Past and Future Steps for Adaptive Storage Data Systems: From Shallow to Deep Adaptivity

Stratos Idreos, Manos Athanassoulis Niv Dayan, Demi Guo, Mike S. Kester, Lukas Maas, and Kostas Zoumpatianos

Harvard University

Abstract. Data systems with adaptive storage can autonomously change their behavior by altering how data is stored and accessed. Such systems have been studied primarily for the case of adaptive indexing to automatically create the right indexes at the right granularity. More recently work on adaptive loading and adaptive data layouts brought even more flexibility. We survey this work and describe the need for even deeper adaptivity that goes beyond adjusting knobs in a single architecture; instead it can adapt the fundamental architecture of a data system to drastically alter its behavior.

Key words: adaptive data systems, adaptive indexing, adaptive storage

1 Adaptive Data Systems

Data systems design is a massive collection of small decisions that collectively make the architecture of a data system [19, 1, 8]. Typically data systems come with several tuning knobs that allow adjusting the system for different workloads and even for different hardware. As data systems have grown more complex over the years, the same is true for their tuning knobs. As a result, this task is offloaded to human experts, database administrators, that understand very well both the internals of the data system at hand and the specifics of the target application and environment.

One of the most important tuning knobs that dramatically affects performance is the selection of secondary and primary indexes that should be materialized to support a given application. This choice has attracted a lot of attention over the years given its drastic effect on performance and the complexity of making the right decisions. The main solution that helped scale such tuning choices is the introduction of auto-tuning tools. These tools automate the discovery of good candidate indexes and propose a possible configuration to the database administrator. In turn, the administrator may approve or edit the final tuning options [11]. Such tools rely primarily on what-if analysis and interaction with the optimizer to obtain estimated query costs should a candidate index set be materialized. There has been significant progress over the years including work that periodically reconsiders the tuning choices [9, 10].

A big part of why the choice of indexes is such a critical factor is that it truly affects the architecture of a data system as it changes how data is stored and accessed. With the continuing trend of CPUs improving at a faster pace than memory, having the right data layout is one of the biggest deciding factors in performance of data systems as it allows us to minimize data movement. Having the correct index, i.e., the one that contains just enough relevant data (attributes) and in the best layout for the given query, means that all appropriate queries will use this optimal storage layout to access data while otherwise they would use a plain scan or a less appropriate index resulting in the need to move around many more data (pages).

While the work on auto-tuning tools has provided significant advances, it has limitations when it comes to varying workloads and data exploration scenarios [26]. In such cases, the workload cannot always be predicted up front. This means that auto-tuning tools cannot rely on a known workload to perform what-if analysis. Similarly, if the workload is changing unpredictably and often, a tool that only periodically triggers auto-tuning cannot always capture such changes.

Another disadvantage of this process is that it relies on expensive actions that materialize complete indexes. This is not a design problem of auto-tuning tools; it is rather an inherent feature of how indexing is performed, i.e., fully generating a candidate index. With bigger data sets, though, index generation actions can take a significant amount of time (e.g., it can be in the order of several hours). By the time indexing is done, the workload might have already changed and any indexing action is now irrelevant. If anything, it can even be a bottleneck because now the system must maintain unneeded indexes during updates.

Finally, another fundamental bottleneck with the indexing process is the involvement of a human in the loop. Auto-tuning tools helped minimize this involvement and scale the process but the fact that a human database administrator is still needed means that the cost of ownership of a data system is high and it slows down the process.

Adaptive indexing, adaptive partitioning, and adaptive layouts came to address the above issues by promising data systems that can automatically generate the right indexes and layouts without a human in the loop and without the requirement to know the workload up front, quickly adapting to the workload changes with small incremental actions [22, 25, 24, 30, 37, 17, 31, 15, 27, 13, 20, 32, 23, 28, 14, 33, 35, 6, 34, 36, 38, 3, 21, 5, 4, 2, 29, 7]. The main innovation in these works comes from the fundamental feature of integrating physical data reorganization actions to query processing and triggering lightweight reorganization purely based on query requests.

In this paper, we briefly survey recent advances on adaptive storage work in modern data systems, highlight common principles, open problems and challenges. We term this line of work as shallow adaptivity in data systems, i.e., work that while it changes crucial knobs in the data system architecture it does not fundamentally affect the system's primary design. We then motivate the need for a new class of systems that can (automatically) perform changes at a deeper architectural level which results in system instances that would typically be considered different systems (not just differently tuned instances of the same system). We term this direction as deep adaptivity and we highlight some examples of capabilities that such deep adaptive systems should have.

There have been numerous other efforts in the general area of adaptivity. Most prominently work on adaptive query processing allows systems to change their query processing strategy by adapting to data and query properties [12, 18]. While this is one more extremely important direction with equally exciting past work and future opportunities (including possible synergy between all above directions), in this paper we focus solely on adaptive storage work that primarily changes the way data is stored and uses this as a way to modify the system behavior and properties.

2 Shallow Adaptivity

In this section, we briefly survey work on adaptive storage. We first highlight common principles and motivation. Towards the end, we discuss several open research challenges.

Principle 1: Lazy Tuning. The primary principle behind adaptive storage works is that critical tuning decisions that typically happen as part of the tuning and initialization phase of a data system, can happen during query processing time instead. The motivation for this step is twofold. First, it allows users and applications to minimize data to query time. That is, minimize the time taken for all steps required to set-up the system and ingest data from the moment that data becomes available to the moment one can issue a query. Typical steps in modern systems, include loading data into the system, choosing indexes and materialized views, and setting up several knobs. The second reason why this is a good idea is that it enables us to delay tuning decisions that may otherwise lock the system into a suboptimal state. Instead, if we take tuning decisions after we have seen the actual workload, tuning decisions can better match the desired state.

Principle 2: Continuous Layout Adaptation. Expanding from above, the second principle of adaptive storage works is that every request from the outside world (e.g, a read or write query), is treated as advice on how to store the data. This property means that the system can adapt its access patterns exactly to what the workload needs and as the workload evolves. In this way, even if the properties of the workload are not known up front, or even if they change over time, adaptive storage can capture such changes and react by physically reorganizing the data towards what would be the optimal for the running workload.

Principle 3: On-the-Fly Adaptation. Third, adaptation actions happen as part of query processing. This property is important to minimize the cost of adaptation. Adaptive storage relies on physical data reorganization which implies read and write actions to transform data between different formats. Performing the adaptation actions during query processing, means that adaptation algorithms can piggy back on I/O actions that would happen anyway for the active queries. Most typically, we may read data just once to answer a query and to perform adaptation on the relevant data. In all advanced adaptive storage works query processing and physical reorganization happen as part of the same algorithm and as part of the same pass over the data. This is key to achieve good performance.

Principle 4: Incremental Adaptation. Fourth, adaptation actions are lightweight actions that incrementally improve the state of the system. This is important to maintain low overhead for active queries that run in parallel with adaptation actions. This results in systems that are always in a transient state in terms of their storage layout. The exact state, and thus the exact data layout, at a given time depends on the types of queries the system has processed in the (recent) past. Small incremental actions have the added benefit of allowing to balance (read/write) trade-offs at a fine granularity.

Adaptive Storage. Work on adaptive storage for modern systems began with the idea of adaptive indexing in column-stores with dense and contiguous columns [22]. The idea is that instead of having to fully sort columns to create a good index, database cracking incrementally and continuously partitions columns using query predicates as pivots. This happens during query processing and effectively cracking builds a new select operator that handles both a read request and physical reorganization at the same time. As more queries are processed, columns are further partitioned; every read query needs to touch at most two pieces of a column and given that for the same column, pieces become smaller and smaller as we get more queries, performance quickly improves. Effectively the result is similar to what we would get from a fully sorted column and a binary search access path, i.e., the result of every select operator is a contiguous area in the partitioned column.

Later work on cracking pushed this idea deeper in the db kernel to be able to handle more complex queries and to mitigate any side-effects of physical reorganization. For example, work on sideways cracking showed how to process queries over more than one attributes (columns) by adaptively propagating physical reorganization across columns [24]. Work on cracking updates shows how to update such columns while maintaining the side-effects of past adaptive actions by lazy merging of updates [23]. Work on partial cracking demonstrates that the storage overhead of auxiliary indexes can be managed by both vertical and horizontal partitioning based on the workload [24]. Subsequently, work on transactional processing revealed that while indeed adaptive storage turns read queries into write queries (due to on-line adaptation), the fact that adaptive storage only affects data structure and not contents provides significant flexibility to mitigate transactional concerns [14, 13]. In addition, to work around the fact that certain query sequences may provide bad partitioning schedules for columns, stochastic cracking introduced the idea that queries should indeed be treated as advice on how to store the data but they should be complemented by random reorganization actions that help balance off any bad patterns [17]. Other algorithmic efforts have focused on understanding the trade-off of how much reorganization effort should go into every query and how this affects the pace of adaptation [25, 36]. Finally work on utilizing multi-core enables adaptation and full utilization of modern hardware by taking advantage of the inherent partitioned nature [31, 36, 6]. More recent proposals also utilize idle CPU cycles in multi-core environments to keep improving the state of the system in no peak workload periods [30].

Other than work on columnar systems, adaptive storage has also been studied for more traditional B-tree settings which triggered the work for balancing initialization costs with adaptation pace (more important when data needs to be persistent to disk) [16] as well as for more operation such as joins [28] and lightweight indexes [32].

Furthermore, work on distributed environments has utilized adaptive storage locally in each node to gain local improvements in the performance of each node [34] or even across nodes for complex operations by carefully selecting and adapting where data resides [29].

In addition, adaptive storage has been studied for the case of hybrid storage systems that provide a balance between row-stores and column-stores. Static solutions to this problem decide upfront which hybrid layout should be materialized based on the expected workload. This is exactly the same as the index tuning problem. Adaptive storage in this case can decide the right layout on-the-fly based on the access patterns required by the queries [5, 7]. A new design point here is that in order to utilize the different data layouts such as an adaptive system also needs to generate code that matches the right layout. This allows to minimize both cache misses and instruction misses and branches in the code [5].

Adaptive storage is not only about physically reorganizing existing data; it is also about how we ingest data to form the desired layout. Adaptive approaches in this direction choose to wait until queries arrive to decide which data to bring into the system and in what form [21, 2, 4]. This removes even more overhead from the initialization phase of a data system as now we can skip the data loading step completely. The additional design point here is that touching raw data repeatedly as more queries arrive can be expensive and so adaptive works in this area are primarily concerned with minimizing this cost, e.g., by creating indexes on top of raw files.

Other than the relational model, adaptive storage has also been studied in time-series scenarios for nearest neighbor queries [37, 38]. While the concepts remain the same there are some interesting differences with time-series. More importantly given that there is no strict global order across time-series there is no notion of performing small adaptation actions that refine an index. Instead, what the adaptive storage works do in this case is that they incrementally build the index by lazily indexing time-series as queries arrive. The adaptive time-series index builds initially a basic tree index but without populating it with any time series (using only high level representations) and then it actually inserts time-series in the index only when a relevant query arrives. This amortizes the cost of building the index across numerous queries in the same way as adaptive storage works do in the relational model.

Benchmarking. Benchmarking adaptive storage work requires a different approach than in traditional works. Here, the performance of the system evolves as we process more queries. Thus, each adaptive storage approach has to be evaluated over a window of time and expose properties about how the performance changes over time [15]. There are three main metrics: 1) How much slower is the very first query of adaptive storage compared to the default performance of a system (e.g., the standard one assuming no optimization has been applied via tunning)? 2) How fast the adaptive storage becomes faster than the standard approach (e.g., what is the crossover point)? 3) Does adaptive storage become as good as the optimal approach that we would get after tunning and how fast does this happen?

Open Topics. Overall there has been significant progress in adaptive storage both in terms of individual techniques and in terms of developing the concepts and principles. There are numerous open topics. Some of the most prominent open challenges include using machine learning to make tuning decisions when it is not easy to create a precise model that describes the effect of certain choices. A typical example in this direction is the choice of which column or sets of columns to optimize in a column-store. Another important open topic is studying the trade-offs of persistence vs. query response time and updates. This creates a triangle of choices that are mutually exclusive, i.e., any action taken to optimize one of them, hurts one or both of the other two. Finally, another important direction is merging adaptive indexing with traditional auto-tuning tools such that we can utilize both known workload knowledge if available and adapt to tight time budgets and workload changes. For example, while existing auto-tuning tools work only when there is enough workload knowledge and idle time to go through the tuning phases, an approach that merges both auto-tunning and adaptive indexing properties has the potential to be able to utilize 1) any workload knowledge even if it is not complete and 2) any idle time even if it is not enough to create all indexes we know we should create.

3 Deep Adaptivity

While the premise of the above work, termed as shallow adaptivity, is promising, it still has a fundamental limitation. It only works within a given architecture design. That is, it takes critical tuning steps that primarily have to do with data layout and devises adaptive/lazy approaches to adjust the layout to the workload. However, the options supported are within the design space of the original architecture. For example, in the case of generating the right indexes for a column-store system, adaptive storage decides which indexes to generate, when and how fine-grained the indexes should be. At all states, though, the fundamental architecture of the system does not change; it is still a column-store and behaves well only for the set of properties for which we can tune a column-store.

We term as deep adaptivity, a future class of data systems that would be able to cross architecture designs, i.e., designs that are typically considered fundamentally different systems today. A fundamentally different architecture is one that has significant design differences which typically stems from focusing to specific workloads, functionalities and hardware.

There are numerous reasons why such a path would lead to desirable systems. The primary one is similar to the need for adaptivity but at an even higher level. With ever more complex and diverse workloads today, industry and sciences need support from different kinds of systems which in turn increases the complexity of managing and tuning these systems as well as the cost of ownership. Having a single system that can change its behavior drastically would decrease costs and make access to data generally easier to manage instead of needing a new tailored system for every different scenario we need to support within a business. One of the most typical examples is the ability to support drastically different or even varying read/write ratios.

Perhaps even more importantly, a system that can change its shape drastically can better adapt to evolving hardware. As it stands today, new hardware appears faster than the pace at which software can follow. Having systems that are inherently flexible to adapt their design in more fundamental ways than the knobs exposed today would allow for easier adaptation.

As an example of possible deep design changes, consider changes that are fundamentally shifting the shape of underlying access methods in data systems. Given that access methods define the read/write performance of systems such a step does affect the overall performance. However, introducing new access methods or drastically changing existing ones in most systems that are older than a few years quickly becomes increasingly hard and is only worthwhile if we are close 100% sure about the final performance impact (which is a huge problem by itself). This restricts innovation and system adaptation to change. In this way, even the ability to manually adapt and test the design of systems in a low-overhead way would be a significant step even if it does not happen completely automatically.

Deep adaptivity requires several steps such as high level languages to express design (and change), methods to quickly test the impact of possible changes, allowing for semi-automatic human in the loop design, and most importantly fully mapping the possible design space. Similarly to shallow adaptivity, deep adaptivity needs to be evaluated against new metrics that capture the ability to change and adapt quickly.

4 Summary

This paper briefly surveys past work on adaptive storage and introduces the need to go beyond shallow adaptivity, to enable deeper architectural changes either automatically or semi-automatically. Having the ability to easily change the shape of a data system in a drastic way means that we can easily test possible alternative designs and quickly adapt to new application features or hardware, a process that typically may take several months or even years for mature systems with accumulated complexity.

References

1. D. J. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.
2. I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 241–252, 2012.
3. I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB in action: adaptive query processing on raw data. *Proceedings of the VLDB Endowment*, 5(12):1942–1945, 2012.
4. I. Alagiannis, R. Borovica-Gajic, M. Branco, S. Idreos, and A. Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. *Communications of the ACM*, 58(12):112–121, 2015.
5. I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: A Hands-free Adaptive Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1103–1114, 2014.
6. V. Alvarez, F. M. Schuhknecht, J. Dittrich, and S. Richter. Main memory adaptive indexing for multi-core systems. In *Tenth International Workshop on Data Management on New Hardware, DaMoN 2014, Snowbird, UT, USA, June 23, 2014*, pages 3:1–3:10, 2014.
7. J. Arulraj, A. Pavlo, and P. Menon. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2016.
8. M. Athanassoulis and S. Idreos. Design Tradeoffs of Data Access Methods. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Tutorial*, 2016.
9. N. Bruno and S. Chaudhuri. To tune or not to tune?: a lightweight physical design alerter. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 499–510, 2006.
10. N. Bruno and S. Chaudhuri. An Online Approach to Physical Design Tuning. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 826–835, 2007.
11. S. Chaudhuri and V. R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 146–155, 1997.
12. A. Deshpande, J. M. Hellerstein, and V. Raman. Adaptive query processing: why, how, when, what next. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 806–807, 2006.
13. G. Graefe, F. Halim, S. Idreos, H. Kuno, and S. Manegold. Concurrency control for adaptive indexing. *Proceedings of the VLDB Endowment*, 5(7):656–667, 2012.
14. G. Graefe, F. Halim, S. Idreos, H. A. Kuno, S. Manegold, and B. Seeger. Transactional support for adaptive indexing. *The VLDB Journal*, 23(2):303–328, 2014.
15. G. Graefe, S. Idreos, H. Kuno, and S. Manegold. Benchmarking adaptive indexing. In *Proceedings of the TPC Technology Conference on Performance Evaluation, Measurement and Characterization of Complex Systems (TPCTC)*, pages 169–184, 2010.
16. G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 371–381, 2010.

17. F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *Proceedings of the VLDB Endowment*, 5(6):502–513, 2012.
18. J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. A. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Eng. Bull.*, 23(2):7–18, 2000.
19. J. M. Hellerstein, M. Stonebraker, and J. R. Hamilton. Architecture of a Database System. *Foundations and Trends in Databases*, 1(2):141–259, 2007.
20. S. Idreos. *Database Cracking: Towards Auto-tuning Database Kernels.* PhD thesis, University of Amsterdam, 2010.
21. S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 57–68, 2011.
22. S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2007.
23. S. Idreos, M. L. Kersten, and S. Manegold. Updating a cracked database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 413–424, 2007.
24. S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 297–308, 2009.
25. S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *Proceedings of the VLDB Endowment*, 4(9):586–597, 2011.
26. S. Idreos, O. Papaemmanouil, and S. Chaudhuri. Overview of Data Exploration Techniques. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Tutorial*, pages 277–281, 2015.
27. P. Karras, A. Nikitin, M. Saad, R. Bhatt, D. Antyukhov, and S. Idreos. Adaptive Indexing over Encrypted Numeric Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 171–183, 2016.
28. Z. Liu and S. Idreos. Main Memory Adaptive Denormalization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2253–2254, 2016.
29. Y. Lu, A. Shanbhag, A. Jindal, and S. Madden. Adaptdb: Adaptive partitioning for distributed joins. *PVLDB*, 10(5):589–600, 2017.
30. E. Petraki, S. Idreos, and S. Manegold. Holistic Indexing in Main-memory Column-stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2015.
31. H. Pirk, E. Petraki, S. Idreos, S. Manegold, and M. L. Kersten. Database cracking: fancy scan, not poor man's sort! In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*, pages 1–8, 2014.
32. W. Qin and S. Idreos. Adaptive Data Skipping in Main-Memory Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2255–2256, 2016.
33. S. Richter, J.-A. Quiané-Ruiz, S. Schuh, and J. Dittrich. Towards zero-overhead static and adaptive indexing in Hadoop. *The VLDB Journal*, 23(3):469–494, 2013.
34. S. Schuh and J. Dittrich. AIR: adaptive index replacement in hadoop. In *31st IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2015, Seoul, South Korea, April 13-17, 2015*, pages 22–29, 2015.
35. F. M. Schuhknecht, A. Jindal, and J. Dittrich. The Uncracked Pieces in Database Cracking. *Proceedings of the VLDB Endowment*, 7(2):97–108, 2013.

36. F. M. Schuhknecht, A. Jindal, and J. Dittrich. An experimental evaluation and analysis of database cracking. *The Very Large Database Journal VLDBJ*, 25(1):27–52, 2016.
37. K. Zoumpatianos, S. Idreos, and T. Palpanas. Indexing for interactive exploration of big data series. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1555–1566, 2014.
38. K. Zoumpatianos, S. Idreos, and T. Palpanas. ADS: the adaptive data series index. *The Very Large Database Journal VLDBJ*, 25(6):843–866, 2016.