AnyOLAP: Analytical Processing of Arbitrary Data-Intensive Applications without ETL

Felix Schuhknecht
Johannes Gutenberg-University
Mainz, Germany
schuhknecht@uni-mainz.de

Aaron Priesterroth
Johannes Gutenberg-University
Mainz, Germany
aprieste@students.uni-mainz.de

Justus Henneberg
Johannes Gutenberg-University
Mainz, Germany
henneberg@uni-mainz.de

Reza Salkhordeh
Johannes Gutenberg-University
Mainz, Germany
rsalkhor@uni-mainz.de

ABSTRACT

The volume of data that is processed and produced by modern data-intensive applications is constantly increasing. Of course, along with the volume, the interest in analyzing and interpreting this data increases as well. As a consequence, more and more DBMSs and processing frameworks are specialized towards the efficient execution of long-running, read-only analytical queries. Unfortunately, to enable analysis, the data first has to be moved from the source application to the analytics tool via a lengthy ETL process, which increases the runtime and complexity of the analysis pipeline.

In this work, we advocate to simply skip ETL altogether. With AnyOLAP, we can perform online analysis of data directly *within* the source application and *while* it is running. In the proposed demonstration, the audience will get the chance to put AnyOLAP to the test on a set of data-intensive applications that are supposed to be analyzed while they are up and running. As the entire analysis pipeline of AnyOLAP will be exposed to the audience in form of live and interactive visualizations, users will be able to experience the benefits of true online analysis firsthand.

PVLDB Reference Format:
Felix Schuhknecht, Aaron Priesterroth, Justus Henneberg, Reza Salkhordeh. AnyOLAP: Analytical Processing of Arbitrary Data-Intensive Applications without ETL. PVLDB, 14(12): 2823 - 2826, 2021.
doi:10.14778/3476311.3476354

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://gitlab.rlp.net/fschuhkn/anyolap_public.

1 INTRODUCTION

Modern data-intensive applications process and produce more and more data. They perform complex, often incremental computations on large input datasets, potentially modify them, and produce correspondingly large results. Of course, the produced data is supposed to be analyzed afterwards, which is often carried out in an analytical DBMS [8, 16, 18] or analytics framework [6, 7, 17]. However, before this analysis can happen, a cumbersome ETL process must be carried out, which consists of the following three steps: (1) *Extract*: The application materializes the results externally, e.g. in form of a CSV file. (2) *Transform*: The results are potentially transformed so that they are ready to be loaded into the analytics system, e.g. by reformatting all dates. (3) *Load*: The content of the CSV file is loaded into the proprietary database of the system. After this ETL process, the results are finally ready to be analyzed and interpreted by running a corresponding query in the system.

A step in the right direction is in-situ query processing [5, 13], which directly operates on raw text files. However we believe that even extraction can be eliminated from the pipeline. Instead of extracting the data from the application, it should be directly available for analysis *within* the application. Only this approach allows true *online* analytical processing, which is happening side-by-side with the running application. This is exactly the approach taken by AnyOLAP, which drastically simplifies the analysis pipeline as shown in Figure 1.

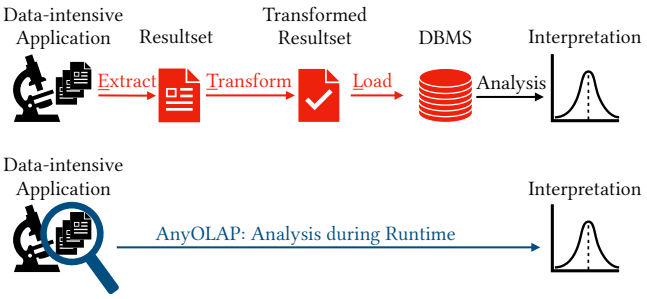

Figure 1: A typical analysis pipeline including an ETL process (top) vs our AnyOLAP pipeline without ETL (bottom).

But how can we access, analyze, and interpret the internal data of an application during its runtime? One option would be to modify the application such that it exposes its data in a shared memory space. This shared data could then be analyzed by an external system. Unfortunately, such a modification typically imposes deep and complex changes in the memory management of the application. Moreover, each and every application to analyze would require manual and careful adaptation for shared memory.

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476354

In AnyOLAP, we follow a minimally invasive approach: Instead of modifying the application that we want to analyze, we *attach* AnyOLAP to it in order to gain access to its internal memory. The key to do so lies in so-called *function interposing*, which is available in different variants for Linux [1], Mac OSX [2], and Windows [3] – here, we focus on the implementation in Linux. Interposing allows AnyOLAP to intercept allocation requests and get a handle for the virtual and physical memory that is in use by the application. Using this handle, AnyOLAP can take virtual snapshots of the memory regions of interest. AnyOLAP assures that these virtual snapshots remain consistent and are not affected by the concurrent execution of the host application. These virtual snapshots can then be transformed and interpreted by user-written analytical tasks. To ease usage and interpretation, AnyOLAP provides a GUI, which visualizes intercepted memory regions and the corresponding analysis process in a live and interactive fashion.

2 ANY OLAP

To enable analytics in a minimally invasive fashion, AnyOLAP utilizes a combination of rather exotic techniques, which we will describe at a high level in the following section.

Function interposing [1–3] forms the basis of AnyOLAP. In general, it works as follows: If an application calls a function from a dynamic library, the definition of the function is resolved during runtime. For example, if an application calls *mmap()*, the call is resolved at runtime by the definition of *mmap()* in the GNU C Library. The idea of function interposing is to *preload* a library containing an *alternative* definition of *mmap()*. As a consequence, the call is resolved by the alternative definition instead of the default one. In AnyOLAP, interposing has one essential purpose: To get a handle on the virtual and physical memory of the application. Precisely, we hijack *mmap()*, *mremap()*, and *munmap()*, as these are typically called by general-purpose allocators, such as *malloc()*, to allocate, resize, and free large virtual memory regions. Of course, AnyOLAP can be extended to hijack further calls if required, as long as the called functions are linked dynamically. The goal of this approach is to install a custom memory manager in the application, which enables us to take *virtual snapshots*.

A call to *mmap()* returns a newly allocated virtual memory region. By default, this virtual memory region is backed by *anonymous* physical memory. Anonymous means that the user cannot get a handle on it – it is transparently managed by the OS. This is a problem: To create efficient virtual snapshots [10, 11, 15], where physical memory is shared between the virtual memory of the application and the corresponding virtual snapshot, we need to get a handle for the physical memory first. To get this handle, we provide the following alternative definition of *mmap()*: Instead of returning a virtual memory area that is backed by anonymous physical memory, we return a virtual memory area that is backed by a so called *main-memory file* f. The pages of a main-memory file are again mapped to anonymous physical pages by the OS. As we can freely map virtual pages to file pages [14], this main-memory file is effectively our handle to the physical memory of the application.

2.1 Virtual Snapshotting

With the physical memory at hand, we are now able to create lightweight virtual snapshots: Assume we have a virtual memory region w of four virtual pages w_0 to w_3 that are mapped to the four file pages f_0 to f_3. If we now take a virtual snapshot s with respect to w, then s_0 to s_3 will map to the same four file pages f_0 to f_3. Thus, w and s share their physical memory. Figure 2 visualizes the situation right after s has been taken.

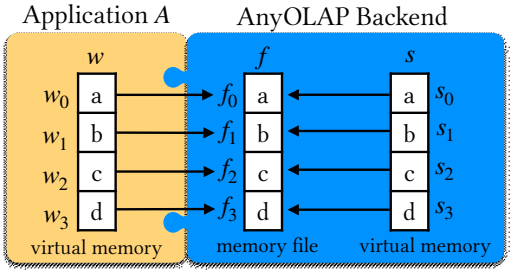

Figure 2: AnyOLAP takes a virtual snapshot s with respect to w. Both s and w map to the same file pages.

Of course, s should provide a consistent view for analysis *while* the application is writing to w. To realize this behavior, we implement a manual copy-on-write (CoW) mechanism, as described in detail in [14]. When creating the virtual snapshot s with respect to w, we set the memory protection of w to *read-only*. Thus, when the application intends to write to a virtual page, a segmentation fault is triggered. By default, this would terminate the application. However, AnyOLAP installs a *custom segmentation fault handler* that catches the fault. Then, we perform a manual CoW, which duplicates the physical page and adjusts the mapping as visualized in Figure 3. As a consequence, no write to w will be visible through s. Of course, the same holds in the other direction as well: A write to s, i.e., to transform the data of the snapshot, will not be visible in the host application.

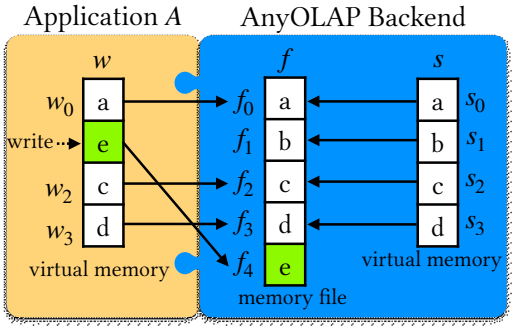

Figure 3: A write to w_1 triggers a manual CoW, where w_1 is remapped to an unused file page f_4, before the write is performed. As the corresponding page s_1 in the snapshot still maps to file page f_1, the snapshot remains consistent.

In summary, being able to create virtual snapshots offers two essential advantages over classical extraction: (1) Creating a snapshot is lightweight as only a memory mapping needs to be initialized [14]. (2) Only pages that are modified by the host application are actually copied.

2.2 Overhead

Of course, interposing an application and adapting its memory management comes at a performance price. In the following, we evaluate in a micro-benchmark how much the application is influenced by the creation of the snapshot. In our evaluation, the application creates an array of 10000 pages and AnyOLAP takes a snapshot on this area. Now, the application iterates two times over this array sequentially and updates the first x percent of its data. Figure 4 shows the results without AnyOLAP being attached (red) as well as with it being attached (blue). We can see that the overhead of the first iteration depends on the amount of data that is written, as this directly correlates with the amount of CoW that is happening. In the second iteration, the overhead vanishes, as the memory of the application has been separated from the memory of the snapshots.

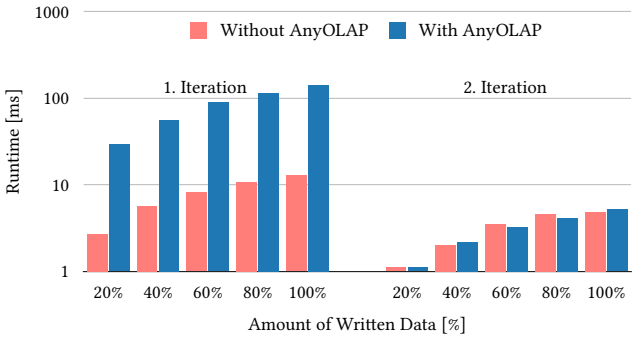

Figure 4: Overhead of AnyOLAP on the application.

3 DEMONSTRATION

Let us now discuss how the analysis pipeline looks like and how it can be used in practice. Along with that, we describe how the audience will use AnyOLAP to analyze data-intensive applications during the demonstration.

To simplify the interaction with AnyOLAP, we will provide an easy-to-use GUI on top of the backend. Via our GUI, users are able to configure the analysis pipeline, initiate the analysis process, and interact with AnyOLAP while the analysis is running. We realized the GUI in the Python framework Dash [4] as a web application, which communicates with the backend written in C. The AnyOLAP GUI visualizes the following information: (1) A **timeline** showing all (intercepted) memory in the virtual address space of the application (y-axis) with respect to the execution time (x-axis). (2) **Write accesses** that are happening to the memory. The monitoring granularity and frequency can be freely adjusted by the user. Visualizing accesses can help the user to understand the behavior of the application and to classify the different memory regions. (3) All taken **snapshots** with respect to the memory region on which they have been taken along with the point of creation time. (4) The **results of analysis tasks**, which have been computed on the snapshots.

3.1 Sample Application and Analysis Tasks

To understand the workflow, let us discuss analysis process for a sample application A, which performs an out-of-place radix sort on an array of integers. The algorithm maintains two large arrays w_1 and w_2, where w_1 initially contains the input data. Then, an out-of-place radix partitioning step partitions the data over to w_2 with respect to a portion of the integer. Afterwards, w_2 is copied back to w_1 and the next round starts. Besides w_1 and w_2, the algorithm also maintains a histogram h that is required to count how many elements are moved into each partition.

We want to analyze the following two properties during runtime: (`t1`) How many duplicates does the input contain? (`t2`) What is the current sortedness of the working set? For each of these properties, we create a so-called *analysis task*. An analysis task is written in C against a simple interface that receives a snapshot as its argument. Listing 1 shows the code for task `t2` (sortedness). Note that modifying an analysis task only requires a recompilation of AnyOLAP – no recompilation of the host application is necessary.

```
float t2_analyze(void* snapshot, size_t length) {
   float sortedness = 0.0;
   int* array = (int*) snapshot;
   size_t numEntries = length / sizeof(int);
   for (int i = 1; i < numEntries; i++)
      if (array[i-1] < array[i]) sortedness += 1.0;
   float numPairs = numEntries - 1;
   return (sortedness /  numPairs) * 100.0;
}
```

Listing 1: Analysis task `t2`, which computes the sortedness of an array of integers of a given snapshot.

3.2 Workflow

Let us now go through the actual analysis workflow, which is visualized in Figure 5. First, we upload the binary of A. Second, we select the *analysis mode* we want to perform. In *interactive mode*, which we will discuss in the following, the user triggers the creation of a snapshot on a particular memory region manually from the GUI. In *automatic mode*, snapshots are taken and analyzed automatically at a configurable frequency.

By starting the analysis, the AnyOLAP library is first preloaded in the background. Then, A is executed and continuously produces a timeline of the run. This timeline shows a live visualization of all detected memory allocations, as shown by the light blue rectangular areas in Figure 5. We can clearly identify w_1 and w_2 as the rectangles `m1` and `m2`. The dark blue dots within the rectangles mark the detected write accesses. We can identify the sequential copying into `m1` as well as the random writes into the individual partitions within `m2`. To create snapshots and to perform an analysis on them, we can now select one or multiple analysis tasks. We first select task `t1`, choose the input region `m1` and take the snapshot `m1_s1`. The backend will execute `t1_analyze(m1_s1, sizeof(m1_s1))` and provide the result to the GUI. By hovering over the visualized snapshot (red block), we can learn that 12% of the integers are duplicates.

Next, we select analysis task `t2` to analyze the sortedness during the run. We trigger the creation of a virtual snapshot on the working memory `m2`, resulting in the virtual snapshot `m2_s1`. The backend executes `t2_analyze(m2_s1)` and provides the result, which states that the array is already sorted by 23% after 10 seconds of execution. In contrast to `t1`, we want to execute task `t2` multiple times, as the sortedness changes over time. After 30 seconds, we thus create another virtual snapshot `m2_s2` to see how far we got. From the

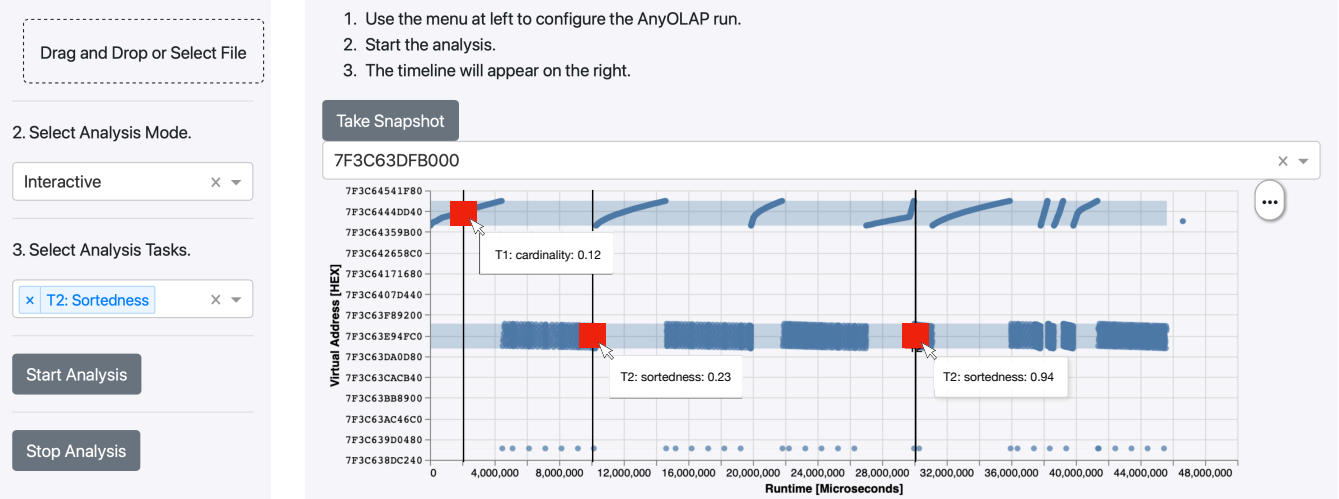

Figure 5: The GUI and the timeline of an AnyOLAP run. All intercepted memory allocations, their lifetime, as well as all taken snapshots are visualized. Further, the executed analysis functions are shown. Hovering the snapshot reveals the result. (Multiple visualizations have been merged to show the result of all executed tasks in one figure.)

execution of `t2_analyze(m2_s2)`, we learn that the sortedness has already reached 94%.

3.3 Limitations

While AnyOLAP removes the need to extract the data, it imposes certain requirements on the host application and the user. As we have seen in Listing 1, we require an analysis task to provide an *interpretation* of the snapshot in form of type information, as it cannot be extracted from the host application. Consequently, (a) the user must have an understanding of the internal data representation of the host application and (b) the data representation must be sufficiently simple. While this is not the case for all applications, we observed that this holds for many data-intensive applications from the HPC domain, such as [9, 12].

3.4 Data-intensive Demonstration Applications

Generally, AnyOLAP supports the analysis of arbitrary binaries. For the demonstration, we provide the following set of data-intensive applications to test:

(1) **A Collection of Sorting Algorithms**: A set of sorting algorithms (radix-sort, quick-sort, insertion-sort). The user will be able to monitor the access pattern of these algorithms as well as to analyze their progress. (2) **Optimization Problem Solving**: A search-based solver for a difficult optimization problem. The audience can view the development of the best solution found so far by the solver. Additionally, we showcase two real-world applications from the HPC domain: (3) **LULESH**: The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics [9] rooted in physics solves a Sedov blast problem to model hydrodynamics. The audience can observe how high-locality applications behave. (4) **NAMD**: The Nanoscale Molecular Dynamics program [12] from medicine simulates large biomolecular systems using spatial decomposition. The audience can analyze its memory access pattern and identify different phases of its runtime.

ACKNOWLEDGMENTS

We would like to thank Markus Vieth for significant contribution to the continuous development and improvement of this project.

REFERENCES

[1] July, 2021. https://man7.org/linux/man-pages/man8/ld.so.8.html
[2] July, 2021. https://blog.timac.org/2012/1218-simple-code-injection-using-dyld_insert_libraries
[3] July, 2021. https://www.microsoft.com/en-us/research/project/detours/
[4] July, 2021. https://dash.plotly.com
[5] Ioannis Alagiannis, Renata Borovica-Gajic, Miguel Branco, et al. 2015. NoDB: efficient query execution on raw data files. *Commun. ACM* 58, 12 (2015), 112–121.
[6] Michael Armbrust, Reynold S. Xin, Cheng Lian, et al. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD 2015, Melbourne, Victoria, Australia, May 31 - June 4*. ACM, 1383–1394.
[7] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 137–150.
[8] Franz Färber, Sang Kyun Cha, et al. 2011. SAP HANA database: data management for modern business applications. *SIGMOD Rec.* 40, 4 (2011), 45–51.
[9] Ian Karlin, Jeff Keasler, and JR Neely. 2013. *Lulesh 2.0 updates and changes*. Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA.
[10] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE 2011, April 11-16, 2011, Hannover, Germany*. IEEE Computer Society, 195–206.
[11] Henrik Mühe, Alfons Kemper, and Thomas Neumann. 2011. How to efficiently snapshot transactional data: hardware or software controlled?. In *DaMoN 2011, Athens, Greece, June 13, 2011*. ACM, 17–26.
[12] Mark T. Nelson, William Humphrey, Attila Gürsoy, et al. 1996. NAMD: a Parallel, Object-Oriented Molecular Dynamics Program. *Int. J. High Perform. Comput. Appl.* 10, 4 (1996), 251–268.
[13] Matthaios Olma, Manos Karpathiotakis, et al. 2020. Adaptive partitioning and indexing for in situ query processing. *VLDB J.* 29, 1 (2020), 569–591.
[14] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. 2016. RUMA has it: Rewired User-space Memory Access is Possible! *PVLDB* 9, 10 (2016), 768–779.
[15] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. Accelerating Analytical Processing in MVCC using Fine-Granular High-Frequency Virtual Snapshotting. In *SIGMOD 2018, Houston, TX, USA, June 10-15*. ACM, 245–258.
[16] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, et al. 2005. C-Store: A Column-oriented DBMS. In *VLDB, Trondheim, Norway*. ACM, 553–564.
[17] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, et al. 2010. Spark: Cluster Computing with Working Sets. In *HotCloud'10, Boston, MA, USA, June 22*. USENIX Association.
[18] Marcin Zukowski, Peter A. Boncz, Niels Nes, and Sándor Héman. 2005. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Eng. Bull.* 28, 2 (2005), 17–22.