NoDB: Efficient Query Execution on Raw Data Files

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, data loading evolves to a major bottleneck. 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. For such applications data collections keep growing fast, even on a daily basis, and we are already in the era of *data deluge* where we have much more data than what we can move, store, let alone analyze.

Our contribution in this paper is 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 costs. 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.

Our implementation over PostgreSQL, called PostgresRaw, is able to avoid the loading cost completely, while matching the query performance of plain PostgreSQL and even outperforming it in many cases. We conclude that NoDB systems are feasible to design and implement over modern database architectures, bringing an unprecedented positive effect in usability and performance.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems - Query Processing; H.2.8 [**Database Applications**]: Scientific Databases

General Terms

Algorithms, Design, Performance

Keywords

Adaptive loading, In situ querying, Positional map

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.
SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.
Copyright 2012 ACM 978-1-4503-1247-9/12/05 ...$10.00.

1. INTRODUCTION

We are now entering 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.

Motivation. Although Database Management Systems (DBMS) remain overall the predominant data analysis technology, they are rarely used for emerging applications such as scientific analysis and social networks. 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 few attributes might be relevant for the task, the entire data must first be loaded inside the database. For large amounts of data, this means a few hours of delay, even with parallel loading across multiple machines. 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. The problem with the situation today is in many ways similar to the past, before the first relational systems were introduced; 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.

The lessons learned in the past four decades indicate that in order to efficiently cope with the data deluge era in the long run, we will need to rely on the fundamental principles adopted by database management technology. That is, we will need to build extensible systems with declarative query processing and self-managing optimization techniques that will be tailored for the data deluge. A growing part of the database community recognizes this 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 [4, 14, 15, 20, 22, 24, 28].

The NoDB Philosophy. We recognize a 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 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 in situ querying as the principal way to manage data in a database and propose extending traditional query processing architectures to work in situ.

Querying directly raw files, i.e., without loading, has long been a feature of database systems. For instance, Oracle calls this feature external tables. Unfortunately, such features are hardly sufficient to satisfy the data deluge demands, since they repeatedly scan entire files for every query. 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.

Adaptive Data Loads. We presented the idea of adaptive data loads, as an alternative to full a priori loading in an earlier vision paper [15]. This paper makes numerous and significant contributions, towards demonstrating the feasibility and the potential of that vision. Using a mature and complete implementation over a modern row-store database system, we identify and overcome fundamental limitations in NoDB systems. Most importantly, we show how to make raw files first-class citizens without sacrificing query processing 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 row-stores to conform to the NoDB philosophy, identifying limitations and opportunities in the process.

Contributions. Our contributions are as follows.

- We convert a traditional row-store (PostgreSQL) into a NoDB system (PostgresRaw), and discover that the main bottleneck is the repeated access and parsing of raw files. Therefore, we design an innovative adaptive indexing mechanism that makes the trip back to the raw data files efficient.
- We demonstrate that the query response time of a NoDB system can be competitive with a traditional DBMS. We show that PostgresRaw provides equivalent or faster access on the TPC-H dataset (scale factor 10) compared to PostgreSQL, even without prior data loading.
- We show that NoDB systems provide quick access to the data under a variety of workloads (micro-benchmarks) as well as different file formats. 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, identifying fundamental overheads such as data type conversion.

The rest of the paper is organized as follows. Section 2 discusses related work and distinguishes it from the NoDB philosophy. Section 3 presents straw-man approaches to support in situ query processing and describes the key concepts of this philosophy. Section 4 presents our NoDB prototype, called PostgresRaw, emphasizing the key changes required to turn a modern row-store DBMS into a NoDB system. Section 5 presents a thorough experimental evaluation demonstrating the behavior and potential of NoDB systems. Section 6 presents the trade-offs that in situ querying brings, while Section 7 explains the research opportunities arising from such an approach. Finally, Section 8 concludes the paper.

2. 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. In this section, we discuss some topics such as auto tuning tools, adaptive indexing, information extraction and external files and their relation to the NoDB philosophy.

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 including sets of indexes, statistics and views to use for a specific workload [1, 2, 3, 5, 7, 9, 10, 25, 29, 30]. More recently, these ideas have been extended to support online indexing [6, 27], hence removing the need to know the workload in advance. The workload is discovered on-the-fly, with periodic reevaluations of the physical design. 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 [11, 12, 13, 16, 17, 18, 19]. 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 database systems offer the ability to query raw data files directly with SQL, i.e., without loading data, similarly to our approach. 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. 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 [21], 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 [26].

In situ Processing. Several researchers have recently identified the need to reduce data analysis time for very large data processing tasks [4, 8, 14, 15, 22, 23, 28]. For instance, Idreos et al. [15] present a vision towards such a system, based on adaptive data loads. The current paper, however, makes important advances to these goals by (a) presenting a complete system prototype, (b) introducing novel data structures to index raw files, hence making raw files first-class citizens in the DBMS, (c) tightly integrating adaptive loads, caching and indexing while implementing in situ access into a modern DBMS, (d) identifying fundamental limitations and opportunities, and (e) showing detailed benchmarks, specifically tailored to address unique performance charac-

teristics of an in situ system, as well as highlighting key features and proposing guidelines for future in situ designs.

3. QUERYING RAW DATA

In this section, we introduce the NoDB philosophy. We first adopt a straw-man approach to in situ query processing, 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. The design of PostgresRaw is presented in the next section.

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.

3.1 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 immediately 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 as well as binary data. 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. From an engineering point of view, this calls for an integration of the loading code with the scan code.

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. We maintain this assumption, as automated schema discovery is a well-studied problem orthogonal to the work presented here. 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 essentially 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 into persistent tables, then every future query needs to perform loading from scratch, which is a major overhead. This is the default setting and usage of external files. 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 a single or very reduced number of queries.

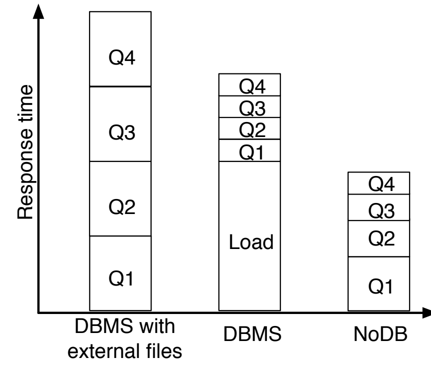


Figure 1: Improving user interaction with NoDB

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. Again, without statistics the query plans are poor, with suboptimal choices of operator order or algorithms to use. The lack of statistics and indexing means that straw-man techniques do not provide query processing 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.

3.2 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 the rest of this paper 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 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. Our experiments

demonstrate that the costs involved in raw data access significantly deteriorate 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.

4. POSTGRESRAW: BUILDING NODB IN POSTGRESQL

In this section, we discuss the design of our NoDB prototype, called PostgresRaw, implemented by modifying PostgreSQL 9.0. 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, exploitation of statistics in PostgresRaw and discuss updates. 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 are challenging for an in situ engine and a very common data source, presenting an ideal use case for PostgresRaw. Since data is stored in a character-based encoding such as ASCII, conversion is expensive and fields are variable length. Handling CSV files requires a wider combination of techniques than handling e.g. well-defined binary files, which could be similar to database pages.

4.1 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. This requires finding the end-of-line delimiter, which is determined by scanning each character, one by one, until the end-of-line delimiter is found. Once all tuples have been identified, PostgresRaw must then search for the delimiter separating different values in a row (which is usually a comma for CSV files). The final step is to transform those characters into their proper binary values depending on the respective attribute type. Having the binary values at hand, PostgresRaw feeds those values in a typical DBMS query plan. Overall, these extra parsing and tokenizing actions represent a significant overhead that is unique for 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. One way to reduce the tokenizing costs is to abort tokenizing tuples as soon as the required attributes for a query have been found. This occurs at a per tuple basis. For example, if a query needs the 4th and 8th attribute of a given table, PostgresRaw needs to only tokenize each tuple of the file up to the 8th attribute. 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. Consider again the example of a query requesting the 4th and 8th attribute of a given table. If the query contains a selection on the 4th attribute, PostgresRaw must convert all values of the 4th attribute to binary. However, PostgresRaw with selective parsing delays the binary transformation of the 8th attribute on a per tuple basis, until it knows that the given tuple qualifies. The last is important since the transformation to binary is a major cost component in PostgresRaw.

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 required for a given query. In PostgresRaw, tuples are only created after the *select* operator, i.e. after knowing which tuples qualify. This also requires carefully mapping of the current tuple format to the final expected tuple format.

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.

4.2 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 Requirements. In general, we expect variable-length attributes in raw format, i.e., the same attribute X appears in dif-

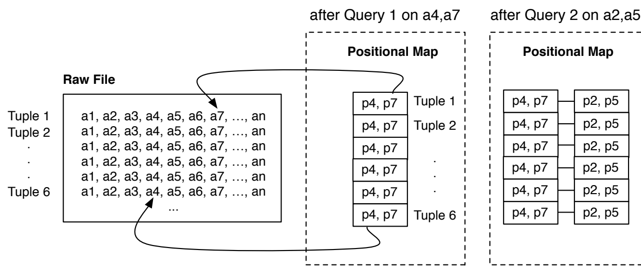


Figure 2: An example of indexing raw files

ferent positions in different tuples. The requirement to support variable-length attributes demands the positional map to store positions for every tuple in a table. To minimize the storage requirements, PostgresRaw uses run-length encoding to store its positions. Holding relative positions reduces storage requirements per position and is compatible with how row-store databases process data, i.e., one tuple at a time.

Storage Format. The dynamic nature of the positional map requires a physical organization that is easy to update. It must also incur low cost, to minimize its overhead 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.

Scheduling Map Accesses. In addition, PostgresRaw maintains a higher level data structure, a plain array, which contains the order of attributes in the map in respect to their order in the file. It is used by PostgresRaw to quickly determine the position of a given attribute in the positional map and the order in which multiple attributes are accessed. For example, if a query requests the 4th and the 8th attribute and both are indexed in the map but the 8th attribute is indexed before the 4th attribute, then for each tuple PostgresRaw retrieves first the position of the 8th attribute and then the position of the 4th attribute. Furthermore, in order to minimize the costs of retrieving information from the map PostgresRaw retrieves first all positions for the attributes needed in the where clause of a query. It will then access the map for positional information of select clause attributes, only for tuples that qualify from the where clause.

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 attribute *A* is the 4th attribute of the raw file and the map contains positional information for the 4th and the 5th attribute, then PostgresRaw does not need to tokenize the 4th attribute; it knows that, for each tuple, attribute *A* consists of the characters that appear between two positions contained in the map. Similarly, if a query is looking for the 9th attribute of a raw file, while the map contains information for the 4th and the 8th attribute, PostgresRaw can still use the positional map to jump to the 8th attribute and parse it until it finds the 9th attribute. This incremental parsing can occur in both directions, so that a query requesting the 10th attribute with a positional map containing the 2nd and the 12th attributes, jumps initially to the position of the 12th attribute and tokenizes backwards.

Pre-fetching. 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; it brings the benefit of temporal and spatial locality when reading the map while not disturbing the parsing and tokenizing phases with map accesses and positional computation. All pre-fetched and pre-computed positions are stored in a temporary map in the same structure as the positional map; the difference is that the temporary map contains only the positional information required by the current query and that all positional information has been pre-computed. The temporary map is dropped once the current query finishes its parsing and tokenizing phase.

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. We currently use an LRU policy to maintain the map.

Along with dropping parts of the positional map, our current implementation supports writing parts of the positional map from memory to disk. Positional information that is about to be evicted from the map can be stored on disk using its original storage format. Thus, we can still regulate the size of the positional map while maintaining useful positional information that is still relevant but not currently used by the queries. Accessing parts of the positional map from disk increases the I/O cost, yet it helps to avoid repeating parsing and tokenizing steps for workload patterns we have already examined.

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, 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.

4.3 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. Once a disk block of the raw file has been parsed during a scan, PostgresRaw caches the binary data immediately. 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.

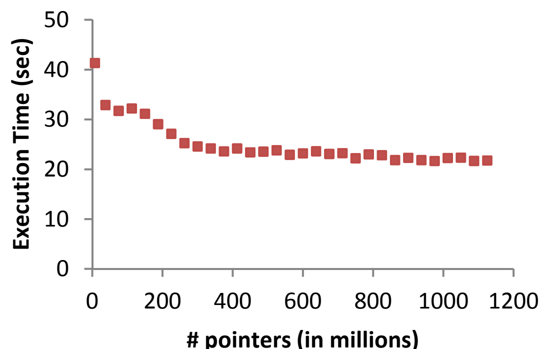


Figure 3: Effect of the number of pointers in the positional map

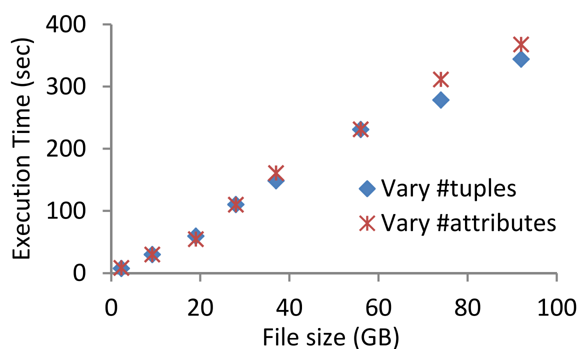


Figure 4: Scalability of the positional map

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. Nevertheless, NoDB systems should differentiate between string and other attribute types depending on the character-encoding scheme. For instance, for ASCII data, numerical attributes are significantly more expensive to convert to binary. Thus, the PostgresRaw cache always gives priority to attributes more costly to convert. Overall, the PostgresRaw cache can be seen as the place holder for adaptively loaded data.

4.4 Statistics

Optimizers rely on statistics to create good query plans. Most important choices have to do with 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.

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.

4.5 Updates

The design of PostgresRaw allows for two different kinds of updates: a) external and b) internal. Via external updates a user can directly update one of the raw data files without using NoDB (e.g. via a text editor) or simply add a new data file. On the other hand, internal updates can be triggered by SQL statements to the NoDB engine. As soon as raw data files are updated or added, their contents are immediately available for querying without further delays as there are no initialization costs in NoDB.

Adding new data files or externally updating a data file in an append-like scenario can be handled by updating the low-level information collected in the positional map and the data stored in the cache. When a new file is added, no auxiliary NoDB data structures have been created for this file in the past so no actions are required. On the other hand, coping with appends on existing files requires extending the positional map and the cache to include the new information the first time the respective file is queried.

Supporting in place updates is more complicated, especially in the case of the positional map. A change in a position of an attribute in the data file might call for significant reorganization. Nevertheless, being an auxiliary data structure, the positional map can be dropped and recreated when needed again.

5. EXPERIMENTAL EVALUATION

In this section, we present an experimental analysis of PostgresRaw. Since PostgresRaw is implemented on top of PostgreSQL, the direct comparison between the two systems is particularly 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 execution engine. We study the performance using both fine tuned micro-benchmarks and well-known benchmarks and workloads. PostgresRaw demonstrates a clear self-organizing behavior; by exploiting caching, indexing and on-the-fly statistics, it outperforms existing in situ query processing proposals while at the same time it provides comparable per query performance to that of traditional database systems where all loading costs need to be paid up front.

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.

5.1 Micro-benchmarks

In the first part of the experimental analysis, we study the behavior of PostgresRaw in isolation, i.e., we study the effect of the various design choices, the positional map and caching techniques. For this part, we use micro-benchmarks in order to perform the proper sensitivity analysis on parameters that affect performance. Later on, we demonstrate results on known benchmarks and workloads.

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 $[0 - 10^9)$.

5.1.1 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 simple select project queries. We refer to queries as random, because they may ask for any attribute of the raw file. Each query asks for 10 random attributes of the raw file. Selectivity is 100% as there is no WHERE clause. 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

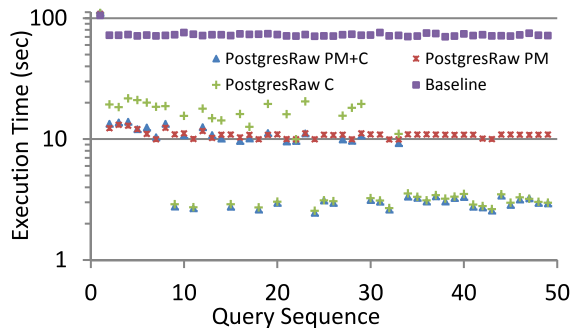


Figure 5: Effect of the positional map and caching

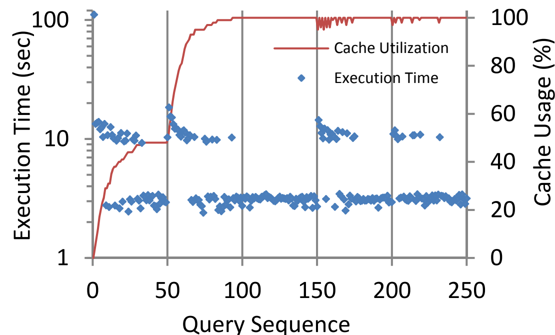


Figure 6: Adapting to changes in the workload

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 collected, 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, we incrementally add more projection attributes to queries 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 do not store positions for every tuple in the file but only for positions accessed by the most recent queries. In this experiment, the size of the positional map varies from 350 MB to 13.9 GB.

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.

5.1.2 Positional Maps and Caching

The following experiment investigates the behavior of PostgresRaw when exploiting both the positional map and caching or only one of them. The set up is as follows. We create 50 queries, where each query randomly projects 5 columns of the raw file. As in previous experiments, there is no WHERE clause; selectivity is 100%. We study four variations of PostgresRaw. The first variation, 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. The third variation, called PostgresRaw C, uses only the cache and an additional minimal map maintaining positional information only for the end of lines in the raw file. The final version, called PostgresRaw PM+C, combines all previous techniques. Again, in this experiment, we do not set a limit on the storage space for the positional map and the cache; however, their combined size always remains below 1.4 GB.

The response time for each query in the sequence is plotted in Figure 5. The first query is the most expensive for all PostgresRaw variations. Given that there is no a priori knowledge to exploit, all PostgresRaw variations, need to touch the raw file to extract the needed data; 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 as of the second query 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 assistance 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 only a small subset of the attributes are required in a tuple. When only caching is used, there is a noticeable difference in performance. Caching achieves optimal performance only when all the requested attributes happen to be cached. Nevertheless, if some attributes are missing from the cache, PostgresRaw needs to parse the raw file, which significantly 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 whole query sequence.

5.1.3 Adapting to Workload Changes

In this experiment, we demonstrate that PostgresRaw progressively and transparently adapts to changes in the workload. The set up of the experiment is as follows. We use the same raw file as in the previous experiments but the query sequence is expanded to 250 queries. As before, queries are select project queries. Each query refers to 5 random attributes of the file and there is no WHERE clause. 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 only to columns $1-50$. The cache is initially empty and so is the positional map. After executing 32 queries all data in this part of the file is cached; the cache does not increase any more and performance remains stable. In the second epoch, queries retrieve data between columns $51-100$. The size of the cache increases even more in order to index the new columns. Performance fluctuates as some queries can fully exploit the cache and have faster response times while others need to go back to the raw file so they pay the extra cost. After the second epoch, the cache is full and all queries enjoy good performance. During the third epoch, we launch random sets of queries requesting columns in the set $1-100$, i.e the same regions used in the pre-

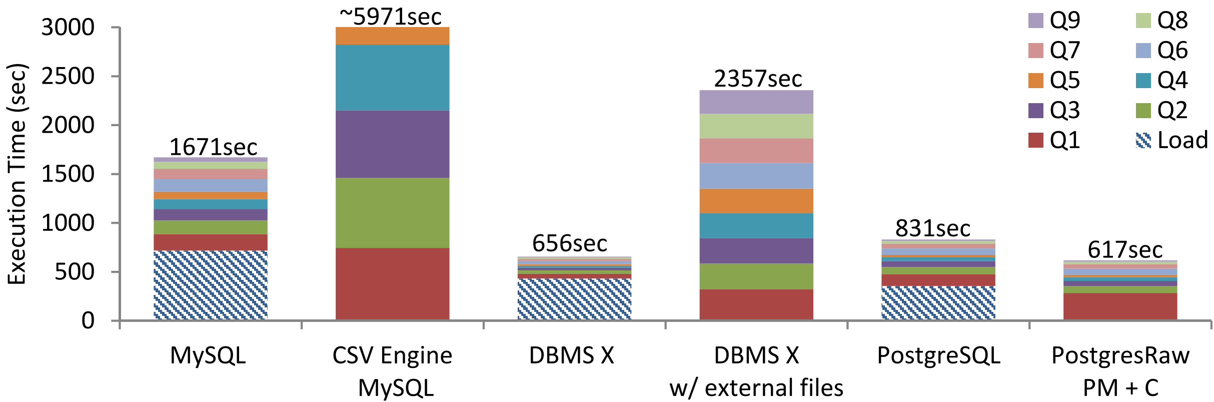


Figure 7: Comparing the performance of PostgresRaw with other DBMS

vious two epochs. Since PostgresRaw has built a complete cache of this region, no I/O or parsing is required and the system achieves optimal performance. 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 implements a LRU replacement policy in its cache and drops previously cached data to accommodate the new requests. During the last epoch, the workload again slightly shifts to the region of columns 85 – 135. The effect is that again PostgresRaw needs to replace parts of its cache while parts of the requested data have to be retrieved from the raw 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.

5.1.4 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 as if they were database tables. Therefore, for MySQL and DBMS X we include two sets of performance results; (a) using external files functionality, 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. For all systems, we boost the bulk loading procedure by enabling the file system cache (with asynchronous I/O).

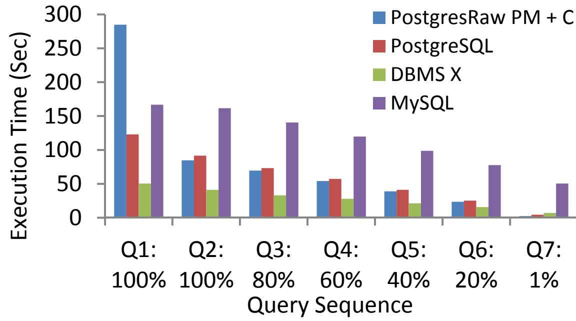
For the first experiment, we study the cumulative time needed to run a sequence of queries with each system. We use a sequence of 9 queries where we also vary selectivity and projectivity. All queries have one selection predicate in the WHERE clause and then project and run aggregations on the rest of the attributes. The first query has 100% selectivity and requires all attributes of the raw 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 vary selectivity, decreasing selectivity at steps of 20% at a time. Then, the final 4 queries are again similar to the first query with the difference that they decrease projectivity 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 take into account only 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.

In addition to demonstrating cumulative response times, in the following experiments we report on individual query response times as we vary the selectivity and projectivity. We do not include external files in this comparison as the respective response times are over an order of magnitude slower. For MySQL, DBMS X and PostgreSQL queries are submitted over previously loaded data but the loading time is not taken into account here; buffer caches are cold, however. As in our previous experiment, selectivity and projectivity is incrementally decreased during the query sequence.

Figure 8(a) shows the results as the selectivity decreases from 100% to 1% with projectivity constant at 100%. Similarly, Figure 8(b), depicts the performance with constant selectivity (100%) while projectivity decreases from 100% to 10%. The first query is similar in both graphs; selectivity is 100% and projectivity is 100%. This is the worst possible query for PostgresRaw; with an empty map and cache, it forces PostgresRaw to parse and tokenize the complete raw file. PostgresRaw is merely 2.3 times slower in the first query than PostgreSQL, but PostgresRaw actually outperforms PostgreSQL for the remaining queries even though it is performing in situ data accesses and sharing the same relational query execution engine. For all systems, as selectivity and projectivity decreases, performance improves as less computation effort is needed. Moreover, the improvement of PostgresRaw over PostgreSQL increases since we are bringing only the useful attribute values in the CPU caches. PostgresRaw improves even more as in addition to computation costs that have to do with the query components, e.g., aggregations, it can also decrease parsing and tokenizing costs via selective parsing and tokenizing actions.

Low selectivity and projectivity drastically reduce the query execution time in PostgresRaw, making it competitive with state-of-the-art DBMS without requiring data loading. The positional map allows us to read only the data required to answer queries, avoiding



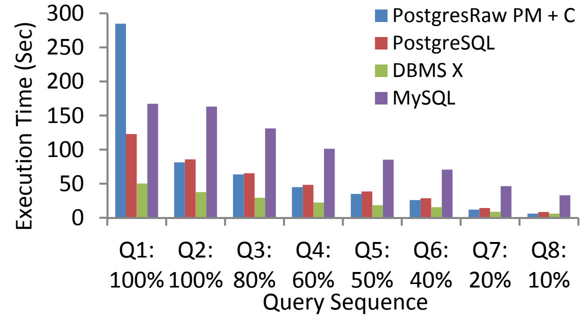


Figure 8: PostgresRaw performance compared to other DBMS as a function of (a) selectivity and (b) projectivity

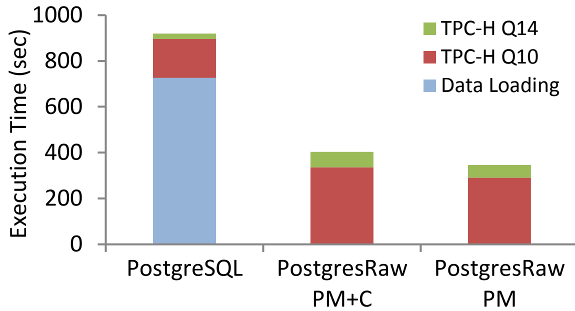


Figure 9: PostgreSQL Vs. PostgresRaw when running two TPC-H queries that access most tables

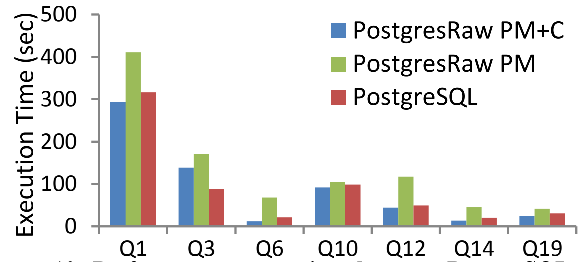


Figure 10: Performance comparison between PostgreSQL and PostgresRaw when running TPC-H queries

unnecessary processing. At the same time, caching further reduces the response time for commonly accessed data. 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.

5.2 TPC-H Workload

In the following experiment, we compare the behavior of PostgresRaw against PostgreSQL, using the TPC-H decision support benchmark scale factor 10. We use two variations of PostgresRaw. The first has the positional map enabled but caching disabled (PostgresRaw PM), while the second version has both positional maps and caching enabled (PostgresRaw PM+C). In this experiment, we allow unlimited storage space for the positional map and the cache. In TPC-H, tuples have few attributes and each attribute has a narrow width which narrows the effectiveness of the positional map.

Figure 9 shows the execution time for Queries 10 and 14 of TPC-H. Query 10 has a join over 4 tables (Customer, Orders, Lineitem, Nation), which requires reading data from four separate files, while Query 14 touches two tables (Orders and Lineitem). Tables Orders and Lineitem are the largest table in TPC-H. In all cases, the systems are cold. For PostgreSQL data must be loaded before queries can be submitted. PostgresRaw does not require any a priori loading, so queries can be submitted directly. Figure 9 shows that PostgresRaw is faster as long as positional maps are enabled (regardless of caching). The caching version is slower due to the overhead of creating and populating the cache.

Now that PostgreSQL and PostgresRaw are "warm", we submit a larger subset of TPC-H queries. Figure 10 shows the results. (The remaining queries were not implemented because their performance is either very poor in conventional PostgreSQL, or relied on functionality not yet fully implemented in the PostgresRaw prototype, such as views.) PostgresRaw with the positional map is always slower than PostgreSQL: 3*x* slower when running Q6 while approximately 25% slower in Q1.

When the cache is enabled, however, the PostgresRaw PM+C is faster than PostgreSQL in most of the queries even though PostgreSQL initially spent 802 seconds loading data. The goal of this experiment is not to conclude that PostgresRaw is faster than PostgreSQL but to show that an in situ query engine can be competitive with a conventional DBMS in important scenarios.

5.3 Alternative File Format: FITS

To demonstrate the applicability of PostgresRaw across multiple file formats, we also implemented the FITS (Flexible Image Transport System) file format. FITS is widely used in astronomy to store, transmit, manipulate and archive data. For instance, the Sloan Sky Digital Survey (SDSS) data is available in FITS.

FITS files can store multidimensional arrays (e.g. raw images acquired by telescopes) or tables (e.g. containing information about observed stars and galaxies). Tables can be stored in either ASCII or binary. A single FITS file can contain multiple images and tables. The file header, stored in ASCII, describes the contents of each payload including additional metadata information. A widely used tool to handle FITS files is the C library CFITSIO developed by NASA. It allows users to read data and apply filters to rows, columns or spatial regions.

The FITS-enabled PostgresRaw allows users to query FITS files containing binary tables directly, using regular SQL statements. We support FITS binary tables instead of ASCII tables because querying ASCII tables is very similar to CSV. Binary file formats however, are significantly different and pose different challenges. For instance, while parsing may not be required since each tuple and attribute is usually located in a well-known location, techniques such as caching become more important.

Unlike previous experiments, we cannot compare PostgresRaw directly with traditional databases, since they do not support loading of FITS files into tables. In fact, a major advantage of the PostgresRaw philosophy is that it allows database technology, such as declarative queries, to be executed over data sources that would otherwise not be supported. Therefore, we compare PostgresRaw with a custom-made C program that uses the CFITSIO library and procedurally implements the same workload as PostgresRaw.

Figure 11 illustrates the time breakdown for a set of queries over a single FITS file of 12 GB. The machine used is a dual CPU Intel Xeon X5660 (2.80GHz, 6 cores, 12MB cache) with 48 GB of

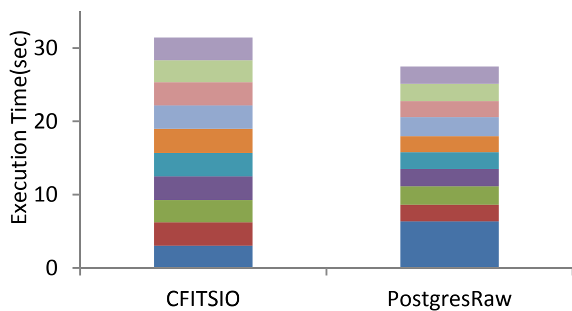


Figure 11: PostgresRaw in FITS files

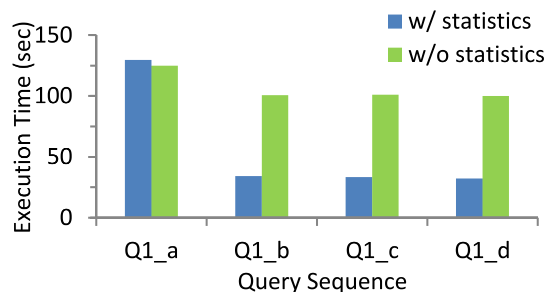


Figure 12: Execution time as PostgresRaw generates statistics

RAM. Each query executes a MAX, MIN or AVG over columns of float values in a table with approximately 4.3 million rows. The CFITSIO column includes the query execution times of the custom-made CFIOTS C program. In both cases, the file system caches are warm. If the caches were cold, the first query execution time would be approximately 16 seconds in both systems; the remaining queries would take the same time as in Figure 11.

The PostgresRaw implementation for FITS uses caching. Interestingly, the CFITSIO implementation also benefits from caching but only at the file system level. From the results in Figure 11, it is apparent that the data structures used in PostgresRaw to cache data are more appropriate.

There are three observations from these results: a) the CFITSIO approach leads to nearly constant query times since the entire file must be scanned for every query, while in PostgresRaw there is a performance gain after the first query (where caches are built), b) after as a few as 10 queries, the data-to-query time in PostgresRaw is lower than in CFITSIO, and c) it is much easier to submit a query to PostgresRaw, since only SQL statement is required, while each CFITSIO query requires a different, custom-made C program.

5.4 Statistics in PostgresRaw

In our final experiment, we demonstrate the behavior of PostgresRaw when statistics are created on-the-fly during query processing. The set up is as follows. We use 4 instances of TPC-H Query 1, as generated by the TPC-H query generator. 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 12 shows the response times when running all 4 queries. The first query uses the same plan in both versions of PostgresRaw and is used to initialize 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 have essentially 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.

6. TRADE-OFFS FOR IN SITU QUERYING PROCESSING

In situ querying, although desirable in theory, is thought to be prohibitive in practice. Executing queries directly over raw data files incurs significant additional overhead to the execution path, when compared to query execution over tables with 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. The result is an in situ system that is competitive with a DBMS under certain workloads, but without requiring data to be loaded in advance. In situ query execution introduces a new set of trade-offs, which require further analysis:

Data Type Conversion. In PostgresRaw, data is stored in the source data files. For ASCII files, the in situ engine 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. If the source data files are in binary, data conversion is also not an overhead, as demonstrated with the FITS file format where the PostgresRaw prototype is competitive with a custom-made C program. Caching can also be used where data conversion is expensive and caches are cheap to maintain. An example are CSV files (ASCII strings) containing integer attributes. Converting each value to a binary integer is expensive, but integers take little space in memory, making them good candidates for caching. Different data types, however, affect NoDB performance in different ways and should be taken into account when deciding the data to cache or load directly.

File Size vs. Database Size. Loading data into a DBMS creates a second copy of the data. This copy, however, can be stored in an optimized manner depending on the database schema: 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. For instance, 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, so if the schema is very small, parts of the page are unused. In addition, larger tuples cause unpredictable behavior due to the use of slotted pages in the DBMS. PostgresRaw, however, is robust to complex schemas, since it does

not rely on page structures that may overflow. This difference is illustrated in Figure 13, where the query response time of PostgreSQL is compared with PostgresRaw for a sequence of queries while varying the width of the attributes from 16 to 64. In the case of PostgreSQL, the query response time worsens significantly as the size of each attribute increases. The slowdown in PostgresSQL is at least 20 times and up to 70 times, while in PostgresRaw it is less significant: usually around 50% and at most 6 times.

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 or partitions, 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 an extra 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 only need to frequently analyze small subsets of the data; such scenarios are increasingly common. For instance, scientific users have new datasets available regularly.

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, if we take into account the proliferation of vendors offering data import/export tools. The NoDB philosophy, however, achieves database independence, since the raw data files remain as the main data repository.

7. 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 they also do not require 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 in the future. Data compression can also be applied, where beneficial. Deciding the proper memory and storage layout is an open research question. Rows, columns and hybrids all have comparative advantages and disadvantages. Nevertheless, a NoDB system benefits uniquely from avoiding to take these decisions in advance. In NoDB, physical layout decisions can be done online, and change overtime as the workload changes; every time data is brought from raw files, such a system can make dynamic decisions regarding the physical layout to use, based on the workload.

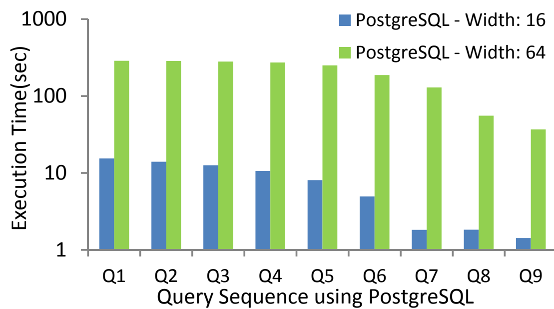
Adaptive Indexing. Furthermore, the NoDB philosophy brings new opportunities towards achieving fully autonomous database systems, i.e., systems that require zero initialization and administration. Recent efforts in database cracking and adaptive indexing [11, 12, 13, 16, 17, 18, 19] demonstrate the potential for incrementally building and refining indexes without requiring an administrator to tune the system, or knowing the workload in advance. Still, though, all data has to be loaded up front, breaking the adaptation properties and forcing a significant delay in data-to-query time. We envision that adaptive indexing can be exploited and enhanced for NoDB systems. A NoDB-like system with adaptive indexing can avoid both index creation and loading costs, while providing full-featured database functionality. The major challenge is the design of adaptive indexing techniques directly on raw files.

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, have the opportunity to exploit idle time as best as possible, loading and indexing as much of the relevant data as possible. The rest of the data remains unloaded and unindexed until relevant queries arrive. A NoDB tuning tool should take into account raw data access costs, I/O costs in addition to the typical query workload based parameters used by modern auto tuning tools. The NoDB philosophy brings new opportunities in exploiting every single bit of idle time or workload knowledge even if that is not enough for the complete initialization effort a traditional DBMS would do.

Updates. Traditionally, external files are considered as read-only sources with no support for inserts or updates. Immediate access on updated raw data files provides a major opportunity towards decreasing even further the data-to-query time and enabling tight interaction between the user and the database system. In order to optimize performance in such cases, a NoDB system needs to maintain and many times extend its auxiliary data structures including caches, indexes, and statistics so that any change in the data files be reflected immediately in any incoming NoDB queries. Despite being a technical challenge, the effort would bring substantial gains for users. In particular, it would allow seamless integration of existing data analysis tools together with NoDB systems.

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 very 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 NoDB queries. For instance, as soon as a user opens a CSV file in a text editor, NoDB can be notified through the file system layer and, in a background process, start tokenizing the parts of the text file currently being read by the user. Future NoDB queries can benefit from this information to further reduce the query response time. Obtaining this information is reasonably cheap since the data has already been read from disk by the user request and is in the file system buffer cache. A file system interface also allows a NoDB



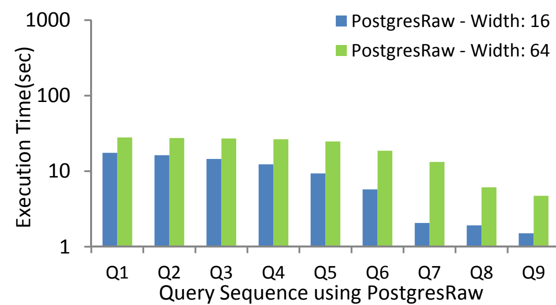


Figure 13: Varying attribute width in PostgreSQL vs PostgresRaw

system to maintain its caches up to date, even if the user changes the raw data files outside of NoDB.

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 in detail the transformation of a modern and widely used row-store, PostgreSQL, into a NoDB prototype system, which we call PostgresRaw. In depth experiments on PostgresRaw demonstrate competitive performance with traditional database systems, both on micro-benchmarks as well as on challenging, real-world workloads and benchmarks. 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 and only as required, allowing users to explore new data quickly and greatly improving the usability of database systems.

The NoDB philosophy does not stop here however. We also describe numerous 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] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*, 2000.

[3] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*, 2004.

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

[5] N. Bruno and S. Chaudhuri. Automatic physical database tuning: a relaxation-based approach. In *SIGMOD*, 2005.

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

[7] S. Chaudhuri and V. R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*, 1997.

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

[9] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic SQL tuning in Oracle 10g. In *VLDB*, 2004.

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

[11] G. Graefe, S. Idreos, H. Kuno, and S. Manegold. Benchmarking adaptive indexing. In *TPCTC*, 2011.

[12] G. Graefe and H. Kuno. Adaptive indexing for relational keys. *ICDEW*, 0:69–74, 2010.

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

[14] 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.

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

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

[17] S. Idreos, M. L. Kersten, and S. Manegold. Updating a cracked database. In *SIGMOD*, 2007.

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

[19] 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.

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

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

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

[23] K. Lorincz, K. Redwine, and J. Tov. Grep versus FlatSQL versus MySQL: Queries using UNIX tools vs. a DBMS, 2003.

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

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

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

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

[28] 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.

[29] G. Valentin, M. Zuliani, D. C. Zilio, G. Lohman, and A. Skelley. DB2 Advisor: An optimizer smart enough to recommend its own indexes. In *ICDE*, 2000.

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