# ECOS: Evolutionary Column-Oriented Storage

Syed Saif ur Rahman, Eike Schallehn, and Gunter Saake

Faculty of Computer Science,
Otto-von-Guericke University, Magdeburg, Germany
{srahman,eike,saake}@ovgu.de

**Abstract.** As DBMS has grown more powerful over the last decades, they have also become more complex to manage. To achieve efficiency by DBMS tuning is nowadays a hard task carried out by experts. This development inspired the ongoing research on self-tuning to make DBMS more easily manageable. We present a customizable self-tuning storage manager, we termed as Evolutionary Column-Oriented Storage (ECOS). The capability of self-tuning data management with minimal human intervention, which is the main design goal for ECOS, is achieved by dynamically adjusting the storage structures of a column-oriented storage manager according to data size and access characteristics. ECOS is based on the Decomposed Storage Model (DSM). It supports customization at the table-level using five different variations of DSM. ECOS also proposes fine-grained customization of storage structures at the column-level. It uses hierarchically-organized storage structures for each column, which enables autonomic selection of the suitable storage structure along the hierarchy using an evolution mechanism (as hierarchy-level increases). Moreover, for ECOS, we proposed the concept of an evolution path that provides a reduction of human intervention for database maintenance. We evaluated ECOS empirically using a custom micro benchmark showing performance improvement.

**Keywords:** column-oriented storage, evolving hierarchically-organized storage structures, customization, autonomy.

## 1 Introduction

Efficient data management demands continuous tuning of a database and a DBMS. The need for tuning a DBMS is driven by changes, such as database size, workloads, schema design, hardware, and application specific data management needs. Existing DBMS need extensive human intervention for tuning, which contributes to a major portion of the total cost of ownership for data management [7]. Self-tuning is the solution to reduce the tuning cost through minimizing the human intervention [22]. However, researchers are united on one conclusion that the self-tuning based solutions are the biggest challenge in the database domain because of the inherent complexity of existing DBMS architectures. Their functionalities are tightly integrated into their monolithic engines, and it is difficult to assess the impact of tuning of one knob on another [6].

In this paper, we present a customizable and online self-tuning storage manager. As a key design concept, we propose the selection of an appropriate storage

© Springer-Verlag Berlin Heidelberg 2011

model and data/index storage structure through customization. This design decision is according to the suggestion from the work of Chaudhuri and Weikum [6]. ECOS supports fine-grained customization at the table-level and column-level according to the recommendations/results from [2,10,12]. We also identified the need to autonomically change the existing data and index storage structure to more appropriate ones with the changing data management needs according to our previously published results in [18]. We named our solution Evolutionary Column-Oriented Storage (ECOS), which is based on the existing Decomposed Storage Model (DSM) [10]. It uses hierarchically-organized storage structures for each column with an innovative evolution mechanism, which enables autonomic selection of the most suitable storage structure along the hierarchy (as the levels of a hierarchy increase). Furthermore, we present four possible variations to standard 2-copy DSM to reduce its high storage requirement. We evaluated ECOS empirically using the custom micro benchmark and our results show that ECOS self-tunes the storage structure while maintaining the required performance. Additionally, it also gives minor performance gains. Furthermore, we propose a mechanism called evolution path to define the storage structure evolution, which reduces the need for human intervention for long-term database maintenance.

This paper is organized as follows. Section 2 defines the problem and justifies the motivation for the proposed design. Section 3 explains the concepts of ECOS and evolution path in detail. Section 4 introduces the prototype implementation and gives details of the empirical evaluation of the proposed concepts using a custom micro benchmark. Section 5 outlines the related work. Section 6 concludes the paper with hints for the future work.

## 2 Problem Statement and Motivation

Specific storage structures have characteristics suitable for certain data sizes and access patterns. As both of these aspects may change over the course of data usage, there is no single storage solution that provides optimal performance in every situation. Therefore, we propose an autonomic adjustment of the storage structures. In this section, we explain the motivation for some critical design decisions in ECOS. To explain the problem in detail, we take the LINEITEM table of the TPC Benchmark$^{\mathrm{TM}}$H (TPC-H) [17] schema as an example. We generated the benchmark data with the scale factor of one and gathered statistics for the LINEITEM table as shown in Table 1.

***Why column-oriented storage model?*** The column-oriented storage model is derived from earlier work of DSM [10]. DSM is a transposed storage model [4] that stores all values of the same attribute of the relational conceptual schema relation together [10]. Copeland and Khoshafian in [10,20] concluded many advantages of DSM including simplicity (Copeland and Khoshafian related it to RISC [16]), less user involvement, less performance tuning requirement, reliability, increased physical data independence and availability, and support of heterogeneous records. These advantages give strong motivation for the use of the DSM in a self-tuning storage manager.

***Why customization at the column-level?*** Table 1 includes some characteristics of the LINEITEM table. We can observe that distinct data count (cardinality) for all columns is different. We further looked into the TPC-H queries that access the LINEITEM table (general observation) and predicted (using a layman-approach) the workload and data access patterns for columns. We identified that four columns involve read-intensive workload and three columns involve ordered data access as shown in Table 1. The differences in distinct data count, workload, and data access pattern for different columns raise the need for the support of storage structure customization at the column-level. If a storage manager supports the column-level customization of storage structures, we can hypothetically customize the LINEITEM table columns as shown in Table 1.

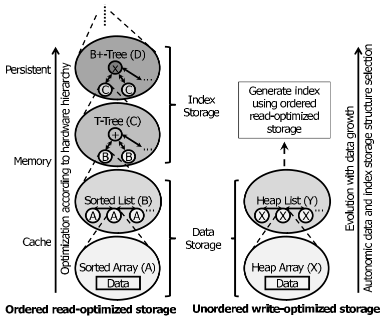

Persistent
Memory
Cache
Optimization according to hardware hierarchy
B+-Tree (D)
T-Tree (C)
Sorted List (B)
Sorted Array (A)
Data
Index Storage
Data Storage
Generate index using ordered read-optimized storage
Heap List (Y)
Heap Array (X)
Evolution with data growth
Autonomic data and index storage structure selection
Ordered read-optimized storage
Unordered write-optimized storage

**Fig. 1.** Evolving hierarchically-organized storage structures

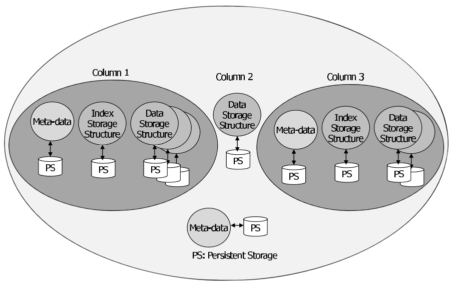


**Fig. 2.** Evolutionary column-oriented storage

***Why hierarchically-organized storage structures?*** A hierarchical organization of storage structures is a composition of similar or different storage structures in a hierarchy as depicted in Figure 1. Hierarchically-organized storage structures provide an opportunity for autonomic selection of appropriate storage structures along the hierarchy. We suggest that a new storage structure will be appropriate because we can use the existing data and gathered statistics during previous operations on existing storage structures to make better decisions for the next appropriate storage structure selection. The usage of hierarchically-organized storage structures is also motivated by the possible optimization of the storage structure hierarchy according to a hardware hierarchy and data management needs. For example, consider the memory hierarchy in the modern hardware. We optimize storage structures for cache, main memory, and persistent storage in the specified order. As shown in Figure 1, the lowest level of hierarchy is using arrays, which are optimized for cache. On the second level above, T-Trees are used, which are optimized for main memory. At the third level, B+-Tree is used, which is optimal for persistent storage. Previously published results from Bender et al. [5], Chen et al. [8], and Morzy et al. [14] also influenced our decision for the use of hierarchically-organized storage structures.

**Table 1.** TPC-H LINEITEM table observed statistics, possible customization, and anticipated evolution

| Column Name | Distinct Count | Workload | Data Access | Storage Structure Initial | Storage Structure 1st Evolution | Storage Structure 2nd Evolution |
|---|---|---|---|---|---|---|
| **L_ORDERKEY** | 1500000 | | | Sorted Array | Sorted List | B+-Tree |
| **L_COMMENT** | 4501941 | | | Sorted Array | Sorted List | Hash Table |
| **L_DISCOUNT** | 11 | Read-Intensive | | Sorted Array | | |
| **L_SHIPMODE** | 7 | | | Heap Array | | |
| **L_SHIPINSTRUCT** | 4 | | | Heap Array | | |
| **L_RECEIPTDATE** | 2554 | | | Heap Array | Heap List | |
| **L_COMMITDATE** | 2466 | | Ordered | Sorted Array | Sorted List | |
| **L_SHIPDATE** | 2526 | | Ordered | Sorted Array | Sorted List | |
| **L_LINESTATUS** | 2 | | | Heap Array | | |
| **L_RETURNFLAG** | 3 | | | Heap Array | | |
| **L_TAX** | 9 | Read-Intensive | | Sorted Array | | |
| **L_EXTENDEDPRICE** | 933900 | Read-Intensive | | Sorted Array | Sorted List | B+-Tree |
| **L_QUANTITY** | 50 | Read-Intensive | Ordered | Sorted Array | | |
| **L_LINENUMBER** | 7 | | | Heap Array | | |
| **L_SUPPKEY** | 10000 | | | Heap Array | Heap List | |
| **L_PARTKEY** | 200000 | | | Sorted Array | Sorted List | Hash Table |

## 3 Evolutionary Column-Oriented Storage

In this section, we explain the concepts of ECOS in detail. We introduce and explain four DSM based schemes proposed to reduce the high storage requirement of standard 2-copy DSM. We also discuss the concepts of the table and the column customization, hierarchical organization and evolution of the storage structures, and the evolution path.

### 3.1 Table-Level Customization

ECOS is a customizable and online self-tuning storage manager. We use the term storage manager in its standard meaning for DBMS, i.e., a component to physically store and retrieve data. Data storage efficiency is assumed to be the main goal for a storage manager. By storage structure, we mean the data structure used by the storage manager to physically store data and indexes. ECOS stores data according to the column-oriented storage model, where each column stores a key/value pair of data. ECOS suggests two customizations for each table in a database, i.e., at the table-level and at the column-level. At the table-level, we customize, how columns are stored physically for a logical schema design. We use five variations of DSM for table customization, i.e., Standard 2-copy DSM [10], Key-copy DSM (KDSM), Minimal DSM (MDSM), Dictionary based Minimal DSM (DMDSM), and Vectorized Dictionary based Minimal DSM (VDMDSM). The motivation for proposing and testing different variations of

**Table 2.** DSM

| Columnk0 | | Columnk1 | | Columnk2 | |
|---|---|---|---|---|---|
| Key | Value | Key | Value | Key | Value |
| **k1** | 731 | **k1** | 20090327 | **k1** | Jana |
| **k2** | 137 | **k2** | 20071201 | **k2** | Tobias |
| **k3** | 173 | **k3** | 20010925 | **k3** | Christian |
| **k4** | 371 | **k4** | 20090327 | **k4** | Tobias |
| **k5** | 317 | **k5** | 20090327 | **k5** | Tobias |
| **k6** | 713 | **k6** | 20010925 | **k6** | Jana |

(a) Columns clustered on key

| Columnv0 | | Columnv1 | | Columnv2 | |
|---|---|---|---|---|---|
| Key | Value | Key | Value | Key | Value |
| **k2** | 137 | **k3** | 20010925 | **k3** | Christian |
| **k3** | 173 | **k6** | 20010925 | **k1** | Jana |
| **k5** | 317 | **k2** | 20071201 | **k6** | Jana |
| **k4** | 371 | **k1** | 20090327 | **k2** | Tobias |
| **k6** | 713 | **k4** | 20090327 | **k4** | Tobias |
| **k1** | 731 | **k5** | 20090327 | **k5** | Tobias |

(b) Columns clustered on value

**Table 3.** MDSM

| Columnk1 | | Columnk2 | |
|---|---|---|---|
| Key | Value | Key | Value |
| **k1** | 20090327 | **k1** | Jana |
| **k2** | 20071201 | **k2** | Tobias |
| **k3** | 20010925 | **k3** | Christian |
| **k4** | 20090327 | **k4** | Tobias |
| **k5** | 20090327 | **k5** | Tobias |
| **k6** | 20010925 | **k6** | Jana |

(a) Columns clustered on key

| Columnv0 | |
|---|---|
| Key | Value |
| **k2** | 137 |
| **k3** | 173 |
| **k5** | 317 |
| **k4** | 371 |
| **k6** | 713 |
| **k1** | 731 |

(b) Primary key columns clustered on value

DSM arise from high storage requirement of standard 2-copy DSM. The details for the five variations of DSM are as follows:

***Standard 2-copy DSM.*** DSM is a transposed storage model [4], which pairs each value of a column with the surrogate of its conceptual schema record as key [10]. It suggests storing two copies of each column, one copy clustered on values, whereas another copy is clustered on keys. DSM is depicted in Table 2. We argue that for a self-tuning storage manager, 2-copy DSM is the most suitable storage model. It is easy to implement and easy to use, moreover, it does not require human intervention to identify which column to cluster or index, instead it is done in a uniform way [20]. To justify our argument, we evaluated standard 2-copy DSM with four other variations and found it the most appropriate one. The results are presented in Section 4.

***Key-copy DSM (KDSM).*** KDSM is the first variation of DSM that we propose to reduce the high storage requirement of the standard 2-copy DSM. KDSM stores the data similar to DSM, i.e., for each column, data is stored in values, whereas keys are unique numeric values that relate attributes of a row together. All columns are clustered on the keys. However, unlike DSM, we store an extra copy of only key columns (primary key or composite primary key) clustered on values. This design alteration reduces the storage requirement of KDSM, but it increases the access time for read operations that involve non-key columns in search criteria. However, for read operations with the key column in the search criteria it performs similar to DSM with less storage requirement as shown in Section 4. We propose the use of KDSM for tables that only require querying data using key columns.

***Minimal DSM (MDSM).*** MDSM stores the data similar to DSM except that we do not store any extra copy for any columns thus reducing the high storage requirement of DSM to a minimum. Instead, the design idea of MDSM is to store primary key columns clustered on values, whereas non-primary key columns are clustered on key as depicted in Table 3. MDSM performs similar to DSM and KDSM for the read operations with search criteria on key column attributes, but it performs worse for the read operations with non-key column attributes

**Table 4.** Dictionary columns for DMDSM and VDMDSM

| Dict. Column 0 | |
|---|---|
| **Keyd0** | **Valued0** |
| **d02** | 137 |
| **d03** | 173 |
| **d05** | 317 |
| **d04** | 371 |
| **d06** | 713 |
| **d01** | 731 |

| Dict. Column 1 | |
|---|---|
| **Keyd1** | **Valued1** |
| **d11** | 20090327 |
| **d12** | 20071201 |
| **d13** | 20010925 |

| Dict. Column 2 | |
|---|---|
| **Keyd2** | **Valued2** |
| **d23** | Christian |
| **d21** | Jana |
| **d22** | Tobias |

(a) Dictionary columns

**Table 5.** DMDSM

| Columnv0 | |
|---|---|
| **Keyv0** | **Valuev0** |
| **k2** | d02 |
| **k3** | d03 |
| **k5** | d05 |
| **k4** | d04 |
| **k6** | d06 |
| **k1** | d01 |

(a) Primary key columns clustered on value

| Columnk1 | | Columnk2 | |
|---|---|---|---|
| **Key** | **Value** | **Key** | **Value** |
| **k1** | d11 | **k1** | d21 |
| **k2** | d12 | **k2** | d22 |
| **k3** | d13 | **k3** | d23 |
| **k4** | d11 | **k4** | d22 |
| **k5** | d11 | **k5** | d22 |
| **k6** | d13 | **k6** | d21 |

(b) Columns clustered on key

**Table 6.** VDMDSM

| Vector Column | |
|---|---|
| **Key** | **Value** |
| **v1** | d01,d11,d21 |
| **v2** | d02,d12,d22 |
| **v3** | d03,d13,d23 |
| **v4** | d04,d11,d22 |
| **v5** | d05,d11,d22 |
| **v6** | d06,d13,d21 |

(a) Vector column

in search criteria as shown in Section 4. Our results in Section 4 suggest that if we do not have any space constraint and we do need access using non-key attributes, this scheme is not appropriate.

***Dictionary based Minimal DSM (DMDSM).*** To improve the performance of MDSM, we introduced DMDSM, which stores the unique data for each column separately as the dictionary column. DMDSM is inspired from the concept of the dictionary encoding scheme, which is frequently used as light-weight compression technique in many column-oriented data management systems [1]. In DMDSM, for each main column, values are the keys for the data from dictionary column as depicted in Table 5. All dictionary columns are clustered on value. All other concepts for the DMDSM are similar to MDSM. This scheme gives us the provision to exploit our innovative concept of evolving hierarchically-organized storage structures to its maximum potential for dictionary columns.

***Vectorized Dictionary based Minimal DSM (VDMDSM).*** VDMDSM is an extension of DMDSM, such that it stores the values (i.e., dictionary column keys) for all columns together as the vector column, i.e., instead of saving each column separately, it generates the vector of all attributes in the row and stores it as a value for vector column as depicted in Table 6. Similar to DMDSM, VDMDSM provides the opportunity to exploit the benefit of evolving hierarchically-organized storage structures to their full potential for dictionary columns.

## 3.2 Column-Level Customization and Storage Structure Hierarchies

Once we select the appropriate storage model scheme from above-mentioned schemes at the table-level, we move forward to customize the columns as explained next. At the column-level, we customize the storage structure for each column. Each column is initially customized as either ordered read-optimized or unordered write-optimized storage structure. For ordered read-optimized storage structures, we store data in sorted order with respect to key or value, whereas for unordered write-optimized storage structure, we store data according to insertion order. In the above-mentioned schemes, dictionary columns are always stored as ordered read-optimized storage structures.

***Evolving hierarchically-organized storage structures.*** ECOS utilizes the hierarchically-organized storage structure for data and index storage, such that a storage structure at each new level of hierarchy is composed of multiple lower level storage structures as depicted in Figure 1. The storage structures that we discuss in this paper include heap array, sorted array, heap list, sorted list, B+-Tree, T-Tree, and hash table. Before we continue our discussion, we outline the hierarchically-organized storage structures, which we use further in our discussion. At the lowest level of hierarchy, we used:

**Sorted array:** Optimized for read-access with minimal space overhead. No need to instantiate a buffer manager or an index manager to manage an array.
**Heap array:** Optimized for write-access with minimal space overhead.

At the next level, we used composite storage structures:

**Sorted list:** Sorted list is composed of multiple sorted arrays. It requires the instantiation of a buffer manager for managing multiple sorted arrays.
**Heap list:** Heap list is composed of multiple heap arrays. It also requires the instantiation of a buffer manager for managing multiple heap arrays.
**B+-Tree:** B+-Tree is composed of multiple arrays as leaf nodes. It requires the instantiation of a buffer manager for managing multiple arrays as well as an index manager to manage the multiple index nodes.

On the higher levels, we used high-level composite (HLC) storage structures:

**HLC SL:** HLC SL is a B+-Tree based structure, where each leaf node is a sorted list. HLC SL instantiates a buffer manager to manage multiple sorted lists and an index manager to manage multiple index nodes. Each sorted list manages its own buffer manager, which ensures the high locality of data for each sorted list.
**HLC B+-Tree:** HLC B+-Tree is a B+-Tree based structure, where each leaf node is also a B+-Tree. HLC B+-Tree instantiates a buffer manager to manage multiple B+-Trees and an index manager to manage multiple index nodes. Each B+-Tree at leaf nodes has its own buffer manager and index manager, which ensures the high locality of data and index nodes for each B+-Tree.

Once a column is customized as either ordered read-optimized or unordered write optimized storage structure, ECOS initializes each column to the smallest possible storage structure, i.e., an ordered read-optimized column is initialized as a sorted array, whereas an unordered write-optimized column is initialized as a heap array. ECOS enforces that each storage structure should be atomic and should be directly accessible using an access API. The reason for this approach is that small storage structures consume less memory and generate reduced binary size for small data management [18]. If we can use them directly, then there is no reason to use them as part of complex storage structures[1], such as

[1] We use storage structure as a common term for both data storage structure and index storage structure.

B+-Tree or T-Tree; avoiding the overheads of complexity associated with these storage structures. This approach ensures that using smallest suitable storage structures, desired performance is achieved using minimal hardware resources for small database management.

***Storage capacity limitations for predictable performance.*** ECOS imposes data storage capacity limitations for each storage structure. We enforce this for more predictable performance and to ensure that storage structure performance does not degrade because of unlimited data growth. In ECOS, once the limited storage capacity of a storage structure is consumed, it evolves to a larger more complex storage structure composed of multiple existing ones considering the important factors, such as hardware, the data growth, and the workload. For ordered read-optimized data storage, a sorted array is evolved into a sorted list. For unordered write-optimized data storage, a heap array is evolved into the heap list. The evolution of storage structure is an important event for assessing the next suitable storage structure by analyzing the existing data and the previously monitored workload. Similarly, each new storage structure also has a definite data storage capacity limitation and, once again, as it is consumed, ECOS further evolves and increases the hierarchy of the hierarchically-organized storage structures.

***API consistency to hide complexity and ensure ease of use.*** To hide the complexity of different storage structures over different levels of hierarchy, ECOS keeps the interface for all storage structures consistent. We provide a standard interface to access columns with simple, Put(), Get(), and Delete() functionality with record as argument. It is invisible to an end-user, which storage structure is currently in use for each column.

***Automatic partitioning.*** ECOS separates physical storage for each column to reduce the I/O contention for storing large databases. For large columns, it also separates the data for a column into multiple separate physical storage units, which is similar to horizontal partitioning. In Figure 2, at a minimum each column has its own separate physical storage. With the growth of data, each column may spread over multiple physical storage units. For example, for storage structures of Table 1, each sorted list or heap list will be stored in a separate data file, whereas each B+-Tree or T-Tree will be stored in a separate index file. These physical storage units may be stored on the single hard disk, or they may spread across the network.

## 3.3 Evolution and Evolution Paths

By evolution, we mean the transformation of a storage structure from an existing form into another form such that the previous form becomes an integral and atomic unit of the new form autonomically. Evolution path is the mechanism to define how ECOS evolves a smallest simple storage structure into a large complex storage structure. It consists of many storage structure/mutation rules pair entries that ECOS uses to identify, how to evolve the storage structures. Each

**Table 7.** Example for evolution paths

| **Storage Structure: Initial** | **Mutation Rules** | **Storage Structure: 1st Evolution** | **Mutation Rules** | **Storage Structure: 2nd Evolution** |
|---|---|---|---|---|
| **Sorted array** | **Event:**<br>Sorted array=Full<br>**Heredity based selection:**<br>Workload=Read intensive<br>Data access=Unordered<br>**Mutation:**<br>=> Evolve (Sorted array − >Sorted list) | Sorted list of sorted arrays | **Event:**<br>Sorted list=Full<br>**Heredity based selection:**<br>Workload=Read intensive<br>Data access=Ordered<br>**Mutation:**<br>=> Evolve (Sorted list − >B+-Tree) | B+-Tree of sorted lists(As leaf nodes for data storage) |
| **Sorted array** | **Event:**<br>Sorted array=Full<br>**Heredity based selection:**<br>Workload=Read intensive<br>Data access=Ordered<br>**Mutation:**<br>=> Evolve (Sorted array − >B+-Tree) | B+-Tree of sorted arrays(As leaf nodes for data storage) | **Event:**<br>B+-Tree=Full<br>**Heredity based selection:**<br>Workload=Read intensive<br>Data access=Ordered<br>**Mutation:**<br>=> Evolve (B+-Tree − >HLC (B+-Tree based)) | HLC of B+-Tree(As leaf nodes) |
| **Sorted array** | **Event:**<br>Sorted array=Full<br>**Heredity based selection:**<br>Workload=Write intensive<br>Data access=Unordered<br>**Mutation:**<br>=> Evolve (Sorted array − >Heap array) | Heap list based on heap array mutation rules | | |
| **Heap array** | **Event:**<br>Heap array=Full<br>**Heredity based selection:**<br>Workload=Write intensive<br>Data access=Ordered<br>**Mutation:**<br>=> Evolve (Heap array − >Heap list) &<br>Generate (Secondary index = Sorted list) | Heap list | **Event:**<br>Heap list=Full<br>**Heredity based selection:**<br>Workload=Write intensive<br>Data access=Ordered<br>**Mutation:**<br>=> Evolve (Heap list− >Hash table) &<br>Evolve (Secondary index = Sorted list − >B+-Tree) | Hash table |
| **Heap array** | **Event:**<br>Heap array=Full<br>**Heredity based selection:**<br>Workload=Write intensive<br>Data access=Unordered<br>**Mutation:**<br>=> Evolve (Heap array − >Heap list) | Heap list | **Event:**<br>Heap list=Full<br>**Heredity based selection:**<br>Workload=Write intensive<br>Data access=Unordered<br>**Mutation:** => Evolve (Heap list − >Hash table) | Hash table |

storage structure can have multiple mutation rules mapped to it. These mutation rules consist of three information elements: Event, Heredity based selection, and Mutation. The event identifies, when this mutation rule should be executed. Different mutation rules can have the same event, but not all of them execute the mutation. The heredity based selection identifies precisely, when evolution should occur based on the heredity information gathered for the existing storage structure. Heredity information means the gathered statistics about the storage structure, e.g., workload type, data access pattern, previous evolution details, etc. The mutation defines the actions that should be executed to evolve the storage structure. Example of an evolution path is shown in Table 7. We envision that common DBMS maintenance best practices can be documented using the evolution path mechanism. ECOS assumes that DBMS vendors provide the evolution paths that best suit their DBMS internals, with the provision of alteration for a database administrator. The only liability for configuration that lies with database designers and administrator is to have a look at the evolution path for the DBMS and if needed, alter it with desired changes. Evolution process in

ECOS is autonomic, and it exploits evolution path to automatically evolve the storage structures, i.e., our approach for self-tuning is online.

Consider the L_ORDERKEY column of the LINEITEM table as shown in Table 1. Suppose, as a database designer, we design this table. According to our application design, we select the L_ORDERKEY column as a part of the primary key. As we already discussed in Section 3, we have to customize each column as either ordered read-optimized or unordered write-optimized. Therefore, we customize the L_ORDERKEY column as ordered read-optimized. At the initial design time, we design according to the domain knowledge, our experiences, and predictions. As a designer, it is difficult to guarantee, how much this column grows, and how long it takes to reach that size. When we customize the column as ordered read-optimized, it is internally initialized as a sorted array. Now for the L_ORDERKEY column, three initial rows of the sample evolution path of Table 7 are relevant.

As we mentioned in Section 3, ECOS limits the storage capacity for each storage structure. Therefore, the initial sorted array has a certain data storage capacity limit. For example, consider it as 4KB. As long as data is within the 4KB limits, sorted array is the storage structure for the L_ORDERKEY column, and we gather the heredity information for the column, such as the number of Get(), the number of Put(), the number of Delete(), the number of range Get() (for range queries), the number of Get() for all records (for scan queries), etc. What heredity information should be gathered may vary from one implementation to another. Here, we simplify our discussion by assuming that a system can identify using heredity information that the workload is either read-intensive or write-intensive and the access to data is either ordered (range queries) or unordered (point or scan queries).

The moment the storage limit of the sorted array is consumed, an event is raised for notification. This event triggers all three initial mutation rules of Table 7. Now heredity based selection identifies, which one of them to execute. We suppose that for the L_ORDERKEY column, the workload is read-intensive and the data access is unordered, this scenario executes the first mutation rule of Table 7, which evolves the existing sorted array into a sorted list. Now sorted list is the new storage structure, and it is also constrained with the storage limit according to the design principle of ECOS. As long as the L_ORDERKEY column data is within the storage limit of the sorted list, heredity information is gathered, and it is used for the next evolution.

It is observed from Table 1 that only half of the LINEITEM columns, i.e., eight out of sixteen with high data growth evolve during the first evolution. The rest of the columns can be stored within an array (either heap array or sorted array). Furthermore, only half of the columns with high data growth, i.e., four out of eight, which are evolved during the first evolution evolve again during the second evolution (i.e., L_ORDERKEY, L_COMMENT, L_EXTENDEDPRICE, and L_PARTKEY). The final state of the table presented in Table 1 shows that each column is using the appropriate storage structure (we assume for explanation) according to the stored data size and observed workload. We can add more parameters for evolution decisions, but we only used limited parameters (i.e., data size, workload, and data access) to keep our discussion simple and understandable. Table 1 shows only the evolution for dictionary columns for the LINEITEM table as they utilizes the benefits of evolving hierarchically-organized storage structures to their full potential. Before we conclude this section, to avoid any confusion, we want to disclaim that the terms and concepts of evolution, evolution path, mutation rules, and heredity information used in this paper have no relevance with their counterpart in evolutionary algorithms or any other non-relevant domain.

## 4 Implementation and Empirical Evaluation

In this section, we provide the details of our micro benchmark and the evaluation results for ECOS[2]. The data and index storage structures that we have implemented in the existing ECOS prototype implementation are the same as we have discussed in Section 3.2. To simplify our discussion, we present the results involving sorted array, sorted list, and HLC SL.

### 4.1 Micro Benchmark Details

For ECOS evaluation, we set up a micro benchmark with repeated insertion, selection, and deletion of data using API based access methods. The data contain keys in ascending, descending, and random order, which also represents their insertion, selection, and deletion order in the database. For different columns, the number of records (cardinality) is kept different. We defined seven columns with two unique non-null columns, one of them used as a primary key. We used three different widths for columns, i.e., 16, 85, and 4096 bytes to assess the impact of tuple width on performance of different storage schemes. All storage structures used in a micro benchmark operate in main-memory. For ECOS evaluation, we used CPU cycles and heap memory as resources. We used OpenSuse 11.2 operating on Intel(R) Core(TM)2 Duo CPU E6750 @ 2.66GHz with four GB of RAM. We measured execution speed by taking the average of CPU cycles observed over multiple iterations of the micro benchmark. We used Valgrind tools suite [21] to measure the heap usage.

### 4.2 ECOS Performance Improvement

To demonstrate the performance gain using ECOS, we presented our observation of the effect of an increase in data size on performance of different storage structures in [18,19]. According to our observation in [18,19], we suggest the performance gain and reduced resource consumption using the evolving storage structures because evolving storage structures attempt to use minimal/simple storage structures (such as sorted array for small data management) as long as possible using the definitions from evolution paths. To demonstrate the evolving storage structures evolution, we present the evaluation results for evolving

[2] Please refer to the web link for all related publications and prototype evaluation binaries: `http://wwwiti.cs.uni-magdeburg.de/~srahman/CellularDBMS/index.php`

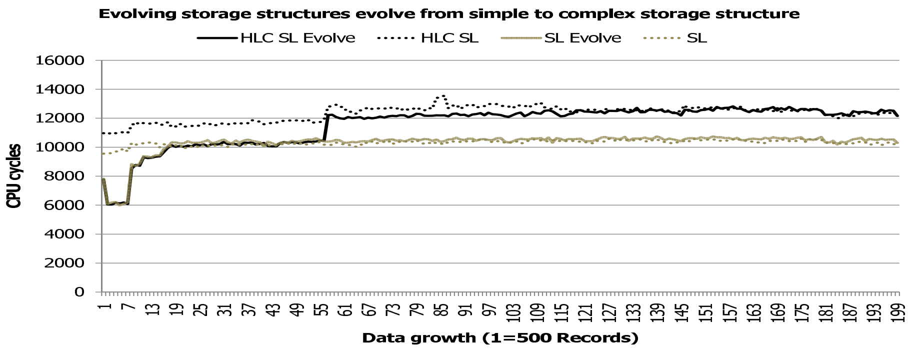


**Fig. 3.** Evolving HLC SL storage structure evolution

HLC SL storage structure in Figure 3 (due to space constraint the evaluation results for evolving HLC B+-Tree can be found in [19]). It can be observed in Figure 3 that the evolving storage structure HLC SL evolves with the data growth. It can be observed that the HLC SL storage structure consume more CPU cycles in comparison with sorted list and sorted array. This behavior is due to the complexity of the storage structure, which is meant to be used for extremely large data sizes. The HLC SL storage structure automatically partitions the data and uses separate buffer and index managers for each partition, which is not the requirement for presented 500K records storage. However, we forced storage structures to evolve to HLC SL for 500K records for the purpose of demonstration of the evolution concept.

In Figure 4 and 5, we present the performance comparison of different DSM based schemes that we explained in Section 3. The results in Figure 4 and 5 show that DSM and KDSM perform better for evaluation with search criteria on key-attributes, whereas for evaluation with search criteria on non-key attributes DSM outperforms the other schemes. It is observed that storage requirement for DSM is highest, whereas the storage requirement is the lowest for VDMDSM. It is

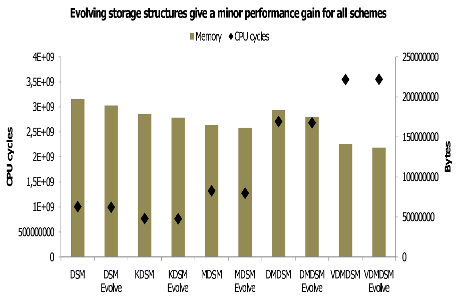


**Fig. 4.** Performance comparison of different DSM based schemes in ECOS with primary key based search criteria

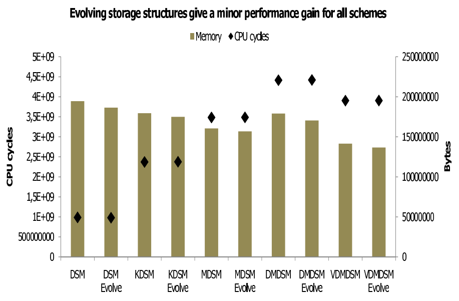
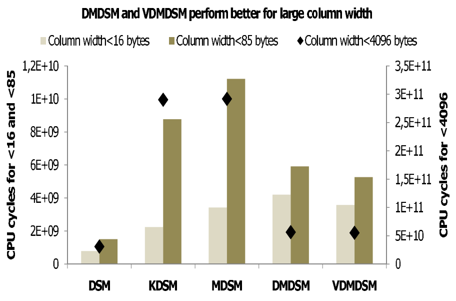


**Fig. 5.** Performance comparison of different DSM based schemes in ECOS with non-key based search criteria



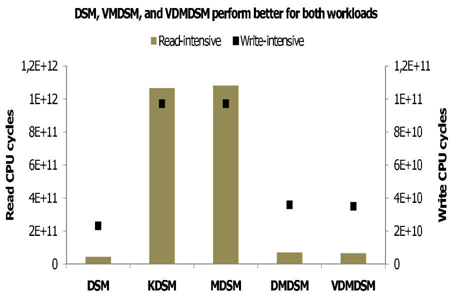


**Fig. 6.** Performance improvement for dictionary based DSM schemes for large column width

**Fig. 7.** Performance comparison of different DSM based schemes in ECOS for read and write intensive workloadsf

also observed that evolving storage structures perform better than fixed storage structures with minor performance gains. As we have discussed in Section 1, our work is based on the ideology from Chaudhuri and Weikum presented in [6]. They used the notion of “gain/pain ratio” to discuss the overall gain of their proposed approach. They advocate the ideology of less complex, more predictable, and self-tuning RISC-style components with minor compromise on performance to achieve overall improvement in “gain/pain ratio”. Our results show the minor performance gain, which should be a good achievement considering the overall benefits we achieve in terms of simplicity, predictability, and self-tuning.

It can be observed in Figure 6 that dictionary based schemes performance is improved and becomes comparable with standard 2-copy DSM scheme for large tuple width. However, KDSM and MDSM still perform poor. We also analyzed the performance difference for different DSM schemes on both the read-intensive and write-intensive workloads. It is observed in Figure 7 that for write-intensive workload DSM outperforms other schemes; however, for the read-intensive workload differences in performance between the 2-copy DSM and the dictionary based DSM schemes is minimum. This is a promising result for dictionary based schemes, and it shows their potential to act as a better alternative to 2-copy DSM after overcoming their short comings.

## 5 Related Work

Hierarchically-organized storage structures have already been in use in the data warehousing domain. Morzy et al. in [14] proposed a hierarchical bitmap index for indexing set-valued attributes. Later, Chmiel et al. in [9] extended that concept to present hierarchically-organized bitmap indexes for indexing dimensional data. Bender et al. proposed cache-oblivious B-Trees [5] that perform the optimal search across different hierarchical memories with varying memory levels, cache size, and cache line size. Fractal prefetching B+-Trees [8] proposed by Chen et al. are the most relevant work for the ECOS and is similar in concept to cache-oblivious B-Trees with an additional concept of prefetching. Fractal prefetching

B+-Trees are optimized for both cache and disk performance, which is also a goal for the ECOS. However, the ECOS concepts do not restrict the use of any fixed structure; instead it suggests the use of different storage structures in the hierarchy to support an efficient use of underlying hardware.

An automated tuning system (ATS) [11] is a feedback control mechanism that automatically adjusts the tuning knobs using the defined tuning policies according to the monitoring statistics. ECOS also works in similar fashion as suggested in ATS. ECOS also monitors and adjust storage structures with changing data management needs. Malik et al. in [13] suggested the benefit of online physical design techniques and proposed an online vertical partitioning technique for physical design tuning. Similarly, ECOS also operates in online fashion. Automated physical design research focuses on finding the best physical design structure for a running workload, e.g., indexes, materialized views, partitioning, clustering, and views [3]. Existing automated physical design tools assume the workload as a set of SQL statements [3]. These tools use the query optimizer to identify the appropriate physical design selection from various proposed candidate designs [15]. ECOS also performs automated physical design, but at the different level, i.e., at the storage manager level. It does not rely on a query optimizer. Furthermore, ECOS is designed with the motivation of exploring new architectures for developing self-tuning DBMS instead of developing techniques to self-tune existing ones.

## 6 Conclusion and Future Work

In this paper, we presented ECOS, a customizable and online self-tuning storage manager. ECOS and evolution paths enable and use the fine-grained customization of storage structures at the table-level and column-level. In addition, ECOS and evolution paths allow storage structures to autonomically evolve (to more suitable storage structures) with the change in the data management needs, to maintain the desirable performance while keeping the human intervention at a minimum. We also presented a detailed evaluation and discussion of ECOS and evaluation paths showing the performance improvement and reduced resource consumption. As future work, we plan to enhance the presented dictionary based DSM schemes for better performance. ECOS self-tuning design makes it a suitable candidate for emerging cloud computing platforms for data services. We also intend to investigate the efficient utilization of multi-core and many-core parallel processors using the presented evolution mechanism. Once query processing is implemented, we want to integrate the presented evolution mechanism with query processing, and then we will be able to evaluate the ECOS using the full TPC-H benchmark. Transaction management is also an implementation specific future work for our ECOS prototype.

**Acknowledgments.** Syed Saif ur Rahman is a HEC-DAAD Scholar funded by Higher Education Commission of Pakistan and NESCOM, Pakistan.

## References

1. Abadi, D.J., Madden, S.R., Ferreira, M.C.: Integrating compression and execution in column-oriented database systems. In: SIGMOD, pp. 671–682 (2006)
2. Abadi, D.J., Madden, S.R., Hachem, N.: Column-stores vs. row-stores: how different are they really? In: VLDB, pp. 967–980 (2008)
3. Agrawal, S., Chu, E., Narasayya, V.: Automatic physical design tuning: workload as a sequence. In: SIGMOD, pp. 683–694 (2006)
4. Batory, D.S.: On searching transposed files. ACM Trans. Database Syst. 4(4), 531–544 (1979)
5. Bender, M.A., Demaine, E.D., Farach-Colton, M.: Cache-oblivious B-trees. In: FOCS, pp. 399–409 (2000)
6. Chaudhuri, S., Weikum, G.: Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In: VLDB, pp. 1–10 (2000)
7. Chaudhuri, S., Weikum, G.: Foundations of automated database tuning. In: SIGMOD, pp. 964–965 (2005)
8. Chen, S., Gibbons, P.B., Mowry, T.C., Valentin, G.: Fractal prefetching B+-Trees: optimizing both cache and disk performance. In: SIGMOD, pp. 157–168 (2002)
9. Chmiel, J., Morzy, T., Wrembel, R.: HOBI: Hierarchically Organized Bitmap Index for Indexing Dimensional Data. In: Pedersen, T.B., Mohania, M.K., Tjoa, A.M. (eds.) DaWaK 2009. LNCS, vol. 5691, pp. 87–98. Springer, Heidelberg (2009)
10. Copeland, G.P., Khoshafian, S.N.: A decomposition storage model. SIGMOD Rec. 14, 268–279 (1985)
11. Hellerstein, J.L.: Automated tuning systems: Beyond decision support. In: CMG, Computer Measurement Group, pp. 263–270 (1997)
12. Holloway, A.L., DeWitt, D.J.: Read-optimized databases, in depth. Proc. VLDB Endow. 1, 502–513 (2008)
13. Malik, T., Wang, X., Burns, R., Dash, D., Ailamaki, A.: Automated physical design in database caches. In: ICDE Workshop, pp. 27–34 (2008)
14. Morzy, M., Morzy, T., Nanopoulos, A., Manolopoulos, Y.: Hierarchical Bitmap Index: An Efficient and Scalable Indexing Technique for Set-Valued Attributes. In: Kalinichenko, L.A., Manthey, R., Thalheim, B., Wloka, U. (eds.) ADBIS 2003. LNCS, vol. 2798, pp. 236–252. Springer, Heidelberg (2003)
15. Papadomanolakis, S., Dash, D., Ailamaki, A.: Efficient use of the query optimizer for automated physical design. In: VLDB, pp. 1093–1104 (2007)
16. Patterson, D.A., Ditzel, D.R.: The case for the reduced instruction set computer. SIGARCH Comput. Archit. News 8, 25–33 (1980)
17. TPC-H, `http://www.tpc.org/tpch/`
18. ur Rahman, S.S.: Using Evolving Storage Structures for Data Storage. In: FIT, pp. 30:1–30:6 (2010)
19. ur Rahman, S.S., Schallehn, E., Saake, G.: ECOS: Evolutionary Column-Oriented Storage. Tech. Rep. FIN-03-2011, Department of Technical and Business Information Systems, Faculty of Computer Science, University of Magdeburg (2011)
20. Valduriez, P., Khoshafian, S.N., Copeland, G.P.: Implementation Techniques of Complex Objects. VLDB, 101–110 (1986)
21. Valgrind, `http://www.valgrind.org`
22. Weikum, G., Moenkeberg, A., Hasse, C., Zabback, P.: Self-tuning database technology and information services: from wishful thinking to viable engineering. In: VLDB, pp. 20–31 (2002)