

Interactivity, Scalability and Resource control for efficient KDD support in DBMS

Matthias Gimbel and Michael Klein and P.C. Lockemann

Universität Karlsruhe, Fakultät für Informatik,
Am Fasanengarten 5, 76128 Karlsruhe, GERMANY

Abstract. The conflict between resource consumption and query performance in the data mining context often has no satisfactory solution. This not only stands in sharp contrast to the need of the analysts for interactive response times, but also makes the seamless integration of data mining operators into common multiuser database systems a difficult and (so far) not very prosperous task. We believe that an efficient solution to the problems of database support for KDD has to affect the whole query processing from the data access on disk up to the complex data mining operators. The basic idea of our framework is to provide resource efficiency and interactivity through precise control over the order in which data is processed from the index structure through the whole query tree. It consists of an index that is basically an extension of the UB-Tree and allows to translate efficient data access patterns into various data orderings. Our KDD-algebra exploits these orderings to allow the control of resource consumption in each operator while providing interactive response times and pipelined query processing. This paper describes the framework and shows its benefits in preprocessing and in the parallel and interactive detection of outliers.

1 Introduction

Database support for decision support applications and especially for data mining tasks has to deal with the dilemma between the huge amounts of data to be analyzed, the complexity of data mining algorithms and the demand of the analysts for interactive response times. The mentioned factors cannot easily be fulfilled by traditional operator algebras and, up to now, have effectively prevented the true integration of data mining support into standard database system kernels.

The demand for interactivity stems from the iterative and explorative nature of KDD processes, in which the analyst analyzes results, modifies his queries and reevaluates them in order to find segments of data where interesting facts can be found and isolate the nuggets of knowledge. This demand forbids one standard database approach in case of resource shortage: Splitting of query processing in successive tasks and swapping out intermediate results in order to save resources. The explorative nature of this process leads to the next problem: The well-known database optimization and partitioning techniques stemming from traditional application areas, where pre-planning is possible because of static access patterns, lose much of their value. So, at first glance the dilemma only seems to be solvable with tremendous computing power and main memory resources. But even classic data parallel execution schemes promising adequate scalability often suffer from the same problems in this context.

In this paper, we describe what is necessary to make database support for large scale query processing in KDD resource-efficient, scalable and interactive. The basic idea is to provide an index structure that allows to efficiently provide several data orderings that can be used by the KDD query algebra to compute final results on partial data while at the same time adhering to resource limits without the need of swapping out data. This allows for interactive query processing with controllable resources and especially for non-blocking pipelining in parallel environments.

The paper is organized as follows. After a survey of related work, in section 3 we will first analyze common preprocessing operators in our KDD algebra to find out, what properties of the processed data stream can be useful to achieve our goals. In section 4, we will describe the concept of data stream quality, which is the solution to these problems. Section 5 shows, how such qualities can be provided efficiently. After a short demonstration of the benefits our framework provides in query processing we describe in section 6, how interactive data mining algorithms can benefit from this framework to allow for parallel and interactive query processing in KDD. A summary and a survey on open problems and further work close the paper.

2 Related Work

In the recent years, database support for KDD concentrated on the development of special languages to express queries for data mining results. Starting with the query language DMQL [5] and leading up to the formulation of a sophisticated abstract data mining algebra in [7], there have been many attempts to design languages for data mining queries. What's still missing is the integration of KDD with database system kernels. So we take a look, where scalability, interactivity and resource questions play a role in database literature.

One way to achieve scalability is parallelization. The majority of the classic database approaches rest on the assumption of expensive inter-node communications. This resulted in the preferential use of data parallelism (node splitting), which means that query processing is mainly driven by the physical placement of data. Pure data parallelism leads to scalability problems in case of data skew and limits interactivity by splitting query processing in tasks. Pipelining was seen mainly as a technique for shared memory (SMP) machines, and even there the problems were regarded as particularly serious [3]. Nevertheless, there has been substantial work in the area of pipelining implementations of complex operators [15], cost models [16, 12] and optimization strategies [9].

With the advent of interactive, adaptive and online query processing aiming at giving the user early insight into query results, there has been an increasing interest in resource efficient and blocking-free query processing over the past three years. The work in [6] extends the pipelined hash join with out-of-core techniques to limit its massive resource consumption. The paper proposes to swap parts of the hash tables to disk and defines two processing stages: a regular stage that works similar to the original algorithm, and a clean-up stage where the portions of the hash table swapped to disk are processed. The work in [13] extends this with a third reactive phase, which is activated when regular input is blocked. Overall these various phases lead to complex pipeline scheduling problems solved by the specialized scheduling scheme [14]. Another di-

rection in online query processing is trading interactivity for accuracy. The goal is to provide early results that reflect the trends within the data, and do so with an increasing level of confidence as query processing progresses. The most prominent operator in this context is the ripple join [4], which provides the possibility to adjust the rate at which the inputs are processed to allow the user to favor the "most interesting" input. The area of adaptive query processing deals with the goal of providing partial results under conditions of incomplete input data at a limited bandwidth. A good survey of the area of stream processing can be found in [1].

Data mining algebra, pipelining, stepwise accuracy and partial results seem to be orthogonal concepts on the way towards interactivity and resource efficiency. We believe, however, that our approach is capable of adding novel aspects to the discussion in the aforementioned fields.

3 The source of the Problems

Starting from the conviction that a true integration of data mining and database systems leads over the integration of KDD-specific operators into the query algebra of database systems, we first analyzed common preprocessing operators used in KDD for sources of resource consumption and lacking interactivity because of blocking. As seen in Figure 1, this includes standard database operators like select, join, etc. as well as KDD specific operators (sample, history etc.). For reasons of space, in this paper we limit to the following four operators:

- GROUP, which calculates an aggregation function f of tuples coinciding on the values of attributes in some fixed list T_U (like in the known SQL construct)
- JOIN, which joins two relations using the join attributes T_U and T_V
- SAMPLE, which provides a randomly chosen sample of g tuples from the relation
- NORMALIZE, which does a linear rescaling of attribute A_{R_i} so that the values fit into a given interval $[a, b]$.

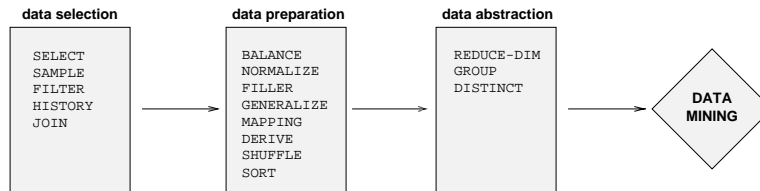


Fig. 1. KDD operators in our Algebra

To identify the blocking and resource intensive steps, we have to look deeper into the implementations of those operators. Since the various algebraic operators (or more precisely: their implementations) differ widely in their complexity and resource cost, it seems to make sense to concentrate on splitting the implementations of the complex operators.

One possible implementation for the GROUP operator is the following collect-BlockGroup:

```
collectBlockGroup( $R, T_U, f$ )
1. [collect] Collect tuples with the same value in  $T_U$  in a
   hash table and provide them one after the other.
2. [block] Calculate the aggregated attributes block wise
```

Here, the collect step can be separated from calculation step. Note that the second step runs with little resource demand in linear time and is not blocking.

The sortMergeJoin implementation for the JOIN operator works similar. It sorts the tuples of both relations according to the join attributes and merges them like a zipper:

```
sortMergeJoin( $R, S, \text{join attributes } T_U, T_V$ )
1a. [sort] Sort the first relation  $R$  by  $T_U$ 
1b. [sort] Sort the second relation  $S$  by  $T_V$ 
2. [merge] Merge the relations
```

Here also, the sorting steps can be separated. Then, the following merge step can be executed, again with little resource demand, and is not blocking. Sorting, however, is blocking.

If we consider the countPickSample implementation for the SAMPLE or the count-MinmaxNormalize implementation for the NORMALIZE operator we observe a similar result:

```
countPickSample( $\text{Relation } R, \text{sample size } g$ )
1. [count] Determine the number  $M$  of tuples
2. [pick] Let 1 of  $n = \frac{M}{g}$  tuples pass
```

```
countMinmaxNormalize( $R, A_{R_i}, \text{new interval } [a, b]$ )
1. [count] Determine the minimum and maximum of the values of  $A_{R_i}$ 
2. [minmax] Rescale the value of  $A_{R_i}$  to fit into  $[a, b]$ 
```

Again, the second step can be performed in a non-blocking manner with little resource consumption whereas the first step is blocking.

We may summarize our observations as follows.

Dividing the operator implementations is often possible

Many of the complex operators seem to possess algorithms that process the data stream in two steps. The first step serves to prepare the data for the following steps, which can process it in a fast, non-blocking manner with little resource consumption.

Data preparation steps are limited

There seem to be only a few different operators for data preparation steps. In the examples presented there were **ordering** according to selected attributes (achieved by SORT), **grouping** by selected attributes (achieved by COLLECT), **counting tuples** and determining the **range of values** of an attribute. In principle, it is possible to treat those preprocessing steps as separate operators that can be executed by different implemen-

tations. What is important is that the preparation levels attained by executing a single pre-processing step may be useful in more than one subsequent processing step, provided it is not destroyed by the following step. This opens the chance to avoid the preparation step in some cases.

Avoiding data preparation is the clue for interaction

Blocking operators can slow down the response time and are therefore not suitable for interactive data processing. Hence, our goal must be to find a way to substitute blocking data preparation steps by non-blocking ones or to avoid them altogether. The idea is to strictly keep track of the already achieved preparation levels in the pipeline. By cleverly combining and reusing these preparation levels we hope to avoid many costly preparation steps and go right on to process the tuples in the non-blocking calculation step. The next section introduces the necessary concepts.

4 Data Stream Quality

The previous section served to provide an intuitive understanding of the problem. What we need now is a systematic approach to describe the state of data preparation. We introduce data stream quality as the central concept. The previous section already indicated some useful data qualities: sortedness, continuity, and the knowledge of cardinality or extremal values. Below we formalize these qualities and present theorems that capture useful relationships between the different qualities.

4.1 Definitions

To describe continuity and sortedness, we assume equality (denoted by $d_1 =_{T_U} d_2$) and (lexicographic) ordering on tuples (denoted by $d_1 <_{T_U} d_2$) with regard to a list of attributes T_U as given without formal definition. In consequence, we also use \leq_{T_U} in the expected manner. With these definitions, we are able to formalize the data qualities:

The most important data stream quality is the lexicographic order of the tuples in a data stream with regard to a list of attributes, for example resulting from the preparation of the SORT operator:

Definition 1. A data stream $D = [d_1, \dots, d_M]$ has data stream quality **sorted ascendingly** regarding to a list T_U of attributes, in characters $\mathbf{S}^+([T_U])$ iff $i < j \implies d_i \leq_{T_U} d_j$ for all $i, j \in \{1, \dots, M\}$.

In analogy to that, the decreasing order (\mathbf{S}^-) can be defined.

In a continuous data stream, all tuples having the same T_U values occur in a consecutive block. As described in the previous section, the COLLECT operator can produce this quality.

Definition 2. A data stream $D = [d_1, \dots, d_M]$ has data stream quality **continuous** regarding to the list $T_U = [A_{U_1}, \dots, A_{U_n}]$ of attributes, written as $\mathbf{C}([T_U])$ iff for all $i, j \in \{1, \dots, M\}$, $d_i = d_j \implies d_i =_{T_U} d_k$ for all k with $i \leq k \leq j$.

As a special case of continuity, we have uniqueness. We call a data stream **distinct**, if no two tuples are equal regarding to T_U .

Definition 3. A data stream $D = [d_1, \dots, d_M]$ has the data stream quality **distinct** regarding to the list $T_U = [A_{U_1}, \dots, A_{U_n}]$ of attributes, written as $\mathbf{D}([T_U])$ iff $d_i =_{T_U} d_j \implies i = j$ for all $i, j \in \{1, \dots, M\}$.

In contrast to the above data stream qualities, the following ones are not related to the ordering of the tuples in the stream, but provide information about tuple counts and value ranges. Because of that, they describe the data stream as a whole.

The data stream quality achieved by the COUNT-operator is the knowledge of the tuple count. It is independent from a list T_U of attributes.

Definition 4. A data stream $D = [d_1, \dots, d_M]$ has the data stream quality **known tuple count**, in characters **num** iff M is known with the arrival of d_1 .

Similar to the knowledge of the number of tuples, the knowledge about extreme values increases the data stream quality of a stream.

Definition 5. A data stream $D = [d_1, \dots, d_M]$ has the data stream quality **known attribute range**, in characters **min** or **max** iff $\min_{T_U}(D)$ or $\max_{T_U}(D)$ is known with the arrival of d_1 .

4.2 Theorems

In the following section, some of the dependencies between the different data qualities will be presented as theorems. The proofs are easily seen. T_U denotes a list of attributes.

Theorem 1. If a data stream is sorted regarding to T_U , it is also sorted with regard to every prefix of T_U .

Theorem 2. If a data stream is sorted regarding to T_U , it is also continuous with regard to T_U .

Theorem 3. If a data stream is distinct regarding to T_U , it is also distinct with regard to every permutation of T_U .

Theorem 4. If a data stream is continuous regarding to T_U , it is also continuous with regard to every permutation of T_U .

Theorem 5. If a data stream is distinct regarding to T_U , it is also distinct in an extension of this list by further attributes.

Theorem 6. If a data stream is distinct regarding to T_U , it is also continuous in T_U .

Theorem 7. If a data stream is increasingly (decreasingly) sorted, its minimum (maximum) is also known with the arrival of the first tuple.

4.3 Data stream quality requirements and transformations

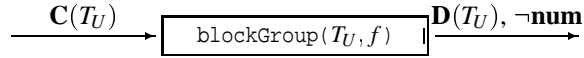
Given the data stream qualities, we now examine how different operator implementations can exploit them. We concentrate – as an example – on the GROUP operator as a complex operator that is particularly important to KDD. For each implementation we present the minimum input data stream quality that is necessary for a correct working of the implementation as well as the change of data quality as a result of the transformation by the implementation.

The GROUP operator has three implementations which differ in their minimum input data stream quality, their memory requirements and their blocking capabilities. The traditional implementation is hashGroup, which uses a hash table to calculate the aggregation results for each combination of T_U .



For this implementation no data stream quality requirements are needed. Because each combination of T_U values produces exactly one result tuple, the output stream is distinct with regard to T_U . However, the knowledge of the number of tuples is lost. Furthermore, the hash function usually destroys all existing orders. The disadvantages of this implementation are the great amount of memory necessary to store all results at the same time and the blocking output (denoted by the bar near the end of the box), which restricts the use in a pipeline.

If the data stream quality at the input is at least continuous with respect to the list of the grouping attributes, the blockGroup implementation can be used.



It applies the aggregation function block by block and therefore does not need to store more than one block of continuous tuples at the same time. Therefore, the amount of required memory is usually far less than with the hashGroup implementation. Moreover, if the input is sorted according to a list of attributes, that property also holds for the output. The knowledge about minimum and maximum values for attributes which are not included in T_U are lost along with these attributes. This implementation is delaying (denoted by the small bar at the end of the box) because the first result tuple cannot be output until the first block has been processed.

The noopGroup is used if the data stream is already distinct in T_U . Here, the aggregation function can be applied tuple by tuple without blocking or delaying.

As a result, the different implementations have shown that with rising data stream quality memory requirements decrease and pipelining capabilities rise. As this principle turns out to hold for most of the complex operators of the KDD process, we can hold it as a general principle.

5 Providing Data Stream Qualities

Introducing data stream quality does not by itself solve the problem that the implementations of the operators used for establishing data stream qualities are blocking and need many resources. In this section we look for an access method that can produce

data streams with desired data stream qualities right off the disk. We will see that there is a trade-off between the efficiency of the index and the level of delivered data stream quality. Therefore we introduce the notion of pseudo-quality that allows us to control the desired degree of data stream quality.

5.1 The index structure

Traditional clustering indices or combinations of several one-dimensional indices support only one or very few attributes and in general allow sequential reading in just one dimension, so they aren't appropriate for our demands, where we have to deal with mass data and don't know in advance, which attributes will be subject of interest during query evaluation. Therefore, in order to achieve equal treatment of multiple attributes and allow for sequential disk access, we chose a multidimensional index structure based on the Peano (Z) space filling curve [11] for physical clustering of the multidimensional data. The Peano curve was chosen because of the efficient calculation of Z-addresses and borders by algorithms operating on the bit representation of the coordinates. The basic ideas are similar to the UB-tree [2] with its range query algorithm and TETRIS algorithm [10]: The space-filling curve is used to map the multidimensional data tuples (represented as points in multidimensional space) to one-dimensional addresses reflecting the points position on this curve while preserving locality as far as possible. These addresses are then used to store the tuples using a conventional one-dimensional access method (such as in our case, a clustering B*-tree). The range-query algorithm for this data structure works as illustrated in Figure 2a: The segments of the space filling curves lying in the query box (shaded) are calculated and the data belonging to these segments can then be read sequentially from disk using the one-dimensional addresses of the entry/exit points of the curve with the query box. The small numbers in the figures denote the order in which these curve segments are processed. For this range query, we see that four segments are processed, which typically requires four random accesses.

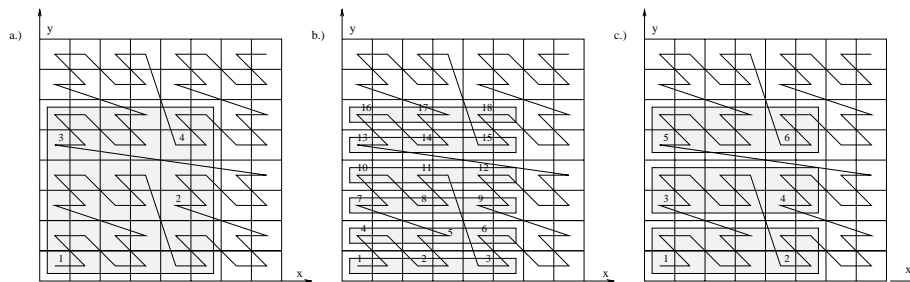


Fig. 2. Range query, sorted, pseudo-sorted

In our work, we extended the UB-tree algorithms by some optimizations for sequential reading in a clustering index and the support for various data stream qualities in arbitrary dimensions. To deliver the data satisfying a particular range query according to a specified data stream quality, e.g. sorted, our algorithm works as follows. To

provide the sort order, the query box is segmented into regions which are processed region by region in the desired sorting order. For each region, the standard range query algorithm is used. Figure 2b shows how the range query of figure 2a can be presented in sorted order along dimension y (data stream quality $\mathbf{S}^+(y)$). In order to deliver the data in this order, the index splits the query box into flat regions of height 1 and processes them one after another. As we can see by counting the corresponding line segments, this approach is not very efficient: In our simplified example, 18 segments are processed (and correspondingly 18 index accesses are performed).

The solution for this problem is shown in picture 2c: By a controlled increase in the block size of the segmentation, we are able to improve the index performance at the expense of a controlled degradation of our data stream quality criteria. The resulting data stream can be seen as a concatenation of blocks. Data stream quality criteria (in our case sort order) are only valid between these blocks, but within these blocks, the values appear in arbitrary order. We call such degraded data stream qualities *pseudo-qualities*. These qualities will be defined in the following section. Note that the number of different values per block is limited by the block size. In our example, we can see that by allowing the index to deliver several (in our example 2) values of the sorting attribute per block) the number of index accesses can be reduced to 6. This approach can easily be generalized to higher dimensions. Instead of two-dimensional ranges, we get higher dimensional boxes which are processed one after another.

In the case of continuity, we have a similar approach. The only difference is, that the desired property that no value combination in the grouping attributes appears in more than one block leaves more degrees of freedom to the shape of those blocks in multidimensional space, so the number of entry/exit points with the space filling curve can be reduced rendering the index more efficient.

5.2 Pseudo Qualities

In the last section we have seen what data stream qualities can be efficiently provided by our extended UB-Index, when we tolerate slight degradations from our strict quality criteria. We include these in our formal framework by the following additional definitions.

Definition 6. [Pseudo-Sorting]

A data stream D_R has the data stream quality **pseudo-sorted $_k$ ascendingly** with respect to T_U and k , in short $\mathbf{PS}_k^+([A_{U_1}, \dots, A_{U_n}])$ iff there exists a partitioning of D_R into consecutive streams $D_R = D_1 \circ \dots \circ D_b$, with

1. for all $g \in \{1, \dots, b\} : |\{\Pi_{T_U}(d_i) \mid d_i \in D_g\}| \leq k$
2. for all $g, h \in \{1, \dots, b\}$ with $g < h$ and for all $d \in D_g$ and $e \in D_h : d <_{T_U} e$

In other words, a data stream is pseudo-sorted if it can be partitioned into consecutive blocks such that each block has at most k distinct values of the sort key (in arbitrary order) and the key ranges of successive blocks form a non-overlapping ascending sequence of ranges.

Similar to the above definition, we define pseudo-continuity. Here it is enough to request that each value combination appears in exactly one block:

Definition 7. [Pseudo-Continuity]

A data stream D_R has the data stream quality **pseudo-continuous_k** with respect to attribute list T_U and k , in short $\mathbf{PC}_k([A_{U_1}, \dots, A_{U_n}])$ iff there exist a partitioning of D_R into consecutive streams $D_R = D_1 \circ \dots \circ D_b$, with

1. for all $g \in \{1, \dots, b\}$: $|\{\Pi_{T_U}(d_i) \mid d_i \in D_g\}| \leq k$
2. for all $g, h \in \{1, \dots, b\}$ with $g \neq h$ and for all $d \in D_g$ and $e \in D_h$: $d \neq_{T_U} e$

With these definitions we are now able to formulate the following theorems. T_U denotes a list of attributes. Theorems 8 to 10 show that sorting is a special case of pseudo-sorting ($k = 1$).

Theorem 8. *If a data stream is sorted with respect to T_U , then it is also pseudo-sorted_k for every k .*

Theorem 9. *A data stream is sorted with respect to T_U , iff it is pseudo-sorted₁ with respect to T_U .*

Theorem 10. *If a data stream is pseudo-continuous_k (pseudo-sorted_k) with respect to T_U , then it also pseudo-continuous(pseudo-sorted) for every multiple of k .*

Theorem 11. *If a data stream is pseudo-sorted_k with respect to T_U then it also is pseudo-continuous_k.*

Theorem 12. *If a data stream is continuous with respect to T_U , then it is pseudo-continuous_k for every k .*

Theorem 13. *A data stream is continuous with respect to T_U , iff it is pseudo-continuous₁.*

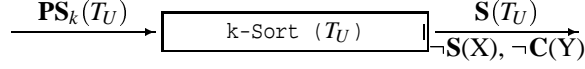
Theorem 14. *If a data stream is pseudo-continuous_k with respect to T_U , then it is pseudo-continuous_k with respect to every permutation of T_U .*

5.3 Operators dealing with Pseudo-Qualities

Since most of the common implementations cannot make direct use of the pseudo-qualities described in the past section, we need specific operators that transform pseudo data qualities into strong data qualities. These operators work on each block in the data stream (the index inserts markers into the data stream to delimit individual blocks) and release the result with the desired quality to the next operator. Our goal is to replace the blocking operators SORT and COLLECT by non-blocking operators that transform pseudo-sorted and pseudo-continuous data streams to sorted and continuous data streams, respectively.

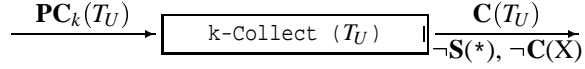
From the SORT operator, which sorts an entire input stream with respect to a given attribute list T_U , we derive the k -SORT operator. This operator differs from SORT by accepting only pseudo-sorted inputs and respecting the block boundaries: After sorting an entire block with a maximum of k different values for the sort key, the operator knows that all tuples in the following blocks are greater with respect to the sort attributes and

therefore it can release the result and clear its internal data structures. In our notation, the k -Sort operator looks as displayed below. Note that in general, all data stream qualities are lost with the exception of those that refer to a prefix or an extension of T_U .



X stands for all attribute sequences that are neither a prefix of T_U nor an extension of T_U . Y denotes all attribute lists, that are neither a prefix nor a superset of T_U .

In analogy, k -Collect transforms pseudo-continuous data streams into continuous streams. After collecting the values of an entire block (e.g in a hashtable) and reading a block boundary, it can be sure, that none of the value combinations in T_U seen in this block will ever appear in subsequent blocks. So, it can release the continuous result and clear its internal data structures. The operator is depicted below in our notation. Here, in general, all data stream qualities involving attribute lists different from T_U are lost.



X stands for all attribute lists that do not contain all attributes from T_U .

With our specialized index and the operators described above, it is now possible to provide all desired data stream qualities in a non-blocking and resource-efficient way.

5.4 An example

Now our framework is complete to demonstrate the effects in query execution. We look at the following simple query that computes from a table of account movements the average sum each customer (denoted by `FirstName` and `LastName`) has moved in each transaction.

```
SELECT FirstName, LastName, AVG(sum) as meanCash
FROM Accounting
GROUP BY FirstName, LastName
ORDER BY LastName, meanCash
```

The resulting operator tree is displayed in Figure 1. To be able to use an efficient `blockGroup` implementation for the `GROUP BY` that processes blocks of continuous values, the data stream should be continuous with respect to `[FirstName, LastName]` (in short, $\mathbf{C}(\mathbf{F}, \mathbf{L})$). For the `Sort` operator, the data stream should be either sorted by `LastName` (then we would use the `blockSort` implementation, that sorts blocks with the same value in its first attribute), or pseudo-sorted (then we would use the new k -Sort implementation). In our case, it turns out that $\mathbf{PS}(\mathbf{L}, \mathbf{F})$ is the optimum configuration for the index access: Because of Theorem 11 we know that this also means $\mathbf{PC}(\mathbf{L}, \mathbf{F})$, which can be used by the `k-collect` operator to provide continuity to the `blockGroup` implementation. The output of this operator is reduced in cardinality, but still pseudo-sorted in `(L, F)`. It can therefore be sorted with the `k-Sort` operator to provide strict sorting in `LastName` to the `blockSort` operator, which now produces the desired order.

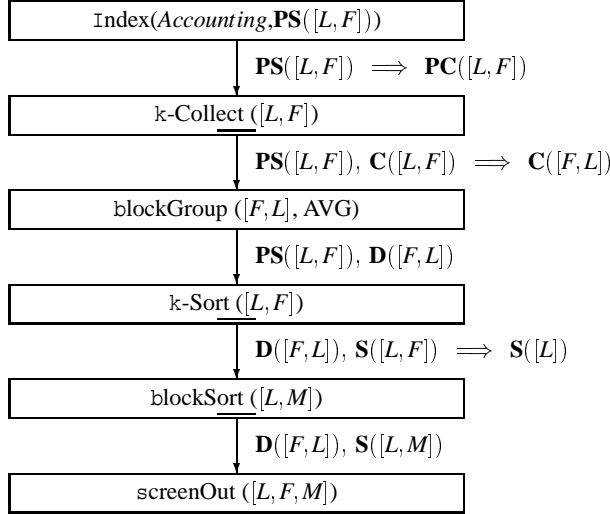


Table 1. Pipeline for Example Query

This simple example shows the power of our framework: The index structure can efficiently provide initial data stream qualities, the k-implementations of data preparation operators can be non-blocking, because the block markers in the pseudo-ordered data stream allow them to release their actual results and proceed with the next block, resource consumption can be effectively controlled by varying the block size k , data preparation operators can be inserted where they're really needed (and, e.g. be placed after initial selections) and the delivered data stream qualities (that formerly existed only within individual implementations of complex operators) are made explicit and can be used by several operators at once.

In our first experiments on the 1 GB-TPC-D data set, a pipelined implementation of the algebra could indeed reduce response times by a factor of up to 20. In addition, against pure in-memory execution without data stream qualities, we pay an overhead of less than 30% in overall response time. For larger datasets, when conventional execution has to swap out intermediate results, we even get better overall response times because of the possibility to effectively control resource consumption. These results show that we are able to achieve interactive query processing while enhancing scalability and efficiency.

6 Parallel and interactive Data Mining

After having gained some insight into preprocessing, we now look at how this approach can be combined with data mining functionality to achieve parallel and interactive data mining. As an example we look at the problem of finding distance based outliers as defined in [8]. A data tuple is called a (p,D) -outlier, if at least fraction p of the tuples in the dataset are outside a distance of D from the observed tuple. A simple operator

to calculate these outliers would be the nested-loop operator: The distances from each tuple to the others is calculated and those lying within distance D are counted. If we get over the threshold defined by p , the tuple is no outlier and the next tuple is analyzed. Clearly, this algorithm is not efficient and only usable, if the whole data fits in memory because index access to find neighboring tuples would make things only worse. In [8] the problem is alleviated by several block- and cell based approaches aiming to reduce disk accesses and complexity. Based on our data stream qualities, we are able to take a different approach: If the data stream has the quality sorted according to some attribute, we are able to exploit this to increase interactivity and save resources. The corresponding `windowOutlier` operator works as shown in figure 3. On the data stream ordered by attribute A_{U1} , we define a window of size D , which is shifted over this stream as data flows through the operator. As a tuple enters the window (e.g. tuple 7), it is compared to all others within the window. Each time the distance between two tuples is lower than D , each tuples counter is increased. As a tuple leaves this window (as in this case, tuple 3), it is output as an outlier if its counter is below the threshold.

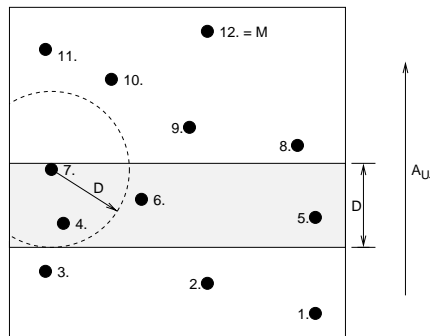


Fig. 3. Pipelining with the `windowOutlier` operator

We see, that this implementation is not blocking and resource efficient, because only the tuples within D need to be stored in memory and compared at a time.

7 Conclusion

We set out to improve efficiency, scalability and interactivity in KDD. The main idea was to isolate the blocking, resource-intensive operators. We identified the data preparation operators as the ones that tend to interrupt the continuity of a data stream. By introducing the notion of data stream quality and by employing advanced index structures that support an algebra of data-stream-quality-aware operators we were able to demonstrate substantial improvements in sequential and parallel query execution. These features together with the possibility to control resource consumption at optimization time seem to open a wide area of applications for our techniques beyond KDD. Such applications may range from the integration of analysis functionality into mainstream commercial databases all the way across embedded databases, up to pipelined distributed

query processing in Grid Computing. Still, we feel we just scratched the surface. We plan to extend our work towards an extensive investigation of the combination of our index structure with various data quality levels. In addition we plan to further enhance scalability making use of replicated input data by means of our index structure. Then do we feel confident enough to tackle the issue of interleaving more data mining algorithms with our improved algebra.

References

1. S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 2001.
2. R. Bayer. The universal b-tree for multidimensional indexing. Technical Report TUM-I9637, TU München, November 1996.
3. D.J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.
4. P.J. Haas and J.M. Hellerstein. Ripple joins for online aggregation. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 287–298. ACM Press, 1999.
5. J. Han, Y. Fu, W. Wang, K. Koperski, and O. Zaiane. Dmql: A data mining query language for relational databases. In *Proceedings of the SIGMOD'96 Workshop on Research Issues on Data Mining and Knowledge Discovery*, pages 27–34, Montreal, Kanada, June 1996.
6. Z. Ives, D. Florescu, M. Friedmann, A. Levy, and D.S. Weld. An adaptive query execution system for data integration. In *Proceedings of the ACM SIGMOD Conference*, 1999.
7. T. Johnson, L.V.S. Lakshmanan, and R.T. Ng. The 3w model and algebra for unified data mining. In *Proceedings of the 26th VLDB Conference*, Kairo, Egypt, 2000.
8. E.M. Knorr and R.T. Ng. Algorithms for mining distance-based outliers in large datasets. In *Proceedings of the 24th VLDB Conference*, New York, USA, 1998.
9. S. Manegold, F. Waas, and M.L. Kersten. On optimal pipeline processing in parallel query execution. Technical report, CWI, Amsterdam, February 1998. <http://www.cwi.nl/ftp/CWIreports/INS/INS-R9805.ps.Z>.
10. V. Markl, M. Zirkel, and R. Bayer. Processing operations with restrictions in rdbms without external sorting: The tetris algorithm. In *Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Australia*, pages 562–571. IEEE Computer Society, 1999.
11. J.A. Orenstein and T.H. Merrett. A class of data structures for associative searching. In *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, April 2-4, 1984, Waterloo, Ontario, Canada*, pages 181–190. ACM, 1984.
12. M. Spiliopoulou, M. Hatzopoulos, and C. Vassilakis. A cost model for the estimation of query execution time in a parallel environment supporting pipeline. *Computers and Artificial Intelligence*, 1996.
13. T. Urhan and M.J. Franklin. Xjoin: A reactively-scheduled pipelining join operator. *IEEE Data Engineering Bulletin*, 2000.
14. T. Urhan and M.J. Franklin. Dynamic pipeline scheduling for improving interactive performance of online queries. In *Proceedings of the 27th Intl. Conference on Very Large Data Bases*, 2001.
15. A.N. Wilschut and P.M.G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 68–77, Miami Beach, December 1991.
16. A.N. Wilschut and S.A. van Gils. A model for pipelined query execution. In *Proceedings of the MASCOTS93 Symposium*, 1993.