Flexs – A Logical Model for Physical Data Layout

Hannes Voigt, Alfred Hanisch, and Wolfgang Lehner

Database Technology Group,
Technische Universität Dresden,
01062 Dresden, Germany
{firstname.lastname}@tu-dresden.de
http://wwwdb.inf.tu-dresden.de/

Abstract. Driven by novel application domains and hardware trends database research and development set off to many novel and specialized architectures. Particularly in the area of physical data layout, specialized solutions have shown exceptional performance for specific applications. This trend is great for research and development and for those in need of top-level performance first and foremost. For those with moderate performance needs, however, a universal but flexible database system has the benefit of lower TCO. Regarding physical data layout, the more general systems are fairly inflexible compared to the variety of physical data layouts available in specialized systems. Particularly, the macroscopic characteristics, i.e., how the data is grouped and clustered, are generally hard-coded and cannot be changed by configuration. We present Flexs, a declarative storage description language for the macroscopic characteristics of physical data layouts. Flexs allows describing physical data layouts ranging from the row and column store layouts to data layouts for irregular data such as vertical schema. Using Flexs, a storage engine can be configured to use a specific physical data layout. Flexs contributes to make specialized physical data layouts available to the broad majority of universal database systems.

1 Introduction

The drastic expansion of the database ecosystem to novel application domains as well as new hardware trends sparked a new exciting age of database technology. Widely challenging the traditional database system architecture in new fields [28], the database community created the wide and diverse range of database systems available today. Specialized systems provide exceptional performance and features for specific applications that have not been seen before.

Among other techniques, these systems typically build on a physical data layout different from the traditional row-orientation. Column-oriented data layouts [15,9,27] showed to be advantageous for most analytic applications. In multi-tenancy databases, clustering data along versions and schema extensions is a preferable strategy [4]. Efficient processing of geo-spatial data often benefits from

N. Bassiliades et al. (eds.), *New Trends in Database and Information Systems II*,
Advances in Intelligent Systems and Computing 312, DOI: 10.1007/978-3-319-10518-5_7

a grid representation [11]. Other layouts such as interpreted record [7] or vertical schema [2,12,1] are favored for applications with flexible schemas or sparse tables such as product catalogs or clinical information systems.

Specialization is great for bringing database technology to new frontiers and boosting its performance. However, it is not always affordable to every extent and for every customer. No matter if they run 10 database systems or 10 000 systems, companies require standardization, unification, and consolidation to keep total cost of ownership (TCO) under control [26]. Any additional system increases the overall complexity of an IT landscape. It requires additional maintenance and additional attention by staff trained on the details of that system. Any additional system adds additional interactions with other systems and by that it implies further need for supervision and further potential cause for failures. With all that additional complexity and cost, specialized systems are only worthwhile where absolute top performance is needed. In most cases, a single system with a configurable physical data layout comes at a much lower TCO, but would still be able to serve a diverse range of workloads well enough. Configurability also simplifies the adaption of a system to evolving workloads and requirements.

Physical data layouts differ in (1) microscopic characteristics, such as, used data structures, applied compression techniques, etc., and in (2) macroscopic characteristics, i.e. how the data is grouped and clustered. To some extent, most systems allow influencing the microscopic characteristics of their physical data layout. For instance, compression can be tuned or different data structures can be configured. Macroscopic characteristics, however, are generally hard-coded and cannot be changed by configuration.

In this paper, we present Flexs. Flexs is a declarative storage description language for the macroscopic characteristics of the physical data layout. It provides a generic way to configure the grouping and clustering of data. In contrast to other modeling approaches such as list comprehension, Flexs explicitly includes the schema elements entity types and attributes into the layout description. Flexs supports a wide range of layouts, such as row-oriented, column-oriented, interpreted record, or vertical schema. We introduce the Flexs notation with the help of three examples and present the grammar that specifies the Flexs language.

The remainder of the paper is structured as follows. Section 2 introduces the Flexs notation. Section 3 details an adaptive materilization strategy necessary to implement a storage engine that is configurable with Flexs. Finally, we discuss related work in Section 4 and conclude the paper in Section 5.

2 Flexs Notation

Flexs is a modeling language that allows describing various physical data layouts for structured data, specifically the macroscopic aspects of the physical data layouts. Commonly, structured data models organize values with the user-given specifiers entity types, entities, and attributes. The physical data layout determines how a database management system organizes the elements of structured data (entity types, entities, attributes, and values) on the physical storage to preserve their logical structure.

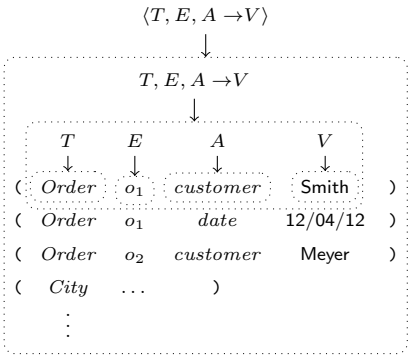

Fig. 1. Vertical schema layout

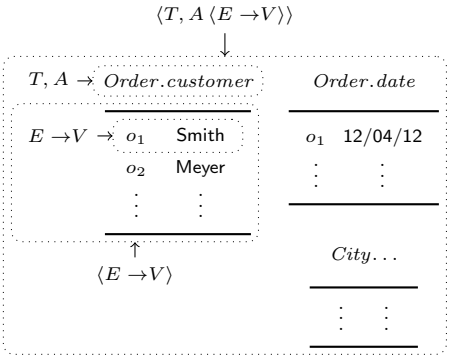

Fig. 2. BATs layout

The first and most simple way to preserve logical structure is to store logically related elements physically next to each other. For instance, the vertical schema as illustrated in Figure 1 stores an entity type, an entity, an attribute, and the value that belongs to the specific combination in one coherent physical block. In Figure 1, the first block lists Smith as the value of the attribute *customer* for the entity o_1 with type *Order*. In Flexs, we describe such a block by listing the domains of its elements. T denotes the domain of entity types, E denotes the domain of entities, A denotes the domain of attributes, and V denotes the domain of values Consequently, $T, E, A \rightarrow V$ describes a physical block in the vertical schema layout. The arrow $\rightarrow$ indicates that T, E, A uniquely identify a block and functionally determine V in this mental model. To represent more than a single value, the database system uses a whole set of similar blocks to represent a complete data set. In Flexs, we denote such a block set as $\langle T, E, A \rightarrow V \rangle$. Chevrons $\langle X \rangle$ indicate the repetition of the embedded block structure X. The order of the blocks in a block set is insignificant. The commas in the Flexs notation do not have a particular meaning but serve better visual separation of the domain symbols.

A second way to preserve logical structure is nesting. For nesting, let's have a look at the binary association tables (BATs) [9] layout. Figure 2 shows the same data represented in BATs. A single BAT consists of an entity type, an attribute, and a set of entity–value pairs. Hence, we can denote the header of a BAT in Flexs as T, A and the body of a BAT – set of entity–value pairs – can be denoted as the block set $\langle E \rightarrow V \rangle$. In all, a BAT forms a block $T, A \langle E \rightarrow V \rangle$. This block nests block set of entity–value pairs to represent the fact these entity–value pairs are logically related to the entity type–attribute pair in BAT's header. Again, the complete physical data layout consists of multiple blocks like these and is denoted as $\langle T, A \langle E \rightarrow V \rangle \rangle$.

A third way to preserve logical structure is the physical order. Normally, the physical order of blocks within a block set is insignificant. Netherless, some layouts utilize the physical order, e.g. the traditional row store store layout, as

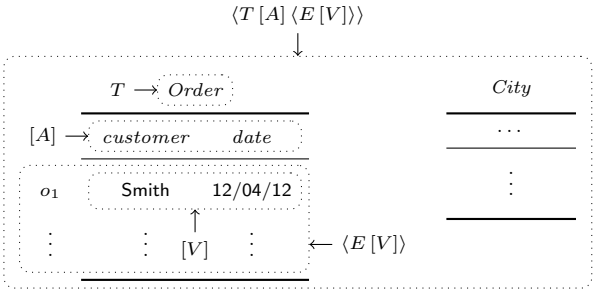

Fig. 3. Row store layout

shown in Figure 3. For each entity type, the row store maintains an ordered set of attributes. In Flexs, we denote such an ordered set with brackets: $[A]$. All values of an entity are also stored in order, namely in the order of the attributes of the corresponding entity type. Hence, the row store layout $\langle T\,[A]\,\langle E\,[V]\rangle\rangle$ represents the logical relations between attributes and values exclusively by their order.

Next to three discussed layouts, Flexs can also describe various other layouts as shown in Table 1. The table shows each layout with its respective Flexs expression and an example of the resulting block set. Note that Flexs explicitly considers the role of the schema elements entity types and attributes within the physical data layout. The inclusion of schema elements allows Flexs to support physical data layout for irregular data, e.g., interpreted record and vertical schema. Further it allows Flexs to support also layouts that have not been widely considered so far. For instance, the tagging layout shown in Table 1 allows to physically represent multifaceted entities as they exist in the Freebase data model [8]. This clearly distinguishes Flexs from other modeling approaches such as list comprehension.

Flexs is formally defined by a grammar as shown in Figure 5. To be valid, a Flexs expression has to follow the grammar. The central element is the block set definition (<blockset>). A block set can be defined as ordered (<ordered>) or as a block set without particular order (<unordered>). Ordered block sets are defined on single domains. Further, ordered block sets are only allowed in pairs in Flexs expressions, where one of the two ordered block sets has to be defined on the domain of values V. Normal block set definitions consist of header domains (<header>), included domains (<include>), and nested block sets (<nesting>). There must be at least one header domain; included domains and nested block sets are optional. Among the four domains, we distinguish between the specifier domains – entity types T, entities E, and attributes A – and the values domain V. We are not considering values to have an identity of their own; value identity derives from the entity type, the entity, and the attribute a value belongs to. Consequently, V is only allowed in ordered block sets or as included domain in unordered block sets.

Table 1. Various physical data layouts

Vertical schema	$\langle T, E, A \rightarrow V \rangle$	$\langle Order, o_1, customer \rightarrow$Smith $\mid Order, o_1, date \rightarrow$12/04/12 $\mid Order, o_2, customer \rightarrow$Meyer $\mid \ldots \rangle$
Row store	$\langle T\,[A]\,\langle E\,[V] \rangle \rangle$	$\langle Order, [customer \mid date]\,\langle o_1, [$Smith $\mid$ 12/04/12$]$ $\mid o_2, [$Meyer $\mid$ 12/05/12$] \rangle \rangle$
Column store	$\langle T\,[E]\,\langle A\,[V] \rangle \rangle$	$\langle Order, [o_1 \mid o_2]\,\langle customer, [$Smith $\mid$ Meyer$]$ $\mid date, [$12/04/12 $\mid$ 12/05/12$] \rangle \rangle$
BATs	$\langle T, A\,\langle E \rightarrow V \rangle \rangle$	$\langle Order, customer\,\langle o_1 \rightarrow$Smith $\mid o_2 \rightarrow$Meyer$\rangle$ $\mid Order, date\,\langle o_1 \rightarrow$12/04/12 $\mid o_2 \rightarrow$12/05/12$\rangle \rangle$
Interpreted record	$\langle T\,\langle E\,\langle A \rightarrow V \rangle \rangle \rangle$	$\langle Order\,\langle o_1, \langle customer \rightarrow$Smith $\mid date \rightarrow$12/04/12$\rangle$ $\mid o_2, \langle customer \rightarrow$Meyer $\mid date \rightarrow$12/05/12$\rangle \rangle \rangle$
BigTable	$\langle T\,\langle E, A \rightarrow V \rangle \rangle$	$\langle Order\,\langle o_1, customer \rightarrow$Smith $\mid o_1, date, \rightarrow$12/04/12 $\mid o_2, customer \rightarrow$Meyer $\mid \ldots \rangle \rangle$
XML-like	$\langle E \rightarrow T\,\langle A \rightarrow V \rangle \rangle$	$\langle o_1 \rightarrow Order\,\langle customer \rightarrow$Smith $\mid date \rightarrow$12/04/12$\rangle$ $\mid o_2 \rightarrow Order\,\langle customer \rightarrow$Meyer $\mid \ldots \rangle \rangle$
Tagging	$\langle E\,\langle T \rangle\,\langle A \rightarrow V \rangle \rangle$	$\langle o_1\,\langle Order \rangle\,\langle customer \rightarrow$Smith $\mid date \rightarrow$12/04/12$\rangle$ $\mid o_2\,\langle Order \rangle\,\langle customer \rightarrow$Meyer $\mid \ldots \rangle \rangle$

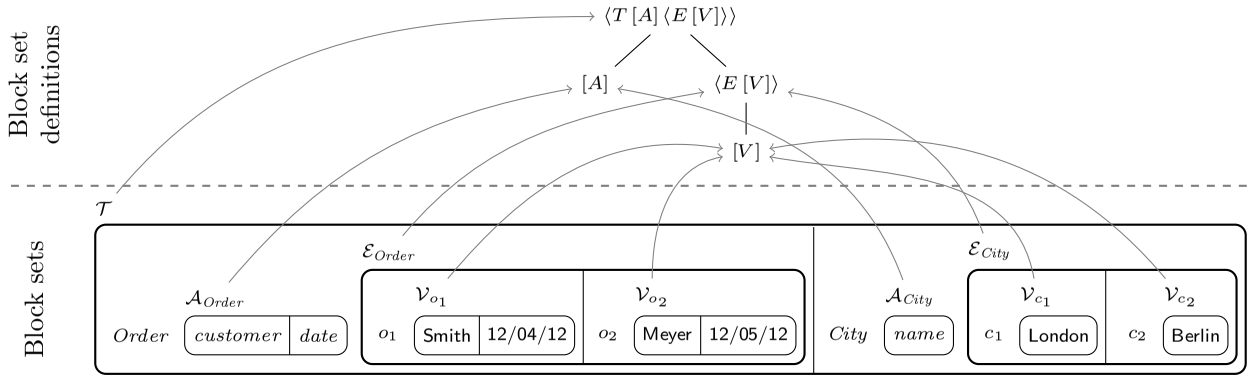

Fig. 4. Block set with the row store layout $\langle T\,[A]\,\langle E\,[V] \rangle \rangle$

```
  <blockset> ::= <ordered> | <unordered>
   <ordered> ::= '[' <domain> ']'
 <unordered> ::= '⟨' <header> <include>? <nesting>? '⟩'
    <header> ::= <domain>+
   <include> ::= '→' <domain>+
   <nesting> ::= <blockset>+
   <domain> ::= <specifiers> | <values>
<specifiers> ::= 'T' | 'E' | 'A'
    <values> ::= 'V'
```

Fig. 5. Flexs grammar

When parsed, a Flexs expression results in an abstract syntax tree of block set definitions as shown in the upper half of Figure 4. These block set definitions provide the necessary information to physically arrange and retrieve data on storage. Like the block set definitions, the physical layout described by the expression is hierarchical. On each hierarchy level the block set definition is instantiated with one or more block sets, so that each block set is typed by a particular block set definition from the abstract syntax tree. The topmost block set definition is only instantiated once, while lower-level block set definitions are likely to be instantiated multiple times, depending on the data.

Block sets, the instances of block set definitions, contain one or multiple blocks. The blocks contain data elements and may nest lower-level block sets. The definition of a block set defines the structure of its blocks. For instance, in a block set of the definition $\langle E\,[V]\rangle$, all blocks take the form $(e, \mathcal{X})$, where e is an entity and $\mathcal{X}$ is a block set of the definition $[V]$. All blocks in a block set are uniquely identified by their headers, i.e. their elements from the header domains in the definition. In our example, all blocks are uniquely identified by the data element e. A block's data elements from the included domains in the block set definition form the block include.

As an example, Figure 4 illustrates the block sets resulting from the row store layout $\langle T\,[A]\,\langle E\,[V]\rangle\rangle$ given our example data. In the figure, the rounded rectangles mark block sets, while the vertical lines separate blocks within block sets. The top most block set $\mathcal{T}$ is an instance of $\langle T\,[A]\,\langle E\,[V]\rangle\rangle$ as indicated by the arrow pointing from the block set to its block set definition. Consequently, it contains blocks consisting of an entity type as header and two nested block sets of the form $[A]$ and $\langle E\,[V]\rangle$. $\mathcal{T}$ contains two of such blocks: $(\mathit{Order}, \mathcal{A}_{\mathit{Order}}, \mathcal{E}_{\mathit{Order}})$ and $(\mathit{City}, \mathcal{A}_{\mathit{City}}, \mathcal{E}_{\mathit{City}})$. The order of these two blocks as given in the figure is insignificant. Further down in the hierarchy, for instance, the block set $\mathcal{E}_{\mathit{Order}}$ is of the form $\langle E\,[V]\rangle$ and contains the blocks $(o_1, \mathcal{V}_{o_1})$ and $(o_2, \mathcal{V}_{o_2})$. $\mathcal{V}_{o_1}$ and $\mathcal{V}_{o_2}$ are both ordered block sets of the form $[V]$, each containing a pair of blocks, (Smith), (12/04/12) and (Meyer), (12/05/12), respectively. Here, the order of the blocks is essential since it allows connecting the values to their attributes stored in the ordered block set $\mathcal{A}_{\mathit{Order}}$.

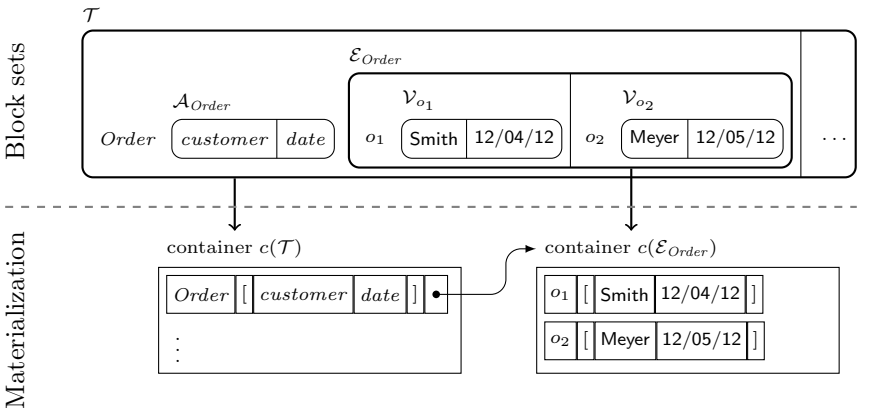

Fig. 6. Block set materialization for $\langle T\,[A]\,\langle E\,[V]\rangle\rangle$

3 Block Set Materialization

With Flexs a storage engine can be configured to a specific physical data layout. The storage engine then has to arrange the data accordingly in storage. Therefore the storage engine simply can materialize the blocks of the topmost block set consecutively in storage. The materialization of a block embeds the materialization of all nested block set. For the majority of layouts, however, this simple materialization strategy results in very large physical blocks. These large physical blocks and particularly the nested block sets are hard to maintain and cannot be efficiently queried. Selective queries would have to read a lot of unnecessary data.

Efficient retrieval and maintenance of blocks require indirection steps in the materialization. An indirection step separates nested block sets into a dedicated materialization. In the materialization of the nesting block a reference points to the dedicated materialization of the nested block set. To have an indirection step for each nested block set is not ideal either. Too many indirection steps lower the retrieval performance as well; they reduce locality and increase the number of references to resolve. The optimal set of indirection steps depends on the data set and the configured layout.

For instance, the materialization strategy used in Figure 6 is absolutely reasonable for regular relational data, where we have orders of magnitude more entities than attributes and entities are only a few hundred bytes of size. However, with a very large number of attributes or large blob values the embedded block sets $\mathcal{A}_{Order}$, $\mathcal{V}_{o_1}$, $\mathcal{V}_{o_2}$, etc. would be significantly larger. If so, a different materialization strategy may be more efficient.

We propose an adaptive strategy to avoid the embedding of very large block sets. When a database is created, the adaptive strategy starts with a single physical container for the topmost block set. All other block sets are materialized in an embedded way. With more data being inserted, the block sets grow. For

every block set where the embedded materialization size exceeds a configured threshold, e.g., the size of a memory page or a disk block, the adaptive strategy introduces an indirection step and moves the block set to a dedicate materialization. Note that this reorganization is applied to relatively small block sets only (just above the threshold), so that operational overhead is small. Regardless of the configured layout, the adaptive strategy finds the optimal set of indirection steps.

4 Related Work

Flexs aims at supporting a wider range of applications in a single system, by offering a configurable specialization of the physical data layout. Most prominently, OLTP workloads favor a row-oriented data layout while OLAP workloads profit from a column-oriented layout. Supporting these two workloads in a single system has been the aim of multiple works in recent years.

Index-only plans try to emulate column-oriented processing in a row-based system. Additionally to the base relational, it adds an unclustered B+Tree index on every column. With the help of index intersection, the database system can then answer queries without reading the base relation [21,23]. The efficiency of index-only plans can be increased by exploiting the parallel processing capabilities of modern hardware [13]. Microsoft further exploits this idea by introducing a dedicated column store index [19]. Trojan Columns [18] follows the same line but omits the base relation completely. Still, the index-only plans remain an OLAP-focused add-on to a row-oriented database system. They do not offer the generality of Flexs.

A very early approach of combining OLTP and OLAP is Fractured Mirrors [22]. Fractured Mirrors leverages the fact, that disk-based databases usually replicate data to multiple disks. If the replicates hold the data in different physical layouts, queries can access the data in their preferred physical layouts. As presented, Fractured Mirrors supports exactly the two representations, row store and column store. Both are implemented as different scan operators in the database engine. Generally, the idea is orthogonal to Flexs. Both would make an appealing combination.

HyPer [20] is a main memory database system with OLTP and OLAP support. It runs OLAP queries concurrently and isolated from the OLTP queries, by utilizing hardware-supported page shadowing. In addition to the strictly sequential execution of OLTP write operations, this eliminates the need for locking or any other kind of currency control, resulting in outstanding transaction throughput and query response times. The physical data layout, though, is not the authors' main concern. HyPer uses the same physical layout for all data; globally configured to either row-oriented or column-oriented. Other physical data layouts are not supported.

SAP HANA [14,25] is another hybrid main memory database system for OLTP and OLAP. In its hybrid relational engine, it combines SAP's row store database engine P*Time [10] and SAP's column store engine TREX. HANA's academic

sidekick [24] uses MaxDB as row-store engine. The setup, though, is the same: Two relational engines wired to the same query processor. During table creation it is decided which engine manages the table. The well-known open source database system MySQL also follows the multiple engine concept. Here, an interface separates data storage from the query processor including SQL parser and query optimizer. Over the years, many storage engines have been developed each implementing its own physical data layout. Instead of supporting a hard-coded set of physical data layouts, Flexs aims at a freely configurable physical data layout implemented in a single engine.

HYRISE [16] is also a hybrid main memory storage engine. It partitions tables vertically into configurable column groups, where each column group is stored directory compressed as if it is a single column. Hence, in its heart, HYRISE is a main memory column store that can be configured to mimic a row store or mixed layouts. The idea of bundling columns together that are typically accessed was already used earlier in the Data Morphing approach [17]. It is an extension of PAX [3]. PAX organizes pages like a column store for better cache performance. Data Morphing improves that by exploiting column groups. All these approaches focuses on OLTP and OLAP. Other workloads favoring different physical data layouts are not considered.

While the aforementioned approaches focus on the combination of OLTP and OLAP workloads RodentStore [11] takes a broader hold of the topic. Like Flexs, RodentStore envisions a freely configurable data layout, but with a different focus than Flexs. RodentStore describes the physical data layout as nested lists of values. How the lists are nested and which values they contain can be configured with a storage algebra building on list comprehensions. This allows a wide range of physical data layouts including row store, column store, and grid layout. With the focus rather on spatial data, RodentStore includes some representations not covered by Flexs, such as the utilization of space filling curves. The RodentStore concept assumes strictly regular relational data, though, and does not consider the role of specifiers in the data representation. Hence, it lacks support for physical data layouts most suitable for irregular data. It would be interesting to see how both concepts, RodentStore and Flexs could be combined.

Another notable approach on increasing physical data independence is GMAP [29]. GMAP offers a generalized definition language for access paths. Where traditional access paths are fixed to the logical data model, GMAP defines the data stored in an access path with query-like statements. This provides great flexibility in the physical design for the DBA. As RodentStore, GMAP assumes regular data and does not consider the role of specifiers in the data layout. GMAP's focus is to allow the consolidation of associated entities from different domains in a single access path. It also allows the replication of entities over multiple access paths. GMAP offers a great way to define data subsets which require different physical data layouts. In that respect, GMAP and Flexs complement each other perfectly.

Finally, GENESIS [5,6] is a project from the mid-eighties that developed detailed storage models for database systems including aspects of the physical data

layout. The aim, however, was not to strengthen the physical data independence of database systems but to speedup their development. Instead of providing an exhaustive description of a database system's storage layer, it focuses on capturing its macroscopic characteristics.

5 Conclusion

In this paper, we presented Flexs. Flexs allows describing the macroscopic characteristics of physical data layouts, i.e. how data elements are grouped on the physical storage. It can express common layouts such as row store, column store or BATs. Flexs also can serve as a foundation for mixed physical data layouts – a direction we plan to explore next. In contrast to list comprehensions, Flexs is not limited to regular data; it supports also layouts for irregularly structured data, such as interpreted record and vertical schema. A Flexs expression describes a hierarchical structure of nested block sets, which contain blocks of data elements and thereby determine how data elements can be selected and scanned. We also proposed an adaptive materialization strategy that finds the optimial set of indirection steps to allow for efficient retrieval and maintanence of the data. Flexs is not meant to provide top-level performance but allow combining the characteristics of various physical data layouts in a single system.

References

1. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.J.: Scalable Semantic Web Data Management Using Vertical Partitioning. In: VLDB 2007 (2007)
2. Agrawal, R., Somani, A., Xu, Y.: Storage and Querying of E-Commerce Data. In: VLDB 2001 (2001)
3. Ailamaki, A., DeWitt, D.J., Hill, M.D., Skounakis, M.: Weaving Relations for Cache Performance. In: VLDB 2001 (2001)
4. Aulbach, S., Seibold, M., Jacobs, D., Kemper, A.: Extensibility and Data Sharing in evolving multi-tenant databases. In: ICDE 2011 (2011)
5. Batory, D.S.: Modeling the Storage Architectures of Commercial Database Systems. ACM Transactions on Database Systems 10(4) (1985)
6. Batory, D.S., Barnett, J.R., Garza, J.F., Smith, K.P., Tsukuda, K., Twichell, B.C., Wise, T.E.: GENESIS: An Extensible Database Management System. IEEE Transactions on Software Engineering 14(11) (1988)
7. Beckmann, J.L., Halverson, A., Krishnamurthy, R., Naughton, J.F.: Extending RDBMSs To Support Sparse Datasets Using An Interpreted Attribute Storage Format. In: ICDE 2006 (2006)
8. Bollacker, K.D., Evans, C., Paritosh, P., Sturge, T., Taylor, J.: Freebase: A Collaboratively Created Graph Database For Structuring Human Knowledge. In: SIGMOD 2008 (2008)
9. Boncz, P.A., Kersten, M.L.: MIL Primitives for Querying a Fragmented World. The VLDB Journal – The International Journal on Very Large Data Bases 8(2) (1999)
10. Cha, S.K., Song, C.: P*TIME: Highly Scalable OLTP DBMS for Managing Update-Intensive Stream Workload. In: VLDB 2004 (2004)

11. Cudré-Mauroux, P., Wu, E., Madden, S.: The Case for RodentStore: An Adaptive, Declarative Storage System. In: CIDR 2009 (2009)
12. Cunningham, C., Graefe, G., Galindo-Legaria, C.A.: PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS. In: VLDB 2004 (2004)
13. El-Helw, A., Ross, K.A., Bhattacharjee, B., Lang, C.A., Mihaila, G.A.: Column-Oriented Query Processing for Row Stores. In: DOLAP 2011 (2011)
14. Färber, F., Cha, S.K., Primsch, J., Bornhövd, C., Sigg, S., Lehner, W.: SAP HANA Database - Data Management for Modern Business Applications. SIGMOD Record 40(4) (2011)
15. French, C.D.: "One Size Fits All" Database Architectures Do Not Work for DDS. In: SIGMOD 1995 (1995)
16. Grund, M., Krüger, J., Plattner, H., Zeier, A., Cudré-Mauroux, P., Madden, S.: HYRISE - A Main Memory Hybrid Storage Engine. The Proceedings of the VLDB Endowment 4(2) (2010)
17. Hankins, R.A., Patel, J.M.: Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In: VLDB 2003 (2003)
18. Jindal, A., Schuhknecht, F., Dittrich, J., Khachatryan, K., Bunte, A.: How Achaeans Would Construct Columns in Troy. In: CIDR 2013 (2013)
19. Larson, P.Å., Clinciu, C., Hanson, E.N., Oks, A., Price, S.L., Rangarajan, S., Surna, A., Zhou, Q.: SQL Server Column Store Indexes. In: SIGMOD 2011 (2011)
20. Kemper, A., Neumann, T.: HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: ICDE 2011 (2011)
21. Mohan, C., Haderle, D.J., Wang, Y., Cheng, J.M.: Single Table Access Using Multiple Indexes: Optimization, Execution, and Concurrency Control Techniques. In: Bancilhon, F., Zhang, J., Thanos, C. (eds.) EDBT 1990. LNCS, vol. 416, pp. 29–43. Springer, Heidelberg (1990)
22. Ramamurthy, R., DeWitt, D.J., Su, Q.: A Case for Fractured Mirrors. In: VLDB 2002 (2002)
23. Raman, V., Qiao, L., Han, W., Narang, I., Chen, Y.L., Yang, K.H., Ling, F.L.: Lazy, Adaptive RID-List Intersection, and Its Application to Index Anding. In: SIGMOD 2007 (2007)
24. Schaffner, J., Bog, A., Krüger, J., Zeier, A.: A Hybrid Row-Column OLTP Database Architecture for Operational Reporting. In: Castellanos, M., Dayal, U., Sellis, T. (eds.) BIRTE 2008. LNBIP, vol. 27, pp. 61–74. Springer, Heidelberg (2009)
25. Sikka, V., Färber, F., Lehner, W., Cha, S.K., Peh, T., Bornhövd, C.: Efficient Transaction Processing in SAP HANA Database – The End of a Column Store Myth. In: SIGMOD 2012 (2012)
26. Stonebraker, M.: Stonebraker on NoSQL and enterprises. Communications of the ACM 54(8) (2011)
27. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E.J., O'Neil, P.E., Rasin, A., Tran, N., Zdonik, S.B.: C-Store: A Column-oriented DBMS. In: VLDB 2005 (2005)
28. Stonebraker, M., Çetintemel, U.: "One Size Fits All": An Idea Whose Time Has Come and Gone. In: ICDE 2005 (2005)
29. Tsatalos, O.G., Solomon, M.H., Ioannidis, Y.E.: The GMAP: A Versatile Tool for Physical Data Independence. The VLDB Journal – The International Journal on Very Large Data Bases 5(2) (1996)