The Mimicking Octopus: Towards a one-size-fits-all Database Architecture

Alekh Jindal

Supervised by: Prof. Dr. Jens Dittrich

Information Systems Group, Saarland University
http://infosys.cs.uni-saarland.de

ABSTRACT

Modern enterprises need to pick the *right* DBMSs e.g. OLTP, OLAP, streaming systems, scan-oriented systems among others, each tailored to a specific use-case application, for their data managing problems. This makes using specialized solutions for each application costly due to licensing fees, integration overhead and DBA costs. Additionally, it is tedious to integrate these specialized solutions together. Alternatively, enterprises use a single specialized DBMS for all applications and thereby compromise heavily on performance. Further, a particular DBMS (e.g. row store) cannot adapt and change into a different DBMS (e.g. streaming system), as the workload changes, even though much of the code and technology is replicated anyways.

In this paper we discuss building a new type of database system which fits several use-cases while reducing costs, boosting performance, and improving the ease-of-use at the same time. We present the research challenges in building such a system. We believe that by dropping the assumption of a fixed store, as in traditional systems like row store and column store, and instead having a flexible storage scheme we can realize much better performance without compromising the cost. We outline OctopusDB as our plan for such a system and discuss how it can mimic several existing as well as newer systems. To do so, we present the concept of *storage view* as an abstraction of all storage layouts in OctopusDB. We discuss how the heterogenous optimization problems in OctopusDB can be reduced to a single problem: *storage view selection*; and describe how a Holistic Storage View Optimizer can deal with it. We present simulation results to justify our core idea and experimental evidence on our initial prototype to demonstrate our approach. Finally, we detail the next steps in our work.

1. INTRODUCTION

1.1 Background

Database management systems started off as monolithic systems. However, database engineers soon started tuning their performance for specific applications. Consequently, currently we are witnessing a split of data management systems into several specialized solutions [20, 21] e.g. OLTP for transactional queries, OLAP for read-only complex reporting queries, DSMS (data stream management systems) for continuous window queries, and search engines for read-only keyword queries. It started in the mid-nineties [10] when the database engineers understood that the DBMSs of that time were ill-equipped to cope with the size of the datasets and complexity of OLAP-queries. Therefore a separate type of system was forked from the one size fits all DBMS code line [11]. That system is based on a column store and became one of the most popular and successful approaches for OLAP; products include SAP BI Accelerator, InfiniDB and Paraccel. At the same time other types of systems were forked including DSMS (data stream management systems) [23]; products include StreamBase. In addition, search engines developed into a separate community sometimes re-inventing DBMS technology. Yet these people are unwilling to use DBMS systems as a backend [22]. As a result, we have one specialized system per use-case application.

1.2 Motivation

A typical enterprise, today, employs a variety of data managing applications, and hence a variety of DBMSs. For instance, a banking enterprise uses an OLTP system for real time transactions, OLAP system for business intelligence and analytics and Streaming system for stock trading. Additionally, in many cases applications need to adapt to new requirements or evolve over time and usage; possibly requiring a switch to a different DBMS for optimal performance. For instance, in the banking enterprise the stock data prior to a time window may be pushed into a Archival system. These heterogenous systems, however, need to be integrated by copying data from one database to another using complex ETL-style data pipelines. Moreover, these *specialized* systems have their specialized vendors and DBAs, thereby incurring further licensing and maintenance costs. Obviously, all of this leads to extra costs in terms of development costs, maintenance costs, and DBA costs. So rather than making the world of data management easier, we have created a *zoo* of systems that sometimes has the opposite effect: it makes life of a company harder and more costly. We agree that for companies who invest a lot into connecting the different species in their zoo, it will eventually lead to a well-integrated and efficient overall system. Still, we believe that the zoo-keeping costs are non-trivial, especially for small to medium-sized business. We also believe that adapting such a zoo to new requirements, changing workload, or new types of applications may be prohibitive.

Moreover, one technology (e.g. row stores, column stores etc.) per system does not always deliver the best performance. For example, consider a university database with Student and Lecture ta-

Copyright is held by the author/owner(s).
VLDB 2010 PhD Workshop, September 13, 2010, Singapore

bles. Now for a query workload requesting students based on lecture types, credit points or instructor, we need to access only few attributes from the Lectures table but most (or all) attributes from Student table. For such a use-case scenario it makes sense to store the Student table in a row store and the Lecture table in a column store, thereby deriving the maximum performance. However, such a configuration is not possible to achieve in traditional *per-application* database systems, except in Fractured Mirrors [17]. Fractured Mirrors, however, has an exorbitant update cost. Moreover, as students graduate and hence do not attend lectures anymore, their details are not fetched by our query workload. Therefore, to improve performance it might make sense to partition the cold data (graduated students) in the Student table into a column store, utilizing compression, and the hot data (current students) into a row store. Again, such a configuration is impossible to achieve in a *single* traditional database system.

Further, we analyzed the TPC-H benchmark to illustrate how the use-case scenario and the workload could determine the right DBMS technology. For each query in the benchmark we marked the attributes referenced by it in each of the 8 relations. Our analysis revealed that each relation has a different attribute access pattern such that some attributes are referenced more than others while some attributes are not referenced at all e.g. `retailprice`, `comment` in `PART` relation. Further, several groups of attributes are co-referenced in the same queries. Thus, a single type of store (row, column) may not suit all relations. For instance, relations `LINEITEM`, `NATION` and `REGION` have several attributes referenced in the same queries and therefore a row store could be more suited. On the other hand, relations `PART`, `SUPPLIER`, `CUSTOMER` and `ORDER` have only few attributes referenced and hence column store would be the better choice. This analysis hints for a more holistic approach to database storage design.

2. PROBLEM STATEMENT

In this section we discuss the problem we focus on and describe our overall goal. As highlighted in the previous section: (1) A single specialized DBMS may not deliver the best performance for all applications, (2) Modern enterprises, anyway, end up having a zoo of database systems, (3) It is tedious to stitch together (complex ETL-style data pipelines) and costly to maintain (development, licensing, DBA costs) different database systems, and (4) Optimal performance over changing workloads, in a single or a zoo of DBMSs, remains an issue. Therefore, we need a *one-size-fits-all* database system which automatically adapts to different use-case scenarios and betters performance at the same time. We state our research goal as follows:

Research Goal. *A one-size-fits-all database system which caters well to all existing and newer data management use-cases and adapts automatically to initial configuration as well as to changing workload; all with improved performance, lowered cost and better maintainability at the same time.*

We discuss the research challenges in our problem in the following section, related work in Section 4, OctopusDB as our approach to the problem in Section 5, preliminary results in Section 6, and next steps in Section 7.

3. RESEARCH CHALLENGES

Below we discuss the major research challenges associated with our problem.

3.1 Different Storage Layouts under a single umbrella

Row stores are typically used in transactional processing systems (OLTP). Column stores on the other hand make heavy use of compression and are used in read-oriented workloads. Many people argue that these are completely different systems [2]. Further, streaming systems may not have any store at all. The *one-size-fits-all* system needs to cater the different storage layouts for different use-case scenarios. Additionally, the system need to adapt and interchange the storage layouts to changing workoads. The challenge, therefore, is to have a flexible storage scheme by bringing the different layouts into a single system. This brings to the fore three additional issues that we have to care about.

First, layout selection and maintenance is a major concern with having different layouts in a single system. This is because with changing workloads, the system needs to automatically decide the most appropriate layouts to create, maintain them with future updates and finally decide upon when, if at all, to discard them altogether. Second, query processing across different storage layouts is another important issue. Ideally, we would want the query processor to abstract much of the functionality across different layouts. At the same time, we would not want to completely miss out the storage layout specific optimizations e.g. compression in column stores.

Finally, apart from row and column layouts of the full table, the system can also create other sub-structures to boost performance. For instance, it can crack the table, not only horizontally as in [14] but also vertically, depending upon which part of it is accessed by the incoming queries; partition the tables horizontally or vertically in case of a definitive workload information; or create materialized views on any subset of the data. The challenge is to automatically create and manage this inventory of storage layouts. Further, all layouts discussed so far store data in rectangular fashion. It would be interesting to even consider non-rectangular storage layouts for the given data. The underlying idea in a *one-size-fits-all* system remains the same: relax the fixed layout assumption. The unlimited possibilities thereafter offer unique research challenges.

3.2 Automatic Adaptive Bifurcation instead of administered Eventual Integration

Currently, companies having several types of database systems spend a lot of time and effort to eventually integrate them together. As pointed before, these costs are non-trivial for small to medium sized companies. Further, the integrated system is less adaptive to a change in workload followed by a consequent addition or removal of a database system. Therefore, starting with a bag of database systems in the first place might end up with a loosely integrated system which is difficult to manage and expensive to maintain. Instead, since much of the code and technology of different DBMSs is anyways replicated, we can start with a single system for all applications.

The challenge, therefore, is to adaptively bifurcate the system into specialized technology (row, column etc.) depending upon the workload. The system needs to continuously monitor the workload and reassess its configuration. This makes sense because (1) we have a tightly integrated system (2) we abstract much of the code and technology and fork out only the necessary one, and (3) the system is fairly simple initially and is later adapted to be only as much complex as needed. We discuss the last point in more detail below.

3.3 Simplicity Vs Optimization

Simplicity is an important consideration while designing

database systems. In several cases too much of optimization in a database system is an overkill. For example, the materialized views created for each operator output in MonetDB [3] can be detrimental for changing workloads. Simplicity pays off in terms of performance and house-keeping. The challenge, therefore, is to strike the right balance between simplicity and optimization in the single system. Of course, simplicity might be traded for performance by incorporating more complex optimizations as the workload gets more sophisticated. But the key is to avoid any overkill.

Another aspect of simplicity is to support several use-cases in the minimal configuration, i.e. the system should first try to *mimic* as many specialized systems as possible before upgrading the configuration. This is necessary because changing configuration could be an overkill as well as expensive. Other people have also tried to develop systems which mimic more than one system. For example, [4] tries to mimic a column store in a row store. The challenge, however, is to determine the limiting point for the mimic.

Finally, a database system which is always initialized in the most rudimentary configuration might be quite slow till it adapts to the initial workload. Depending upon the adaptability speed, this slow start can be quite expensive. Additionally, we may incur the startup cost always, even when the system is reset or the data is ported to another instance. Instead, the system should be able to set its initial configuration, depending upon the initial workload, in order to derive the maximum performance straightaway. The challenge again lies in deciding how much of the simplicity should be lost at the very outset.

4. RELATED WORK

Traditional DBMS Landscape. Several papers have claimed that one size does not fit all. It started with [10] who noted that DBMSs do not work well for DSS-type workloads. This work lead to one of the first column-oriented data warehouses: SybaseIQ. Later on, other authors supported the idea of different types of database systems for different markets as well [20, 21]. This split the landscape into at least four different systems: SearchEngines (read-only inverted index), OLTP (transactional row-store), OLAP (read-only column-store) and DSMS (continuous window queries on unbounded streams) which originated from append-only databases [23]. However, the major difference of OLTP and OLAP are different access patterns and missing ACID semantics in OLAP. Furthermore, it is somewhat easier to use compression in column stores, see [1] for a comprehensive tutorial. However, a recent paper has argued that column stores may be efficiently emulated on row stores as well [4]. In any case, note that OctopusDB may make use of the existing optimizations for column stores and/or row stores. The advantage of OctopusDB is that we are not restricted to a particular store and workload. In addition, as we show in this paper, the boundary of OLTP/OLAP and DSMS may also be removed into a unified OctopusDB system. Furthermore, there have been several efficient approaches already to implementing search engines as an *application* on top of an OLTP database [12, 24]. Thus, given these and other recent advancements, it is questionable whether search engines will survive as a separate code base.

Lightweight Systems. Recently, a paper claimed that even in the traditional OLTP market existing DBMSs can be beaten by a large factor [22, 15]. This approach, HStore, is basically a stripped down version of an OLTP row-store. Again, this stripped down system could be emulated in OctopusDB as well. Thus, HStore is orthogonal to what we propose. Another line of systems has recently appeared exploring array-oriented systems for scientific applications [7]. Their major difference is an array data model with additional scientific operators. We believe that OctopusDB could also be extended to offer an Array Storage View. However, that discussion is beyond the scope of this paper.

Scanning Systems. Due to the dramatic changes in hardware (random access is hardly becoming better, sequential access is improving by up to 50% per year), index access pays of less and less. Therefore several authors have proposed to drop indexed-based query plans entirely and resort to scanning. [13] proposes techniques to reduce the scanning costs on modern hardware architectures. [18] proposed per-tuple constant-time processing on a row-store. [27] examines shared-scans, i.e. multiple query results are computed using a single scan. This idea is extended in [26] to so-called clock-scans, i.e. continuously running shared scans. Finally, [5] proposed to compute multiple star queries simultaneously using a static shared operator pipeline and a clock scan as its input. All these techniques show the viability of scan-based plans and we believe that all those techniques may be integrated into OctopusDB. Still we believe that a system should be able to offer index access for highly selective queries as well. OctopusDB offers this option.

Rodent store [8] allows DBAs to declare the database store using an algebra. We agree with the authors of [8] that currently considerable functionality is duplicated as row-stores and column-stores are two separate lines of development that share however considerable common technology. However, Rodent Store still assumes that there has to be a store. Furthermore, it simply provides an abstract way to declare a store in an OLTP or OLAP-style system. No unified approach with streaming systems is provided as it is the case for OctopusDB. In addition, no updateable storage view mechanism is present in Rodent Store. GMAP [25] presents a DDL for defining physical structures (similarly to [8]). In contrast to OctopusDB, GMAP does neither handle unification with streaming systems, automatic store selection and adaption, recovery, nor union queries.

Cracked databases, e.g. [14], break database tables into pieces by piggy-backing index-reorganization requests to individual queries, i.e. queries are interpreted as hints to break the database into pieces. Therefore cracked databases have similarities with partial indexing [19] and adaptive indexing [9], i.e. dynamic adjustment of index granularity to a given workload. Furthermore in contrast to OctopusDB, cracked databases assume a column-store (the authors mention that it could work on a row-store as well). In contrast, we do not assume a fixed store. In addition, in cracked databases the store may not be exchanged as it is the case in OctopusDB. Therefore cracked databases are orthogonal to what we propose.

Finally, self-tuning DBMSs [6] have looked at ways to automate index selection. We believe that some of these techniques may be extended to support OctopusDB. At the same time we believe that automatic storage view selection in OctopusDB is a hard problem justifying a separate research study.

5. OUR APPROACH: OCTOPUSDB

In this section we outline OctopusDB as our approach to a *one-size-fits-all* database system. We first describe the architecture of OctopusDB and subsequently discuss the core ideas in it.

5.1 Architecture

Figure 1 shows the main components of OctopusDB. The system contains components similar to the ones known from traditional DBMSs: transaction manager, query optimizer, recovery manager, purging and checkpointing and query and storage catalogs. The most striking difference, however, is the *primary log store* and the *storage view store*. The external user has a single programming

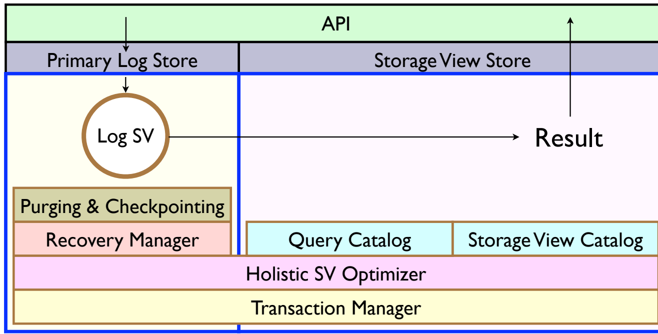

Figure 1: OctopusDB Architecture

interface (API) to interact with the overall system. Internally, OctopusDB passes all inputs from the user to the primary log store and retrieves results to user queries from storage view store.

5.2 Storage Views

OctopusDB departs from existing database systems by not having any fixed hard coded store e.g. row store, column store by default. Instead, OctopusDB records all database operations to a sequential log, called *primary log*, by creating appropriate logical log records. Each log entry contains a unique log sequence number, the database operation performed and the input parameters. OctopusDB stores the primary log persistently on durable storage (hard disk or SSD) following the write-ahead logging (WAL) protocol. It later creates arbitrary physical representations of the log, called *Storage Views* (SVs), depending on the workload. For instance, OctopusDB can create Row SV for transactional queries, Column SV for read-only queries. Additionally, it can create different (or same) SVs for different partitions of the data e.g. *hot* and *cold* data. Further, OctopusDB can create Index SVs or any other materialized views on any subset (or full) data. Finally, OctopusDB considers the primary log as just another SV, a Log SV, and can even decide to do away with it e.g. to emulate a streaming system. Thus, by having a completely flexible storage layer, in the form of SVs, OctopusDB addresses the research challenge of having *different storage layouts under a single umbrella* and has two advantages: (1) It can mimic OLTP, OLAP, Streaming Systems as well as several other types of database systems, (2) It dramatically improves the overall performance by morphing into any hybrid combination of these systems.

5.3 Storage View Selection

OctopusDB provides a system interface for users to insert, update and query data. Internally, it automatically figures out which SVs to create, maintain and iterate for the user queries. The user is not overwhelmed with the low levels decisions. The goal, therefore, is to come up with the most suitable SVs for the given database operations, while keeping the user agnostic to the internal data representation at the same time. Thus, the concept of Storage Views in OctopusDB reduces database tuning to a single optimization problem: *storage view selection*. More specifically, by automatically deciding upon the right storage views to create, OctopusDB addresses the research problem of having a *automatic "adaptive bifurcation" instead of administered "eventual integration"*.

5.4 Holistic Storage View Optimizer

Since OctopusDB has a single optimization problem, as described above, it has a single *Holistic Storage View Optimizer* to deal with it. In general, each class of SV may implement its own access algorithms optimized for the particular storage structure. For instance, a Row SV may use row-wise compression and row-oriented iteration, e.g. [13]. In contrast, a Col SV may implement column-oriented compression and vectorized iteration [3]. Outside those SVs OctopusDB's holistic storage view optimizer then implements any appropriate techniques to:

(1.) speed-up query processing, i.e. pick the most promising physical execution plan to compute a query;
(2.) apply updates to any SV in the SV store, i.e. pick the best update method like a batch-oriented differential update or log-structured merge-trees;
(3.) decide on the SVs to create and keep in the SV store, i.e. whether to materialize a new SV or drop an existing one;
(4.) combine results spanning several SVs, e.g. to join data from a row, a column store, and a streaming window.

The holistic SV optimizer is responsible for maintaining the primary SV (the log) and to create and maintain further secondary SVs. To do this, the optimizer uses a scan (index and full table) and update cost model to decide upon the appropriate SVs to create and subsequently maintain. Additionally, the optimizer uses a SV transformation cost model to decide whether to transform one SV into another. Finally, the optimizer is also responsible for deciding whether to drop a SV altogether. In this way, the holistic SV optimizer manages a *storage view lattice*, depending on the use-case and the workload, within the Storage View Store of the OctopusDB. For example, consider a travel database. The optimizer may create a Column Storage View for customer data, and a Row and Index Storage Views on ticket data. It can later archive tickets before a specified date into a Column Storage View. Given such a SV lattice, the optimizer speeds-up query processing by picking the most appropriate input Storage Views for a given query. Thus, the holistic SV optimizer in OctopusDB decides on the research problem of *simplicity vs optimization*.

6. PRELIMINARY RESULTS

In this section we provide the simulation insight and the experiment evidence of our approach in OctopusDB. But first we describe the use-case scenario used in our analysis below.

Use-Case Scenario. Consider a flight-booking system with a table `TICKETS` containing data on flight tickets; `CUSTOMERS` containing data on customer. Queries select tickets using predicates on different attribute subsets of `TICKETS`. For all selected tickets we retrieve all attribute values of matching customers. We assume that `TICKETS` is frequently updated. Thus, index maintenance on `TICKETS` is too expensive. This is a real-world example as proposed in [26]. This scenario calls for having a column layout on `TICKETS` and a row layout on `CUSTOMERS`. However, this flexible layout is not supported by current DBMSs. The update rate also precludes using fractured mirrors.

6.1 Simulation Insight

6.1.1 Flexible Layout

In this simulation we evaluate the scan costs for Row, Column, Indexed Row, and Indexed Column layouts over queries with varying selectivity and number of referenced attributes. The idea is to show that a single layout may not be suitable for the entire selectivity-referenced attributes space.

Setup. We use a simple cost model to estimate the random and sequential I/O costs for Row and Column layouts. Index scans incur an extra B^+-Tree index lookup cost followed by the

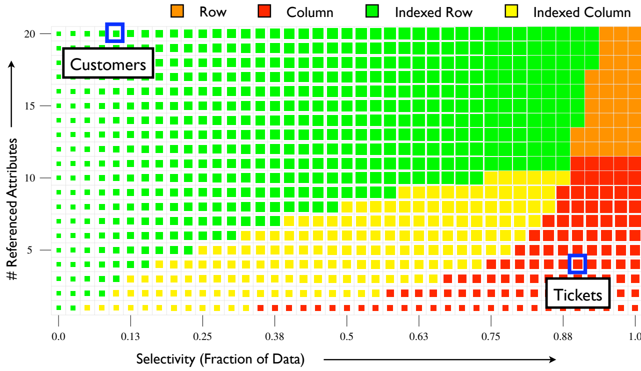

Figure 2: Scan Costs

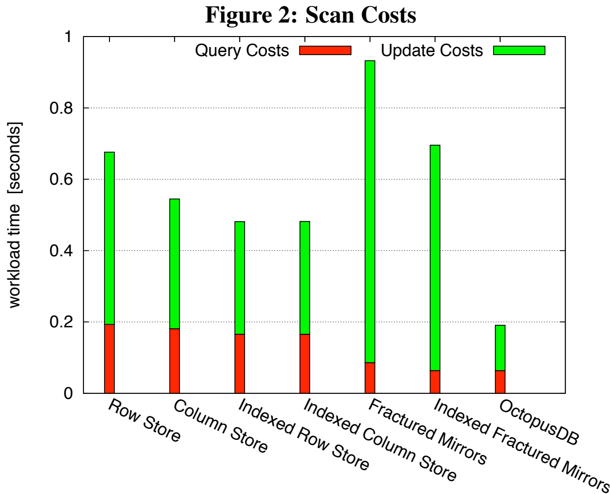

Figure 3: Workload Costs for Different Systems

respective Row or Column scan of the selected tuples. We vary number of referenced attributes from 1 till 20 and selectivity from 0.0 till 1.0.

Result. Figure 2 shows the result of the simulation. For each selectivity-attributes pair, we depict the cheapest layout (Row, Column, Indexed Row, Indexed Column) by a square of size proportional to the fraction of its cost to the maximum cost in the entire space. We can see four distinct regions in the figure where Row, Column, Indexed Row and Indexed Column are the best layouts respectively. Particularly, we observe that less number of referenced attributes favor Column layouts as compared to Row layouts. Additionally, higher selectivity (lesser tuples selected) favor indexed layouts as compared to unindexed ones.

Conclusion. From the result above we conclude that a single layout is not suitable for all use-cases. The right layout depends upon the workload. Hence, a flexible storage scheme makes sense for a *one-size-fits-all* database system.

6.1.2 Comparison with specialized DBMSs

In this simulation we compare the performance of OctopusDB on a given workload with Row Store, Column Store, Index Row Store, Indexed Column Store, Fractured Mirrors and Indexed Fractured Mirrors. The idea is to show that a *one-size-fits-all* system, like OctopusDB, can still outperform each of these specialized systems.

Setup. The Ticket and Customer tables contain $100,000$ and $20,000$ tuples respectively and 20 attributes each. The workload consists of equal number of scan and update queries having selectivity of 0.9 and 0.1 on Ticket and Customer tables respectively. Additionally, the queries access 4 and 20 attributes on Ticket and Customer tables respectively. We use the same cost model as in the previous simulation to estimate the total query and update costs.

Result. Figure 3 shows the overall workload time for different systems. We further break the overall time into query and update costs, shown as stacked bars. We can observe that only Indexed Fractured Mirrors match OctopusDB in terms of query cost. However, it has far higher update cost. Thus, as a result of flexible storage layout, OctopusDB outperforms all other database systems in terms of overall performance by a factor of up to 5.

Conclusion. From the above result we conclude that a single *one-size-fits-all* database system can not only adapt to several use-cases but can also outperform traditional database systems by a large margin.

6.2 Experimental Evidence

In this experiment we show how OctopusDB may automatically adapt SVs to a query response time requirement. We assume that the user specified that workload should not take longer than 0.1 sec.

Setup. Our first prototype of OctopusDB is a main memory implementation in Java 1.6. However, note that OctopusDB is not limited to main memory scenarios but could also be run as an external memory system. From our use-case scenario, we use `Tickets` and `Customers` records having 20 attributes each for our experiment. For each measurement, the workload contains a batch of 40 randomly picked scan and update queries in the ratio 1:3. We pick the search key attribute for scan queries using zipfian distribution with a skewness factor of 4; scan queries have a selectivity of 0.01. The experiment was executed on a medium-sized computing node. We used a single Intel Xeon Quadcore, 2.66Ghz (E5430) with 16GB of main memory. The operating system was Linux 2.6.27.7-9-xen.

Result. Figure 4 shows the automatic adaption of SVs for Tickets and Customers to query time requirement of 0.1sec when scaling database size. The figure shows the evolvement of workload time for both relations, the sum, as well as the SV transformation costs. With the increase in database size, we observe that several SVs become successively expensive beyond tolerable limits. OctopusDB copes with it by transforming the SVs to more suitable ones. This is impossible to achieve in traditional database systems.

Conclusion. OctopusDB can adapt over changing database size (use-case) by keeping a flexible storage layer. It meets the performance requirements by successively transforming the SVs into more suitable ones. Therefore, storage view selection is a core problem and a single holistic optimizer addressing it makes sense.

7. NEXT STEPS

7.1 Picking the Right layout

One of the core strengths of OctopusDB is picking the right SV (and the layout). We would therefore like to focus on picking the right layouts. We can pick the layout (row, column) on a per-table basis. Additionally, we can partition attributes in any arbitrary manner and store groups of attributes together i.e. pick the layout on a per-attribute basis; row and column layouts being the two extremes of the variety of possible layouts. Horizontal partitioning is

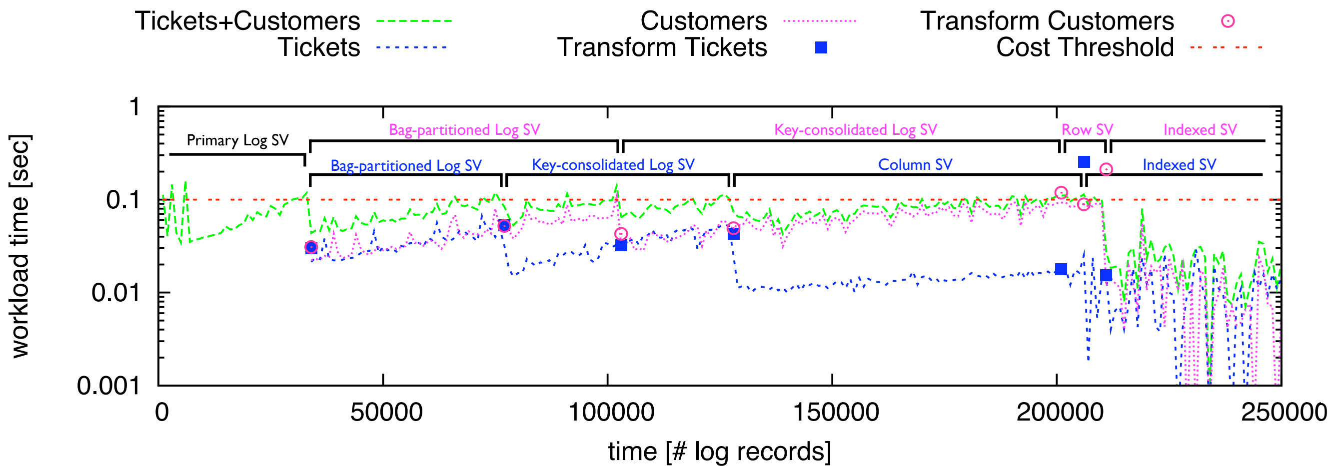

Figure 4: Automatic adaption in OctopusDB

another aspect which can play an important role. However, arriving at the right vertical and horizontal partitioning, adapting it over changing workload and considering even non-rectangular layouts is a challenge. Our next step is to investigate this challenge and come up with alternative techniques for dynamically arriving at the most suitable layout.

7.2 SV Compression

Compression is one of the most distinguishing features of column stores. Some people have argued that row stores can emulate efficient compression as well [4]. Our focus is to look into how compression plays an important role in the entire range of possible SVs; Row and Column SVs being just special cases. Because of the varying tuple size in vertically partitioned layouts, tuple level compression is a challenge. Our next step is to investigate that in more detail.

7.3 SV Maintenance

The bag of SVs in OctopusDB need to be maintained for further updates. View maintenance is an old and extensively researched problem in database systems. However, heterogeneous nature of the SVs is a unique challenge in OctopusDB. Further, the different possible layouts in OctopusDB differ from a fixed standard layout in traditional view maintenance. Therefore it needs a separate attention and is our next step of research.

7.4 OctopusDB Benchmarking

Next, to justify our claim that OctopusDB will be able to outperform other specialized systems, we will test our extended OctopusDB prototype on several different benchmarks: TPC-E for OLTP, TPC-H for OLAP, and Linear Road for Streaming systems among others. Finally, the existing benchmarks are for specialized applications (OLTP, OLAP etc) whereas increasingly there is a demand for combined OLTP and OLAP systems [16, 26]. Therefore, reflecting the newer demands, we would like to come up with a new benchmark for *one-size-fits-all* database systems.

8. REFERENCES

[1] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column oriented Database Systems (Tutorial). *PVLDB*, 2(2), 2009.

[2] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. Row-stores: how different are they really? In *SIGMOD*, 2008.

[3] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.

[4] N. Bruno. Teaching an Old Elephant New Tricks. In *CIDR*, 2009.

[5] G. Candea et al. A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses. In *PVLDB*, 2009.

[6] S. Chaudhuri and V. R. Narasayya. Self-Tuning Database Systems: A Decade of Progress. In *VLDB*, 2007.

[7] P. Cudré-Mauroux et al. A Demonstration of SciDB: A Science-Oriented DBMS (Demo). In *PVLDB*, 2009.

[8] P. Cudré-Mauroux et al. The Case for RodentStore: An Adaptive, Declarative Storage System. In *CIDR*, 2009.

[9] J.-P. Dittrich, P. M. Fischer, and D. Kossmann. AGILE: adaptive indexing for context-aware information filters. In *SIGMOD*, 2005.

[10] C. D. French. “One size fits all” database architectures do not work for DSS. In *SIGMOD*, 1995.

[11] C. D. French. Teaching an OLTP Database Kernel Advanced Data Warehousing Techniques. In *ICDE*, 1997.

[12] S. Heman et al. Efficient and Flexible Information Retrieval using MonetDB/X100. In *CIDR*, 2007.

[13] A. L. Holloway et al. How to Barter Bits for Chronons: Compression and Bandwidth Trade Offs for Database Scans. In *SIGMOD*, 2007.

[14] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *CIDR*, 2007.

[15] R. Kallman et al. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System (Demo). In *PVLDB*, 2008.

[16] H. Plattner. A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. In *Keynote Talk, SIGMOD*, 2009.

[17] R. Ramamurthy, D. J. DeWitt, and Q. Su. A Case for Fractured Mirrors. In *VLDB*, 2002.

[18] V. Raman et al. Constant-Time Query Processing. In *ICDE*, 2008.

[19] M. Stonebraker. The Case For Partial Indexes. *SIGMOD Rec.*, 18(4):4–11, 1989.

[20] M. Stonebraker and U. Cetintemel. ”One Size Fits All”: An Idea Whose Time Has Come and Gone (Abstract). In *ICDE*, 2005.

[21] M. Stonebraker et al. One Size Fits All? Part 2: Benchmarking Studies. In *CIDR*, 2007.

[22] M. Stonebraker et al. The End of an Architectural Era (It’s Time for a Complete Rewrite). In *VLDB*, 2007.

[23] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous Queries over Append-Only Databases. In *SIGMOD*, 1992.

[24] M. Theobald, H. Bast, D. Majumdar, R. Schenkel, and G. Weikum. TopX: efficient and versatile top- query processing for semistructured data. *VLDB J.*, 17(1):81–115, 2008.

[25] O. G. Tsatalos et al. The GMAP: A Versatile Tool for Physical Data Independence. *VLDB J.*, 5(2):101–118, 1996.

[26] P. Unterbrunner et al. Predictable Performance for Unpredictable Workloads. In *PVLDB*, 2009.

[27] M. Zukowski, S. Heman, N. Nes, and P. A. Boncz. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *VLDB*, 2007.