NoDB in Action: Adaptive Query Processing on Raw Data

Ioannis Alagiannis[⋆] Renata Borovica[⋆] Miguel Branco[⋆] Stratos Idreos[‡] Anastasia Ailamaki[⋆]

[⋆]EPFL, Switzerland
{ioannis.alagiannis, renata.borovica, miguel.branco, anastasia.ailamaki}@epfl.ch

[‡]CWI, Amsterdam
stratos.idreos@cwi.nl

ABSTRACT

As data collections become larger and larger, users are faced with increasing bottlenecks in their data analysis. More data means more time to prepare the data, to load the data into the database and to execute the desired queries. Many applications already avoid using traditional database systems, e.g., scientific data analysis and social networks, due to their complexity and the increased *data-to-query* time, i.e. the time between getting the data and retrieving its first useful results. For many applications data collections keep growing fast, even on a daily basis, and this *data deluge* will only increase in the future, where it is expected to have much more data than what we can move or store, let alone analyze.

In this demonstration, we will showcase a new philosophy for designing database systems called NoDB. NoDB aims at minimizing the data-to-query time, most prominently by removing the need to load data before launching queries. We will present our prototype implementation, PostgresRaw, built on top of PostgreSQL, which allows for efficient query execution over raw data files with zero initialization overhead. We will visually demonstrate how PostgresRaw incrementally and adaptively touches, parses, caches and indexes raw data files autonomously and exclusively as a side-effect of user queries.

1. INTRODUCTION

We are in the era of data deluge, where the amount of data outgrows the capabilities of query processing technology. Many emerging applications, from social networks to scientific experiments, are representative examples of this deluge, where the rate at which data is produced exceeds any past experience. Scientific analysis such as astronomy is soon expected to collect multiple Terabytes of data on a daily basis, while web-based businesses such as social networks or web log analysis are already confronted with a growing stream of large data inputs. Therefore, there is a clear need for efficient big data processing to enable the evolution of businesses and sciences to the new era of data deluge.

A growing part of the database community recognizes the need for significant and fundamental changes to database design, ranging from low-level architectural redesigns to changes in the way users interact with database systems [1, 4, 5, 7, 8, 9, 10].

The NoDB Philosophy. We recognize a new need, which is a direct consequence of the data deluge: the need to minimize or eliminate the data-to-query time. The data-to-query time is of critical importance as it defines the moment when a database system becomes usable, and thus useful. Modern database architectures, however, are based on fundamental principles that represent a major bottleneck for data-to-query time, most notably, the need to load data before submitting queries.

The NoDB design philosophy argues for creating new database kernel designs that minimize the data-to-query time, while also making database systems more friendly and accessible to end-users. This philosophy changes the way a user interacts with a database system, primarily by eliminating one of the most important bottlenecks, i.e., data loading. We advocate in situ querying as the principal way to manage data in a database and propose extending traditional query processing architectures to work in situ.

The overall NoDB vision was initially presented at CIDR 2011 [5], while the first feasibility study and experimental system, PostgresRaw, was presented at SIGMOD 2012 [2].

Contributions and Demo. We demonstrate PostgresRaw, a full NoDB system based on PostgreSQL. Our demonstration of PostgresRaw aims at a) introducing the NoDB philosophy through a system implementation and b) demonstrating the extent at which NoDB 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 members a complete visual insight into the behavior of PostgresRaw and the trade-offs that come with in situ query processing. In addition, we present a comparison between PostgresRaw and other widely-used DBMS in an interactive way with the audience by organizing a “friendly” race between the systems.

Innovation. PostgresRaw immediately starts processing queries without any data preparation or loading steps. As more queries are processed, response times improve due to the adaptive properties of PostgresRaw. We visually demonstrate these effects by observing internal components, such as the indexes and caches on raw data files, which allow PostgresRaw to adaptively and continuously improve its performance. Audience members will see how the indexing and caching structures of the system evolve as additional queries arrive, or when the workload changes.

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 the width of the attributes may significantly change the behavior of a NoDB system. Additionally, the graphical interface provides access to PostgresRaw specific execution configuration parameters. For instance, the user can enable

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The xth International Conference on Very Large Data Bases.
Proceedings of the VLDB Endowment, Vol. X, No. Y
Copyright 20xy VLDB Endowment 2150-8097/11/XX... $ 10.00.

or disable the NoDB components of PostgresRaw and 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

The NoDB philosophy draws inspiration from several decades of research on database technology and is related to a plethora of research topics. We briefly discuss related work in this section.

External Files. Querying directly raw files, i.e., without loading, has long been a feature of database systems. For instance, Oracle calls this feature external tables. External files, however, can only access raw data with no support for advanced database features such as DML operations, indexes or statistics. Therefore, external files require every query to access the entire raw data file, as if no other query did so in the past. In fact, this functionality is provided mainly to facilitate data loading tasks and not for regular querying. Instead, we propose to redesign the query processing layers of database systems to incrementally and adaptively query raw data files directly, while automatically creating and refining auxiliary structures to speed up future queries.

Physical Design. Work on auto tuning tools [3] for automating the physical design process and work on adaptive indexing [6] to incrementally refine indexes is highly relevant; both these directions aim at making the process of initializing a database system much easier. The first one by eliminating the need for hard workload analysis, and the second via introducing incremental and adaptive indexes. Still though the data needs to be loaded. NoDB goes a step further by studying the data-to-query time problem at its very root, i.e., before data is even loaded. As such, it is rather an orthogonal and complementary approach to auto-tuning tools and adaptive indexing.

3. POSTGRESRAW ARCHITECTURE

In this section, we discuss the design of our NoDB prototype, called PostgresRaw, implemented by modifying PostgreSQL 9.0. The main bottleneck of in situ query processing is the access to raw data. The design of PostgresRaw is geared towards improving access on raw data (a) by speeding up the steps required via raw data indexing and (b) by eliminating the need to access hot raw data via caching.

In the remaining of this section we assume that raw data is stored in comma-separated value (CSV) files. CSV files are challenging for an in situ engine, considering the high conversion cost and the fact that fields may be variable length. Nonetheless, being a common data source, they present an ideal use case for PostgresRaw.

Query plans in PostgresRaw. When a query submitted to PostgresRaw references relational tables that are not yet loaded, PostgresRaw needs to access the respective raw file(s). PostgresRaw overrides the *scan* operator with the ability to access raw data files directly, while the remaining query plan, generated by the optimizer, works without changes compared to a conventional DBMS.

Parsing and Tokenizing Raw Data. Every time a query needs to access raw data, PostgresRaw has to perform parsing and tokenization of the raw data. Having the binary values at hand, PostgresRaw feeds those values in a typical DBMS query plan.

Selective Tokenizing. PostgresRaw reduces the tokenizing costs by opportunistically aborting tokenizing tuples as soon as the required attributes for a query have been found. This occurs at a per tuple basis. Given that CSV files are organized in a row-by-row basis, selective tokenizing does not bring any I/O benefits; nonetheless, it significantly reduces the CPU processing costs.

Selective Parsing. In addition to selective tokenizing, PostgresRaw also employs selective parsing to further reduce raw file access costs. PostgresRaw needs only to transform to binary the values required for the remaining query plan.

Selective Tuple Formation. To fully capitalize on selective parsing and tokenizing, PostgresRaw also applies selective tuple formation. Therefore, tuples are not fully composed but only contain the attributes needed for a given query. In PostgresRaw, tuples are only created after the *select* operator, i.e. after knowing which tuples qualify.

3.1 Indexing

Adaptive Positional Map. The adaptive positional map further reduces parsing and tokenizing costs. It maintains low level metadata information on the structure of the flat file, which is used to navigate and retrieve raw data faster. This metadata information refers to positions of attributes in the raw file. For example, if a query needs an attribute X that is not loaded, then PostgresRaw can exploit this metadata information that describes the position of X in the raw file and jump directly to the correct position without having to perform expensive tokenizing steps to find X.

Map Population. The positional map is created on-the-fly during query processing, continuously adapting to queries. Initially, the positional map is empty. As queries arrive, PostgresRaw adaptively and continuously augments the positional map. The map is populated during the tokenizing phase, i.e., while tokenizing the raw file for the current query, PostgresRaw adds information to the map. PostgresRaw learns as much information as possible during each query. For instance, it does not keep maps only for the attributes requested in the query, but also for attributes tokenized along the way; e.g. if a query requires attributes in positions 10 and 15, all positions from 1 to 15 may be kept.

Exploiting the Positional Map. The information contained in the positional map can be used to jump to the exact position of the file or as close as possible. PostgresRaw opts to determine first all required positions instead of interleaving parsing with search and computation. Pre-fetching and pre-computing all relevant positional information allows a query to optimize its accesses on the map.

Adaptive Behavior. The positional map is an adaptive data structure that continuously indexes positions based on the most recent queries. This includes requested attributes as well as patterns, or combinations, in which those attributes are used. As the workload evolves, some attributes may no longer be relevant and are dropped by the LRU policy. Similarly, combinations of attributes used in the same query, which are also stored together in chunks, may be dropped to give space for storing new combinations. Populating the map with new combinations is decided during pre-fetching, depending on where the requested attributes are located on the current map. The distance that triggers indexing of a new attribute combination is a PostgresRaw parameter. In our prototype, the default setting is that if all requested attributes for a query belong in different chunks, then the new combination is indexed.

3.2 Caching

PostgresRaw also contains a cache that temporarily holds previously accessed data, e.g., a previously accessed attribute or even parts of an attribute. If the attribute is requested by future queries, PostgresRaw will read it directly from the cache.

The cache holds binary data and is populated on-the-fly during query processing. Once a disk block of the raw file has been parsed during a scan, PostgresRaw caches the binary data immediately. To

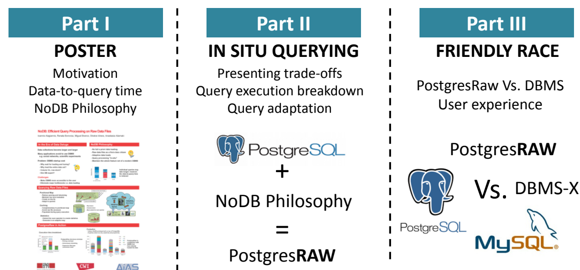

Figure 1: Demonstration walkthrough

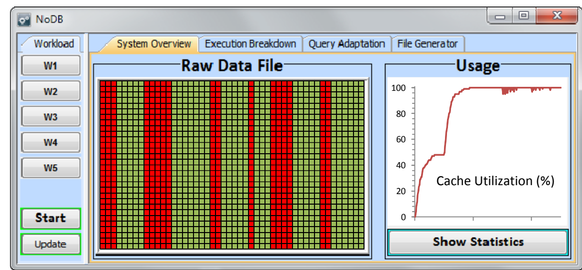

Figure 2: The System Monitoring Panel

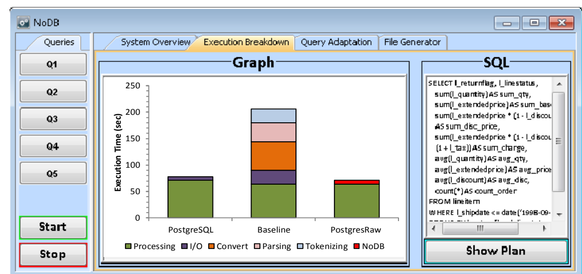

Figure 3: The Query Execution Breakdown Panel

minimize the parsing costs and to maintain the adaptive behavior of PostgresRaw, caching does not force additional data to be parsed, i.e., only the requested attributes for the current query are transformed to binary. The cache follows the format of the positional map such that it is easy to integrate it in the PostgresRaw query flow, allowing queries to seamlessly exploit both the cache and the positional map in the same query plan.

The size of the cache is a parameter than can be tuned depending on the resources. PostgresRaw follows the LRU policy to drop and populate the cache. Overall, the PostgresRaw cache can be seen as the place holder for adaptively loaded data.

3.3 Statistics

Optimizers rely on statistics to create good query plans. Most important plan choices depend on the selectivity estimation that helps ordering operators such as joins and selections. Creating statistics in modern databases, however, is only possible after data is loaded.

We extend the PostgresRaw *scan* operator to create statistics on-the-fly. We carefully invoke the native statistics routines of the DBMS, providing it with a sample of the data. Statistics are then stored and are exploited in the same way as in conventional DBMS. In order to minimize the overhead of creating statistics during query processing, PostgresRaw creates statistics only on requested attributes, i.e., only on attributes that PostgresRaw needs to read and which are required by at least the current query. As with other features in PostgresRaw, statistics are generated in an adaptive way; as queries request more attributes of a raw file, statistics are incrementally augmented to represent bigger subsets of the data.

4. DEMONSTRATION

The demonstration will be executed in three parts. The first part introduces the audience to the NoDB philosophy and the motivation behind minimizing the data-to-query time. The second part presents the NoDB philosophy in action, particularly its tradeoffs, giving a detailed insight into the system. The third part provides a direct comparison between our implementation, PostgresRaw, and other conventional DBMS. This third part is executed as a "friendly race" between systems, followed with a strong visual component and audience participation. Figure 1 summarizes the three parts.

4.1 Part I: Introduction to NoDB

In this part of the demonstration, we use a poster to introduce the audience to the NoDB philosophy and explain how in situ query processing can be used to minimize data-to-query time. Furthermore, we show the design of our NoDB prototype and we explain how PostgresRaw accesses the raw data files adaptively and incrementally without any previous data loading. Finally, we illustrate how the positional map and the flexible caching structure are used to enhance performance of future queries.

4.2 Part II: Describing NoDB systems

The second part of the demo illustrates the trade-offs with in situ query processing. For this, we show how the adaptive indexing mechanism maintains positional information over a variety of different datasets. Similarly, we show how the dynamic caches cope with different raw data and queries.

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 storage space occupied by the positional map and the caching structures and we visualize which parts of the raw data files are known to the positional map, caches or both. We allow the user to vary the available space for indexing and caching in order to examine the impact of these parameters on the performance. In addition, we provide usage statistics regarding the accessed attributes of the raw data file. Finally, the interface allows users to enable or disable some system components, e.g. the caches. An example screenshot of the user interface is shown in Figure 2.

The structure and the data type of the input raw data files are crucial for any database system, drastically affecting its performance. Therefore, our demonstration allows users to change the type of raw data files. For instance, tuples with fewer attributes or smaller attributes limit the effectiveness of the positional map. At the same time, caching should give priority to attributes that are more expensive to parse and cheaper to maintain in memory e.g. integer attributes. Therefore, we allow the user to directly generate their own input comma-separated value (CSV) files and choose parameters such as the number of attributes and the number of tuples in the file, the width of attributes, as well as the type of the input data.

Query Execution Breakdown. To highlight the difference between in situ and conventional database query execution, we monitor the query execution in PostgresRaw and PostgreSQL. Then, we present a time execution breakdown (shown in Figure 3). We show two variations of PostgresRaw. The first variation (PostgresRaw PM+C) combines positional map and caching while the second one

(Baseline) does not use any of the aforementioned techniques and constitutes the naive way of accessing external files. This component allows the user to examine how the positional map and the caching structure reduce in situ query processing overheads. Both systems share the same query execution engine. Therefore, the direct comparison between the two systems will help us understand the impact of in situ querying.

Query Adaptation. To demonstrate how PostgresRaw progressively adapts to changes in the workload, we vary queries such as to trigger changes both in the positional map and in the cache. In this scenario, we use simple Select-Project queries that are organized into epochs. The queries within each epoch refer to a specific part of the input data file, representing their exploratory behavior. As the workload evolves, new access patterns are observed, new combinations of attributes are indexed or cached and old information may no longer be relevant and will be evicted from our structures. We will show how PostgresRaw adapts to these changes during query execution and how the contents of the positional map and the cache evolve. We will visually illustrate this behavior using our graphical interface by properly shading the area of the input file that is queried in each epoch.

Updates. In this scenario, we allow the users to perform updates directly on the raw data files without using PostgresRaw. The user can either directly update one of the raw data files in an append-like scenario using a text editor or simply give a pointer to a new data file. In both cases, PostgresRaw is responsible for detecting the change in the input files and update the auxiliary NoDB data structures. The user will be immediately able to query the new or the updated file and observe the changes in the results of the next queries.

4.3 Part III: Friendly race between PostgresRaw and other DBMS

In this part of the demonstration, we compare the behavior of PostgresRaw against conventional DBMS using as a metric the data-to-query time and how it is reflected in user experience. We use MySQL, DBMS X (a commercial system) and PostgreSQL against PostgresRaw with positional maps and caching enabled. For the purpose of this part, we consider the interaction with the audience highly important. Thus, we propose a friendly race among the available database systems focusing on user experience.

Each of the contestants will be responsible for one of the DBMS. All DBMS execute the same sequence of input queries and take as input the same raw data files and the same schema. The data is not loaded in advance into any system. As a result, for conventional DBMS, the contestant will have to load the data before executing the queries.[1] The contestant is also free to tune the configuration parameters of the systems and/or build additional auxiliary data structure such as indices or materialized views. After the "starting shot", all contestants try to get the query results as soon as possible. Finally, each of the contestants will report the total execution time for the workload and the time spent to initialize each of the systems.

The experiment above will highlight a representative use case scenario and a major motivation for NoDB systems. PostgresRaw needs only a pointer to the raw data files and it starts executing queries immediately. On the other hand, the conventional DBMS have to go through a time consuming initialization phase (data loading and tuning). In the end, a few individual queries may take longer to respond in comparison with a traditional system; however, the data-to-query time is reduced and continuously improves over time. Therefore, PostgresRaw has already answered a number of queries while the traditional DBMS have not yet started processing the first query. The aforementioned behavior is particularly attractive for scenarios where the user wants to quickly examine new data in search of certain properties, or quickly skim through a few data attributes relevant to a given task. Our user interface will also provide an automatic demonstration of the above "race". In this case, we use pre-defined scenarios with reasonable choices for data loading and building indices before running the queries. In the end, we report the data-to-query time for each of the systems.

[1]MySQL and DBMS X offer "external files" functionality, which enables direct querying over raw files as if they were database tables. The users can choose to execute the queries using this feature as well.

5. CONCLUSIONS

Very large data processing is increasingly becoming a necessity for modern applications in businesses and in sciences. For state-of-the-art database systems, the incoming data deluge is a problem. With NoDB, we introduce a database design philosophy that turns the data deluge into a tremendous opportunity for database systems. It requires drastic changes to existing query processing technology but eliminates one of the most fundamental bottlenecks present in classical database systems for the past forty years, i.e., the data loading overhead. Until now, it has not been possible to exploit database technology until data is fully loaded. NoDB systems permanently remove this restriction by enabling in situ querying.

This demo showcases PostgresRaw, the first mature NoDB system. Through a graphical user interface the demo allows the user to see the adaptive behavior of PostgresRaw that results in efficient execution over raw data. The system monitors the state of its main components, i.e., indexing and caching, and shows how raw data is touched on demand as more and more queries arrive. 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.

6. REFERENCES

[1] A. Ailamaki, V. Kantere, and D. Dash. Managing scientific data. *Commun. ACM*, 53:68–78, 2010.

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

[3] S. Chaudhuri and V. Narasayya. Self-tuning database systems: a decade of progress. In *VLDB*, 2007.

[4] 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.*, 34:34–41, 2005.

[5] 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.

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

[7] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *SIGMOD*, 2007.

[8] 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. In *VLDB*, 2011.

[9] A. Nandi and H. V.Jagadish. Guided Interaction: Rethinking the Query-Result Paradigm. In *VLDB*, 2011.

[10] M. Stonebraker, J. Becla, D. J. DeWitt, K.-T. Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik. Requirements for Science Data Bases and SciDB. In *CIDR*, 2009.