NoDB: Efficient Query Execution on Raw Data Files[*]

Ioannis Alagiannis[†] Renata Borovica-Gajic[†] Miguel Branco[†] Stratos Idreos[‡] Anastasia Ailamaki[†]

[†]Ecole Polytechnique Fédérale de Lausanne
{ioannis.alagiannis, renata.borovica, miguel.branco, anastasia.ailamaki}@epfl.ch

[‡]Harvard University
stratos@seas.harvard.edu

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 and to load the data into the database before executing the desired queries. Many applications already avoid using database systems, e.g., scientific data analysis and social networks, due to the 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.

We here present the design and roadmap of a new paradigm in database systems, called NoDB, which *do not require data loading while still maintaining the whole feature set of a modern database system*. In particular, we show how to make raw data files a first-class citizen, fully integrated with the query engine. Through our design and lessons learned by implementing the NoDB philosophy over a modern DBMS, we discuss the fundamental limitations as well as the strong opportunities that such a research path brings. We identify performance bottlenecks specific for in situ processing, namely the repeated parsing and tokenizing overhead and the expensive data type conversion. To address these problems, we introduce an adaptive indexing mechanism that maintains positional information to provide efficient access to raw data files, together with a flexible caching structure. We conclude that NoDB systems are feasible to design and implement over modern DBMS, bringing an unprecedented positive effect in usability and performance.

1. INTRODUCTION

We are in the era of data deluge, where the amount of generated 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 disciplines such as astronomy are soon expected to collect multiple Terabytes of data on a daily basis. Similarly, 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.

Motivation. Although Database Management Systems (DBMS) remain overall the predominant data analysis technology, they are rarely used for emerging applications. This is largely due to the complexity involved; there is a significant initialization cost in loading data and preparing the database system for queries. For example, a scientist needs to quickly examine a few Terabytes of new data in search of certain properties. Even though only a few attributes might be relevant for the task, the entire data must first be loaded inside the database. Besides being a significant time investment, it is also important to consider the extra computing resources required for a full load and its side-effects with respect to energy consumption and economical sustainability.

Instead of using database systems, emerging applications rely on custom solutions that usually miss important database features. For instance, declarative queries, schema evolution and complete isolation from the internal representation of data are rarely present. There are a wide variety of competing approaches but users remain exposed to many low-level details and must work close to the physical level to obtain adequate performance and scalability. 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 the system [2, 5, 8, 9, 12, 14, 16, 17, 21].

The NoDB Philosophy. We recognize this new need, which is a direct consequence of the data deluge, and describe the roadmap towards NoDB, a new database design philosophy that we believe will come to define how future database systems are designed. The goal of the NoDB philosophy is to make database systems more accessible to the user by eliminating major bottlenecks of current state-of-the-art technology that increases 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. There are, however, fundamental processes in modern database architectures that represent a major bottleneck for data-to-query time. The NoDB philosophy changes the way a user interacts with a database system by eliminating one of the most important bottlenecks, i.e., data loading. We advocate querying over raw data, *in situ* (i.e., in its original place) as the principal way to manage data in a database and we propose to redesign the query processing layers of database systems to incrementally and adaptively query raw data files in situ, while automatically creating and refining auxiliary structures to speed up future queries.

Adaptive Data Loads. We originally introduced the idea of adaptive data loading in an earlier vision paper [9]. The current paper makes numerous and significant contributions, towards demon-

[*]The original version of this paper was published in Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (Scottsdale, Arizona, USA).

strating the feasibility and the potential of that vision. Using a mature and complete implementation over a modern DBMS, we identify and overcome fundamental limitations in NoDB systems. Most importantly, we show how to make raw files first-class citizens without sacrificing query performance. We also introduce several innovative techniques such as selective parsing, adaptive indexing structures that operate on the raw files, caching techniques and statistics collection over raw files. Overall, we describe how to exploit current relational databases to conform to the NoDB philosophy while identifying limitations and opportunities in the process.

Contributions. Our contributions are as follows.

- We convert a traditional relational database (PostgreSQL) into a NoDB system (PostgresRaw), and discover that the main bottlenecks are the repeated access and parsing of raw files. Therefore, we design an innovative adaptive indexing mechanism that makes the trip back to the raw files efficient.
- We demonstrate that the query response time of a NoDB system can be competitive with a traditional DBMS, even without prior data loading.
- We show that NoDB systems provide quick access to the data under a variety of workloads. PostgresRaw query performance improves adaptively as it processes additional queries and it quickly matches or outperforms traditional DBMS, including MySQL and PostgreSQL.
- We describe opportunities with the NoDB philosophy, as well as challenges such a research path brings.

2. QUERYING RAW DATA

In this section, we introduce the NoDB philosophy. For ease of presentation, we first discuss a straw-man approach to in situ querying, where every query relies exclusively on raw files for query processing. Then, we address the weaknesses of the straw-man approach by introducing the core concepts of NoDB that enable efficient access to raw data.

Typical Storage and Execution. A row-store DBMS organizes data in the form of tuples, stored sequentially one tuple after the other in the form of slotted pages. Each page contains a collection of tuples as well as additional metadata information to help in-page navigation. These pages are created during the loading process. Before being able to submit queries, the data must first be loaded, which transforms it from the raw format to the database page format. During query processing the system brings pages into memory and processes the tuples. In order to create proper query plans, i.e., to decide the operators and their order of execution, an optimizer is used, which exploits previously collected statistics about the data. A query plan can be seen as a tree where each node is a relational operator and each leaf corresponds to a data access method. The access methods define how the system accesses the tuples. Each tuple is then passed one-by-one through the operators of a query plan. The NoDB philosophy needs to be integrated with the aforementioned design for efficient and adaptive query execution.

2.1 Straightforward Approaches

We describe two straightforward ways to directly query raw data files. The first approach is to simply run the loading procedure whenever a relevant query arrives: when a query referring to table *R* arrives, only then load table *R*, and immediately evaluate the query over the loaded data. Data may be loaded into temporary tables that are discarded after processing the query, or it may be loaded into persistent tables stored on disk. These approaches however, significantly penalize the first query, since creating the complete table before evaluating the query implies that the same data needs to be accessed twice, once for loading and once for query evaluation.

A better approach is to tightly integrate the raw file accesses with the query execution. This is accomplished by enriching the leaf operators of the query plans, e.g., the *scan* operator, with the ability to access raw data files. Therefore, the *scan* operator tokenizes and parses a raw file on-the-fly, creates the tuples and passes them to the remaining of the query plan. The key difference is that data parsing and processing occur in a pipelined fashion, i.e., the raw file is read from disk in chunks and once a tuple or a group of tuples is produced, the *scan* immediately passes those tuples upstream.

Both straw-man techniques require that the proper schema be known a priori; the user needs to declare the schema and mark all tables as in situ tables. Other than that, both techniques represent a straightforward implementation of in situ query processing; they do not require significant new technology other than a careful integration of existing loading procedures with query processing.

Limitations of Straightforward Approaches. The approaches discussed above are similar to the external files functionality offered by modern database systems such as Oracle and MySQL. Such solutions are not viable for extensive and repeated query processing. For example, if data is not kept in persistent tables, then every future query needs to perform loading from scratch, which is a major overhead. Materializing loaded data into persistent tables however, forces a single query to incur all loading costs. Therefore, such approaches are only viable if a user needs to fire few queries.

Neither straw-man technique allows the implementation of important database systems functionality. In particular, given that data is not loaded, there is no mechanism to exploit indexing; modern database systems do not support indexes on raw data. Without index support, query plans for straw-man techniques rely only on full scans, incurring a significant performance degradation compared to a DBMS with loaded data and indexes. In addition, the optimizer cannot exploit any statistics, since statistics in a modern DBMS are created only after data is loaded. The lack of statistics and indexing means that straw-man techniques do not provide query processing performance comparable to a modern DBMS and any time gained by skipping data loading is lost after only a few queries.

Even though in situ features, such as external files, are important for the users, current implementations are far from the NoDB vision of providing an *instant gateway to the data*, without losing the performance advantages achieved by modern DBMS.

2.2 The NoDB Philosophy

The NoDB philosophy aims to provide in situ access with query processing performance that is competitive with a database system operating over previously loaded data. In other words, the vision is to completely shed the loading costs, while achieving or improving the query processing performance of a traditional DBMS. Such performance characteristics make the DBMS usable and flexible; a user may only think about the kind of queries to pose and not about setting up the system in advance and going through all the initialization steps that are necessary today.

The design we propose in this work takes significant steps in identifying and eliminating or greatly minimizing initialization and query processing costs that are unique for in situ systems. The target behavior is visualized in Figure 1. It illustrates an important aspect of the NoDB philosophy; even though individual queries may take longer to respond than in a traditional system, the data-to-query time is reduced, because there is no need to load and prepare data in advance or to fine tune the system when different queries arrive. In addition, performance improves gradually as a function

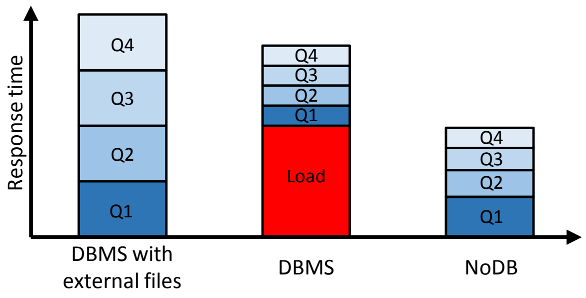

Figure 1: Improving user interaction with NoDB.

of the number of queries processed.

New Challenges of NoDB systems. The main bottleneck of in situ query processing is the access to raw data. The costs involved in raw data access significantly degrade query performance. In a traditional DBMS, parsing raw data files is more expensive than accessing database pages. The NoDB philosophy aims at making raw data a first-class citizen, integrating raw data access in an abstract way into the query processing layer, allowing query processing without a priori loading. However, a NoDB system can only be useful and attractive in practice if it achieves performance levels comparable to a modern DBMS. Therefore, the main challenge for a NoDB system is to minimize the cost of accessing raw data.

From a high level point of view, we distinguish between two directions; the first one aims at minimizing the cost of raw data access through the careful design of data structures that can speed-up such accesses; the second one aims at selectively eliminating the need for raw data access by careful caching and scheduling raw data accesses. The final grand challenge is to come up with a seamless design that integrates such features into a modern DBMS.

3. POSTGRESRAW: BUILDING NODB IN POSTGRESQL

In this section, we discuss the design of our NoDB prototype, called PostgresRaw, implemented by modifying the open-source DBMS PostgreSQL. We show how to minimize parsing and tokenizing costs within a row-store engine via selective and adaptive parsing actions. In addition, we present a novel raw file indexing structure that adaptively maintains positional information to speed-up future accesses on raw files. Finally, we present caching and exploitation of statistics in PostgresRaw. The ideas described in this section can be used as guidelines for turning modern row-stores into NoDB systems.

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

3.1 On-the-fly Parsing

We first discuss aspects related to on-the-fly raw file parsing and essential features such as selective parsing and tuple formation. We later describe the core PostgresRaw components.

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. In a typical CSV structure, each CSV file represents a relational table, each row in the CSV file represents a tuple of a table and each entry in a row represents an attribute value of the tuple. During parsing, PostgresRaw needs first to identify each tuple, or row in the raw file. Once all tuples have been identified, PostgresRaw must then search for the delimiter separating different values and transform those characters into their proper binary values. Overall, these extra parsing and tokenizing actions represent a significant overhead inherent to in situ query processing; a typical DBMS performs all these steps at loading time and directly reads binary database pages during query processing.

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 access costs. PostgresRaw transforms to binary only the values required to answer the query. For example, if a query requests the 4th and 8th attribute of a given file and the query contains a selection on the 4th attribute. PostgresRaw with selective parsing converts all values of the 4th attribute to binary but delays the binary transformation of the 8th attribute, until it knows that the given tuple qualifies.

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

Overall selective tokenizing, parsing and tuple formation help to significantly minimize the on-the-fly processing costs, since PostgresRaw parses only what is necessary to produce query answers.

3.2 Indexing

Even with selective tokenizing, parsing and tuple formation, the cost of accessing raw data is still significant. This section introduces an auxiliary structure that allows PostgresRaw to compete with a DBMS with previously loaded data. This auxiliary structure is a positional map, and forms a core component of PostgresRaw.

Adaptive Positional Map. We introduce the adaptive positional map to reduce 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.

Storage Format. The dynamic nature of the positional map

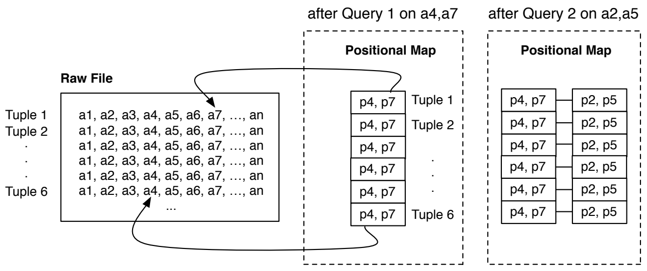

Figure 2: An example of indexing raw files with positional map.

requires a physical organization that is easy to update and incurs low cost during query execution. To achieve efficient reads and writes, the PostgresRaw positional map is implemented as a collection of chunks, partitioned vertically and horizontally. Each chunk fits comfortably in the CPU caches, allowing PostgresRaw to efficiently acquire all information regarding several attributes and tuples with a single access. The map can also be extended by adding more chunks either vertically (i.e., adding positional information about more tuples of already partially indexed attributes) or horizontally (i.e., adding positional information about currently non-indexed attributes). Figure 2 shows an example of a positional map, where the attributes do not necessarily appear in the map in the same order as in the raw file. The positional map does not mirror the raw file. Instead, it adapts to the workload, keeping in the same chunk attributes accessed together during query processing.

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. For example, if a query is looking for the 9th attribute of a file, while the map contains information for the 4th and the 8th attribute, PostgresRaw uses the positional map to jump to the 8th attribute and parse it until it finds the 9th attribute.

Maintenance. The positional map is an auxiliary structure and may be dropped fully or partly at any time without any lost of critical information; the next query simply starts re-building the map from scratch. PostgresRaw assigns a storage threshold for the size of the positional map such that the map fits comfortably in memory. Once the storage threshold is reached, PostgresRaw drops parts of the map to ensure it is always within the threshold limits.

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 a LRU policy. Similarly, combinations of attributes used in the same query, which are also stored together, 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.

3.3 Caching

The positional map allows for efficient access of raw files. An alternative and complementary direction is to avoid raw file access altogether. Therefore, 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. To 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.

3.4 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. 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.

On-the-fly creation of statistics brings a small overhead on the PostgresRaw *scan* operator, while allowing PostgresRaw to implement high-quality query execution plans.

4. EXPERIMENTAL EVALUATION

In this section, we present an experimental analysis of PostgresRaw. PostgresRaw is implemented on top of PostgreSQL 9.0, thus the direct comparison between the two systems is important to understand the impact of in situ querying. We have to point out that PostgresRaw is highly affected by any performance bottlenecks present in PostgreSQL, since they share the same query engine.

All experiments are conducted in a Sun X4140 server with 2 x Quad-Core AMD Opteron processor (64 bit), 2.7 GHz, 512 KB L1 cache, 2 MB L2 cache and 6 MB L3 cache, 32 GB RAM, 4 x 250 GB 10000 RPM SATA disks (RAID-0) and using Ubuntu 9.04.

The experiments presented in this section, use a raw data file of 11 GB, containing $7.5 * 10^6$ tuples. Each tuple contains 150 attributes with integers distributed randomly in the range $[0 - 10^9)$.

4.1 Positional Map

Impact. The first experiment investigates the impact of the positional map. In particular, we investigate how the behavior of PostgresRaw is affected as the map is populated dynamically with positional information based on the workload.

The set up of the experiment is as follows. We create a random set of queries accessing a subset of the attributes found in the raw file. We refer to queries as random, because they may ask for any attribute. Each query asks for 10 random attributes and retrieves all the rows of the file. We measure the average time PostgresRaw needs in order to process all queries with a varying storage capacity for the positional map, from 14.3 MB up to 2.1 GB.

The results are shown in Figure 3. The impact of the positional map is significant as it eventually improves response times by more than a factor of 2. In addition, performance improves rapidly, not requiring the maximum capacity. With little less than the $\frac{1}{4}$ of the pointers (260 million positions) collected, execution time is already only 15% from the full indexed case. After $\frac{3}{4}$ of the pointers are col-

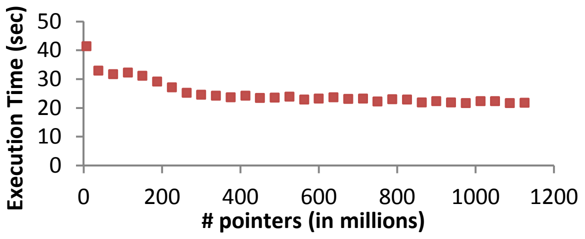

Figure 3: Effect of the number of pointers in the positional map.

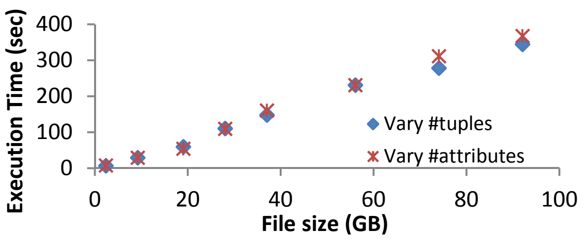

Figure 4: Scalability of the positional map.

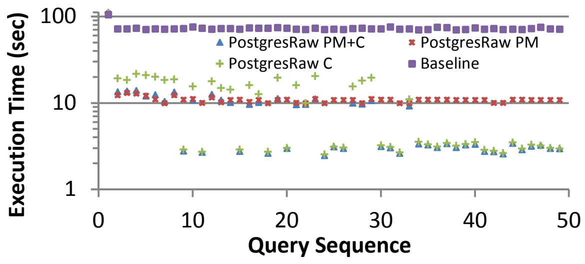

Figure 5: Effect of the positional map and caching.

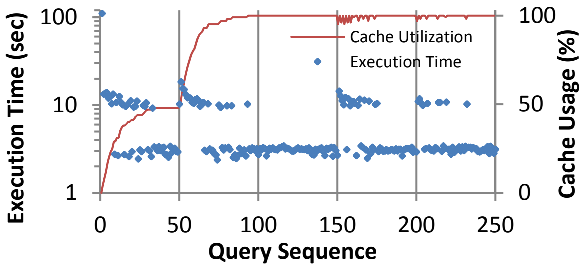

Figure 6: Adapting to changes in the workload.

lected, response time remains constant even though the workload is random. Therefore, PostgresRaw does not need to maintain positional information for the complete raw file, thereby saving significant storage and access costs, without compromising performance.

Scalability. The next experiment investigates the scalability of PostgresRaw when exploiting the positional map. The set up is the same as in the previous experiment with the difference that this time the file size is increased gradually from 2 GB to 92 GB. We use two ways to increase the file size; first, by adding more attributes to the file and second, by appending more rows to the file. In the first case, queries remain the same as before. In the second case, queries incrementally access more attributes as we increase the file size. We ensure that for every case we compare, queries perform similar I/O and computation actions. We allow unlimited storage space for the positional map. Nevertheless, we store only positions accessed by the most recent queries.

Figure 4 depicts the results. For both cases we observe linear scalability; PostgresRaw exploits the positional map to nicely scale as raw files grow both vertically and horizontally.

4.2 Positional Maps and Caching

This experiment investigates the behavior of PostgresRaw when exploiting both the positional map and caching or only one of them. We create 50 queries, where each query randomly accesses 5 columns and all the rows of the raw file. We study four variations. The first one, called Baseline, does not use positional maps or caching, representing the behavior of PostgresRaw as if it were a straw-man external files implementation. The second variation, called PostgresRaw PM, uses only the positional map while the third, called PostgresRaw C, uses only the cache and an additional minimal map with positional information for the end of lines. The final version, called PostgresRaw PM+C, combines all previous techniques.

Figure 5 plots the response time for each query. Since there is no a priori knowledge to exploit, all PostgresRaw variations need to touch the raw file to extract the needed data for the first query; thus, they all show similar performance. Performance improves drastically as of the second query. When the cache and the positional map are enabled the second query is 82-88% faster than the first. The Baseline variation improves slightly mainly due to file system caching and from there on it provides constant performance, which is not competitive with the other variations; every query needs to scan the raw file without any help from indexing and caching.

When only the positional map is used, the first few queries collect metadata information, improving future attribute retrieval by minimizing the parsing and tokenizing costs. The rest of the queries benefit from this information, demonstrating improved and stable performance. The positional map allows PostgresRaw to navigate as close as possible to the required attributes, which is important particularly when a small subset of the attributes is required. When only caching is used, there is a noticeable difference in performance. Caching achieves optimal performance only when all the requested attributes are cached. Nevertheless, if some attributes are missing, PostgresRaw needs to parse the raw file, which increases the overall execution time ($3-5$ times in this example). Figure 5 shows that the combined effects of the positional map and caching achieve the best performance; PostgresRaw PM+C outperforms all other approaches across the entire query sequence.

4.3 Adapting to Workload Changes

In this experiment, we demonstrate that PostgresRaw progressively and transparently adapts to changes in the workload. We use the same raw file as in the previous experiments but the query sequence is expanded to 250 queries. Each query again refers to 5 random attributes of the file. The query sequence is divided into 5 epochs and in each epoch we execute 50 different queries. All queries within the same epoch focus on a given part of the raw file. The maximum size of the cache is limited to 2.8 GB, while the positional map does not exceed 715 MB.

Figure 6 depicts the results, separating each epoch with vertical lines at positions $50, 100, \ldots, 200$. The graph plots both the response time for each query in the sequence and how the size of the PostgresRaw cache evolves as queries are evaluated.

During the first epoch, queries refer to columns 1-50. The cache and the positional map are initially empty. After executing 32 queries all data in this part of the file is cached; the cache does not increase and performance remains stable. In the second epoch, queries retrieve data between columns 51-100. Performance fluctuates as some queries can fully exploit the cache and have faster

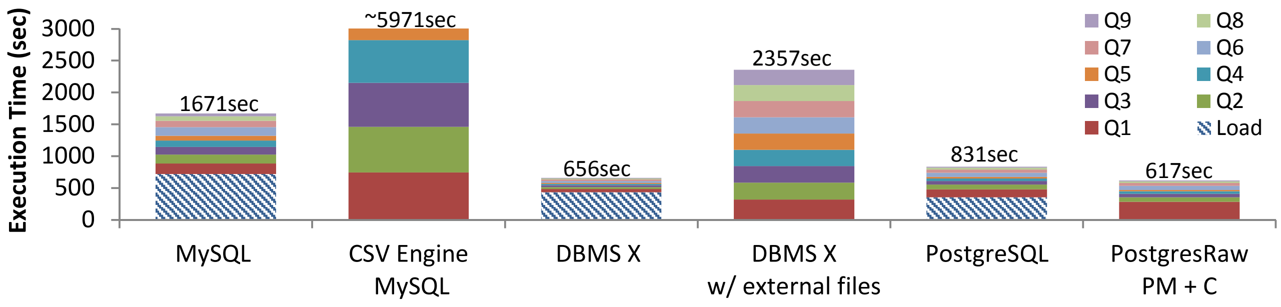

Figure 7: Comparing the performance of PostgresRaw with other DBMS.

response times while others need to go back to the file. After the second epoch, the cache is full and all queries enjoy good performance. During the third epoch, we launch a random set of queries requesting columns between 1-100, i.e., the same regions used in the previous epochs. Since PostgresRaw has built a complete cache of this region, no I/O or parsing is required. In the fourth epoch, queries ask for columns 75-125, i.e., half of the queries hit previously explored areas and half of the queries hit new regions. PostgresRaw uses a LRU replacement policy in its cache and drops previously cached data to accommodate the new requests. During the last epoch, the workload slightly shifts to the region of columns 85-135. PostgresRaw needs to replace parts of its cache while parts of the requested data are retrieved from the file by exploiting the positional map.

Overall, we observe that PostgresRaw gracefully adapts to the changes of the workload. In every epoch, PostgresRaw quickly adapts, adjusting and populating its cache and the positional maps, automatically stabilizing to good performance levels. Additionally, the maintenance of the cache and the positional map do not add significant overhead to query execution.

4.4 PostgresRaw vs other DBMS

In our next experiment we demonstrate the behavior of PostgresRaw against state-of-the-art DBMS. We compare MySQL (5.5.13), DBMS X (a commercial system) and PostgreSQL against PostgresRaw with positional maps and caching enabled. MySQL and DBMS X offer "external files" functionality, which enables direct querying over raw files. Therefore, for MySQL and DBMS X we include two sets of performance results; (a) using external files, and (b) using previously loaded data. For queries over loaded data we also report the time required to load the data; our goal is to show the overall data-to-query time.

We study the cumulative time needed to run a sequence of queries with each system. We use a sequence of 9 queries where we also vary selectivity and projectivity. All queries have one selection predicate and then project and run aggregations on the rest of the attributes. The first query requires all attributes and accesses all rows of the file. This is the worst case for PostgresRaw since we have to pay the whole cost of populating the positional map and the cache up front. The next 4 queries are the same with the difference that they access fewer rows at steps of 20% at a time. Then, the final 4 queries are again similar to the first query with the difference that they refer to fewer attributes at steps of 20% at a time.

Figure 7 shows the results. PostgresRaw achieves the best overall performance. It is competitive with DBMS X and MySQL for this sequence of queries. External files in MySQL (CSV Engine) and DBMS X are significantly slower than querying over loaded data or PostgresRaw, since each query repeatedly scans the entire file. Conventional wisdom indicates that the overhead inherent to in situ querying is problematic. This is indeed the case for straightforward in situ techniques such as external files. Nonetheless, these results show that the in situ overhead is not a bottleneck if we apply more advanced techniques that amortize the overhead across a sequence of queries, allowing for quick access to the data. Compared to PostgreSQL, PostgresRaw shows a significant advantage (25.75% in this case) even though it uses the same query engine. PostgreSQL is 53% slower than DBMS X if we consider the query execution time (without the loading costs). PostgresRaw, on the other hand, manages to be 6% faster than DBMS X even though it uses the same engine as PostgreSQL; by avoiding the loading costs, PostgresRaw has already answered the first 4 queries when DBMS X starts processing the first query.

Overall, PostgresRaw shows that it is feasible to amortize the overheads inherent to in situ querying over a sequence of queries, making an in situ system competitive with a conventional DBMS without requiring a priori data loading.

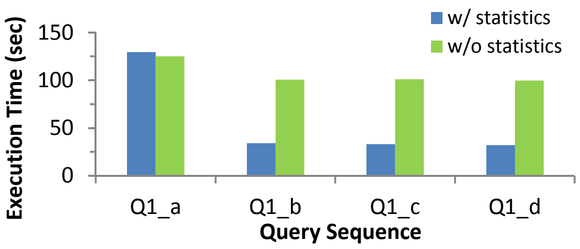

Figure 8: Execution time as PostgresRaw generates statistics.

4.5 Statistics in PostgresRaw

In our final experiment, we demonstrate the behavior of PostgresRaw when statistics are created on-the-fly during query processing. We use 4 instances of TPC-H decision support benchmark Query 1. We compare two versions of PostgresRaw. The first one generates statistics on-the-fly in an adaptive way, while the second one does not generate or exploit statistics at all.

Figure 8 shows the response times when running all 4 queries. The first query uses the same plan in both versions of PostgresRaw and initializes the positional map and the caching as well. Collecting statistics adds an additional overhead of 4.5 seconds in the execution time of the first query. PostgresRaw analyzes and creates statistics only for the attributes required for the current query. After the first query, the rest of the queries have different behavior even though they follow the same query template. In the PostgresRaw version with statistics support, queries run three times faster in comparison with the version without statistics. By examining the query plans, we notice that the optimizer selects a different set of operators and changes the ordering of operators in PostgresRaw with statistics which explains the improvement in performance. Generating the statistics on-the-fly adds only a small overhead, while it significantly improves query plan selection.

5. IN SITU QUERYING: TRADE-OFFS

In situ querying, although desirable in theory, is thought to be prohibitive in practice. Executing queries directly over raw data files incurs additional overhead to the execution, when compared to query execution over previously-loaded data. Nonetheless, our PostgresRaw implementation demonstrates that auxiliary structures reduce the time to access raw data files and amortize the overhead across a sequence of queries. In situ query execution, however, introduces a new set of trade-offs, which require further analysis:

Data Type Conversion. For ASCII files, PostgresRaw must convert the data into its proper type, e.g., from string to integer. Conventional DBMS perform this conversion only once at loading time. To alleviate the data type conversion overhead, PostgresRaw only converts the attributes in the tuple that are actually needed to answer a query. Nonetheless, data type conversion is not always an overhead: if a raw data file consists of variable-length strings, then PostgresRaw over CSV files is actually faster than a conventional DBMS because there is no need to convert data nor create secondary copies when loading data into a DBMS. Different data types, however, affect NoDB performance in different ways and should be taken into account when deciding which data to cache.

File Size vs. Database Size. Loading data into a DBMS creates a second copy of the data. This copy can be stored in an optimized manner: e.g., integers stored in a database page (in binary) likely take less space than in ASCII. Nonetheless, there are cases where a second copy does not imply less data. Variable-sized data stored in fixed-size fields usually takes more space in a database page rather than in its raw form. Therefore, depending on the workload, in situ engines can benefit from keeping data in its raw form.

Complex Database Schemas. DBMS support complex database schemas with large number of tables and columns within a table. Nonetheless, complex schemas usually require a DBA to tune vendor-specific configuration settings. For instance, a commercial DBMS we tested does not allow a row to be split across pages; if there are many columns within a table, or columns have large fields, the DBA must manually increase the page size, buffer pool and table space. These configurations are not straightforward. and are also subjected to additional limitations: e.g., pages must also have a minimum number of rows. In addition, larger tuples cause unpredictable behavior due to the use of slotted pages in the DBMS.

Types of Data Analysis. Current DBMS are best suited to manage data that is loaded only once or rarely in an incremental fashion, with well-known and rarely changing workloads. DBMS require physical design steps for best performance, such as creating indexes, which are time-consuming tasks. In situ databases, however, are more suited for users that need to explore data without having to load entire datasets. Users should be willing to pay a penalty during the early queries, as long as they do not need to create data loading scripts. In situ databases are also useful when there are large datasets but users need to frequently analyze small subsets of the data; such scenarios are increasingly common.

Integration with External Tools. DBMS are designed to be the main repository for the data, which makes the integration of DBMS data with external tools inherently hard. Techniques such as ODBC, stored procedures and user-defined functions aim to facilitate the interaction with data stored on the DBMS. Nonetheless, none of these techniques is fully satisfactory and in fact, this is a common complaint of scientific users, who have large repositories of legacy code that operates against raw data files. Migrating and reimplementing these tools in a DBMS would be difficult and likely require vendor-specific hooks. The NoDB philosophy significantly facilitates such data integration, since users may continue to rely on their legacy code in parallel to systems such as PostgresRaw.

Database Independence. DBMS store data in database pages using proprietary and vendor-specific formats. The DBMS has complete ownership over the data, which is a cause of concern for some users. The NoDB philosophy, however, achieves database independence, since the data files remain the main data repository.

6. OPPORTUNITIES

The NoDB philosophy drastically and fundamentally redefines the way database systems are designed. It requires revisiting well-established assumptions and implementation techniques, while also enabling new opportunities, which are discussed in this section.

Flexible Storage. NoDB systems do not require a priori loading, which implies no need for a priori decisions on how data is physically organized during loading. Data that is adaptively loaded can be cached in memory or written to disk in a format that enables faster access. Data compression can also be applied, where beneficial. Deciding the proper storage layout is an open research question. Rows, columns and hybrids all have comparative advantages and disadvantages. Nevertheless, a NoDB system benefits from avoiding to choose in advance. Physical layout decisions can be done online, and change overtime as the workload changes [3].

Adaptive Indexing. The NoDB philosophy brings new opportunities towards achieving fully autonomous database systems, i.e., systems that require zero initialization and administration. Recent efforts in database cracking and adaptive indexing [7, 10, 11, 13] demonstrate the potential for incrementally building and refining indexes without requiring an administrator to tune the system, or knowing the workload. Still, though, all data has to be loaded up front, forcing a delay in data-to-query time. We envision that adaptive indexing can be exploited and enhanced for NoDB systems.

Auto Tuning Tools. In this paper, we have considered the hard case of zero a priori idle time or workload knowledge. Traditional systems assume “infinite" idle time and knowledge to perform all necessary initialization steps. In many cases, though, the reality can be somewhere in between. For example, there might be some idle time but not enough to load all data. Auto tuning tools for NoDB systems, given a budget of idle time and workload knowledge, can exploit idle time to load and index as much of the relevant data. The rest of the data remains unloaded and unindexed until relevant queries arrive. A NoDB tuning tool should consider raw data access costs, I/O costs in addition to the typical query workload based parameters. The NoDB philosophy brings new opportunities in exploiting every single bit of idle time or workload knowledge.

Information Integration. Another major opportunity with the NoDB vision is the potential to query multiple different data sources and formats. NoDB systems can adopt format-specific plugins to handle different raw data file formats. Implementing these plugins in a reusable manner requires applying data integration techniques but may also require the development of new techniques, so that commonalities between formats are determined and reused. Additionally, supporting different file formats also requires the development of hybrid query processing techniques, or even adding support for multiple data models (e.g., for array data).

File System Interface. Another interesting opportunity that comes with NoDB is that of bridging the gap between file systems and databases. Unlike traditional database systems, data in NoDB systems is always stored in file systems, such as NTFS or ext4. This provides NoDB the opportunity to intercept file system calls and gradually create auxiliary data structures that speed up future queries.

7. RELATED WORK

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

Auto-tuning. The NoDB philosophy advocates for minimizing or eliminating the data-to-query time, which is also the goal of auto-tuning tools. Every major database vendor offers offline indexing features, where an auto tuning tool performs offline analysis to determine the proper physical design for a specific workload [1, 6, 18, 22]. More recently, these ideas have been extended to support online indexing [4, 20], hence removing the need to know the workload in advance. These techniques are a significant step forward, but still require all data to be loaded in advance.

Adaptive Indexing. Database cracking and adaptive indexing introduce the notion of incrementally refining the physical design by following and matching the workload patterns [7, 10, 11, 13]. This shares the adaptive goal of the NoDB philosophy, where each query is seen as an advice on how to refine indexes. Nonetheless, similarly to the previous case, existing adaptive indexing techniques also require all data to be loaded up front.

External Files. Most modern DBMS offer the ability to query data files directly with SQL, i.e., without loading. External files, however, can only access raw data with no support for database features such as DML operations, indexes or statistics and require every query to access the entire data file, as if no other query did so in the past. In practice, this functionality is used to facilitate data loading tasks and not for querying. NoDB systems, however, provide incremental data loading, on-the-fly index creation and caching to assist future queries and drastically improve performance.

Information Extraction. Information extraction techniques have been extended to provide direct access to raw text data [15], similarly to external files. The difference from external files is that raw data access relies on information extraction techniques instead of directly parsing raw data files. These efforts are motivated by the need to bridge multiple different data formats and make them accessible via SQL, usually by relying on wrappers [19].

8. 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. In this paper, 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 paper describes the NoDB philosophy, identifies problems, solutions and opportunities. It also describes the transformation of a modern row-store, PostgreSQL, into a NoDB prototype system, which we call PostgresRaw. Experiments on PostgresRaw demonstrate competitive performance with traditional DBMS. PostgresRaw, however, does not require any previous assumptions about which data to load, how to load it or which physical design steps to perform before querying the data. Instead, it accesses the raw data files adaptively and incrementally, allowing users to explore new data quickly and improving the usability of database systems.

The NoDB philosophy does not stop here however. We describe open issues and research challenges for the database community at large. We expect that addressing these new challenges will enable a new generation of database systems that serve the needs of modern applications and users.

9. REFERENCES

[1] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala. Database Tuning Advisor for Microsoft SQL Server 2005. In *VLDB*, 2004.

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

[3] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: A hands-free adaptive store. In *SIGMOD*, 2014.

[4] N. Bruno and S. Chaudhuri. To tune or not to tune?: a lightweight physical design alerter. In *VLDB*, 2006.

[5] J. Cohen, B. Dolan, M. Dunlap, J. Hellerstein, and C. Welton. MAD skills: new analysis practices for big data. *PVLDB*, 2:1481–1492, 2009.

[6] D. Dash, N. Polyzotis, and A. Ailamaki. CoPhy: a scalable, portable, and interactive index advisor for large workloads. *PVLDB*, 4:362–372, 2011.

[7] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, 2010.

[8] J. Gray, D. Liu, M. Nieto-Santisteban, A. Szalay, D. DeWitt, and G. Heber. Scientific data management in the coming decade. *SIGMOD Rec.*, 34:34–41, 2005.

[9] 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.

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

[11] S. Idreos, M. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *SIGMOD*, 2009.

[12] S. Idreos and E. Liarou. dbTouch: Analytics at your fingertips. In *CIDR*, 2013.

[13] 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*, 4:586–597, 2011.

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

[15] A. Jain, A. Doan, and L. Gravano. Optimizing SQL Queries over Text Databases. In *ICDE*, 2008.

[16] M. 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.

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

[18] S. Papadomanolakis and A. Ailamaki. AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. In *SSDBM*, 2004.

[19] M. T. Roth and P. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *VLDB*, 1997.

[20] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. COLT: continuous on-line tuning. In *SIGMOD*, 2006.

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

[22] D. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 Design Advisor: Integrated Automatic Physical Database. In *VLDB*, 2004.