Alpine: Efficient In situ Data Exploration in the Presence of Updates

Antonios Anagnostou *
AUTH
anagnoad@csd.auth.gr

Matthaios Olma
EPFL
matthaios.olma@epfl.ch

Anastasia Ailamaki
EPFL
anastasia.ailamaki@epfl.ch

ABSTRACT

The ever growing data collections create the need for brief explorations of the available data to extract relevant information before decision making becomes necessary. In this context of data exploration, current data analysis solutions struggle to quickly pinpoint useful information in data collections. One major reason is that loading data in a DBMS without knowing which part of it will actually be useful is a major bottleneck. To remove this bottleneck, state-of-the art approaches perform queries *in situ*, thus avoiding the loading overhead. In situ query engines, however, are index-oblivious, and lack sophisticated techniques to reduce the amount of data to be accessed. Furthermore, applications constantly generate fresh data and update the existing raw data files whereas state-of-the art in situ approaches support only append-like workloads.

In this demonstration, we showcase the efficiency of adaptive indexing and partitioning techniques for analytical queries in the presence of updates. We demonstrate an online partitioning and indexing tuner for in situ querying which plugs to a query engine and offers support for fast queries over raw data files. We present Alpine, our prototype implementation, which combines the tuner with a query executor incorporating in situ query techniques to provide efficient raw data access. We will visually demonstrate how Alpine incrementally and adaptively builds auxiliary data structures and indexes over raw data files and how it adapts its behavior as a side-effect of updates in the raw data files.

1. INTRODUCTION

Nowadays, numerous applications in various domains generate and collect massive amounts of data at a rapid pace. Research fields and applications such as network monitoring and sensor data management require broader data analysis functionality to rapidly gain deeper insights from the available data. Analyzing and understanding all available data is impossible in practice due to the data explosion of the last decade. In order to circumvent this problem, scientists use data exploration, which helps to identify and efficiently extract the relevant data before deeper analysis starts. In such cases, the users explore the available data for actionable information, trying to assess the problem space before making a decision [10, 11, 13, 18, 22]. A key observation behind such use cases is that even though datasets grow exponentially, only a small subset of the data is typically relevant [1]. Furthermore, the analysis requires to include any fresh data which is generated at real time.

The data exploration process involves multiple query iterations that progressively focus on smaller parts of the data until reaching a final result. For example, an electricity monitoring company continuously monitors information about the current and aggregate energy consumption, and other sensor measurements such as temperature. To optimize consumption, the company performs predictive analytics over smart home datasets, looking for patterns that indicate energy request peaks and potential equipment downtime [14]. Analyses in this context start by identifying potentially relevant measurements by using range queries and aggregations to identify areas of interests. The analysis focuses on specific data regions for a number of queries, but is likely to shift across the dataset to a different subset of the data. The analysis also incorporates any fresh measurements produced. However, there are two major obstacles in such exploratory tasks, i) the ever growing data amounts, and ii) the shifting areas of interest within a workload.

Traditional data management solutions struggle to serve data exploration tasks. Conventional DBMS require costly actions (i.e., loading and indexing) to provide the desired interactive nature of exploratory analysis. Given the data sizes involved, any transformation, copying, and preparation steps over the data introduces substantial delays before the data can be utilized and queried [3, 4, 19]. In addition, when going through never-before-seen data, no assumptions can be made about the data or queries. A user may become interested in different value ranges and/or attributes. Workload shifts may nullify investments towards auxiliary data structures (e.g., indexes) as predicting in which areas of the dataset to invest is non-trivial.

We recognize the requirement to minimize data to query time while offering low response times. The data-to-query time is of critical importance as it defines the moment a database system becomes usable, and thus useful and offering high query performance. Minimal data-to-query time is important for iterative exploratory analysis in order to increase user productivity.

To address these issues, we propose a database tuner which offers adaptive logical partitioning and indexing for in situ query processing. By executing queries on raw data files, there is no need for data loading, and by partitioning and indexing adaptively, only relevant data will be indexed, thus minimizing the required investment in building a full index both in memory space and execution time. The tuner makes decisions using a randomized algorithm based on the past workload and data statistics gathered during execution. Fur-

*Work done while the author was at EPFL.

SIGMOD'17, May 14-19, 2017, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/3035918.3058743

thermore, we propose an in situ execution technique which enables in-place and append-like updates while querying the raw data file.

Contributions and Demo. We demonstrate Alpine, an in situ query engine incorporating the adaptive tuner and the in situ update techniques. Our demonstration of Alpine aims at a) introducing the adaptive in situ tuner through a system implementation, b) introducing in situ update techniques and c) demonstrating the extent at which the tuner can be adopted by a traditional database system without altering the internals of the query engine. We visually demonstrate the behavior of its core components in a range of scenarios, giving the audience a complete visual insight into the behavior of Alpine and the trade-offs that come with in situ query processing. In addition, we present a comparison between Alpine and other in situ configurations in an interactive way with the audience by organizing a 'friendly' race between the systems.

Innovation. Alpine starts processing queries without any data preparation or loading steps. As more queries are processed, response times improve due to the adaptive decisions of the tuner to build indexes. We visually demonstrate these effects by observing the partitioning and indexing process. Audience members will see how the partitioning and indexing of the system evolve as additional queries arrive, or when the workload changes. Furthermore, when the raw data file is updated, Alpine recognizes automatically the updated subset and minimizes the changes required in its auxiliary structures. We demonstrate the effect of updates in the raw data file by introducing random changes in the file and observing the cost of re-structuring with and without the mechanisms of Alpine.

Visual Experience. The audience has the ability to interact with the system through a graphical interface that allows them to change the input characteristics of the workload. Properties such as the number of attributes and their width, may significantly change the behavior of Alpine. Additionally, the graphical interface provides access to different Alpine configurations. For instance, the user can enable or disable caching, indexing or positional maps and/or specify the amount of storage space which is devoted to internal indexes and caches. Users will be able to change these parameters and observe the impact on performance.

2. RELATED WORK

In recent years, many research efforts propose re-designing the traditional data management architecture to address the challenges and opportunities associated with data exploration scenarios.

Queries over Raw Data. Works from the DBMS and Hadoop ecosystems advocate executing queries over raw data to reduce costs related to data loading [15]. On the DBMS side, Abiteboul et.al. [2] show how to offer database querying and updates over both structured raw data files and data stored in databases. NoDB [4] treats raw data files as first-class citizens of the DBMS, introducing auxiliary data structures such as positional maps to reduce the expensive parsing and tokenization costs of raw data access. RAW [21] introduces code-generated access paths to adapt the query engine to the underlying data formats and incoming queries. Proteus [20] introduces a code-generated query execution engine that enables efficient query execution despite data heterogeneity. SCANRAW [12] is a physical operator for in-situ processing which exploits parallelism to mask the CPU processing costs associated with raw data accesses. DBMS-based approaches either rely on accessing the data via full table scans or require a priori workload knowledge and enough idle time to create the proper indexes. On the other side of raw data querying, Instant Loading [23] parallelizes the loading process for main-memory DBMS and it focuses on making the loading of data that is streamed from high speed source media transparent.

Adaptive Indexing. Database cracking [16, 17] entails incremental refinement of indexes during query processing to tackle the problem of evolving workloads in the context of in-memory column-stores. HAIL [24] proposes an adaptive indexing approach for MapReduce systems. ARF [5] is an adaptive value-existence index which resembles Bloom filters yet is usable in range queries. ARF dynamically increases the used bits for representing hot data areas, and reduces the number for cold ones. Similarly, alternative approaches such as the BF-tree [6] trade query accuracy for storage to produce smaller, yet fast, index structures. Alpine shares the same motivation with adaptive indexing: it builds indexes during query processing and continuously adapts to the workload characteristics instead of investing upfront in index building [7, 8]. Nevertheless, applying existing technology to *in situ* query processing is non-trivial. For example, database cracking [16] refines the physical order of data, and therefore would require expensive duplication of pages coming from the data files.

Online indexing techniques periodically re-evaluate physical design decisions. COLT [25] continuously monitors the workload and periodically creates new indexes and/or drops unused ones. COLT adds overhead on query execution because it obtains cost estimations from the optimizer at runtime. A "lighter" approach which requires fewer calls to the optimizer was proposed in [9].

3. ALPINE ARCHITECTURE

In this section, we discuss the design of Alpine. Alpine uses state-of-the-art in situ querying techniques over which incorporates logical partitioning and fine-grained indexing, thereby reducing the amounts of accessed data. To remain effective despite workload shifts, Alpine introduces an online partitioning and indexing tuner, which calibrates and refines logical partitions and secondary indexes both based on data and query statistics. Furthermore, Alpine enables append-like and in place data updates without disturbing query execution by dynamically adapting its auxiliary data structures. Alpine monitors the queried files for updates and minimizes the effect of changes in its data structures and caches.

3.1 In situ access

Alpine launches queries directly over the original raw data files to avoid the cost of data loading. However, raw data file accesses are expensive thus, Alpine uses in situ querying techniques to speed-up raw data file access by (a) speeding up query processing via raw data indexing and by eliminating the need to access and re-convert hot raw data via caching. To achieve that, Alpine uses positional maps (PM) to reduce raw data access costs. PMs are data structures which are populated on-the-fly and maintain structural information about an underlying raw file. Specifically, they keep the positions of various attributes of the file. During query processing, this information is used to jump to the exact position of an attribute or as close as possible to an attribute, significantly reducing the cost of tokenizing and parsing when a tuple is accessed. Furthermore, Alpine builds binary caches of already converted fields to reduce parsing and data type conversion costs of future accesses.

3.2 Partitioning and Indexing

Alpine specifies a logical partitioning scheme for each attribute in a table. This process starts when a query accesses an attribute for the first time. Alpine logically divides the file into contiguous non-overlapping pieces to manipulate it at a finer granularity. Alpine internally represents each partition by its starting and ending byte within the original file. The Partition Manager triggers the Structure Refiner to iteratively fine-tune the partitioning scheme with every subsequent query, and progressively all partitions reach a state

in which there is no benefit from further partitioning. The efficiency of a partitioning scheme highly depends on the data distribution and the query workload. Therefore, Alpine varies the partitioning technique it uses based on value cardinality. Furthermore, Alpine evaluates the benefit of indexes and suggests the most promising combination of indexes for a given attribute. Every index corresponds to a specific data partition. Depending on the query workload and selectivity, a single partition may have multiple indexes. Since all indexes must be constructed during query processing and with minimal build overhead, it is non-trivial to choose which indexes to build and when to build them. Alpine decides on which index to build based on past queries; timing is based on an online randomized algorithm which has the following inputs: (i) statistics on the cost of full scan, (ii) statistics on the cost of building an index, and (iii) partition access frequency.

3.3 Updates

Alpine enables both append-like and in-place updates directly upon the raw data file and ensures consistent results. To achieve that, Alpine (i) monitors input files for updates at real-time, (ii) stores a summary of the most recent consistent state for reference, (iii) identifies the updated file subsets, and (iv) restructures its internal data structures accordingly.

In order to be able to discover the updated rows in the file and the type of update (append or in-place), Alpine exploits its logical partitioning scheme. For each partition, Alpine stores a 64 byte MD5 hash code of the contents within that partition, the starting and ending positions of the partition in the file, as well as the characters corresponding to those positions. This information is sufficient to identify the existence of an update within a partition as they summarize the size of the partition as well as the content.

In order to recognize whether the input textual file has been updated by another application (e.g., vim), Alpine uses OS support (i.e., inotify). Specifically, Alpine initializes a watchdog which is triggered when a file is written upon and adds a log entry into a queue. This queue contains all updates that have not been addressed by Alpine yet. Alpine checks the queue for new updates both at the beginning of every query as well as during execution. During a running query, Alpine checks for any updates that happened in data that has been already scanned. If such an update has taken place, Alpine re-executes the query as results might be invalid if not all records were read from the same file version.

To identify the type of update, Alpine compares the current state of each partition with the stored one. Thus, Alpine checks whether the partition beginning and ending character has changed or if the MD5 hash code has changed. If the state of each partition matches with the existing one, then the update type is an append. Otherwise, it is an in-place update.

Specifically, for append-like updates Alpine creates a new partition at the end of the file to accommodate the new data and during the first query after an update, builds binary caches, PMs and indexes over them. In-place updates require special care in terms of positional map and index maintenance because they change the internal structure of the file. For example, a value change in the file might require positional map reorganization. Alpine deals with in-place updates during the query following an update. Specifically, Alpine recognizes the updated partitions, updates the positional map, and recreates the other corresponding structures.

4. DEMONSTRATION WALKTHROUGH

We present the demonstration in three parts. In the first part we introduce the audience to the motivation behind minimizing data access and the techniques used by Alpine for data access, partitioning, indexing, updates and the adaptivity mechanism. The second part presents the trade-offs of the indexing and partitioning mechanisms and demonstrates a hands-on experience giving a detailed insight into the system. The third part provides an interactive demonstration of updates over Alpine. The audience will be able to modify a file with in-place and append like updates and monitor at real time the adaptivity of Alpine's data structures. The demonstration has a strong visual component and audience participation.

System setup. The graphical web interface runs on a dedicated virtual machine in our lab and can be accessed by a laptop and tablet at the conference, where the audience can define the parameters for the scenarios, and visualize the results. The experiments run remotely on multiple machines hosting multiple instance of our system. We use multiple machines, in parallel, to measure the response time and execution statistics of the different configurations required for the following scenarios. Each machine has two 8-core Intel Xeon E5-2660 processors (with hyper-threading enabled), 128GB RAM, and seven 300GB 15kRPM SAS 3.5" hard disks. The O/S is a 64-bit Red Hat, with a 2.6.32 kernel.

4.1 Part I: Introduction to in situ DB tuner

In this part of the demonstration, we use a poster to introduce the audience to the techniques used to improve performance for in situ query execution. Initially, we present the requirements of exploratory analysis workloads through a real-world scenario. Subsequently, we describe the relevant in situ querying techniques we used to minimize the cost of data access as well as the design choices minimizing the amount of data access through online indexing and partitioning as well as the adaptivity technique and the cost model it is based on. Furthermore, we present the complications in querying update-able raw data files and the techniques we use for update monitoring and data structure restructuring. Finally we present the design of our prototype, Alpine, and we explain how it couples data access techniques with indexing and partitioning.

4.2 Part II: Indexing and Partitioning tuner

The second part of the demo illustrates the trade-offs of adaptive indexing and partitioning over vanilla in situ query processing through live experimentation. For this, we show how Alpine maintains binary caches and positional information about the raw data files to improve data access performance and how it uses partitions and indexes to minimize the data access. We demonstrate advantages of adaptive data partitioning versus static partitioning by visualizing the performance gains from partitioning of data based on query requirements. Adaptive partitioning is beneficial as it allows to create a smaller number of partitions with homogeneous data, thus increasing the probability of skipping. Furthermore, we demonstrate the efficiency of adaptive indexing through presenting the memory requirements along with the performance gains. Adaptive indexing over partitions reduces the memory requirements of indexing thus allowing Alpine to efficiently access larger datasets.

User Interface. The demonstration uses an interactive graphical user interface to expose run-time statistics of internal system components during query execution. In particular, we monitor the number of partitions and the storage space occupied by the positional map, the caching structures and the indexes over each partition. We allow the user to change the queried attributes and value ranges and we visualize the changes in performance along with execution statistics (number of accessed partitions and selectivity). Finally, the interface allows users to enable or disable some system components, e.g. indexes, the positional maps, and run a variety of Alpine instances to compare live the effect of the structures. An example screen-shot of the user interface is shown in Figure 1.

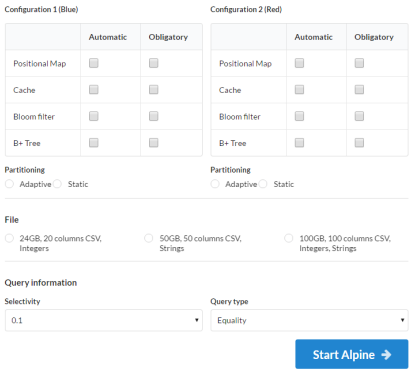

Figure 1: Interface for Tuner

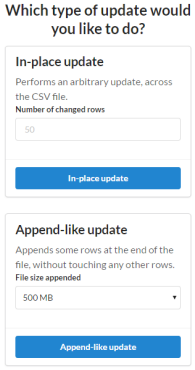

Figure 2: Interface for Updates

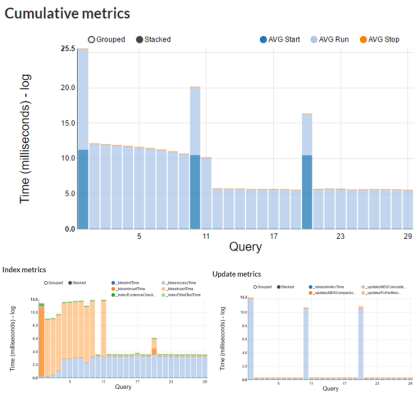

Figure 3: Interface for Updates

Query Execution Breakdown. To highlight the trade-offs in adaptive in situ query processing, we monitor the query execution in Alpine and we present apart from query performance, an execution breakdown presenting the effect of every operation during each query. To demonstrate the difference between adaptive in situ query processing with other in situ approaches, we also visualize the query processing efficiency of different configurations. The default scenario presents a comparison of Alpine using adaptive indexing and partitioning versus a simple in situ query execution using only caches and positional maps. However, the user can enable or disable any of the available data structures.

4.3 Part III: Adapting to random updates

In this part of the demonstration, we present the resilience of Alpine to updates. The graphical user interface provides to the user two different options: (i) append data to the end of the file, (ii) provide a number of lines in the input file to be updated. The position of the lines is randomly selected for uniform update distribution. An example screen-shot of the user interface is shown in Figure 2.

The user interface continuously exposes the run-time statistics along with breakdowns for the cost of query execution. Once an update is submitted, the user interface visualizes the number of partitions updated, the number of re-built indexes, and the cost of updating the data structures (As shown in Figure 3). By storing the state of each partition, Alpine manages to identify the updated partitions and minimize the required restructuring.

5. CONCLUSIONS

Data exploration is increasingly becoming a key process for modern applications in businesses and in sciences Alpine, exploits prior work for in-situ query processing and enriches it with low-overhead adaptive partition and index tuning. Furthermore, until now in-situ engines either prohibited users to update the queried files while executing workloads, or rebuilt all their auxiliary data structures. This demo showcases how Alpine enables users to update the queried files on demand. Through a graphical user interface, the demo allows the user to see the adaptive behavior of Alpine that results in efficient execution over raw data. The system monitors the state of its main components, i.e., partitioning, indexing and caching, and shows the resource consumption during every query execution. By providing an interactive interface, users can set their own scenarios regarding the data input and the various systems knobs, observing the effect of different parameters on the system performance.

Acknowledgments. We would like to thank the reviewers for their valuable comments. This work is partially funded by the EU FP7 programme (ERC-2013-CoG), Grant No 617508 (ViDa).

6. REFERENCES

[1] C. L. Abad, N. Roberts, Y. Lu, and R. H. Campbell. A Storage-centric Analysis of MapReduce Workloads: File Popularity, Temporal Locality and Arrival Patterns. In *IISWC*, 2012.

[2] S. Abiteboul, S. Cluet, and T. Milo. Querying and Updating the File. *PVLDB*, 1993.

[3] A. Abouzied, D. J. Abadi, and A. Silberschatz. Invisible Loading: Access-driven Data Transfer from Raw Files into Database Systems. In *EDBT*, 2013.

[4] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. In *SIGMOD*, 2012.

[5] K. Alexiou, D. Kossmann, and P.-A. Larson. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. *PVLDB*, 2013.

[6] M. Athanassoulis and A. Ailamaki. BF-tree: Approximate Tree Indexing. *PVLDB*, 2014.

[7] M. Athanassoulis and S. Idreos. Design Tradeoffs of Data Access Methods. In *SIGMOD*, 2016.

[8] M. Athanassoulis, M. Kester, L. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. Designing Access Methods: The RUM Conjecture. In *EDBT*, 2016.

[9] N. Bruno and S. Chaudhuri. An Online Approach to Physical Design Tuning. In *ICDE*, 2007.

[10] U. Çetintemel, M. Cherniack, J. DeBrabant, Y. Diao, K. Dimitriadou, A. Kalinin, O. Papaemmanouil, and S. Zdonik. Query Steering for Interactive Data Exploration. In *CIDR*, 2013.

[11] G. Chatzopoulou, M. Eirinaki, and N. Polyzotis. Query Recommendations for Interactive Database Exploration. In *SSDBM*, 2009.

[12] Y. Cheng and F. Rusu. Parallel In-situ Data Processing with Speculative Loading. In *SIGMOD*, 2014.

[13] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. J. DeWitt, and G. Heber. Scientific Data Management in the Coming Decade. *SIGMOD Rec.*, 2005.

[14] IBM. Managing big data for smart grids and smart meters. White Paper.

[15] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? In *CIDR*, 2011.

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

[17] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-memory Column-stores. *PVLDB*, 2011.

[18] S. Idreos, O. Papaemmanouil, and S. Chaudhuri. Overview of Data Exploration Techniques. In *SIGMOD*, 2015.

[19] M. Ivanova, M. Kersten, and S. Manegold. Data Vaults: A Symbiosis Between Database Technology and Scientific File Repositories. In *SSDBM*, 2012.

[20] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast Queries over Heterogeneous Data Through Engine Customization. *PVLDB*, 2016.

[21] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive Query Processing on RAW Data. *PVLDB*, 2014.

[22] M. L. Kersten, S. Idreos, S. Manegold, and E. Liarou. The Researcher's Guide to the Data Deluge: Querying a Scientific Database in Just a Few Seconds. *PVLDB*, 2011.

[23] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. Instant Loading for Main Memory Databases. *PVLDB*, 2013.

[24] S. Richter, J.-A. Quiané-Ruiz, S. Schuh, and J. Dittrich. Towards Zero-overhead Static and Adaptive Indexing in Hadoop. *VLDBJ*, 2014.

[25] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. Colt: Continuous on-line tuning. In *SIGMOD*, 2006.