Basic Components for Building Column Store-based Applications

Andreas Schmidt*† and Daniel Kimmig†
* *Department of Computer Science and Business Information Systems,*
Karlsruhe University of Applied Sciences
Karlsruhe, Germany
Email: andreas.schmidt@hs-karlsruhe.de
† *Institute for Applied Computer Science*
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: {andreas.schmidt, daniel.kimmig}@kit.edu

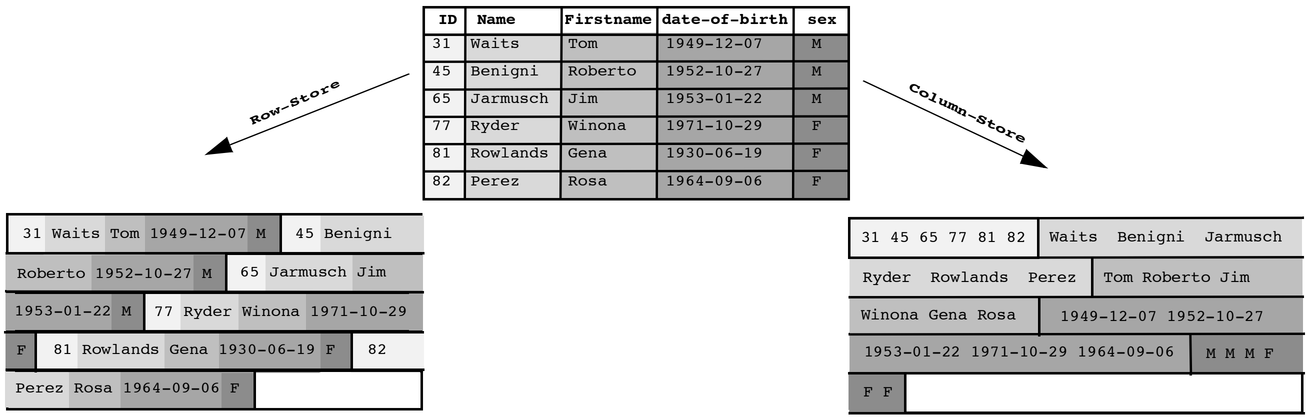
***Abstract*—A constantly increasing CPU-memory gap as well as steady growth of main memory capacities has increased interest in column store systems due to potential performance gains within the realm of database solutions. In the past several monolithic systems have reached maturity in the commercial and academic space. However a framework of low-level and modular components for rapidly building column store based applications has yet to emerge. A possible field of application is the rapid development of high-performance components in various data-intensive areas such as text-retrieval systems. The main contribution of this paper is column-store-kit, a basic building block of low-level components for constructing applications based on column store principles. We present a minimal amount of necessary structural elements and associated operations required for building applications based on our column-store-kit.**

***Keywords*-Column store; basic components; framework; rapid prototyping.**

I. Introduction

Within database systems, values of a dataset are usually stored in a physically connected manner. A *row store* stores all column values of each single row consecutively (see Figure 1, bottom left). In contrast to that, within a *column store*, all values of each single column are stored one after another (see Figure 1, bottom right). In column stores, the relationship between individual column values and their originating datasets are established via Tuple IDentifiers (TID). The main advantage of column stores during query processing is the fact that only data from columns which are of relevance to a query have be loaded To answer the same query in a row store, all columns of a dataset have to be loaded, despite the fact, that only a small portion of them are actually of interest to the processing. On the other side, the column store architecture is disadvantageous for frequent changes (in particular insertions) to datasets. As the values are stored by column, they are distributed at various locations, which leads to a higher number of required write operations exceeding those within a row store to perform the same changes. This characteristic makes this type of storage interesting especially for applications with very high data volume and few sporadic changes only (preferably as bulk upload), as it is the case in, e.g., data warehouses, business intelligence systems or text retrieval systems. Interest in column store systems has recently been reinforced by steady growth of main memory capacities that meanwhile allow for main memory-based database solutions and additionally by the constantly increasing CPU-memory gap [1]: Today's processors can process data much quicker than it can be loaded from main memory into the processor cache. Consequently, modern processors for database applications spend a major part of their time waiting for the required data. Column stores and special cache-conscious [2] algorithms are attempts to avoid this "waste of time". A number of commercial and academic column store systems have been developed in the past. In the research area, MonetDB [3] and C-Store [4] are widely known. Open Source and commercial systems include Sybase IQ, Infobright, Vertica, LucidDB, and Ingres. All these systems are more or less complete database systems with an SQL interface and a query optimizer.

As column stores are a young field of research, numerous aspects remain to be examined. For example, separation of datasets into individual columns result in a series of additional degrees of freedom when processing a query. Abadi et al. [5] developed several strategies as to when a result is to be "materialized", i.e., at which point in time result tuples shall be composed. Depending on the type of query and selectivity of predicates, an early or late materialization may be reasonable. Interesting studies were published about compression methods [6], various index types as well as the execution of join operations, e.g., Radix-Join [1], Invisible Join [7] or LZ-Join [8]. In addition to that, there are attempts at creating hybrid approaches that try to combine the advantages of column and row stores. The main objective of this paper is to present a number of low-level building blocks for constructing applications based on column store systems. Instead of copying the low-level constructs of existing sophisticated column stores, our research work is focused on identifying components



ID	Name	Firstname	date-of-birth	sex
31	Waits	Tom	1949-12-07	M
45	Benigni	Roberto	1952-10-27	M
65	Jarmusch	Jim	1953-01-22	M
77	Ryder	Winona	1971-10-29	F
81	Rowlands	Gena	1930-06-19	F
82	Perez	Rosa	1964-09-06	F

Row-Store
31 Waits Tom 1949-12-07 M 45 Benigni
Roberto 1952-10-27 M 65 Jarmusch Jim
1953-01-22 M 77 Ryder Winona 1971-10-29
F 81 Rowlands Gena 1930-06-19 F 82
Perez Rosa 1964-09-06 F
Column-Store
31 45 65 77 81 82 Waits Benigni Jarmusch
Ryder Rowlands Perez Tom Roberto Jim
Winona Gena Rosa 1949-12-07 1952-10-27
1953-01-22 1971-10-29 1964-09-06 M M M F
F F

Figure 1. Comparison of the layouts of a row store and a column store

and operations that allow for building specialized column store based applications in rapid prototyping fashion. As our components can be composed in a “plug and compute”-style, our contribution is column-store-kit, which is a building block for experimental and prototypical setup of applications within the field of column stores. A possible field of application is the rapid development of high-performance components in various data-intensive areas such as text-retrieval systems.

The paper is structured as follows: In the next section, requirements for our column-store-kit will be outlined. Then, the identified components and corresponding operations will be explained on a logical level. On this basis, various implementations of logical components and operations will be presented. Finally, results will be summarized and an outlook will be given on future research activities.

II. Column Store Principles

Nowadays, modern processors utilize one or more cache hierarchies to accelerate access to main memory. A cache is a small and fast memory which resides between main memory and the CPU. In case the CPU requests data from main memory, it is first checked, whether it already resides within the cache. In this case, the item is sent directly from the cache to the CPU, without accessing the much slower main memory. If the item is not yet in the cache, it is first copied from the main memory to the cache and than further sent to the CPU. However, not only the requested data item, but a whole cache line, which is between 8 and 128 bytes long, is copied into the cache. This prefetching of data has the advantage, that requests to subsequent items are much faster, because they already reside within the cache. Meanwhile, the speed gain when accessing a dataset in the first-level cache is up to two orders of magnitude compared to regular main memory access [9]. Column stores take advantage of this prefetching behaviour, because values of individual columns are physically connected together and therefore often already reside in the cache when requested, as the execution of complex queries is processed column by column rather than dataset by dataset. This also means that the decision whether a dataset fulfills a complex condition is generally delayed until the last column is processed. Consequently, additional data structures are required to administrate the status of a dataset in a query. These data structures are referred to as *Position List*s. A *PositionList* stores the TIDs of matching datasets. Execution of a complex query generates a *PositionList* with entries of the qualified datasets for every simple predicate. Then, the *PositionList*s are linked by *and*/*or* semantics. As an example, Figure 2 shows a possible execution plan for the following query:

```
select name
  from person
 where birthdate < '1960-01-01'
   and sex='F'
```

First, the predicates $birthdate$ <’1960-01-01’ and sex =’F’ must be evaluated against the correponding columns (birthdate and sex) which results in the *PositionList*s *PL1* and *PL2*. These two evaluations could also be done in parallel. Next an *and*-operation must be performed on these two *PositionList*s, resulting in the *PositionList PL3*. As we are interested in the names of the persons that fulfil the query conditions, we have to perform another operation (extract), which finally returns the entries for a TID, specified by the *PositionList PL3*.

III. Concept

The main focus of our components is modeling the individual columns, which can occur both in the secondary store as well as main memory. Their types of representation may vary. To store all values of a column, for example, it is not necessary to explicitly store the TID for each value, because it can be determined by its position (dense

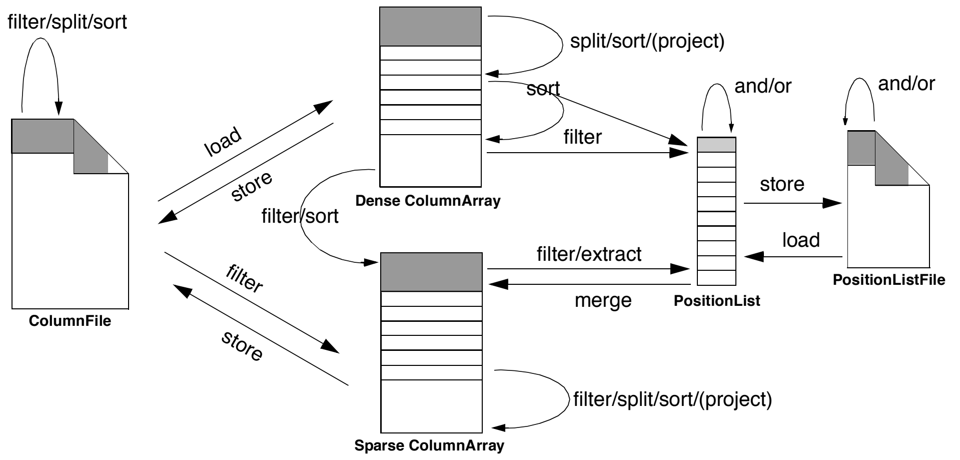


Figure 3. Components and Operations

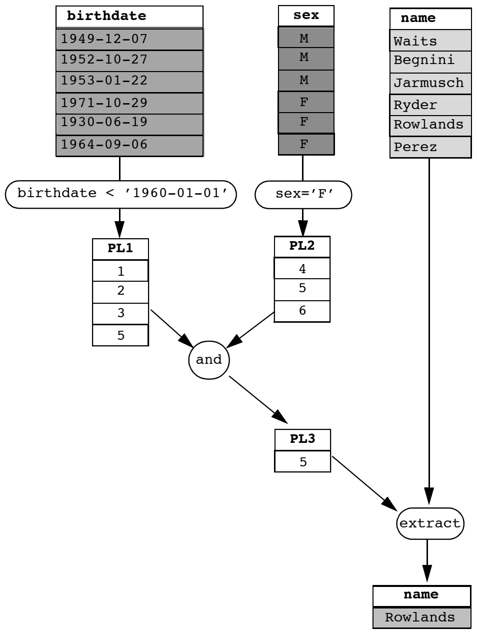


Figure 2. Processing of a query with PositionLists

storage). To handle the results of a filter operation however, the TIDs must be stored explicitly with the value (sparse storage). Another important component is the *PositionList* presented in the previous Section II. Just like columns, two different representation forms are available for main and secondary storage. To generate results or to handle intermediate results consisting of attributes of several columns, data structures are required for storing several values (so-called multi columns). These may also be used for the development of hybrid systems as well as for comparing the performance of row and column store systems. The operations mainly focus on writing, reading, merging, splitting, sorting, projecting, and filtering data. Predicates and/or *PositionList*s are applied as filtering arguments. Figure 3 illustrates a high level overview of the most important operations and transformations between the components. In Section IV they will be described in detail. Moreover, the components are to be developed for use on both secondary store and main memory as well as designed for maximum performance. This particularly implies the use of cache-conscious algorithms and structures.

IV. Presentation of Logical Components

In the following sections, the aforementioned components will be presented together with their structure and their corresponding operations. Section V will then outline potential implementations to reach highest possible performance.

A. Structure

1) ColumnFile: The *ColumnFile* serves to represent a column on the secondary storage. Supported primitive data types are: uint, int, char, date und float. Moreover, the composite type *SimpleStruct* (see below) is supported, which may consist of a runtime definable list of the previously mentioned primitive data types. As a standard, the TID of a value in the *ColumnFile* is given implicitly by the position of the value in the file. If this is not the case, a *SimpleStruct* is used, which explicitly contains the TID in the first column.

2) SimpleStruct: SimpleStruct is a dynamic, runtime definable data structure. It is used within *ColumnFile* as well as within *ColumnArray*s (see below). The *SimpleStruct* plays a role in the following cases:

- Result of a filter query, in which the TIDs of the original datasets are also given.
- Combination of results consisting of several columns.
- Setup of hybrid systems having characteristics of both column and row stores. For example, it may be advan-

tageous to store several attributes in a *SimpleStruct* that are frequently requested together.
- Representation of sorted columns, where TIDs are required. This is particularly reasonable for Join operators or a run-length-encoded compression on their basis.

3) ColumnArray and MultiColumnArray: A *ColumnArray* represents a column in main memory, which consists of a flexible number of lines. The data types correspond to those of the previously defined *ColumnFile*. If the data type is a *SimpleStruct*, it is referred to as *MultiColumnArray*. In addition to the actual column values, the TIDs of the first and last dataset and the number of datasets stored are given in the header of the *(Multi)ColumnArray*. Two types of representations are distinguished:

- **Dense:** The type of representation is dense, if no gaps can be found in the datasets, i.e., if the TIDs are consecutive. In this case, the TID is given virtually by the TID of the first data set and the position in the array and does not have to be stored explicitly (Figure 4, left side). This type of representation is particularly suited for main memory-based applications, in which all datasets (or a continuous section of them) are located in main memory.
- **Sparse:** This type of representation explicitly stores the TIDs of the datasets (Figure 4, right). The primary purpose of a sparse *ColumnArray* is the storage of (intermediate) results. As will be outlined in more detail in Section V, it may be chosen between two physical implementations depending on the concrete purpose.

4) ColumnBlocks and MultiColumnBlocks: Apart from the *(Multi)ColumnArrays* of flexible size, *(Multi)ColumnBlocks* exist, which possess a random, but fixed size. They are mainly used to implement ColumnArrays with their flexible size. In addition, they may be applied in the implementation of an custom buffer management as a transfer unit between secondary and main memory and as a unit that can be indexed.

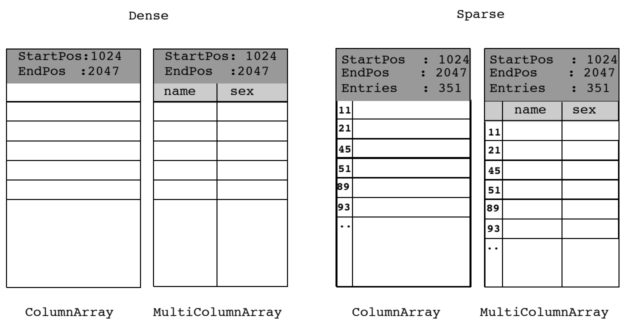


Figure 4. Types of ColumnArrays

5) PositionList: A *PositionList* is nothing else than a *ColumnArray* with the data type *uint(4)* as far as structure is concerned. However, it has a different semantics. The *PositionList* stores TIDs. A position list is the result of a query via predicate(s) on a *ColumnFile* or a *(Multi)ColumnArray*, where the actual values are of no interest, but rather the information about the qualified data sets. *Position Lists* store the TIDs in ascending order without duplicates. This makes the typical and/or operations very fast, as the cost for both operations is $O(|Pl1| + |Pl_2|)$. As will be outlined in Section V, various types of implementations may be applied. Analogously to the *(Multi)ColumnArray*, there is a representation of the *PositionList* for the secondary store, which is called *PositionFile*.

B. Operations

1) Transformations on ColumnFiles: Several operations are defined on *ColumnFiles*. A filter operation (via predicate and/or PositionList) can be performed on a *ColumnFile* and the result can be written to another *ColumnFile* (with or without explicit TIDs). Other operations are the splitting of a *ColumnFile* as well as sorting among different criterias (see Section IV-B6) with and without explicitly storing the TID.

2) Transformations between ColumnFile and (Multi)-Column-Array: ColumnFiles and *(Multi)ColumnArrays* are different types of representation of one or more logical columns. Physically, ColumnFiles are located in the secondary storage, while *ColumnArrays* are located in main memory. Consequently, both types of representations can also be transformed into each other using the corresponding operators.

A *ColumnFile* can be transformed completely or partially into a dense *(Multi)ColumnArray*. If not all, but only certain datasets that match special predicates or *PositionLists* are to be loaded into a *(Multi)ColumnArray*, this can be achieved using filter operations that generate a sparse *(Multi)ColumnArray*. A sparse *(Multi)ColumnArray* may also be transformed into a *ColumnFile*. In this case, the TIDs are stored explicitly in combination with the values. Other operations refer to the insertion of new values and the deletion of values. An outline of the most important operations of *ColumnFiles* is given in Table I.

3) Operations on ColumnArrays: Filter operations can be executed on *(Multi)ColumnArrays* using predicates and/or *PositionLists*. This may result in a sparse *(Multi)ColumnArray* or a *PositionList*. Furthermore, *ColumnArrays* may also be linked with each other by and/or semantics. If the *(Multi)ColumnArrays* have the same structure, the result also possesses this structure. The results correspond to the intersection or union of the original datasets. The result is a sparse *(Multi)ColumnArray*. If *(Multi)ColumnArrays* of differing structure are to be combined, only the and operation is defined. The result is a *(Multi)ColumnArray* that contains a union of all columns of the involved *(Multi)ColumnArrays* and returns the values for the datasets

Table I
OUTLINE OF OPERATIONS ON COLUMNFILES

Operation	Result type
read(ColumnFile)	ColumnArray (dense)
read(ColumnFile, start, length)	ColumnArray (dense)
filter(ColumnFile, predicate)	ColumnArray (sparse)
filter(ColumnFile, predicate-list)	ColumnArray (sparse)
filter(ColumnFile, positionlist)	ColumnArray (sparse)
filter(ColumnFile, positionlist-list)	ColumnArray (sparse)
filter(ColumnFile, predicate-list, positionlist-list)	ColumnArray (sparse)
fileFilter(ColumnFile, predicate)	ColumnFile (explicit TIDs)
fileFilter(ColumnFile, predicate-list)	ColumnFile (explicit TIDs)
fileFilter(ColumnFile, positionlist)	ColumnFile (explicit TIDs)
fileFilter(ColumnFile, positionlist-list)	ColumnFile (explicit TIDs)
fileFilter(ColumnFile, predicate-list, positionlist-list)	ColumnFile (explicit TIDs)
split(ColumnFile, predicate)	ColumnFile, ColumnFile
split(ColumnFile, position)	ColumnFile, ColumnFile
sort(ColumnFile, column(s), direction)	ColumnFile
sort(ColumnFile, Orderlist)	ColumnFile
insert(ColumnFile, value)	Tupel-ID
delete(ColumnFile, Tupel-ID)	boolean
delete(ColumnFile, Positionlist)	integer
delete(ColumnFile, predicate)	integer
delete(ColumnFile, predicate-list)	integer

having identical TIDs. If the *(Multi)ColumnArray*s used as input are dense and if they have the same TID interval, the resulting *MultiColumnArray* is also dense. An outline of the most important operations of *ColumnArray*s is given in Table II. *ColumnArray* may also refer to a *MultiColumnArray*. A *MultiColumnArray*, however, only refers to the version having several columns.

4) Transformation from PositionList to ColumnArray: If the column values of the stored TIDs inside a *PositionList* are needed, an *extract* operation must be performed. Input to this operation is a *PositionList* as well as a *dense (multi) ColumnArray*. The result is a *sparse (Multi) ColumnArray*.

5) Operations between PositionLists: Several *PositionList*s may be combined by *and*, *or* semantics, with the result being a *PositionList*. The result list is sorted in ascending order corresponding to the TIDs. In addition, operations exist to load and store *PositionList*s. An outline of operations of *PositionList*s can be found in Table III.

6) Sorting: One basic operation on *(Multi) ColumnArray*s as well as *ColumnFiles* is sorting. Beside the obvious task to bring the result of a query in a specific order, sorting also plays an important role regarding performance considerations. For the elimination of duplicates, for join operations and for compression using run-length encoding, previous sorting can dramatically improve performance. As a consequence of sorting, the natural order is lost. This is critical for dense columns with implicit TIDs, because the relation to the other column values is lost. The problem can be solved by an additional data structure, similar to a *PositionList* which contains the mapping information to the orginal order of the datasets. Figure 5 gives an example of this situation. The *Multi ColumnArray* on the left side is to be sorted according to the column "name". Additionally to the sorting of the *MultiColumn* (top right), a list is generated which holds the information about the original positions (down right). The list can then be reused by applying it as a sorting criterion to other columns later, as shown in Figure 6.

Table II
OUTLINE OF OPERATIONS ON *ColumnArrays*

Operation	Result type
filter(ColumnArray, predicate)	ColumnArray (sparse)
filter(ColumnArray, predicate-list)	ColumnArray (sparse)
filter(ColumnArray, positionlist)	ColumnArray (sparse)
filter(ColumnArray, positionlist-list)	ColumnArray (sparse)
filter(ColumnArray, predicate-list, positionlist-list)	ColumnArray (sparse)
filter(ColumnArray, predicate)	PositionList
filter(ColumnArray, predicate-list)	PositionList
filter(ColumnArray, positionlist)	PositionList
filter(ColumnArray, positionlist-list)	PositionList
filter(ColumnArray, predicate-list, positionlist-list)	PositionList
and(ColumnArray, ColumnArray)	ColumnArray
or(ColumnArray, ColumnArray)	ColumnArray
and(ColumnArray, ColumnArray)	PositionList
or(ColumnArray, ColumnArray)	PositionList
project(MultiColumnArray, columns)	(Multi)ColumnArray
asPositionList(ColumnArray, column)	PositionList
split(ColumnArray, predicate)	ColumnArray (sparse), ColumnArray (sparse)
split(ColumnArray(dense), position) ColumnArray (dense)	ColumnArray (dense),
split(ColumnArray (sparse), position) ColumnArray (sparse)	ColumnArray (sparse),
store(ColumnArray (dense))	ColumnFile
store(ColumnArray (sparse))	ColumnFile (explicit TIDs)

Table III
OUTLINE OF OPERATIONS ON *PositionLists*

Operation	Result type
load(ColumnFile)	PositionList
store(PositionList)	ColumnFile
and(PositionList, PositionList)	PositionList
or(PositionList, PositionList)	PositionList
read(PositionListFile)	PositionList
store(PositionList)	PositionListFile

7) Compression: Compression plays an important role in column stores [6], as it reduces the data volume that needs to be loaded. Nevertheless, we decided not to include compression in the first prototype. To a certain extent, this constraint can be compensated by the use of dictionary-based compression [10], which will be implemented above the basic components. In later versions, various compression methods will be integrated.

V. IMPLEMENTATION-SPECIFIC CONSIDERATIONS

After presenting the logical structure and the required operations, this section shall now focus on considerations for achieving a performance-oriented implementation.

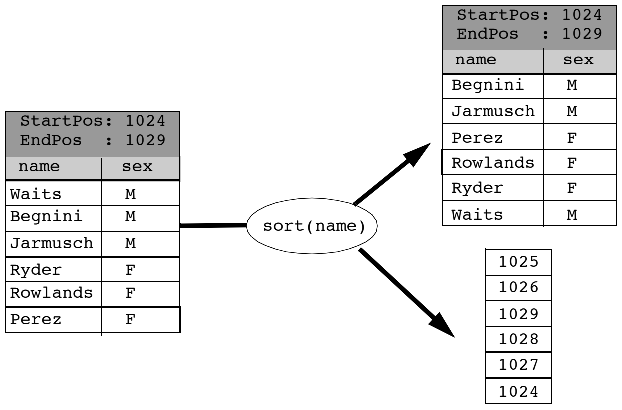


Figure 5. Sorting with explicit generation of an additional mapping list

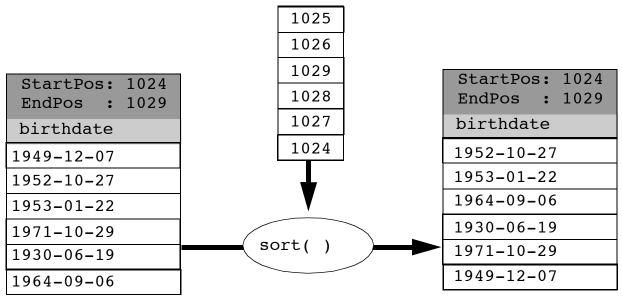


Figure 6. Sorting with explicit given sort-order

Due to the constantly increasing CPU-memory gap, cache-conscious programming is indispensable. For this reason, the implementation was made in C/C++. All time-critical parts were implemented in pure C using pointer arithmetics. The uncritical parts were implemented using C++ classes. The *ColumnBlock* was established as a basic component of the implementation. It is the basic unit for data storage. Its size is defined at creation time and it contains the actual data as well as information on its structure and the number of datasets. The structurization options correspond to those of the *(Multi)ColumnArray*. The *ColumnBlock* also handles all queries by predicates and/or *PositionList*s. A *(Multi)ColumnArray* consists of $1-n$ *ColumnBlock* instances. All operations on a *(Multi)ColumnArray* are transferred to the underlying *ColumnBlock*s.

*PositionList*s play a central role in column store applications. If the *PositionList*s are short (i.e. if they contain a few TIDs only), representation as *ColumnArray* is ideal. Four bytes are required per selected entry. If the lists are very large, however, memory of 40 MB is required for ten million entries, for instance. In this case, a bit vector is recommended for representation. This bit vector indicates using a fixed bit whether every dataset belongs to the set of results or not. If, for example, 100 million data sets exist for a table, only 12.5 MB are required to represent the *PositionList* for certain selectivities. Moreover, the two important operations *and* and *or* can be mapped on the respective primitive processor commands, which makes the operations extremely fast. If *PositionList*s are sparse, bit vectors can be compressed very well using run-length encoding (RLE) (e.g. to a few KB in case of 0.1% selectivity). The necessary operations can be performed very efficiently on the compressed lists, which further increases the performance. An implementation based on the word-aligned hybrid algorithm [11] with satisfactory compression for medium-sparse representations was developed within the framework of the activities reported here [12], [13]. *MultiColumnArray*s may exist in two different physical layouts. In the first version, the n values are written in a physically successive manner and correspond to the classical n-ary storage model (NSM). This type of representation is particularly suited, if further queries are to be performed on this *MultiColumnArray* with predicates on the respective attributes. The individual values of a dataset are stored together in the cache and all attribute values are checked simultaneously rather than successively with the help of additional *PositionList*s (Figure 7, left). The second type of representation corresponds to the PAX format [14]. Here, every column is stored in a separate *ColumnArray*. In addition, a *PositionList* is stored, which identifies the datasets (Figure 7, left). This type of representation is recommended, for instance, for collecting values for subsequent aggregation functions. Several *(Multi)ColumnArray*s may share a single *PositionList*.

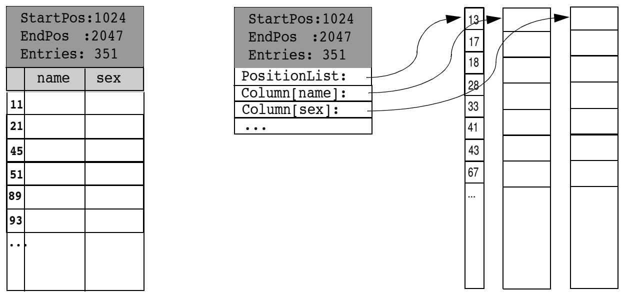


Figure 7. Comparison of storage formats for ColumnArrays

VI. Conclusion

This paper presented a collection of basic components to build column store applications. The components are semantically located below those of the existing column store database implementations and are suited for building experimental (distributed) systems in the field of column store databases. It is planned to use these components to obtain further scientific findings in the area of column stores and to develop data-intensive applications.

VII. Future Work

A first version of the column-store-kit is available without support for compression. The next steps planned are the integration of compression and the use in concrete areas, such as text retrieval systems. A future activity will be the implementation of a scripting language interface for the components. With the help of this interface, it will be possible to assemble the developed components more easily without losing the performance of the underlying C/C++ implementation. In this case, the scripting language act as glue between the components, allowing the developer to build up complex high performance applications with very little effort [15]. As an alternative, a custom domain-specific language (DSL) [16] may be used for building column store applications. A bachelor's thesis [17] focused on the extent to which various degrees of flexibility regarding the structure of *MultiColumnArray*s and expression of the predicates affect the performance. According to the thesis, structural definition at the time of compilation is of significant advantage compared to the runtime behavior. If the implemented flexibility of the *SimpleStruct* is not required at runtime, an alternative implementation may be used. It may be realized by defining a language extension for C/C++, for example. Thus, the respective structures and operations can be defined using a simple syntax. With a number of macros of the C++ preprocessor or a separate inline code expander [18], these could then be transformed into valid C/C++ code.

References

[1] S. Manegold, P. A. Boncz, and M. L. Kersten, "Optimizing database architecture for the new bottleneck: memory access," The VLDB Journal, vol. 9, no. 3, 2000, pp. 231–246.

[2] T. M. Chilimbi, B. Davidson, and J. R. Larus, "Cache-conscious structure definition," in PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation. New York, NY, USA: ACM, 1999, pp. 13–24.

[3] P. A. Boncz, M. L. Kersten, and S. Manegold, "Breaking the memory wall in monetdb," Commun. ACM, vol. 51, no. 12, 2008, pp. 77–85.

[4] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: A Column-oriented DBMS," in VLDB '05: Proceedings of the 31st international conference on Very large data bases. VLDB Endowment, 2005, pp. 553–564.

[5] D. J. Abadi, D. S. Myers, D. J. Dewitt, and S. R. Madden, "Materialization strategies in a column-oriented dbms," in In Proc. of ICDE, 2007, pp. 466–475.

[6] D. J. Abadi, S. R. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in SIGMOD, Chicago, IL, USA, 2006, pp. 671–682.

[7] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. row-stores: How different are they really," in In SIGMOD, 2008, pp. 967–980.

[8] L. Gan, R. Li, Y. Jia, and X. Jin, "Join directly on heavy-weight compressed data in column-oriented database," in WAIM, 2010, pp. 357–362.

[9] P. A. Boncz, M. Zukowski, and N. Nes, "Monetdb/x100: Hyper-pipelining query execution," in CIDR, 2005, pp. 225–237.

[10] C. Binnig, S. Hildenbrand, and F. Färber, "Dictionary-based order-preserving string compression for main memory column stores," in SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data. New York, NY, USA: ACM, 2009, pp. 283–296.

[11] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing bitmap indices with efficient compression," ACM Trans. Database Syst., vol. 31, no. 1, 2006, pp. 1–38.

[12] A. Schmidt and M. Beine, "A concept for a compression scheme of medium-sparse bitmaps," in DBKDA'11: Procceedings of the The Third International Conference on Advances in Databases, Knowledge, and Data Applications. iaria, 2011, pp. 192–195.

[13] M. Beine, "Implementation and Evaluation of an Efficient Compression Method for Medium-Sparse Bitmap Indexes", Bachelor Thesis, Department of Informatics and Business Information Systems, Karlsruhe University of Applied Sciences, Karlsruhe, Germany, 2011.

[14] A. Ailamaki, D. J. DeWitt, and M. D. Hill, "Data page layouts for relational databases on deep memory hierarchies," The VLDB Journal, vol. 11, no. 3, 2002, pp. 198–215.

[15] J. K. Ousterhout, "Scripting: Higher-Level Programming for the 21st Century," IEEE Computer, vol. 31, no. 3, 1998, pp. 23–30.

[16] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," ACM Comput. Surv., vol. 37, no. 4, 2005, pp. 316–344.

[17] M. Herda, "Entwicklung eines Baukastens zur Erstellung von Column-Store basierten Anwendungen" Bachelor's thesis, Department of Informatics, Heilbronn University of Applied Sciences, Germany, Jun. 2011.

[18] J. Herrington, Code Generation in Action. Greenwich, CT, USA: Manning Publications Co., 2003.