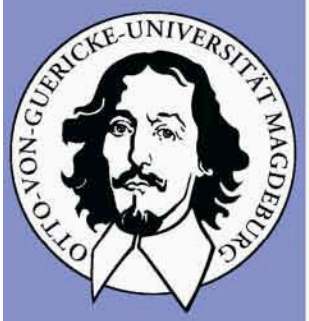

Technical report

Nr.: FIN-03-2011

ECOS: Evolutionary Column-Oriented Storage

Syed Saif ur Rahman, Eike Schallehn, and Gunter Saake

Arbeitsgruppe Datenbanken

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Nr.: FIN-03-2011

ECOS: Evolutionary Column-Oriented Storage

Syed Saif ur Rahman, Eike Schallehn, and Gunter Saake

Arbeitsgruppe Datenbanken

Technical report (Internet)
Elektronische Zeitschriftenreihe
der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
ISSN 1869-5078

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG)

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Syed Saif ur Rahman
Postfach 4120
39016 Magdeburg
E-Mail: srahman@ovgu.de

http://www.cs.uni-magdeburg.de/Technical_reports.html

Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 15.03.2011

Bezug: Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Dekanat

ECOS: Evolutionary Column-Oriented Storage

Syed Saif ur Rahman, Eike Schallehn, Gunter Saake
{srahman, eike, saake}@ovgu.de

Technical Report

Department of Technical and Business Information Systems,
Faculty of Computer Science,
Otto-von-Guericke University,
Magdeburg, Germany

Abstract

As DBMS has grown more powerful over the last decades, they have also become more complex to manage. To achieve efficiency by database tuning is nowadays a hard task carried out by experts. This development inspired the ongoing research on self-tuning to make database systems more easily manageable. In this report, 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 DBMS according to data size and access characteristics. It is based on the Decomposed Storage Model (DSM) with support for customization at the table-level using five different variations of DSM. Furthermore, it also proposes fine-grained customization of storage structures at the column-level. It uses hierarchically-organized storage structures for each column to enable autonomic selection of the suitable storage structure along the hierarchy (as hierarchy-level increases) using an evolution mechanism. 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.

Contents

List of Figures

List of Tables

Listings

1 Introduction

Efficient data management demands continuous tuning of a database and a DBMS. The need for tuning a database system is driven by changes, such as database size, workloads, schema design, hardware, and application specific data management needs. Existing data management systems need extensive human intervention for tuning, which contributes to a major portion of the total cost of ownership for data management [9]. Self-tuning is the solution to reduce the tuning cost through minimizing the human intervention [32]. There are several different self-tuning based solutions for commercial DBMS, e.g., AutoAdmin [7], Oracle automatic SQL tuning [13], DB2 design advisor [33], etc. However, researchers are united on one conclusion that the biggest challenge for self-tuning based solutions is the inherent complexity of existing DBMS architectures because their functionalities are tightly integrated into their monolithic engines, and it is difficult to assess the impact of tuning of one knob on another [8]. This motivates the need to revisit existing DBMS architectures to explore an out of the box and innovative architecture.

In this report, we present a customizable, online self-tuning storage manager. As a key design concept, we propose the selection of an appropriate storage model and data/index storage structure through customization at a fine-granularity. The motivation for this customization is inspired from the work of Chaudhuri and Weikum [8]. The fine-grained customization is supported at the table-level and column-level according to the recommendations/results from [2, 12, 17]. 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 based on our previously published results in [28]. We named our solution Evolutionary Column-Oriented Storage (ECOS), which is based on the existing Decomposed Storage Model (DSM) [12] with the novel capability of automatically evolving the internal data and index storage structures for each column with the growth of data. It uses hierarchically-organized storage structures with an innovative evolution mechanism to enable autonomic selection of the most suitable storage structure along the hierarchy (as the levels of the hierarchy increase). Furthermore, we present four possible variations to standard DSM to reduce the increased storage requirements of the standard 2-copy DSM. We evaluated ECOS empirically using the custom micro benchmark to compare the performance of different DSM based schemes using fixed storage structures as well as proposed evolving storage structures. Our results show that our proposed 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 report is organized as follows. Section 2 defines the problem and the motivation for proposed design. Section 3 explains the concepts of ECOS and evolution path in detail. Section 4 gives brief theoretical explanation of evolving storage structures using time and space complexity analysis. Section 5 briefly outlines ECOS evolution mechanism implementation. Section 6 gives details of empirical evaluation of proposed concepts using custom micro benchmark. Section 7 outlines the related work. Section 8 concludes the report with some hints for future work.

2 Problem Statement and Motivation

Specific storage structures provide characteristics suitable for certain data sizes and access characteristics. 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 TPC Benchmark^{TM}H (TPC-H) [27] schema as an example. We generated the benchmark data with the scale factor of one and gathered statistics for LINEITEM table as shown in Table 1.

Why column-oriented storage model? The column-oriented storage model is derived from earlier work of DSM [12]. DSM is a transposed storage model [5] that stores all values of the same attribute of the relational conceptual schema relation together [12]. In literature, models similar to DSM are also termed as vertical fragmentation [14], vertical partitioning [3], etc. Copeland and Khoshafian in [12,30] concluded many advantages of DSM including simplicity (Copeland and Khoshafian related it to RISC [24]), 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 DSM in self-tuning storage manager.

The column-oriented storage model has recently gained more attention, most of all because of its superior performance for analytical data applications [26]. Accordingly, we see the current developments in column-oriented storage solutions as an opportunity to address the problem of self-tuning storage structures. Nevertheless, basic ideas and concepts are also applicable to the more traditional row-stores and the transfer there could be a point of future research.

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 can classify three types of columns based on distinct data count, i.e., large, medium, and small. We further observed (general observation) the TPC-H queries that access LINEITEM table and predicted (using a layman-approach) the workload and data access pattern for columns. We identified that four columns (i.e., L_DISCOUNT, L_TAX, L_EXTENDEDPRICE, and L_QUANTITY) involve read-intensive workload, whereas three columns (i.e., L_COMMITDATE, L_SHIPDATE, and L_QUANTITY) involve ordered data access. 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 column-level customization of storage structure, we can hypothetically customize LINEITEM table columns as shown in Table 1.

Need for customization is also suggested by other research and commercial data management solutions. C-Store [26] proposed the use of two different stores within same DBMS, i.e., read-optimized and write-optimized stores. Another customization they proposed is that write-optimized store operates in main-memory fashion. Dynamo [15], a highly available key-value store from Amazon, uses pluggable architecture for storage engine. It enables the choice of the storage engine that best suits the data management need for application, i.e., BerkeleyDB can be used to store database of few kilo bytes, whereas for database of large size, MySQL can be used [15]. MySQL DBMS also supports storage engine customization at the table-level.

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 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. Previously published results from Bender et al. [6], Chen et al. [10], and Morzy et al. [21] also motivate our decision for the use of hierarchically-organized storage structures.

Why autonomy? Consider the distinct data count of two large columns, i.e., L_ORDERKEY and L_COMMENT in Table 1. For the benchmark scenario, we generate the data altogether to test our data management solutions, and we customize the storage structure to best suit our desired results. However, in a real world scenario, the data growth is a continuous process. Database designer can predict, how large data can grow and at what rate, but he/she should maintain the database over time.

We can elaborate the problem with two possible scenarios. For example, in a first scenario we suggest a B+-Tree as a suitable storage structure (assume data stored with index) for the L_ORDERKEY column, but what if only after 30 years the expected maximum data size is reached? During the first year, a sorted list could have been good enough to store the data. When we select a complex storage structure for small database management, for each data management operation, we waste resources (cache, memory, and CPU cycles) until and unless data size grows to make the use of the selected storage structure appropriate. For the contrary

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

second possible scenario, a database designer selects a sorted list as a storage structure. However, the data growth is much higher than expected. In a year the sorted list becomes inadequate for the desired performance. The database need maintenance that include changing the storage structure by human intervention.

Another important issue is the change in the workload pattern for a column. It is possible that a column that was previously accessed with write-intensive queries, later on in the lifetime becomes more read-intensive. A classical approach as a solution would require manual analysis of the queries and then the transformation of a table for using appropriate storage structures. Therefore, we suggest that using an autonomic approach of evolving hierarchically-organized storage structures in conjunction with customization at the table and column level, the process of self-tuning data and index storage structures with change in workload can be automated at the storage manager level.

3 Evolutionary Column-Oriented Storage

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

3.1 Table-level Customization

ECOS is a customizable 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 Decomposed Storage Model (DSM) for table customization, i.e., Standard 2-copy Decomposed Storage Model (DSM) [12], Key-copy Decomposed Storage Model (KDSM), Minimal Decomposed Storage Model (MDSM), Dictionary based Minimal Decomposed Storage Model (DMDSM), and Vectorized Dictionary based Minimal Decomposed Storage Model (VDMDSM). The motivation for proposing and testing different variations of DSM arise from high storage requirements of standard 2-copy DSM. The details for the five variations of DSM are as follows:

Table 2: DSM.

Columnk0	
Key	**Value**
k1	731
k2	137
k3	173
k4	371
k5	317
k6	713

Columnk1	
Key	**Value**
k1	20090327
k2	20071201
k3	20010925
k4	20090327
k5	20090327
k6	20010925

Columnk2	
Key	**Value**
k1	Jana
k2	Tobias
k3	Christian
k4	Tobias
k5	Tobias
k6	Jana

(a) Columns clustered on key

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

Columnv1	
Key	**Value**
k3	20010925
k6	20010925
k2	20071201
k1	20090327
k4	20090327
k5	20090327

Columnv2	
Key	**Value**
k3	Christian
k1	Jana
k6	Jana
k2	Tobias
k4	Tobias
k5	Tobias

(b) Columns clustered on value

Standard 2-copy Decomposed Storage Model (DSM) DSM is a transposed storage model [5], which pairs each value of a column with the surrogate of its conceptual schema record as key [12]. It suggests storing two copies of each column, one copy clustered on values, whereas another copy is clustered on keys. We took DSM as the base storage model and then altered it to propose different schemes. We suggest that DSM is suitable for read-intensive workloads where data contain negligible duplicate and NULL values, write and updates are minimal relative to read operations and there are negligible storage constraints. 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 [30]. 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 6.

Table 3: KDSM.

Columnk0	
Key	**Value**
k1	731
k2	137
k3	173
k4	371
k5	317
k6	713

Columnk1	
Key	**Value**
k1	20090327
k2	20071201
k3	20010925
k4	20090327
k5	20090327
k6	20010925

Columnk2	
Key	**Value**
k1	Jana
k2	Tobias
k3	Christian
k4	Tobias
k5	Tobias
k6	Jana

(a) Columns clustered on key

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

(b) Columns clustered on value

Key-copy Decomposed Storage Model (KDSM) KDSM is the first variation of DSM that we propose to reduce the high storage requirements of the standard 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 fewer storage requirements. We propose the use of KDSM for tables that only require querying data using key columns. KDSM allows a conversion to DSM by simply creating a copy of the

non-key columns clustered on values. We suggest that KDSM is suitable for data storage where columns have few duplicates and NULL values. KDSM is shown in Table 3.

Table 4: MDSM.

Columnk1	
Key	**Value**
k1	20090327
k2	20071201
k3	20010925
k4	20090327
k5	20090327
k6	20010925

(a) Columns clustered on key

Columnk2	
Key	**Value**
k1	Jana
k2	Tobias
k3	Christian
k4	Tobias
k5	Tobias
k6	Jana

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

(b) Primary key columns clustered on value

Minimal Decomposed Storage Model (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 4. 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 in search criteria. MDSM can be transformed into KDSM and DSM by creating an extra copy of the key columns clustered on key and non-key columns clustered on values. However, our results in Section 6 suggest that if we do not have any space constraints, this scheme is not appropriate.

Dictionary based Minimal Decomposed Storage Model (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 6. All dictionary columns are clustered on value. All other concepts for the DMDSM are similar to MDSM. DMDSM is suitable for tables with many duplicates or NULL values. In this scheme, for columns, database operators always manipulate numeric data for data

Table 5: 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 6: DMDSM.

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

(a) Primary key columns clustered on value

Columnk1	
Key	**Value**
k1	d11
k2	d12
k3	d13
k4	d11
k5	d11
k6	d13

Columnk2	
Key	**Value**
k1	d21
k2	d22
k3	d23
k4	d22
k5	d22
k6	d21

(b) Columns clustered on key

Table 7: 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

management operations, which execute much faster on modern hardware. Furthermore, it gives us the provision to exploit our innovative concept of evolving hierarchically-organized storage structures to its maximum potential for dictionary columns because they only store non-null unique data and most of them can be stored using simple and small storage structures.

Vectorized Dictionary based Minimal Decomposed Storage Model (VDMDSM) In DMDSM each column stores keys/values, where values are record identifiers from dictionary columns. We can optimize this with a better storage scheme by avoiding the storage of keys for every column separately. 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 7. Similar to DMDSM, VDMDSM provides the opportunity to exploit the benefit of evolving hierarchically-organized storage structures to their full potential for dictionary columns. VDMDSM is suit-

able for tables with many duplicate or NULL values.

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. We use the sorted array and the sorted list as ordered read-optimized data storage structures, whereas the heap array and the heap list is used as unordered write-optimized data storage structure. In the above-mentioned schemes, dictionary columns are always stored as ordered read-optimized storage structures.

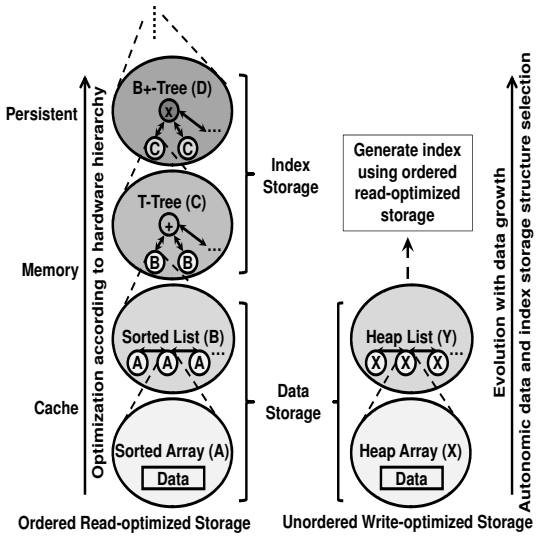

Figure 1: Evolving hierarchically-organized storage structures.

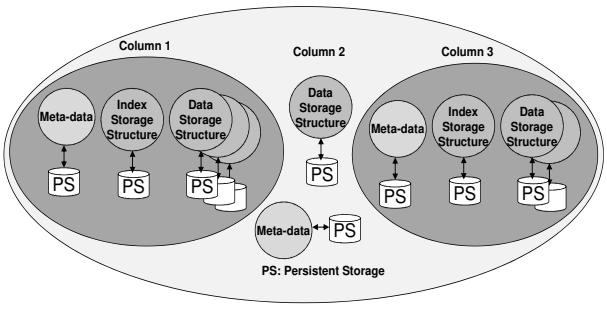

Figure 2: Evolutionary column-oriented storage.

Evolving hierarchically-organized storage structure 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 usage of hierarchically-organized

storage structures is motivated by the possible optimization of the storage structure hierarchy according to hardware hierarchy and data management needs. For example, consider the memory hierarchy in 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 array storage structures, which are optimized for cache. On the second level above, T-Tree storage structure is used, which is optimized for main memory. At the third level, B+-Tree is used, which is optimal for persistent storage.

The storage structures that we discussed in this report include heap array, sorted array, heap list, sorted list, B+-Tree, T-Tree, and hash table. From heap array/list, we mean a storage structure that always appends new data to existing data in chronological order and uses the linear search algorithm to traverse the data. From sorted array/list, we mean storage structures that always maintain the sort order for the data. For data retrieval sorted array uses the binary search algorithm. B+-tree, T-Tree, and hash table operate according to their de facto standards. 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 use:

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 use 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 use 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 manage 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, ECOS initializes each column to smallest possible storage structure, i.e., ordered read-optimized column is initialized as a sorted array, whereas 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 [28]. If we can use them directly, than there is no reason to use them as part of complex storage structures (we use storage structure as a common term for both data storage structure and index storage structure), such as 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 limitation for predictable performance ECOS imposes data storage capacity limitation 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 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, such that the sorted list is composed of multiple sorted arrays linked together. 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. For ordered read-optimized data storage, once sorted list storage capacity is consumed it evolves into new storage structure, such that it becomes part of a new index structure. For example, it becomes the data leaf node of a B+-Tree. For ordered read-optimized

data storage, ECOS does not perform data management operations separately for data and index structures, instead, each operation interact directly with the index structure. Here-onwards, index structure will identify, in which sorted list the data will be stored. For unordered write-optimized storage, operations execute separately on data and index structures, such that first data is inserted into the heap list and then the index structure is updated with the new key or index value. Index structures for unordered write-optimized storage are based on ordered read-optimized storage and will evolve subsequently.

API consistency to hide complexity and ensure ease of use To hide the complexity of different storage structure 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 storage of large database. 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, storage structures of Table 1, each sorted list or heap list is stored in a separate data file, whereas each B+-Tree or T-Tree is 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. This separation also allows using different compression algorithms for each column (or each physical storage unit) based on the data type.

Meta-data for efficient traversal ECOS proposes to maintain important meta-data for efficient traversal of the hierarchically-organized storage structures, which includes count, minimum key/value, and maximum key/value for each storage structure. This avoids the access to unnecessary data and improves the efficiency of hierarchy traversal. ECOS also proposes to maintain the frequently used important aggregates, e.g., summation, average, etc., as the meta-data at every level of hierarchy. The request for these aggregates should be satisfied by accumulating them using the meta-data to reduce the overhead of accessing each value separately to calculate them again and again.

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 storage structure can have multiple mutation rules mapped to it. These mutation rules consist of three information elements, i.e., 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 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 a sample evolution path is shown in Table 8.

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 alter it with desired changes, if needed. 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, as a sample case we customize the L_ORDERKEY column as ordered read-optimized. However, 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. As we customize the column as ordered read-optimized, it is initialized as a sorted array. Now for the L_ORDERKEY column, three initial rows of the sample evolution path of Table 8 are relevant.

As we mentioned in Section 3, ECOS limits the storage capacity of 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,

Table 8: Example for evolution paths.

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

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 point Get() (for point queries), 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) or unordered (point or all).

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 8. 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 8, which evolves the existing sorted array into a sorted list. Now-onwards sorted list is the storage structure for L_ORDERKEY column, 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 columns in LINEITEM table with high data growth (i.e., eight out of sixteen) evolves during first evolution (i.e., L_ORDERKEY, L_EXTENDEDPRICE, L_RECEIPTDATE, L_COMMITDATE, L_SHIPDATE, L_SUPPKEY, L_PARTKEY, and L_COMMENT). The rest of the columns can be stored within an array (either heap array or sorted array). Furthermore, only half of the columns, i.e., four out of eight, which are evolved during first evolution evolve again during the second evolution (i.e., L_ORDERKEY, L_COMMENT, L_EXTENDEDPRICE, and L_PARTKEY). The final state of 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 decision, but we only used limited parameters (i.e., data size, workload, and data access) to keep our discussion simple and understandable.

What heredity information should be gathered for each storage structure, and how to improve the efficiency of storage and retrieval of heredity information is a separate topic. Here, we simplify our discussion with an assumption that we have an efficient and precise mechanism for gathering heredity information. As a sample demonstration of how the LINEITEM table evolves for the sample evolution path in Table 8 is shown in Table 1. 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 report have no relevance with their counterpart in evolutionary algorithms or any other non-relevant domain.

4 Theoretical Explanation

In this section, we provide the theoretical explanation of evolving hierarchically-organized storage structures used in ECOS using the time and space complexity analysis . As we explained in Section 3, we customize a column as either ordered read-optimized storage structure or unordered write-optimized storage structure. In both categories, many different combinations of storage structures are possible, however, we confine our discussion to the storage structures that we implemented in our prototype implementation. We use three parameters that are common for both classes of storage structures, which are as follows:

n = Number of key/value pairs in a storage structure
$T(n)$ = Worst-case running time for operations
$S(n)$ = Worst-case space complexity for storage structure
E_i = Evolution overhead where i = evolution identifier, such that E_i occurs before E_{i+1} and $E_i < E_{i+1}$

4.1 Ordered Read-Optimized Storage Structure

For ordered read-optimized storage structure, we evaluate a storage structure that evolves from a sorted array to a sorted list (of sorted arrays) and then to HLC SL (a B+-Tree based storage structure with sorted lists as data leaf nodes).

Initial storage structure (Sorted array) For sorted array we only have one important parameter to consider, which is as follows:

n_{sa} = Maximum number of key/value pairs that can be stored as a sorted array

The time complexity for different data management operations for a sorted array is as follows:
Get $->$ $\Theta(lg\ n_{sa})$ //Binary search
Put $->$ $\Theta(n_{sa})$
Delete $->$ $\Theta(n_{sa})$

The space complexity for a sorted array is as follows:
$S(n) = O(n_{sa})$

As long as $n \leq n_{sa}$: data storage structure = sorted array. When $n > n_{sa}$ evolution occurs, such that the existing sorted array becomes the part of a new data storage structure, e.g., a sorted list.

First evolution (Sorted array to sorted list) For sorted list we have three important parameters to consider, which are as follows:

n_{sl} = Maximum number of key/value pairs that can be stored as a sorted list
l_{sa} = Number of list blocks (sorted array) in a sorted list
n_p = Number of next and previous pointers in a sorted list

The time complexity for different data management operations for a sorted list is as follows:

$$\begin{aligned}
\text{Get} &-> \Theta(lg\ l_{sa}) + \Theta(lg\ n_{sa}) \\
\text{Put} &-> \Theta(lg\ l_{sa}) + \Theta(n_{sa}) \\
\text{Delete} &-> \Theta(lg\ l_{sa}) + \Theta(n_{sa})
\end{aligned}$$

The space complexity for a sorted list is as follows:

$$\begin{aligned}
l_{sa} &= \tfrac{n_{sl}}{n_{sa}} \text{ //Number of sorted arrays in list} \\
n_p &= l_{sa} * 2 \text{ //Number of next and previous pointers in the sorted list} \\
=> \quad n_p &= \tfrac{n_{sl}}{n_{sa}} * 2 \text{ //Number of next and previous pointers in the sorted list} \\
\therefore \quad S(n) &= O(n_{sl}) + O(\tfrac{n_{sl}}{n_{sa}} * 2)
\end{aligned}$$

As long as $n \leq n_{sl}$: data storage structure = sorted list. When $n > n_{sl}$ evolution occurs, such that the existing sorted list becomes the part of a new storage structure, e.g., B+-Tree, we term this storage structure as HLC SL.

Second evolution (Sorted list to HLC SL) HLC SL is a B+-Tree based storage structure with sorted list as leaf nodes for storing data. For HLC SL we have five important parameters to consider, which are as follows:

n_{bt} = Maximum number of key/value pairs that can be stored in sorted list using HLC SL
l_{sl} = Number of sorted lists as data leaf nodes
t = Minimum degree of HLC SL B+-Tree, such that $t \geq 2$
k = Maximum number of elements in each node, such at each index node can have k-1 keys and k children where k=2t.
h = Height of the HLC SL B+-Tree

The time complexity for different data management operations for a HLC SL with sorted list (of sorted arrays) as its data leaf node is as follows:

Get $->$ $O(\text{t } \log_t l_{sl}) + \Theta(lg\ l_{sa}) + \Theta(lg\ n_{sa})$
Put $->$ $O(\text{t } \log_t l_{sl}) + \Theta(lg\ l_{sa}) + \Theta(n_{sa})$
Delete $->$ $O(\text{t } \log_t l_{sl}) + \Theta(lg\ l_{sa}) + \Theta(n_{sa})$

The space complexity for a HLC SL with sorted list (of sorted arrays) as its data leaf node is as follows:

$l_{sl} = \frac{n_{bt}}{n_{sl}}$ //Number of sorted list as data leaf node
$=> S(n_{bt}) = O(l_{sl})$ //We store one key for each sorted list
$\therefore S(n) = O(l_{sl}) + O(n_{sl}) + O(\frac{n_{sl}}{n_{sa}} * 2)$

As long as $n \leq n_{bt}$: data storage structure = HLC SL. When $n > n_{bt}$ evolution may again occur, however, we confine our discussion to this level. Overall ECOS behavior for our example of ordered read-optimized data storage structure with two levels of evolution can be summarized as follows:
Get:

$$T(n) = \begin{cases} \Theta(lg\ n_{sa}) & \text{if } n \leq n_{sa} \\ \Theta(lg\ l_{sa}) + \Theta(lg\ n_{sa}) & \text{if } n \leq n_{sl} \\ O(tlog_t\ l_{sl}) + \Theta(lg\ l_{sa}) + \Theta(lg\ n_{sa}) & \text{if } n \leq n_{bt} \end{cases}$$

Put:

$$T(n) = \begin{cases} \Theta(n_{sa}) & \text{if } n \leq n_{sa} \\ \Theta(lg\ l_{sa}) + \Theta(n_{sa}) & \text{if } n \leq n_{sl} \\ O(tlog_t\ l_{sl}) + \Theta(lg\ l_{sa}) + \Theta(n_{sa}) & \text{if } n \leq n_{bt} \end{cases}$$

Delete:

$$T(n) = \begin{cases} \Theta(n_{sa}) & \text{if } n \leq n_{sa} \\ \Theta(lg\ l_{sa}) + \Theta(n_{sa}) & \text{if } n \leq n_{sl} \\ O(tlog_t\ l_{sl}) + \Theta(lg\ l_{sa}) + \Theta(n_{sa}) & \text{if } n \leq n_{bt} \end{cases}$$

Space complexity

$$S(n) = \begin{cases} O(n_{sa}) & \text{if } n \leq n_{sa} \\ O(n_{sl}) + O(\frac{n_{sl}}{n_{sa}} * 2) & \text{if } n \leq n_{sl} \\ O(l_{sl}) + O(n_{sl}) + O(\frac{n_{sl}}{n_{sa}} * 2) & \text{if } n \leq n_{bt} \end{cases}$$

4.2 Unordered Write-Optimized Storage Structure

As second example, we discuss write optimized hierarchically-organized storage structures used in ECOS. For unordered write-optimized storage structure, we evaluate a heap array that evolves into heap list and then we generate B+-Tree based index structure on heap list, which further evolves as an ordered read-optimized storage structure.

Initial storage structure (Heap array) For heap array we only have one important parameter to consider similar to sorted array, which is as follows:

n_{ha} = Maximum number of key/value pairs that can be stored as a heap array

The time complexity for different data management operations for a heap array is as follows:

Get $->$ $\Theta(n_{ha})$ //Linear search
Put $->$ $\Theta(1)$
Delete $->$ $\Theta(1)$ //Mark delete
Defragmentation $->$ $\Theta(n_{ha})$ //Linear

The space complexity for a heap array (with defragmentation) is as follows:

$S(n) = O(n_{ha})$

As long as $n \leq n_{ha}$: data storage structure = heap array. When $n > n_{ha}$ evolution occurs, such that the existing heap array becomes part of a new data storage structure, e.g., heap list.

First evolution (Heap array to heap list) For heap list we have three important parameters to consider, which are as follows:

n_{hl} = Maximum number of key/value pairs that can be stored as a heap list
l_{hl} = Maximum number of list blocks(heap array) in heap list
n_p = Number of next and previous pointers in the heap list

The time complexity for different data management operations for a heap list of heap arrays is as follows:

Get $->$ $\Theta(n_{hl})$ //Linear Search
Put $->$ $\Theta(1)$

Delete $->$ $\Theta(1)$ //Mark delete
Defragmentation $->$ $\Theta(n_{hl})$ //Linear

The space complexity for a heap list of heap arrays (with defragmentation) is as follows:

$l_{ha} = \frac{n_{hl}}{n_{ha}}$ //Number of heap arrays in the heap list
$n_p = l_{ha} * 2$ //Number of next and previous pointers in the heap list
$=> n_p = \frac{n_{hl}}{n_{ha}} * 2$ //Number of next and previous pointers in the heap list
$\therefore S(n) = O(n_{hl}) + O(\frac{n_{hl}}{n_{ha}} * 2)$

It can be observed that we do not get any benefit in terms of performance, when we evolve a heap array to a heap list. However, we should also consider here the possibility of evolving to different storage structure, e.g., hash table. Each evolution is the point to observe the statistics that we gather as long as previous storage structure is usable. These statistics gives us insight for the workload on the column. For example, in case of a heap array evolving to hash table, we have following time complexity for new hash table storage structure:

Get $->$ $\Theta(n_{ha})$ //Ignoring the hash calculation and bucket selection overhead
Put $->$ $\Theta(1)$
Delete $->$ $\Theta(1)$ //Mark delete
Defragmentation $->$ $\Theta(n_{hl})$ //Linear

However, for our discussion, here we do not evolve heap list to hash table. As long as $n \leq n_{sl}$: data storage structure = heap list. When $n > n_{sl}$ evolution occurs, however, in unordered write-optimized storage scenario, we do not evolve a heap list to any other storage structure. Instead, we use the heap list as the primary storage structure for data and we generate indexes on it based on the statistics we generated while populating this heap list. Since an index is an ordered data storage structure, we use the evolving storage structure for storing index as we have discuss above in Section 4.1. In this scenario, we assume that according to the gather statistics, we identify B+-Tree as an appropriate index. Here we mean a standard B+-Tree, i.e., leaf node stores the pointer/identifier to data in the heap list.

Second evolution (Heap list with a B+-Tree as an index) For heap list with a B+-Tree as an index, we have five important parameters to consider, which are as follows:

n_{ibt} = Maximum number of keys that can be stored in the B+-Tree
l_{hl} = Number of heap lists for data storage
t = Minimum degree of the B+-Tree, such that $t \geq 2$
k = Maximum number of elements in each node, such at each index node can have k-1 keys and k children where k=2t.
h = Height of the tree

The time for different data management operations for a heap list of heap arrays with B+-Tree as an index is as follows:

Get $->$ $O(\text{t } \log_t \text{n}_{ibt}) + \Theta(1)$
Put $->$ $O(\text{t } \log_t \text{n}_{ibt}) + \Theta(1)$
Delete $->$ $O(\text{t } \log_t \text{n}_{ibt}) + \Theta(1)$ //Mark delete
Defragmentation $->$ $O(\text{t } \log_t \text{n}_{ibt}) + \Theta(\text{n}_{hl})$

The space complexity for a heap list (with defragmentation) of heap arrays with B+-Tree as an index is as follows:

$S(n) = O(n_{ibt}) + O(n_{hl}) + O(\frac{n_{hl}}{n_{ha}} * 2)$
since $n_{ibt} = n_{hl}$ //Number of keys in B+-Tree is same as number of records in a heap list
$\therefore \; S(n) = O(2 * n_{ibt}) + O(\frac{n_{hl}}{n_{ha}} * 2)$

As long as $n \leq n_{ibt}$: data storage structure = heap list with the B+-Tree as an index. When $n > n_{ibt}$ evolution may again occur for index storage structure, however, we confine our discussion to this level. Overall ECOS behavior for our example of unordered write-optimized data storage structure with two level of evolution is as follows:
Get:

$$T(n) = \begin{cases} \Theta(n_{ha}) & \text{if } n \leq n_{hl} \\ O(tlog_t n_{ibt}) + \Theta(1) & \text{if } n \leq n_{ibt} \end{cases}$$

Put:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq n_{hl} \\ O(tlog_t n_{ibt}) + \Theta(1) & \text{if } n \leq n_{ibt} \end{cases}$$

Delete:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq n_{hl} \\ O(tlog_t n_{ibt}) + \Theta(1) & \text{if } n \leq n_{ibt} \end{cases}$$

Defragmentation:

$$T(n) = \begin{cases} \Theta(n_{hl}) & \text{if } n \leq n_{hl} \\ O(tlog_t n_{ibt}) + \Theta(n_{hl}) & \text{if } n \leq n_{ibt} \end{cases}$$

Space complexity

$$S(n) = \begin{cases} O(n_{ha}) & \text{if } n \leq n_{ha} \\ O(n_{hl}) + O(\frac{n_{hl}}{n_{ha}} * 2) & \text{if } n \leq n_{hl} \\ O(2 * n_{ibt}) + O(\frac{n_{hl}}{n_{ha}} * 2) & \text{if } n \leq n_{ibt} \end{cases}$$

It can be observed from above-provided time and space complexity analysis of evolving storage structures that for different database size, we obtain different resource consumption (i.e., we take both CPU time and storage space as resources). To simplify our discussion, we take an example of ordered read-optimized storage. It can be observed that space requirement of complex storage structure, e.g., B+-Tree is high in comparison with the sorted array. Whereas insertion and deletion CPU time for the sorted array is high. However, as we have mentioned and discussed earlier, we restrict the data storage capacity of storage structure. This ensures that we keep the insertion and deletion time for each storage structure within the acceptable limit.

5 Implementation of Evolution Mechanism

In this section, we explain, how we implemented the evolution mechanism in ECOS. Our aim for evolution mechanism implementation was to keep the overheads to be negligible, whereas at the same time ensure that the implementation of evolution should not get tightly coupled with standard storage manager implementation. For this purpose, we used innovative software engineering techniques to ensure that evolution behavior can be added or removed from the storage manager without affecting the other storage manager functionalities. This section contains few details regarding the software engineering techniques that we used for implementing ECOS to give a reader better understanding of how ECOS internals work as a software.

5.1 Monitoring Functionality Implementation

The most important functionality of the evolution mechanism is the monitoring functionality. ECOS monitors existing storage structures to gather the heredity information and to observe the data management operation events (see Section 3.3 for details of heredity information and events). Monitoring functionality in ECOS is implemented using AspectC++[1], which is a set of C++ language extensions to facilitate aspect-oriented programming with C++. Details regarding why and how we used different programming techniques, such as aspect-oriented programming [19] and feature-oriented programming [25] for our prototype implementation can be found in our previously published work in [29].

Below is the code snippet 1 of our monitoring functionality implemented as an aspect (a modular way to separate the common code that otherwise is part of different software components) using AspectC++ language constructs:

Listing 1: Monitoring implementation code snippet

```
1  aspect Monitor {
   Autonom a;
3  MSG _msgid;

5  /*Monitoring for tracing*/
   advice execution("MSG Composite::ICPutData(...)") : before() {
7      Composite *c;
       _msgid = (MSG) * tjp->result();
9      c = tjp->that();
       a.TraceICPutData((RECORD*) tjp->arg(0), (COLUMN*) tjp->arg(1));
11 }
```

[1] "AspectC++", `http://www.aspectc.org/`

```
13 /*Monitoring for possible events, analysis, and fixings*/
   advice execution("MSG Page::PutData(...)") : after() {
15     Page *pg;
       if (tjp->result() != NULL) {
17         _msgid = (MSG) * tjp->result();
           switch (_msgid) {
19             case SUCCESS:
                   //Result is SUCCESS
21                 break;
               case NO_SPACE:
23                 //Result is NO_SPACE
                   pg = tjp->that();
25                 _msgid = a.AnaFixCheckStorageNoSpace(pg);
                   *tjp->result() = _msgid;
27                 if (_msgid == SUCCESS) {
                       a.TraceReset();
29                 }
                   break;
31             case NOT_FOUND:
                   ...
33             default:
                   //Unexpected result
35                 break;
       } } }  ... }
```

In the above mentioned code snippet, the code between line numbers 6 to 11 is an advice code. An advice is used to specify the additional code that could be executed before, after, or at both points (i.e., around) during the flow of program. For example, on the line number 6 in the code snippet 1, the *before* keyword ensures that the advice is executed before the execution of *ICPutData* function.

5.2 Trace Functionality Implementation

Another important functionality of evolution implementation is the trace functionality, which executes before the execution of data management operation and stores the heredity information, such as column information and record details. The advice defined at the line number 6 in the code snippet 1 is a sample trace code. For example, in the above-mentioned code, we gather the statistics for all *ICPutData* function executions, which for presented sample scenario includes taking reference to the involved column and record objects. We use this information to call *ICPutData* again, if it fails to execute successfully.

5.3 Analysis and Fixing Functionality Implementation

The advices between line numbers 14 to 36 in the code snippet 1 define the code that analyzes the execution of different data management functions. It also executes the fixing code if some problem is identified. For example, advice at the line number 14 in the code snippet 1 checks the execution result of the *PutData* function of the *Page* implementation class. If some problem is identified, such as *NO_SPACE* at the line number 22 in the code snippet 1, it executes the code that analyzes the problem based on the recorded trace data and fixes the problem. All functions used in advices in the code snippet 1 are defined in the *Autonom* class. The code snippet 2 from the implementation of *Autonom* class is presented below:

Listing 2: Autonom class implementation code snippet

```
class Autonom {
public:
    //Evolve class implements the evolving functionality
    Evolve evo;
    //Trace function for PutData method
    void TraceICPutData(RECORD* _r, COLUMN* _c);
    //Analysis and fixing functions
    MSG AnaFixCheckStorageNoSpace(Page *p);
    ... };

/*Trace functions*/
void Autonom::TraceICPutData(RECORD* _r, COLUMN* _c) {
    this->evo.r = _r;
    this->evo.c = _c;
    this->evo.isbyval = false; }

/*Analysis and fixation functions*/
MSG Autonom::AnaFixCheckStorageNoSpace(Page *p) {
    //This is for evolving from Sorted Array to Sorted List
    this->evo._msgid = NO_SPACE;
    //Evolution AnaFix function implements the analysis and fixing
    return this->evo.AnaFix(); }
```

It can be observed from the code snippet in 2 that the *Autonom* class contains the implementation of trace functions and uses the *Evolve* class to execute the analysis and fixing. However, we use the above code (i.e., the code snippet 1 and 2) for two fold purpose. As first purpose during analysis we identify, when to evolve the existing storage structure. For example, as shown at the line number 22 in the code snippet 1, each case triggers an event for possible evolution of existing storage structure. Furthermore, as second purpose we also identify, either the

existing storage structure should be evolved into new storage structure or not. The *AnaFix* function at the line number 22 in the code snippet 2 contains the functionality to take this decision. Below we present the code snippet 3 for our evolution code:

Listing 3: Evolution implementation code snippet

```
MSG Evolve::EvolveColumnIM() {
2       //We evolve Sorted array to Sorted list
        //or Heap Array to Heap List
4       //This column should be traced during tracing
        if (this->c == NULL) { return UNRESOLVED; }
6
        //First we change column type to sorted list
8       this->c->columntype = SL;
        //Instantiate new Sorted List
10      this->c->sl = new StorageManager();
        _msgid = c->sl->CreateDatabase(this->c->im->database);
12      if (_msgid != SUCCESS) return _msgid;

14      //Now evolve from Array to List
        //Such that existing Array will become an integral unit of List
16      _msgid = this->c->sl->Evolver(this->c->im);
        if (_msgid != SUCCESS) return _msgid;
18
        //Now after evolution, redo last buffered operation
20      return this->EvolveColumnIMPutData();}

22  MSG StorageManager::Evolver(Page* _page) {
        MSG _msgid = pb->EvolvePB(_page);
24      if (_msgid != SUCCESS) return _msgid;
        this->tuplecount += _page->CountTuples();
26      dd->SetStartPage(_page->GetID());
        dd->SetEndPage(_page->GetID());
28      return SUCCESS;}

30  MSG PageBuffer::EvolvePB(Page* page) {
        tmpPID++;
32      pg[tmpPID - 1] = page;
        pg[tmpPID - 1]->SetID(tmpPID);
34      ++usedPageCount;
        fOnUsedPageCountChanged(evenSink, usedPageCount);
36      return SUCCESS;}
```

In above code snippet, the function *EvolveColumnIM()* at the line number 1 evolves a sorted array storage structure into sorted list storage structure. The $c->im$ object refers to a sorted array storage structure and the $c->sl$ refers to a newly instantiated sorted list storage structure. Each storage structure implements an *Evolver* function, such as the one used at the line number 16 in the code snippet 3.

The *Evolver* function contains the implementation that makes the existing storage structure, which is provided as an argument, an integral component of the newly instantiated storage structure. For example, the *Evolver* function at line number 16 in the code snippet 3 takes a sorted array as an argument and makes it an integral part of the sorted list. For better understanding, the implementation of the *Evolver* function of the sorted list is also provided at line number 22 in the code snippet 3. The *Evolver* function at the line number 22 in the code snippet 3 instantiates a new sorted array and distributes the data of existing sorted array among them equally, than it makes both sorted array a part of the new sorted list. It is a naive implementation that we used to demonstrate the concept, however, the *Evolver* function is an important code fragment. The implementation of an *Evolver* function identifies the associated overhead for an evolution.

Listing 4: ECOS interface code snippet

```
RECORD* crecords =
(RECORD *) malloc(sizeof (RECORD) * <No. of columns>); ...
crecords[<index>].key = <key>;
crecords[<index>].columnindex = <column index>;
crecords[<index>].size = <No. of bytes for value>;
crecords[<index>].value =
(cbyte*) malloc(sizeof(cbyte) * <No. of bytes for value>); ...
_msgid = cell.GetDataNext(crecords); //Scan ...
_msgid = cell.GetData(crecords);//Get record ...
_msgid = cell.PutData(crecords);//Put record ...
_msgid = cell.DeleteData(crecords);//Delete record ...
```

The interface provided to the end-user or external application by ECOS is simple and consistent. Which storage structure is in use by the column?, when it is evolved?, all these aspects are hidden. A sample code snippet to give an insight for ECOS interface is provided as the code snippet 4 above.

6 Empirical Evaluation

In this section, we provide the details of our micro benchmark that we used to generate the evaluation results, the performance comparison of evolving storage structures with fixed storage structures, and the performance comparison of different DSM based schemes (for both fixed storage structures and evolving hierarchically-organized storage structure versions)[2].

The data and index storage structures that we have implemented in the existing ECOS prototype implementation are sorted array, sorted list (of sorted arrays), B+-Tree (with sorted arrays as data leaf nodes), HLC SL (B+-Tree based structure with sorted lists as data leaf nodes), HLC B+-Tree (B+-Tree based structure with B+-Trees as data leaf nodes), heap array, and heap list. To simplify our discussion, we present the results involving sorted array, sorted list, B+-Tree, HLC SL, and HLC B+-Tree.

6.1 Micro Benchmark Details

For ECOS evaluation, we set up a micro benchmark with repeated insertion, selection, and deletion of data using API based access method. 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, number of records ((cardinality)) is kept different to assess the impact of change in data size using ECOS. 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. The reason for selecting these parameters is the change in bottlenecks. In the last two decades, the processor speed has been increasing at the much faster rate of around 60% per annum in comparison with the memory speed that increases only around 10% per year [23]. Therefore, it is essential for DBMS, to make optimal use of increased processing power and large main memories while avoiding the overheads associated with memory latencies. 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 iteration of the micro benchmark. We used Valgrind tools suite [31] to measure the heap usage.

We used a micro benchmark to generate the empirical results. We understand

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

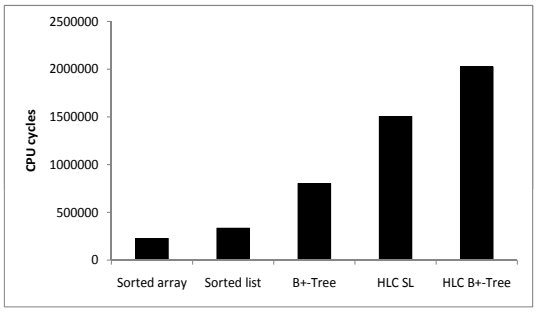

Figure 3: Performance comparison of different storage structures for a single record.

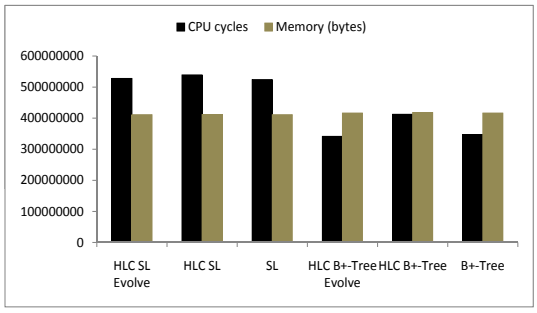

Figure 4: Performance comparison of different storage structures for 4048 records.

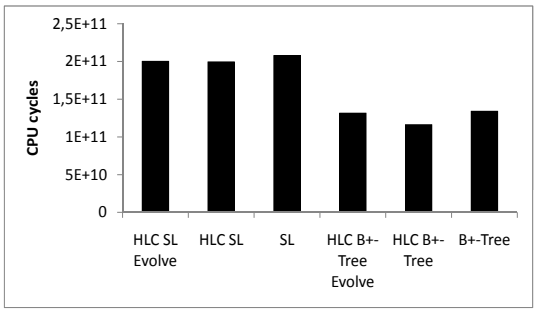

Figure 5: Performance comparison of different storage structures for 100K records.

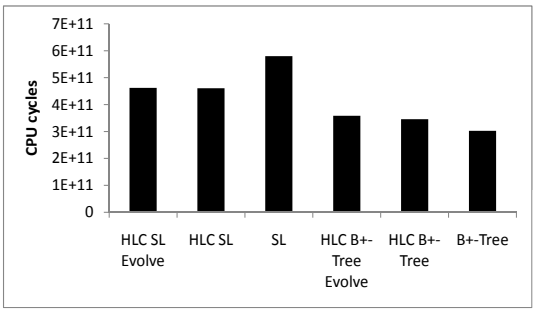

Figure 6: Performance comparison of different storage structures for 500K records.

the need for empirical results using standard benchmarks, such as TPC, however, existing implementation of ECOS is a prototype implementation and can only be tested using a micro benchmark. Furthermore, ECOS is a research prototype with many implementation details still in progress. We are using our best effort to provide reliable and repeatable results that can compare ECOS with the performance of other existing commercial products; however, it is left as part of the future work.

6.2 ECOS Performance Improvement

To demonstrate the performance gain using the ECOS, we first present our observation of the effect of an increase in data size on performance of different storage structures. We executed our benchmark for different storage structures for the different number of records (i.e., single record, 4048 records, 100K records, and 500K records). It can be observed in Figure 3, for a single record sorted array consumes less CPU cycles in comparison with other storage structures. For 4048 records, array consumes much more CPU cycles in comparison with other storage structures therefore we omitted it in Figures 4, 5, and 6. In Figure 4, it can be observed

that for 4048 records, sorted list and B+-Tree based storage structures consume a similar number of CPU cycles and amount of memory. However, Figure 5 and 6 shows that B+-Tree based storage structures perform better for 100K and 500K records. According to the above observation, 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 as long as possible using the definitions from evolution paths, such as sorted array for small data management.

To further clarify the evolving storage structures evolution, we present the evaluation results for evolving HLC SL and evolving HLC B+-Tree storage structure in Figure 7 and 8. In both figures, evolving storage structure evolves with the data growth. It can be seen that both HLC SL and HLC B+-Tree storage structures consume more CPU cycles in comparison with sorted list and B+-Tree. This behavior is due to the complexity of these storage structures, which are meant to be used for extremely large data sizes. These two structures (i.e., HLC SL and HLC B+-Tree) automatically partition the data and uses separate buffer and index managers for each partition, which is not the requirement for presented 500K records storage. However, for the purpose of demonstration of evolution concept we forced storage structures to evolve to HLC SL and HLC B+-Tree level for 500K records. To demonstrate the difference of performance for different DSM based

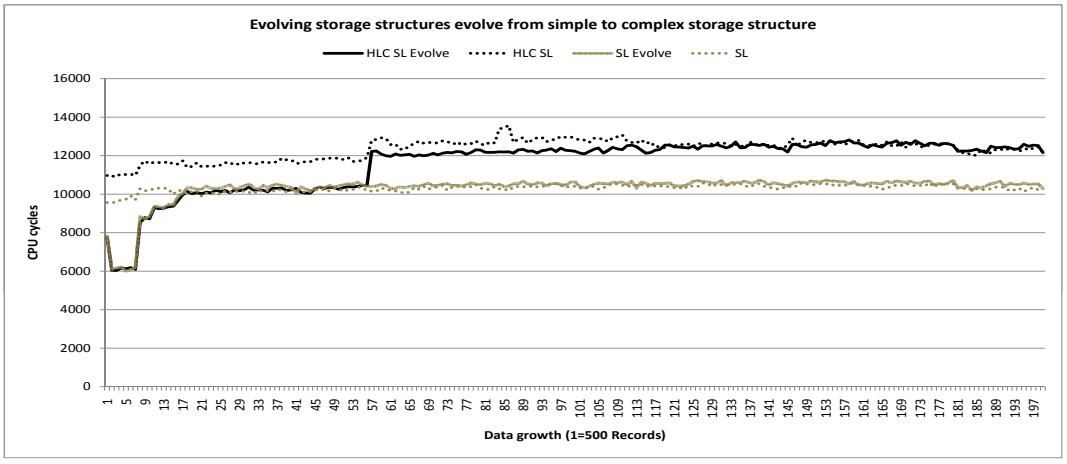

Figure 7: Evolving HLC SL storage structure evolution.

schemes and the performance gains using the evolving storage structures, we executed our micro benchmark in two configurations for all five schemes explained in Section 3. In the first configuration, we instantiated all columns as fixed HLC SL storage structure. In the second configuration, we used evolving HLC SL storage structure, which instantiate all columns as a sorted array on start up and then

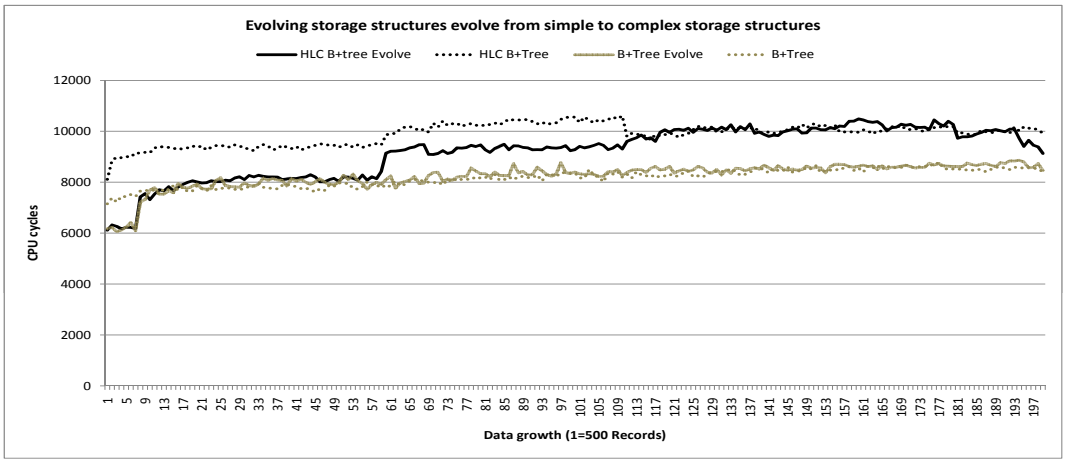

Figure 8: Evolving HLC B+-Tree storage structure evolution.

evolve the column with data growth to a sorted list, and finally to HLC SL (using the evolution path presented in Table 8). As different columns contain the different size of data for two dictionary based schemes, i.e., DMDSM and VDMDSM, in second configuration data of few dictionary columns can be accommodated in a sorted array, few evolve to a sorted list, and rest of the dictionary columns with large data evolves to HLC SL (sample scenario shown in Table 1). In Figure 9 and 10, results for evolving storage structures have evolve keyword appended in front of DSM based scheme name.

In four proposed variations of DSM based schemes in Section 3, we altered the 2-copy DSM by reducing the duplicate copies for each column, which should affect the time for read operations with search criteria on non-key attributes. To assess the impact of proposed change on performance, we evaluated all five schemes in two configurations, i.e., the first configuration with search criteria involving key attribute as shown in Figure 9, and the second configuration with search criteria involving non-key attribute as shown in Figure 10.

The results presented in Figure 9 and 10 shows that DSM and PDSM 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 can also be 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 [8]. 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

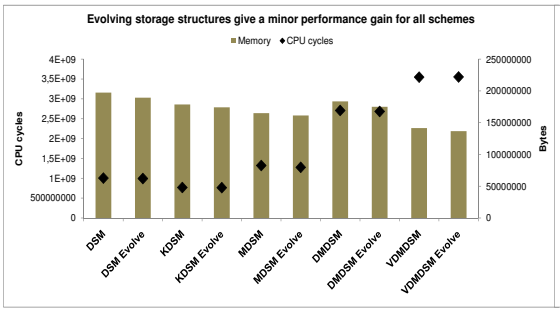

Figure 9: Performance comparison of different DSM based schemes in ECOS with primary key based search criteria.

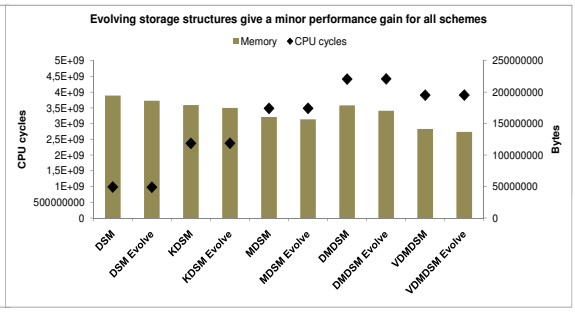

Figure 10: Performance comparison of different DSM based schemes in ECOS with non-key based search criteria.

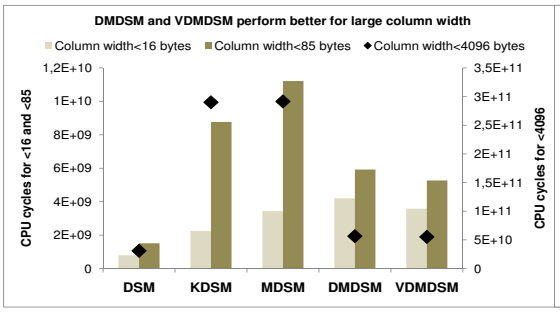

Figure 11: Performance improvement for dictionary based DSM schemes for large column width.

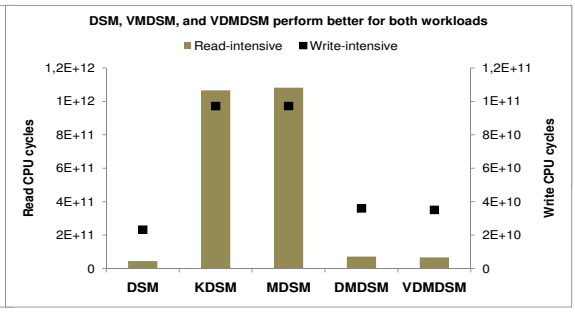

Figure 12: Performance comparison of different DSM based schemes in ECOS for read and write intensive workloads.

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.

The results of Figure 9 and 10 are observed for values with width of 16. We increased the width of value for all columns to 85 and then to 4096 to assess the impact of change in tuple width on performance of different schemes. It can be observed in Figure 11 that dictionary based schemes performance is improved and becomes comparable with standard 2-copy DSM scheme for large tuple width. However, PDSM 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 12 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 DSM if their short comings are overcome.

7 Related Work

Hierarchically-organized storage structures have already been in use in the data warehousing domain. Morzy et al. in [21] proposed a hierarchical bitmap index for indexing set-valued attributes. Later, Chmiel et al. in [11] extended that concept to present hierarchically-organized bitmap indexes for indexing dimensional data. ECOS uses hierarchical organization differently. It allows the use of different data and index storage structures at the different level of hierarchy and increases the hierarchy with data growth autonomically.

Database cracking is an innovative approach proposed by Kersten and Manegold [18]. It proposes the continuous physical reorganization of a database based on the query processing. It cracks the database into manageable pieces based on the user queries to decrease the access time and implementing self-organizing behavior. Our approach is different from the database cracking. ECOS in comparison implements self-organization at the storage manager level. ECOS evolves storage structures with data growth to ensure consistent performance while maintaining minimal resource consumption.

Bender et al. proposed the cache-oblivious B-Trees [6] that performs the optimal search across different hierarchical memories with varying memory levels, cache size, and cache line size. Fractal prefetching B+-Trees [10] proposed by Chen et al. is 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) [16] 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 [20] 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 running workload, e.g., indexes, materialized views, partitioning, clustering, and views [4]. Existing automated physical design tools assume the workload as a set of SQL statements [4]. These tools use query optimizer to identify the appropriate physical design selection from various proposed candidate designs [22]. The cost of using a query optimizer is huge. Papadomanolakis et al. in [22] mentioned that for index selection algorithms on average 90% of running time is spent in the query optimizer. 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 design is motivated from the idea of exploring new architectures for developing self-tuning DBMS instead of developing techniques to self-tune the existing ones.

8 Conclusion

In this report, we presented ECOS, a customizable online self-tuning storage manager and the concept of the evolution path. ECOS and evolution path enables and uses the customization of storage structures at the fine-grained table and column-level. In addition, ECOS and evolution path allows storage structures to autonomically evolve (to more suitable storage structures) with the change in the data management needs. It allows ECOS to maintain the desirable performance while keeping the human intervention at a minimum. We also presented detailed discussion and evaluation of the ECOS and evaluation path showing the performance improvement and reduced resource consumption.

As a future work, we plan to enhance the presented dictionary based DSM schemes for better performance. For ECOS, what heredity information should be gathered for each storage structure, and how to improve the efficiency of storage and retrieval of heredity information is also an important topic for further work. 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. How the presented concepts of evolving hierarchically-organized storage structures in conjunction with the concept of the evolution path can be used for a self-tuning storage manager using the row-oriented storage model is an important open research question. Once query processing is implemented, we want to integrate presented evolution mechanism with query processing, and we will be able to evaluate the ECOS using the TPC-H benchmark. Transaction management is also an implementation specific future work for our ECOS prototype.

References

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[2] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *VLDB*, pages 967–980, 2008.

[3] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.

[4] S. Agrawal, E. Chu, and V. Narasayya. Automatic physical design tuning: workload as a sequence. In *SIGMOD*, pages 683–694, 2006.

[5] D. S. Batory. On searching transposed files. *ACM Trans. Database Syst.*, 4(4):531–544, 1979.

[6] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *FOCS*, pages 399–409, 2000.

[7] S. Chaudhuri and V. Narasayya. Self-tuning database systems: a decade of progress. In *VLDB*, pages 3–14, 2007.

[8] S. Chaudhuri and G. Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *VLDB*, pages 1–10, 2000.

[9] S. Chaudhuri and G. Weikum. Foundations of automated database tuning. In *SIGMOD*, pages 964–965, 2005.

[10] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal prefetching B+-Trees: optimizing both cache and disk performance. In *SIGMOD*, pages 157–168, 2002.

[11] J. Chmiel, T. Morzy, and R. Wrembel. HOBI: Hierarchically Organized Bitmap Index for Indexing Dimensional Data. In *DaWaK*, pages 87–98, 2009.

[12] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. *SIGMOD Rec.*, 14:268–279, 1985.

[13] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic SQL tuning in oracle 10g. In *VLDB*, pages 1098–1109, 2004.

[14] A. P. de Vries, N. Mamoulis, N. Nes, and M. L. Kersten. Efficient image retrieval by exploiting vertical fragmentation. Technical Report INS-R0109, CWI, 2001.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41:205–220, 2007.

[16] J. L. Hellerstein. Automated tuning systems: Beyond decision support. In *CMG*, pages 263–270. Computer Measurement Group, 1997.

[17] A. L. Holloway and D. J. DeWitt. Read-optimized databases, in depth. *Proc. VLDB Endow.*, 1:502–513, 2008.

[18] M. L. Kersten and S. Manegold. Cracking the Database Store. In *CIDR*, pages 213–224, 2005.

[19] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP*, pages 220–242, 1997.

[20] T. Malik, X. Wang, R. Burns, D. Dash, and A. Ailamaki. Automated physical design in database caches. In *ICDE Workshop*, pages 27–34, 2008.

[21] M. Morzy, T. Morzy, A. Nanopoulos, and Y. Manolopoulos. Hierarchical Bitmap Index: An Efficient and Scalable Indexing Technique for Set-Valued Attributes. In *ADBIS*, pages 236–252, 2003.

[22] S. Papadomanolakis, D. Dash, and A. Ailamaki. Efficient use of the query optimizer for automated physical design. In *VLDB*, pages 1093–1104, 2007.

[23] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent RAM. *IEEE Micro*, 17:34–44, 1997.

[24] D. A. Patterson and D. R. Ditzel. The case for the reduced instruction set computer. *SIGARCH Comput. Archit. News*, 8:25–33, 1980.

[25] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP*, pages 419–443, 1997.

[26] 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*, pages 553–564, 2005.

[27] TPC-H. http://www.tpc.org/tpch/.

[28] S. S. ur Rahman. Using evolving storage structures for data storage. In *FIT*, 2010.

[29] S. S. ur Rahman, V. Köppen, and G. Saake. Cellular DBMS: An Attempt Towards Biologically-Inspired Data Management. *Journal of Digital Information Management*, 8:117–128, 2010.

[30] P. Valduriez, S. Khoshafian, and G. P. Copeland. Implementation Techniques of Complex Objects. In *VLDB*, pages 101–110, 1986.

[31] Valgrind. http://www.valgrind.org.

[32] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *VLDB*, pages 20–31, 2002.

[33] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 design advisor: integrated automatic physical database design. In *VLDB*, pages 1087–1097, 2004.