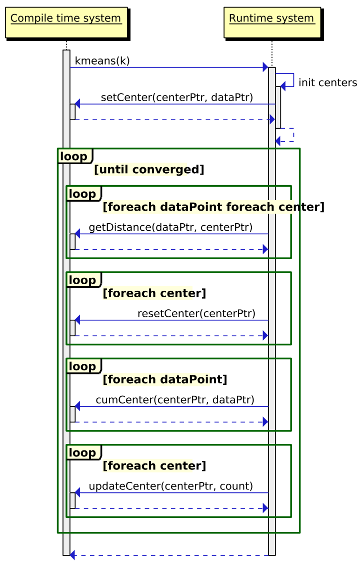

Figure 7.5: The k-Means Algorithm - C++ driven.

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 7.1 shows. The next sections presents an implementation that prevents the algorithms from too many calls between the two systems.

Table 7.1: Generated Function's Calls per Iteration.

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

7.4.2 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 7.1, 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 7.6 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.

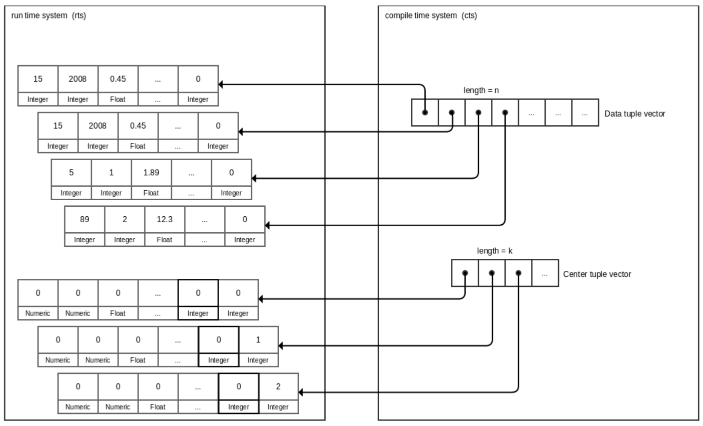

Figure 7.6: k-Means Data Materialization for the LLVM Implementation.

The algorithm is depicted in a sequence diagram in Figure 7.7 and shows the indirection of the calls compared to the sequence diagram in Figure 7.5: 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

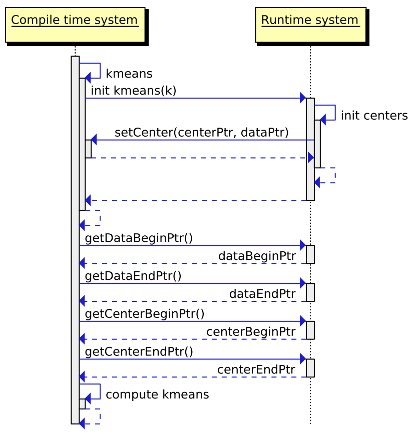

Figure 7.7: The k-Means Algorithm - LLVM driven.

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.

7.4.3 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 [**fisheryates**].
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.

7.5 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 7.8: 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.

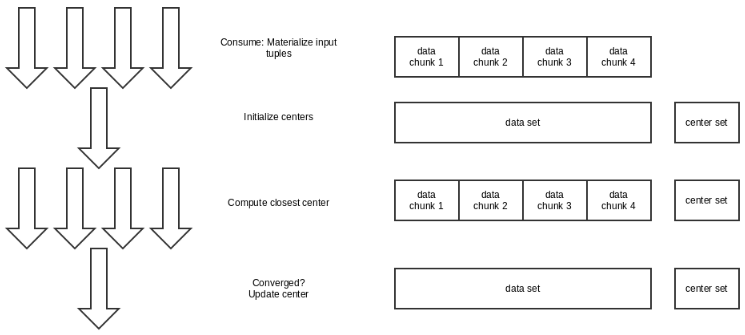

Figure 7.8: The k-Means Algorithm as Parallel Operator.

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.

8 Evaluation

In this section we provide an extensive experimental evaluation comparing the different HyPer implementation against each other, as well as against existing clustering solutions. As a proof of concept, a real world data set and three synthetic data set are used. The data sets are selected in a way that most real world use cases are covered. Intel(R) Xeon(R) CPU X5570 @ 2.93GHz If not stated otherwise, all experiments have been performed on a workstation equipped with 16 cores and 64 MB of main memory.

8.1 Data Sets

3D network: The first data set is a 3D spatial road network data set modelling the road network in North Jutland, Denmark. It is a real world data set available at the UCI Machine Learning Repository and consists of 434874 data points with four dimensions: An id for the road segment, latitude, longitude and altitude. The id is a large integer, the other dimensions are floating point numbers. The size of the data set is 20673913 bytes, therefore a rather small data set.

High dimensional data set: The second dataset is a synthetic data set created on a random uniform distribution. It consists of 50000 data points with 50 dimensions. All dimensions are floating point numbers. The size of the data set is 10000000 bytes, hence also a rather small data set.

Medium size data set: The data set consists of 15 Million data points with four dimensions. It is also a synthetic data set generated on a random uniform distribution. The four dimensions are floating point numbers. The size of the data set is 240000000 bytes and presents the medium size data set in this experiment section.

Large size data set: Finally, the large data set consists of 150 Million data points with ten dimension. It is also synthetically generated by a random uniform distribution of floating point numbers. The size of the data set is 6000000000 Million bytes.

Table 1 depicts the data sets used in this section.

Table 8.1: 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

8.2 Used Tools

As already described in the related research section, there are many tools implementing a kMeans clustering. One of the main criteria of the selection of tools is to make the results comparable. Therefore, 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-Won algorithm, an improvement of the standard Lloyed algorithm. Since kMeans is a non-deterministic algorithm, the number of iterations is another very important criteria to make the results comparable. Running time of kMeans depends much on how many iterations the algorithm has to made, which depends on a random initialization. Therefore the running time should be relative to the number of iterations. Unfortunately, neither Elki nor Scipy's kMeans implementation provide the number of iterations as result. Therefore, a fair comparison is not possible. weka, big data? weka arrf format -> loading time, results etc. R and julia provide good configuration possibilities as well as a result set that contains information about number of iterations, total cost etc. Although both tools are written in a high level language, critical code parts are written in C, C++ and Fortran, without the overhead of an entire database system. Therefore it will be interesting to compare Hyper with julia and R.

8.3 Serial Implementation

Before we look at experiments comparing the performance of HyPer with existing tools, first we compare the serial implementation approaches implemented in HyPer. As show in the previous section, HyPer consists of a runtime system and a compile time system, with C++ and LLVM code, respectively. The two presented implementations vary in the main focus of implementation, i.e. is the kMeans algorithms mostly implemented in C++ or in generated LLVM code. Therefore we compare these implementations first. We measure both the compilation and execution time of HyPer. The compilation time is the time needed for generating LLVM code, while the execution time is the actual algorithm. Since the compilation time can be seen as a setup time, we expect the execution time to be much larger. For each data set, the two HyPer implementations are tested for cluster number k 3,10 and 20. For each k, the algorithms was executed 100 times with a maximum number of iterations of 10. The number is than the median time per iterations in seconds. Table: Network shows the result of the two HyPer implementations for the network data set. The compilation time is similar, while the execution time grows with the cluster number k. Figure Network shows the results as stacked bar charts. The blue area is the compilation time, and the green area is the execution time.

Table 8.2: 3D Network - Time per Iteration.

	HyPer C++		HyPer LLVM	
k	compilation[s]	execution[s]	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

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 still two times the execution time of the LLVM version and for k = 20 it is 1.5 times, respectively.

This gives us the first interesting results. First, the compilation time does not differ for between the llvm and c++ version for the network data set. This is interesting since the C++ version is implementing kMeans in C++ using only LLVM functions, while the second version implements everything in LLVM. Nevertheless, the main functions remain in LLVM for both implementations. Computing the distance, updating the centers, all those functions are generated LLVM code. In C++ these functions gets 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. How about the execution time? Here, the LLVM version shows a far better time per iteration, actually 4, 2 and 1.5 times faster compared to the C++ version for k equals 3, 10 and 20, respectively. There are two explanations: As already described, the C++ implementation has many function calls between the compile time and the runtime system, while the LLVM version does not. These calls are expensive. why? The second advantage is that the algorithm is compiled in LLVM

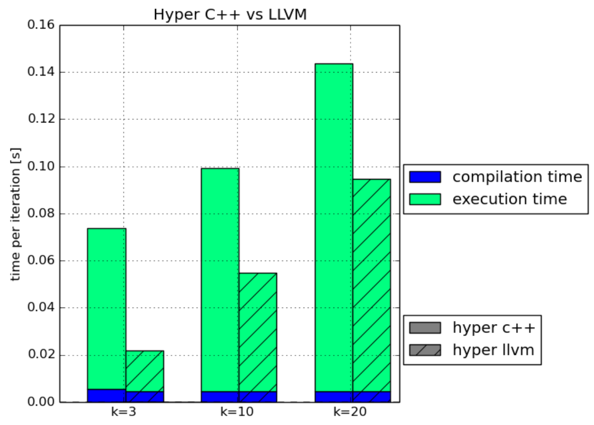

Figure 8.1: 3D Network - Time per Iteration.

code, which produces a very efficient code, optimized on a lower level than C++ code.

Table 8.3: High Dimensional - Time per Iteration.

	HyPer C++		HyPer LLVM	
k	compilation[s]	execution[s]	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: high dim shows the same experiment for the high dimensional data set. This time, the compilation time differs between the c++ and the llvm implementation: The c++ version takes 150The increase of compilation time results in 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. why compilation time differ? We also run the C++ and the LLVM implementation on the medium and large data set. The results are shown in table med and table large. The compilation time is very low again, although the number of instances is much higher compared to the high dimensional data set. This proves that the compilation time is dependent on the dimensionality of the data, and not on the data size.

Figure 3 and 4 depict the compilation and execution in a bar chart. Since the compilation time does not depend on the number of instances but on the dimensions, which are low in that example, the compilation time is not even visible. However, execution time increases with the number of instances. As in all the previous experiments as

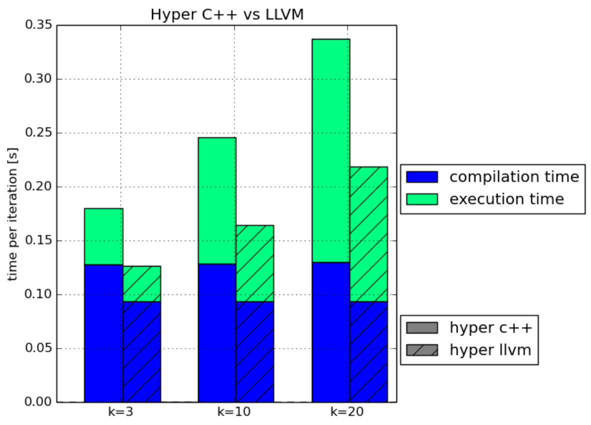

Figure 8.2: High Dimensional - Time per Iteration.

Table 8.4: 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

well, execution time of the LLVM implementation is much faster compared to the C++ version. Regarding the large data set, the LLVM version is faster by around 20 seconds per iteration.

In conclusion, the compilation time is does not grow much if the number of instances gets large but if the number of dimensions grows. Usually, the execution time outnumbers the compilation time by several factors. The only exception is a small, high dimensional data set as shown in the experiment. Here, the compilation can be the bottleneck of the kMeans algorithm.

Another result is that the execution time, growing by number of instances, is much faster for the LLVM implementation since there is almost no communication between C++ and LLVM code. On the other hand, the compilation time is roughly the same. Therefore the LLVM version was always the better choice regarding the running time of the kMeans algorithm.

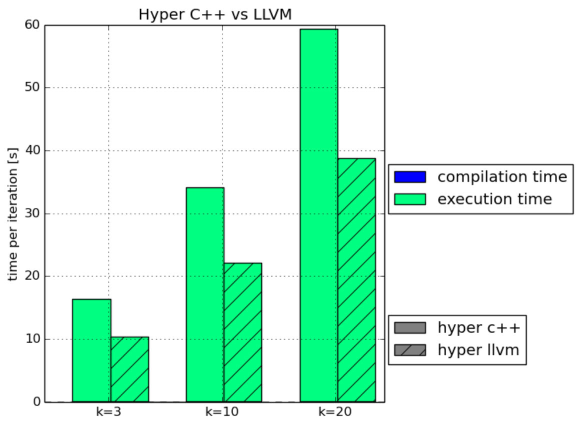

Figure 8.3: Medium Size High Dimensional - Time per Iteration.

Table 8.5: 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

8.4 Performance Test

In this section we compare HyPer to R and julia's implementation of kMeans. All programs are executed in serial. As algorithm, the Lloyed algorithm with a maximum iteration number of ten is chosen. Apart from the large data set, the algorithm is executed 100 times for k equals 3, 10 and 20, respectively. For the large data set, the algorithm is only executed ten times. The result is presented as the time per iteration.

The result of the network data set is presented in Table: network. The median, the 90th and 95th percentile are listed to show the variance of the data. As we already know, the HyPer LLVM implementation outnumbers the C++ version. Julia is the slowest, while R competes quite well with the LLVM version and is even faster for k equals 10 and 20. Figure 1 shows the results as a boxplot. However, all tested programs differ in time per iterations in a few hundred milliseconds, therefore the differences are not very significant.

Table 8.6: 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

Table 8.7: 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	0.09	0.08	0.10	0.14	0.02	0.06	0.10
50	0.31	0.35	0.35	0.08	0.07	0.10	0.08	0.10	0.14	0.02	0.06	nnnn

8.5 Parallel Implementation

So far we compared the serial execution of the HyPer k-Means operator with julia and R. In 7.4 we figured out that the LLVM implementation outperforms the C++ version on all data sets we tested on. In 7.5 we compared the performance against julia and R and showed that HyPer is able to compete with the existing solutions.
However, as data sets grow serial execution takes more and more time, making real-time data mining almost impossible. To improve performance, HyPer operators benefit from multi-threaded execution. In this section we compare the parallel implementation of the HyPer kMeans operator with the LLVM version and the R implementation. For comparison, we use the medium and the large data set.
Table x depicts the time per iteration for k =3, 10 and 20. As before, the medium, the

Table 8.8: 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	0.09	0.08	0.10	0.14	0.02	0.06	0.10
50	0.31	0.35	0.35	0.08	0.07	0.10	0.08	0.10	0.14	0.02	0.06	nnnn

Table 8.9: 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	0.09	0.08	0.10	0.14	0.02	0.06	0.10
50	0.31	0.35	0.35	0.08	0.07	0.10	0.08	0.10	0.14	0.02	0.06	nnnn

Table 8.10: 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	0.09	0.08	0.10	0.14	0.02	0.06	0.10
50	0.31	0.35	0.35	0.08	0.07	0.10	0.08	0.10	0.14	0.02	0.06	nnnn

90th and 95th percentile of 100 iterations on a 16 core machine are presented. For k equals 3, we see that both the R and the LLVM version are faster than the parallel version. However, with growing k, the parallel version outperforms R and the LLVM version. This is also shown in the bar chart in Figure xx.

The reason for performing better than R and LLVM for growing k is the the running time of the parallel execution is almost independent of k. While R and the LLVM version grow by seconds as k grows, the parallel version grows insignificantly by around 200 milliseconds.
Unfortunately, for the medium size data set and small k, the parallel version cannot compete with the LLVM and R algorithm. First, we have to take into account the overhead of the parallel process. Second, we are using the slower C++ version of the HyPer kMeans operator. Even though parts of it are parallelized, we still have all the

Table 8.11: 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	0.09	0.08	0.10	0.14	0.02	0.06	0.10
50	0.31	0.35	0.35	0.08	0.07	0.10	0.08	0.10	0.14	0.02	0.06	nnnn

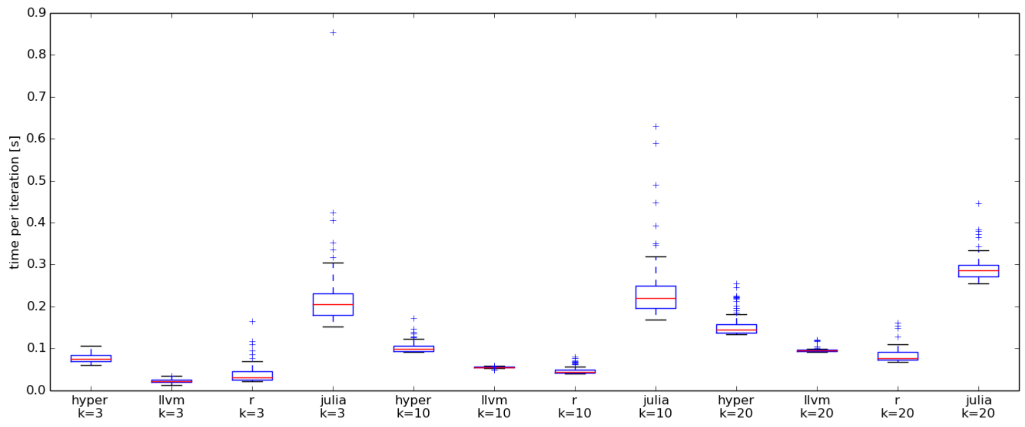

Figure 8.4: 3D Network - Time per Iteration.

Table 8.12: 3D Network - Time per Iteration.

	R			HyPer C++			HyPer LLVM		
k	3	10	20	3	10	20	3	10	20
50	0.03	0.04	0.08	0.08	0.10	0.14	0.02	0.06	0.10
90	0.06	0.06	0.09	0.08	0.10	0.14	0.02	0.06	0.10
95	0.08	0.07	0.10	0.08	0.10	0.14	0.02	0.06	nnnn

downsides of many function calls between the runtime and the compile time system and of the high-level C++ constructs.

Table x shows the same experiment on the large size data set. The table shows the median, the 90th and the 95th percentile. For better visualization, the bar chart in Figure x depicts the median value of the R, LLVM and parallel execution.

This time, however, the parallel version outperforms the R and LLVM version for all k's. Thus, the performance difference gets remarkably as k grows. We experience the same effect as for the medium size data set: A change in k affects only slightly the performance of the parallel version, while it decreases the performance of the serial implementations.
In the following paragraphs we look at the parallel execution in greater detail.

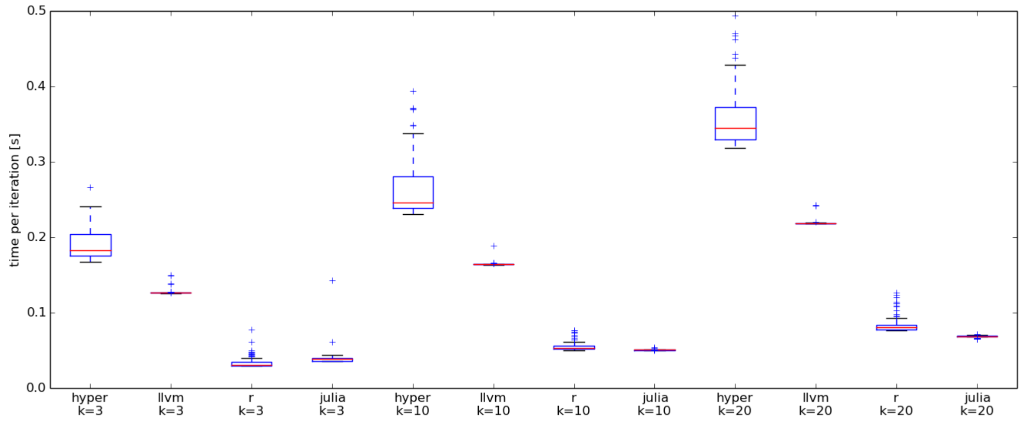

Figure 8.5: High Dimensional - Time per Iteration.

Table 8.13: 3D Network - Time per Iteration.

	R			HyPer C++			HyPer LLVM		
k	3	10	20	3	10	20	3	10	20
50	0.03	0.04	0.08	0.08	0.10	0.14	0.02	0.06	0.10
90	0.06	0.06	0.09	0.08	0.10	0.14	0.02	0.06	0.10
95	0.08	0.07	0.10	0.08	0.10	0.14	0.02	0.06	nnnn

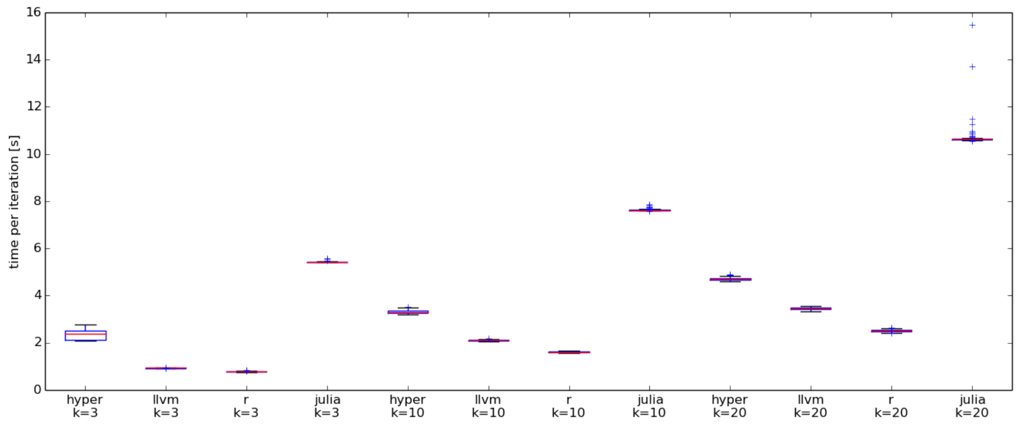

Figure 8.6: Medium Size - Time per Iteration.

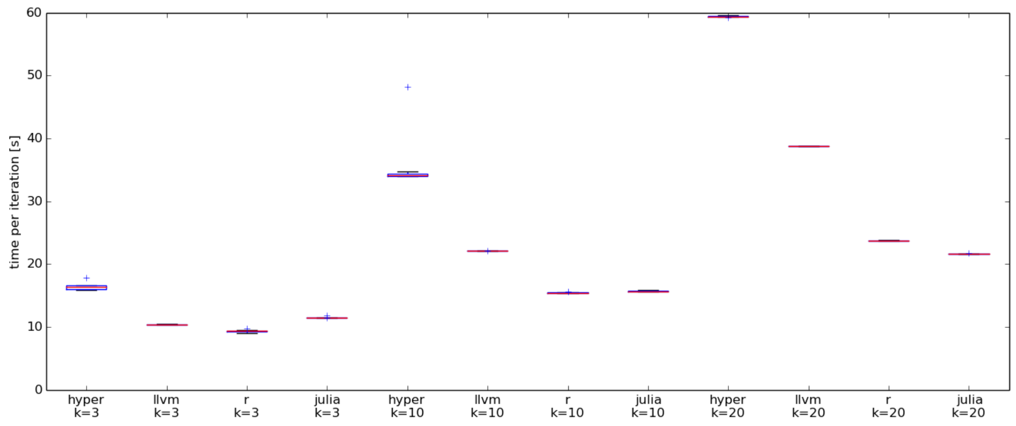

Figure 8.7: Medium Size HD - Time per Iteration.

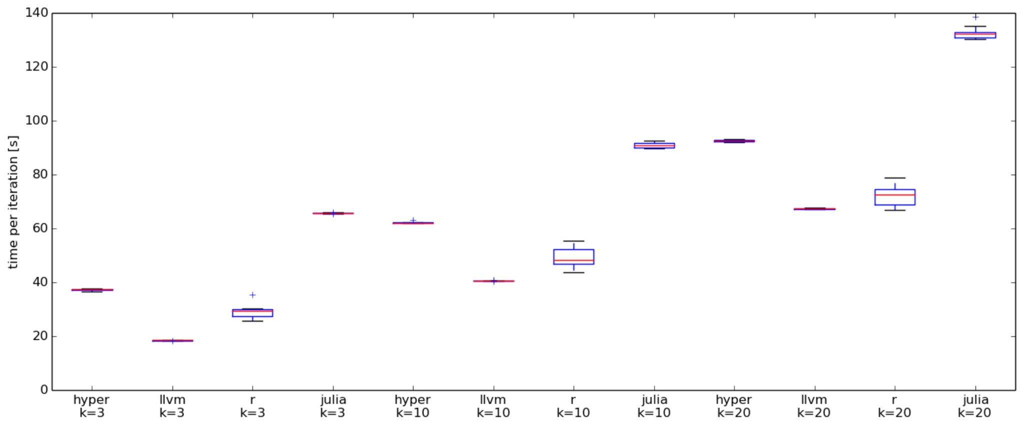

Figure 8.8: Large Size - Time per Iteration.

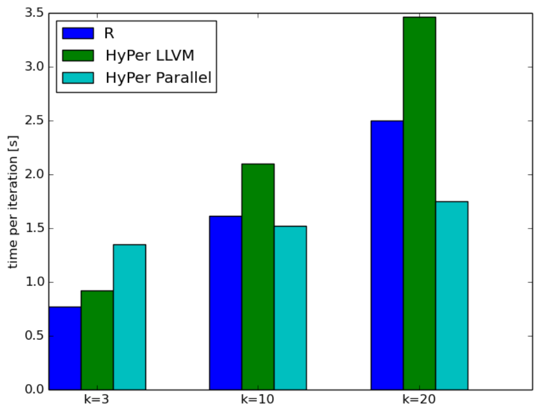

Figure 8.9: Medium Size - Time per Iteration.

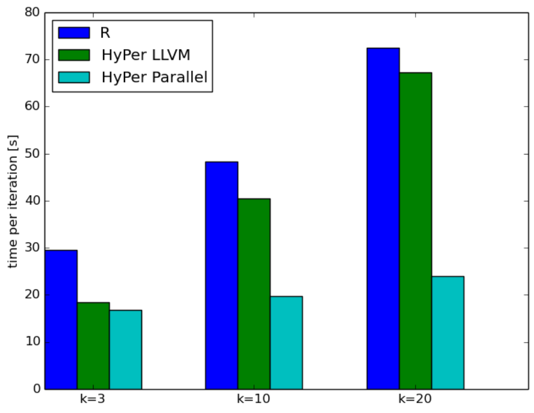

Figure 8.10: Large Size - Time per Iteration.

9 Conclusions and Future Work

9.1 Conclusions

9.2 Future Work

k median

Glossary

computer is a machine that. . . .

Acronyms

TUM	Technische Universität München.

List of Figures

List of Tables