Towards a One Size Fits All Database Architecture*

[Outrageous Ideas and Vision track]

Jens Dittrich Alekh Jindal

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

ABSTRACT

We propose a new type of database system coined OctopusDB. Our approach suggests a unified, *one size fits all* data processing architecture for OLTP, OLAP, streaming systems, and scan-oriented database systems. OctopusDB radically departs from existing architectures in the following way: it uses a logical event log as its primary storage structure. To make this approach efficient we introduce the concept of *Storage Views (SV)*, i.e. secondary, alternative physical data representations covering all or subsets of the primary log. OctopusDB (1) allows us to use different types of SVs for different subsets of the data; and (2) eliminates the need to use different types of database systems for different applications. Thus, based on the workload, OctopusDB emulates different types of systems (row stores, column stores, streaming systems, and more importantly, any hybrid combination of these). This is a feature impossible to achieve with traditional DBMSs.

1. INTRODUCTION

1.1 Background and Motivation

In the past ten years we have seen considerable evidence that there is no *one size fits all* database architecture. We are currently witnessing a split of data management systems into several specialized solutions [20, 21]. For instance, for data warehousing database engineers already understood in the mid-nineties [9] 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 [10]. 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, Paraccel. At the same time other types of systems were forked including DSMS (data stream management systems) [23]; products include StreamBase.

1.2 Research Problem

As a consequence, today's companies have to manage and integrate several types of data management systems. Data has to be copied from one database system to another. To achieve this, complex, ETL-style data pipelines have to be glued together. The different database systems may also use different query languages or dialects. 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. Thus, the research challenge is to build a single efficient system covering the different database use-cases.

2. OCTOPUSDB

In this paper we take a radically new approach: we propose a single type of database system coined *OctopusDB*[1] that is able to mimic the behavior of the different species in the zoo.

Core Idea. The main idea of our system is to drop the assumption that a database system is developed around a central store (be it a row, column, or any hybrid store such as PAX [1] or fractured mirrors [17]) along with an ARIES-style [15] recovery log. OctopusDB does not have a fixed store. In OctopusDB all data is collected in a central log, i.e., all insert and update-operations create logical log-entries in that log. Based on that log we may then define several types of optional *Storage Views*. A *Storage View (SV)* represents all or part of the log in a different (or the same) physical layout. OctopusDB creates SVs transparently and solely based on the workload — and not based on some static decision for a concrete database product and hence a concrete storage layout. This single abstraction has another interesting consequence: the query optimization, view maintenance, index selection, as well as the store selection problems suddenly become a single problem: *storage view selection*, which OctopusDB treats inside a single *holistic storage view optimizer*.

The remainder of this section introduces OctopusDB's data model, primary log, and system interface. Section 3 introduces the concept of Storage View (SV). Section 4 introduces OctopusDB's holistic SV optimizer, Section 5 describes log purging and checkpointing, Section 6 discusses recovery and Section 7 explains con-

*Patent Pending - applied for by Saarland University [7].

[1] An octopus may adapt to its surroundings using a camouflage unmatched by any other species on earth: it may change both the color *and* the texture of its skin. Additionally, some octopus species may even mimic movements and shape thus *impersonating* other species, e.g. http://marinebio.org/species.asp?id=260

5^{th} *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

current transaction isolation in OctopusDB.

2.1 Data Model

The data items managed by OctopusDB are tuples $t_i = (a_1, .., a_{n(i)}),\ 0 \leq i \leq N$ where attributes $a_1, \ldots, a_{n(i)}$ may be of any type. The number of attributes for tuple i is given by $n(i)$. Each tuple is associated to a bag and a key. The bag is used to define subsets of the tuples, e.g. tables, partitions, collections. key identifies a tuple inside a bag. For simplicity, we assume a relational model throughout this paper. Therefore all tuples having the same *bag* share the same set of attributes (=schema).

2.2 The Primary Log Store

OctopusDB does *not* keep a row-, column- or any other store by default. All calls to the system interface are simply recorded in a sequential log, called *primary log*, creating appropriate logical log records. The primary log is itself an SV. OctopusDB stores its log persistently on durable storage (hard disk or SSD) following the write-ahead logging-protocol (WAL). For efficiency reasons we may keep a copy of the log in main memory, however this is no requirement. Each call to the system interface of OctopusDB internally creates a log record with an associated log sequence number *lsn*. As in traditional DBMSs no two log records may have the same lsn, therefore entries to the log are serialized. For the moment, all log records are *logical* and represent a *new state* defined by an operation[2]. Therefore, in contrast to ARIES, our log records do *not* represent changes that have been or should be applied to the database store. Our log simply contains the event history of operations without specifying how these events map to a particular store[3]. Thus, the format of our log record is (lsn, <method>, <parameters>) where <method> denotes the method of the system interface called and <parameters> denotes the parameters passed.

2.3 System Interface

Apart from traditional DBMS components (transaction manager, query optimizer, etc.), OctopusDB has a *primary log store* (to store the primary log) and a *storage view store* (to store additional storage views). To operate these, OctopusDB has a simple yet powerful interface containing the following methods:

registerSV(String svID, Type svType<, additionalPar>): creates and registers an SV of type svType having a unique identifier svID. Additional parameters may be passed to the SV.

registerQuery(String queryID, Query Q<, callback>): registers a query having a unique identifier queryID. An additional callback function may be passed.

snapshot(String outputSVID, String queryID): computes the result of the query and materializes it into the output SV.

maintain(String ouputSVID, String queryID): Same as snapshot, however, future updates will be reflected in outputSVID.

drop(String ID): Drops a query or SV from the system.

query(Query Q)→ Iterator it: Queries and/or modifies data in OctopusDB as specified in Q.

iterate(String ID) → Iterator it: Returns an iterator over the contents of the given query or SV.

Query definitions may be either a relational algebra expression as suggested in [4], SQL, or Pig Latin [16]. We will assume a relational algebra expression throughout this paper.

[2]In addition, transitional log records may be used, e.g. $a = a + 42$.

[3]The major performance advantage of ARIES is that it is using *physical* logging for REDO, i.e. intertwining a particular store with the log is a *feature* of ARIES. However, this feature may also be implemented in OctopusDB without giving up the logical primary log. See Section 6 for details.

3. STORAGE VIEW EXAMPLES

Storage Views (SVs) allow us to define arbitrary physical representations on the log. The main idea is to store the entire or a subset of the log or any other SV using a different physical layout. SVs *always materialize* their data. In general we create a network of SV dependencies with the goal to balance update and query processing costs. The dependency graph between different SVs is called the *SV lattice*. It is similar to the one used in materialized view maintenance (e.g. in data warehouses). However, the SV lattice is more general as it is not restricted to queries only but also has to consider the underlying storage layout. The interface to a SV contains the following private methods:

iterate(String queryID)→ Iterator it: Returns the result of the given query as an unordered iterator it. The query must be restricted to data covered by this SV.

iterationCost(String queryID)→ Cost c: returns the estimated cost c of the given query. In other words, it estimates the cost of **iterate(String queryID)**.

transformationCost(Type svType)→ Cost c: returns the estimated cost c for transforming this SV into svType.

For the flight booking use-case having tickets and customers data, presented in [25] and evaluated in the context of data layouts in [13], we shall now incrementally show the self-adapting API calls made by the system.

Log SV. Initially, the SV store does not contain any SVs; it only contains a single registered join query over tickets and customer data. It would be evaluated by scanning the primary log. However, as the log becomes too big, the system splits it into two logs:

```
registerSV("ticketsLog",LogSV);
registerSV("customersLog",LogSV);
registerQuery("customersOnly",σ_{bag=customers});
registerQuery("ticketsOnly",σ_{bag=tickets});
maintain("ticketsLog","ticketsOnly");
maintain("customersLog","customersOnly");
```

Further, OctopusDB consolidates the different versions for the same `(bag,key)`-pair to only keep the most recent one:

```
registerQuery("custRecent",γ_{recent}(Γ_{bag,key}(customersOnly)));
registerQuery("tickRecent",γ_{recent}(Γ_{bag,key}(ticketsOnly)));
maintain("ticketsLog","tickRecent");
maintain("customersLog","custRecent");
```

Row, Col SVs. The main idea of a Row SV, resp. Col SV, is to create a row store, resp. column store (or both), for any given table. Further, our system can create Row, Col SVs for any static or dynamic partition of the tables. For instance, OctopusDB creates *hot* and *cold* SVs for 7 day query window as follows:

```
registerSV("ticketsCold", ColSV);
registerSV("ticketsHot", ColSV);
registerQuery("tickRecentHot",σ_{time≥now−7days}(tickRecent));
registerQuery("tickRecentCold",σ_{time<now−7days}(tickRecent));
maintain("ticketsCold", "tickRecentCold");
maintain("ticketsHot", "tickRecentHot");
drop("ticketsLog");
```

Index SV. Indexes like B^+-trees, hash indexes, bitmaps, cache-optimized-trees, R-trees, inverted indexes, and so forth are just another type of SVs. The index may also be build on only parts of a table thus mimicking partial indexing [19]. In our use-case, OctopusDB build Index SVs over customers and hot tickets as follows:

```
registerSV("ticketsHotIndex",IndexSV,uncl,key=price);
registerSV("customersIndex",IndexSV,cl,key=id);
registerQuery("tickI1",π_{price,rid}(ticketRecentHot));
registerQuery("custI2",π_{ID,rid}(custRecent));
maintain("ticketsHotIndex","tickI1");
maintain("customersIndex","custI2");
```

In summary, the storage view concept allows us to model several important data managing concepts using a single abstraction only: both types of queries (point-in-time and continuous queries); different database stores (row-, col-, hybrid, etc.); and also the traditional query views (dynamic or materialized).

4. HOLISTIC SV OPTIMIZER

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. [11]. In contrast, a Col SV may implement column-oriented compression and vectorized iteration [2]. Outside those SVs OctopusDB's *Holistic Storage View Optimizer* then implements any appropriate techniques for storage view selection, update propagation and query processing. To perform these, OctopusDB has three *cost models* for Log, Row, Col and Index SVs: (1) A *query cost model*, which models the random and sequential I/O costs, (2) an *update cost model*, which models the minimum of chunk/random update costs, also considering the leaf/node split costs in Index SVs, and (3) a *transformation cost model*, which models the costs to transform one type of SV to another. Below, we briefly describe some of the optimization features in Holistic SV Optimizer.

SV Rearrangement. The holistic SV optimizer can rearrange the SV lattice in order to balance query and update costs. This implies that the SV optimizer decides how to connect the *tail* of an arrow to the existing SV lattice. One particular advantage of using a holistic optimizer is that the query operators can be pushed down through the entire SV store. — even beyond the primary log.

Operator Log-Pushdown. In certain situations, the optimizer may decide to push down some of the selections and projections as follows: (1) we examine the projections of all registered queries and compute the union set of attributes, (2) we push these projections down the lattice until the primary log. Similarly, for selections we (1) compute a conjunctive selection, and (2) push it even beyond the primary log. This means that any incoming log record will be checked even *before* putting it into the Log SV in the primary log.

Adaptive Partial SVs. The holistic SV optimizer can inject additional SVs to speed-up query processing. For instance, it does not make sense to build an index for an entire relation if only parts of that relation are queries. Techniques such as partial indexing [19] can be extended to create *dynamic* partial SVs.

Stream Transformation. For applications having continuous queries, we may only select a *window* of interest over the *unbounded* stream of log records i.e. the primary logical log in OctopusDB. This means the "database store" simply consists of several windows of interest. No other (older) data needs to be kept. OctopusDB can mimic this as follows: (1) do not use a Log SV for the Primary Log Store. (2) route all incoming log records to all relevant queries, (3) push possible updates up the SV lattice. In other words, we are reducing the stream processing problem to a *SV maintenance problem*.

Other Use-Cases. By creating the right SVs OctopusDB can mimic a variety of system e.g. OLTP, OLAP, Streaming Systems. Furthermore, by combining different SVs OctopusDB can emulate newer hybrid systems, for instance combinations of continuous and store (=archival) queries which has been researched heavily in the past years [6]. Table 1 lists several use-cases for OctopusDB.

5. PURGING AND CHECKPOINTING

The primary log may eventually grow too large, especially if the update rate is too high or the database has been up for a while and collected a long log of change operations. In this situation we need to shrink the size of the log. There are several options:

Purge log records for data that is not of interest anymore, e.g. changes older than two years are not needed for OLTP apps.

Compress the log, thus saving the storage space.

Checkpoint i.e. write a *begin checkpoint log record* to the log, create a storage view for all log records older than the begin checkpoint log record, and finally write an *end checkpoint log record*.

Use-Case (traditional systems)	Storage view definition type	Storage view definition example query
row store	Row SV	any
column store	Col SV	any
PAX	PAX SV	any
fractured mirrors	Row SV and Col SV	same query for both
column groups	Row SV and Col SV	$\pi_{a_1,\ldots,a_k}$ $\pi_{a_{k+1},\ldots,a_m}$
index	Index SV	any
indexed row store	Index SV(Row SV)	any
indexed column store	Index SV(Col SV)	any
read-optimized indexed column store **+ differential write-optimized row store**	Index SV(Col SV) Row SV	$\sigma_{t<\text{now}()-1\text{day}}$ $\sigma_{t\geq\text{now}()-1\text{day}}$
partial index	Index SV	$\sigma_{420\leq a_k\leq 42000}$
projection index	Col SV	π_{a_k}
partial projection index	Index SV(Col SV)	$\pi_{a_k}\left(\sigma_{420\leq a_k\leq 42000}\right)$
DSMS	Index SV	$\sigma_{t\geq\text{now}()-5\text{min}}$
DSMS **+ archive**	Index SV and Col SV	$\sigma_{t\geq\text{now}()-5\text{min}}$ $\sigma_{t<\text{now}()-5\text{min}}$
snapshot	any	any
replicated row store	Row SV Row SV	same query for both
query	any	any
dynamic view	any	any
materialized view	any	any

Use-Case (new system)	Storage view definition type	Storage view definition example query
OLTP **+ OLAP**	Row SV Col SV	$\sigma_{t\geq\text{now}()-1\text{day}}$ $\sigma_{t<\text{now}()-1\text{day}}$
DSMS **+ OLTP**	Index SV Row SV	$\sigma_{t\geq\text{now}()-5\text{min}}$ $\sigma_{t<\text{now}()-5\text{min}}$
DSMS **+ archive OLTP** **+ archive OLAP**	Index SV Row SV Col SV	$\sigma_{t\geq\text{now}()-5\text{min}}$ $\sigma_{\text{now}()-1\text{day}\leq t<\text{now}()-5\text{min}}$ $\sigma_{t<\text{now}()-1\text{day}}$
other hybrid	any combination of the above	any

Table 1: Use-Cases of OctopusDB

Then we purge all log records older than the begin checkpoint log record. Depending on the storage view we use for a checkpoint, we can *(a) archive:* Use a RowSV or ColSV, *(b) aggregate:* aggregate part of the log, or *(c) re-checkpoint:* Replace an existing checkpoint in the log with a derived checkpoint.

6. RECOVERY

Logical Recovery. Recovery depends on the purging strategy used. For no purging or checkpointing, since OctopusDB keeps the primary log on durable storage, simply copy the log from durable storage to main memory and OctopusDB is fully recovered. In the background OctopusDB will then re-create all SVs that existed before the crash. Note that the recovery process does not have to put any information on the progress information into the log, e.g. like compensation log records in ARIES [15]. This substantially simplifies the code base of our system. In case the log is purged or checkpointed (i.e. incomplete), we read the log sequentially starting from the oldest entry and collect begin checkpoint log records into a *checkpoint* set. If we find an end checkpoint log record then we remove the corresponding begin checkpoint from the set. If after reading the log *checkpoint* is empty, we proceed as if no log purging or checkpointing ever happened. Otherwise we copy the log to main memory, however ignore all checkpoints missing an end checkpoint log record. After copying this partial log, OctopusDB is recovered. After that, in the background we re-create all checkpoints that did not have an end checkpoint log record.

ARIES-style Physiological Recovery. One might argue that OctopusDB's recovery algorithm gives away some performance by not using page-oriented (physiological) REDO as in ARIES[4]. However, OctopusDB could easily be extended to keep physiological redo information as well. The trick is to write physiological REDO information for each SV separately. UNDO may still be performed using the global logical log. Conceptually, this algorithm then does not differ from ARIES anymore.

7. TRANSACTIONS AND ISOLATION

To support concurrent execution of transactions we extend OctopusDB's system interface by three methods:

beginTA()→ taID: starts a transaction.
commitTA(taID)→ bool: commits transaction.
abortTA(taID)→ bool: aborts transaction.

Furthermore, we extend the methods of OctopusDB's system interface (Section 2.3) to receive an additional taID parameter. Thus, we may define arbitrary transaction sequences. Now let's discuss how to achieve ACID in OctopusDB. As in DBMSs, *Consistency* may be guaranteed by validating a set of integrity constraints at commit time.The *Isolation* algorithm of OctopusDB is a variant of optimistic concurrency control. Its core idea is to append all changes (uncommitted or committed) to the primary log but only to propagate committed data to any secondary SVs. Uncommitted transactions can *read* committed data either from the log or any secondary SV. They are allowed to *write* any data object they desire by adding log records to the log, but: the latter modifications are *not* yet propagated to the secondary SVs. Using this approach *Atomicity* is trivial as only transactions having a commit log record are reflected in a SV and need to be considered by other operations. The same holds for *Durability*: as mentioned before, OctopusDB follows WAL anyway. Since our log records are not condensed into a store in the first place (as in current DBMSs), we do not need undo, redo, before or after images of pages, nor compensation log records to achieve idempotency. Finally, since SV update propagation process is a possible synchronization bottleneck, it could be interesting to improve this to enable eventual or timeline consistency *among* storage views, i.e. trade consistency for performance.

8. RELATED WORK & CONCLUSION

Several authors have supported the idea of different types of database systems for different markets/use-cases [9, 20, 21], splitting 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). OctopusDB is not restricted to a particular store and workload and hence there is no OLTP/OLAP boundary. OctopusDB extends this seamlessness further to DSMS. Due to the dramatic changes in hardware, HStore [22, 14] as well as several scanning techniques [11, 18, 3] propose stripped down or simplistic versions of traditional DBMSs. The design of OctopusDB is simple by default and adds only as much complexity as really needed. Rodent store [5] allows DBAs to declare the database store using an algebra and GMAP [24] presents a DDL for defining physical structures. However, in contrast to OctopusDB, these either still assume a fixed store or do not handle unification with streaming systems, automatic store selection and workload adaption. Cracked databases, e.g. [12], similar to partial indexing [19] and adaptive indexing [8], break database tables into horizontal pieces by piggy-backing index-reorganization requests to individual queries. However, in contrast to OctopusDB, cracked databases assume a fixed column store. Finally, MySQL allows users to plugin application specific custom storage engines. However, they are statically configured and offline installed by an administrator. In contrast, OctopusDB not only creates storage views adaptively over the application life cycle, but can also store any subset of the data in any arbitrary physical representation.

[4]Note that UNDO is logical in ARIES anyway.

Conclusion. This paper opened the book for one-size-fits-all database architecture. We presented OctopusDB as a single system for OLTP, OLAP, streaming databases, as well as several other types of databases. With OctopusDB we are inverting the traditional DBMS development philosophy: a specific store, which is an irrevocable design-decision, built-in into the DBMS and an ARIES-style [15] log-based recovery implemented on top. Instead, in our approach we start with the log (which is totally disconnected from any store) in the first place and if necessary, we define optional SVs on that log suited for a particular workload.

Acknowledgment. Work partially supported by MMCI.

9. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB*, 2001.
[2] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.
[3] G. Candea et al. A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses. In *PVLDB*, 2009.
[4] S. Chaudhuri and G. Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *VLDB*, 2000.
[5] P. Cudré-Mauroux et al. The Case for RodentStore: An Adaptive, Declarative Storage System. In *CIDR*, 2009.
[6] N. Dindar et al. DejaVu: declarative pattern matching over live and archived streams of events (Demo). In *SIGMOD*, 2009.
[7] J. Dittrich and A. Jindal. A method for storing and accessing data in a database system. Patent Application, 2010.
[8] J.-P. Dittrich, P. M. Fischer, and D. Kossmann. AGILE: adaptive indexing for context-aware information filters. In *SIGMOD*, 2005.
[9] C. D. French. "One size fits all" database architectures do not work for DSS. In *SIGMOD*, 1995.
[10] C. D. French. Teaching an OLTP Database Kernel Advanced Data Warehousing Techniques. In *ICDE*, 1997.
[11] A. L. Holloway et al. How to Barter Bits for Chronons: Compression and Bandwidth Trade Offs for Database Scans. In *SIGMOD*, 2007.
[12] S. Idreos and otherts. Database Cracking. In *CIDR*, 2007.
[13] A. Jindal. The Mimicking Octopus: Towards a one-size-fits-all Database Architecture. In *VLDB PhD Workshop*, 2010.
[14] R. Kallman et al. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System (Demo). In *PVLDB*, 2008.
[15] C. Mohan et al. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM TODS*, 17(1):94–162, 1992.
[16] C. Olston et al. Pig Latin: a Not-So-Foreign Language for Data Processing. In *SIGMOD*, 2008.
[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] O. G. Tsatalos et al. The GMAP: A Versatile Tool for Physical Data Independence. *VLDB J.*, 5(2):101–118, 1996.
[25] P. Unterbrunner et al. Predictable Performance for Unpredictable Workloads. In *PVLDB*, 2009.