An Adaptive Query Service for In Situ Querying

Manolis Karpathiotakis

DIAS, I&C, EPFL

***Abstract*—Database management systems have always made the assumption that their input would be loaded within the DBMS, and then queried upon. However, as the amount of data generated increases with rapid rates, this loading process acts as a bottleneck for data analysis, and is seldom leveraged. In addition, prior to being loaded, data need to be converted to a format compatible with the DBMS query engine. The heterogeneity in data formats therefore acts as an additional barrier. As a result, information residing in structured files is often left unexplored.**

In this paper, we examine the NoDB paradigm, which proposes querying data *in situ*. We describe the structures it introduces to overcome the limitations of having to access raw files repeatedly, its adaptive nature, and see how it is manifested in a system extending a traditional DBMS. To address shortcomings of query execution in NoDB, we examine a data-centric query execution model that deviates from the traditional Volcano iterator model. To materialize this model, query compilation techniques are employed, leading to the generation of machine code. Besides bringing performance benefits, these code generation mechanisms can be used in an alternative manner to create a query engine producing data-specific code at query time, thus supporting multiple data formats. In addition, affected by the sheer amounts of data that need to be handled, we present a cloud-based system named Dremel. Dremel natively supports queries over nested data, its entire design is aiming to optimize aggregation queries, and has been shown to scale to petabytes of data.

Finally, inspired by the previous approaches, we briefly present our vision for a lightweight database service which allows users to seamlessly pose interactive queries over raw files distributed over the cloud, regardless their format.

***Index Terms*—In situ querying, Query optimization, Code generation, Data analysis, Cloud computing**

Proposal submitted to committee: June 12th, 2013; Candidacy exam date: June 19th, 2013; Candidacy exam committee: Prof. Karl Aberer, Prof. Anastasia Ailamaki, Dr. Edouard Bugnion.

This research plan has been approved:

Date: _____

Doctoral candidate: _____
(M. Karpathiotakis) (name and signature)

Thesis director: _____
(A. Ailamaki) (name and signature)

Thesis co-director: _____
(if applicable) (name and signature)

Doct. prog. director: _____
(B. Falsafi) (signature)

EDIC-ru/05.05.2009

I. Introduction

THE continuously increasing amount of data produced daily opens up multiple possibilities for extracting useful information via its analysis. Experts from a wide and diverse range of fields, from domain scientists to data warehouse analysts, attempt to gain insights by going through newly acquired data and combining them with existing historical information. Utilizing multiple data sources in this process has a synergistic effect, exposing previously unknown patterns of interest.

However, although there has also been an increase in computing resources, the data produced and the effort placed to analyze it cannot be leveraged, as deriving new information from it is a time-consuming, complex process. The sheer amount of data produced is the basic bottleneck in this process, as even the simplest analysis requires loading everything in a DBMS. There are even scenarios in which very simple data exploration could have indicated that no useful information resides in a batch of data, so it should be ignored. As a result, it is typical for domain scientists to keep their data in structured files, resorting to hand-coded scripts for their exploration.

To make matters worse, when combining information from different data sources, there are no guarantees that the different datasets will follow the relational format or even be expressed in the same format. This heterogeneity results in additional pre-processing costs before being able to access the data in a uniform manner.

The primary reason for the presence of these time-consuming tasks is the requirement of database systems to always have complete control over the data. Therefore, the impedance mismatch between the raw data and the query engine has traditionally been resolved by converting raw data to an internal, database-specific representation and loading them in a database system. Then, queries would take place using a query engine which has been heavily optimized to offer high performance when operating over said internal format.

In this context, we argue that the mismatch between raw data and the query engine must be resolved at the moment of query execution, by employing a query engine that enables querying data in its original format and location. The query engine employed needs to take into account state-of-the-art execution strategies proposed, and also not be restricted to

only processing relational data. Finally, to be able to query the excessive amounts of available data, we consider cloud-based solutions to be the realistic option.

The rest of this paper is organized as follows: Section II introduces the NoDB philosophy, which proposes querying data *in situ* instead of loading them in a traditional DBMS. In Section III we examine a a data-centric query execution model transcending the ubiquitous iterator model used in the majority of DBMS. This model can be also be utilized by a NoDB system to optimize its performance. In Section IV we present Dremel, a cloud-based approach enabling the interactive analysis of datasets having a nested representation, thus examining how hierarchical data can be treated in large scale. Finally, in Section V we briefly present our research plan, outlining our approach for a lightweight, data- and query-adaptive database service over raw data.

II. The NoDB paradigm

The authors of [2] introduce the "NoDB philosophy". As in previous work of theirs [7], they recognize that for data analysis processes to be interactive, it is needed to greatly reduce the data-to-query time. As previously said, traditionally, before a data analysis task can begin, data need to be loaded in a DBMS, even if a significant part of them could prove irrelevant later on. NoDB aims to abolish this major bottleneck and facilitate data exploration by abolishing the data loading phase. Instead of having to load excessive amounts of data in a DBMS, users are to pose queries over data *in situ* instead.

NoDB treats "raw" data as a first-class citizen of a DBMS. The *scan* operators of a NoDB system are to be able to handle not only the binary data format that is understandable by the database, but also raw data. In addition, specialized data structures facilitate raw data access by providing indexing support over raw data. This deep integration of raw data support differentiates NoDB from industrial efforts such as the ones from Oracle and MySQL. These two systems offer support for querying files external to a database, but the approach followed by them leads to poor performance, as files are treated as an external entity, without indexing support, in-database caching, or statistics gathering. NoDB opts instead to treat "raw" data as a first-class citizen of the DBMS.

Such a system prototype has been implemented by the authors of this work. The system, nick-named PostgresRAW, has been built by appropriately overriding the scan operators of the PostgreSQL row store. It provides support for querying raw data stored in comma-separated value (CSV) files.

A. Reducing in situ access overheads

Although the absence of loading reduces the time it takes to start launching queries, it also introduces significant limitations that need to be taken into account. Every time a file is accessed, it needs to be scanned to determine the tuples present (i.e., find the end of line for each row), the tuple fields need to be tokenized based on the delimiter used, and the fields required need to be converted to their original data types.

In order to minimize these costs, various techniques have been implemented over PostgresRAW. A first step includes stopping tuple tokenizing as soon as the attributes necessary to answer a query have been found. To reduce parsing costs as well, PostgresRAW can stop converting attributes of a tuple as soon as this tuple does not satisfy some selection predicate. Furthermore, PostgresRAW performs selective tuple formation, creating tuples comprising only the attributes required by the query. This last optimization can be considered orthogonal to raw data processing, as any traditional DBMS would benefit from this strategy to "push down" the projections present in the query. All in all, by using these techniques, CPU processing costs are reduced, although I/O costs are not affected.

For indexing purposes, PostgresRAW utilizes an auxiliary structure that stores positional information of attributes, aiming to enable faster retrieval of raw data. Whenever an attribute is retrieved from a raw file, its position is inserted in this *positional map*. Subsequent queries requesting this attribute, can use this information to directly jump to the appropriate position, thus reducing parsing and tokenizing costs. Even requests for nearby attributes can be serviced by incrementally parsing from the known positions, so that scanning tuples from their beginning is avoided.

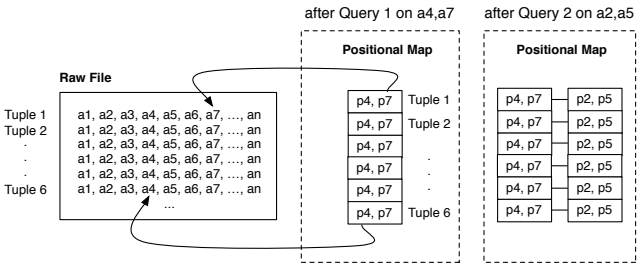


Fig. 1: Example of a positional map

An example of a positional map instance is depicted in Figure 1. Its structure and contents fully adapt to the query workload. It consists of multiple chunks, which are populated incrementally through subsequent accesses to the raw file. "Hot" areas of the file will therefore be indexed in a more fine-grained manner, while areas of no interest will not have their positional metadata in the map. If permitted by the amount of space available, information is also kept about the fields that had to be parsed until the required one has been located. The actual contents within single chunks are also affected by the workload, as attributes accessed together by queries are contained in the same chunk, to facilitate retrieval in the case of frequent combinations.

The adaptive behavior of PostgresRAW is also reinforced by the presence of data caches. The aim of these caches are to further reduce conversion costs. Therefore, the general policy is populating them with values that are expensive to convert (e.g., numerics). Finally, PostgresRAW adaptively generates statistics on the attributes of the file that are touched by queries. The more "popular" some fields are, the more statistics will be gathered for them.

B. Performance of a NoDB system

For a NoDB system to be a viable alternative for users, it needs to offer performance comparable to that of a traditional

DBMS. To this end, and to demonstrate the impact of the techniques proposed, PostgresRAW is evaluated against a number of existing solutions, from systems offering support for external files to systems that traditionally load data and then launch queries over them. Experiments are performed using CSV input consisting of 7.5×10^6 tuples, with each tuple containing 150 integer fields. A subset of the results is depicted in Figure 2. As previously suggested, approaches naively re-visiting raw files in every access cannot be considered a competitive option. However, PostgresRAW even manages to be have performance competitive to state-of-the-art DBMS while allowing rapid access to data. The extra cost of having to access raw data is amortized among the query sequence, and is reduced as the structures used are adaptively refined.

An additional experiment providing useful insights is depicted in Figure 3. In this scenario, PostgresRAW is compared with systems executing queries over pre-loaded data, all starting with a cold cache. The projectivity of the query is reduced between runs, while keeping selectivity at 100%. After the initial query, which is a worst-case scenario for NoDB systems (as the entire file needs to be scanned), PostgresRAW is competitive with the other systems. Actually, as projectivity is reduced, CPU processing in the case of PostgresRAW is reduced more rapidly than in the case of PostgreSQL. This is due to the fact that only necessary attributes are brought in the CPU caches, as mentioned in Section II-A.

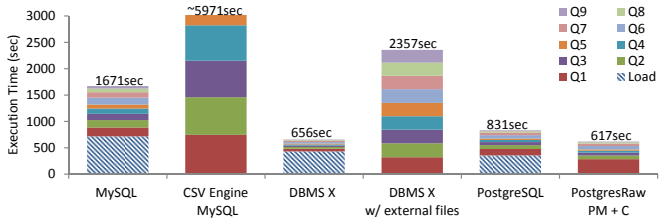


Fig. 2: Comparison of PostgresRAW with other DBMS

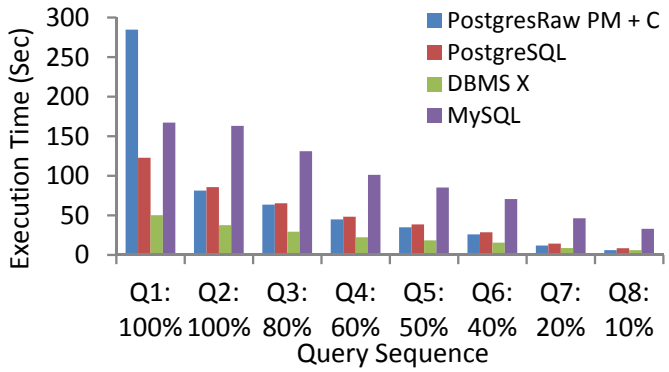


Fig. 3: Comparison of PostgresRAW with other DBMS as a function of projectivity

C. Future of NoDB

The prospects opening when considering the design of a database system that does not require data loading are numerous. As already indicated by the gains from selective tuple formation of PostgresRAW, a NoDB system could even dynamically decide which layout is appropriate for data brought in memory, and opt for a row, columnar or hybrid format depending on the needs of the current query [16]. This flexibility of a NoDB system would be further enhanced if coupled with adaptive indexing mechanisms such as database cracking [8], which has been used in the context of column stores to incrementally create indexes as a side-effect of queries. Finally, as it is typical for datasets produced to be expressed in a variety of formats, a NoDB system could operate over different file types, thus facilitating on-the-fly integration of heterogeneous datasets without having to first convert all data to a binary format understandable by the query engine. The next works we will present aim to provide some insight on how these ideas can be materialized.

D. Summary

Summing up so far, PostgresRAW demonstrated that by utilizing auxiliary data structures and performing user-driven tuning, it is feasible to offer interactive responses without having to incur the initial loading costs. The parsing, tokenization and conversion effort is amortized among multiple queries, thus not penalizing a single one.

III. Compilation of queries

As PostgresRAW has been built on top of PostgreSQL, it inherits its model of query execution, namely the classical Volcano iterator model [6]. Despite its simplicity and generality, this tuple-at-a-time execution model causes poor performance due to the interpretation overhead and the lack of locality it introduces. To handle these shortcomings, the authors of [12] propose utilizing query compilation, thus translating queries to low-level machine code in order to maximize performance. The technique to be presented has been put to use in the HyPer main-memory database system [9].

A. Query execution strategies

When a query is posed in a database system, it is generally processed by a query planner / optimizer, resulting in an algebraic plan. This plan, expressed in the form of a tree, is traditionally interpreted using the Volcano iterator model [6]. Every operator of the plan exposes a general API, consisting of *open()*, *next()* and *close()* function calls. Whenever an operator's *next()* method is called, a request for a new tuple is sent to the operator's children operators. While being a simple and intuitive interface, its very generality actually penalizes performance. The fact that the *next()* function will be called for every tuple leads to increased costs, considering that function calls will take place even for very simple operations. In addition, as these function calls are typically virtual, they lead to frequent branch mispredictions. Finally, the constant changes in control flow lead to poor code locality.

To address these performance concerns, approaches have appeared suggesting block-oriented query processing, i.e. generating more that one tuple with every *next()* call of an operator ([3], [4], [13]). Therefore, the costs resulting from

the multiple calls of functions are significantly reduced. In addition, this type of processing also allows exploitation of modern hardware, such as vectorized execution by using SIMD instructions. On the other hand, by processing blocks, the output of operators needs to be materialized before being provided as input to a subsequent operator. Thus, the pipelining capabilities of the iterator model are no longer available, leading to increased memory bandwidth consumption.

In an attempt to keep the best of both worlds, namely both pipelining capabilities and better utilization of modern hardware without sustaining interpretation overheads, the authors of [12] propose a novel execution model, based on the following guidelines:

- The query execution needs to be data centric instead of operator centric. Execution revolves around data that are kept in the CPU registers as long as possible, even if it means deviating from the operator model introduced.
- Instead of the pull-based nature of the Volcano model, push-based execution is used, thus resulting in better code and data locality.
- Queries are compiled to low-level machine code which is optimized to fully exploit modern hardware.

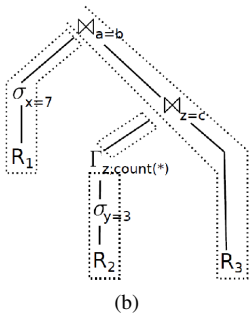
B. Compiled, push-based execution

As mentioned above, the query processing architecture of [12] aims to keep data values in CPU registers as long as possible. To this end, tuples are pushed from the lower operators of the tree to the upper ones. During this process, tuples are materialized only when operators acting as *pipeline breakers* (i.e., they cause values to be evicted from registers and "spilled to memory") are met.

An example of this execution plan for the query of Figure 4a is depicted in Figure 4b. In this example, tuples are materialized only when a group by operator is met, or a hashtable needs to be built for the execution of a join operator. All other operators are applied to the data without moving them out of the CPU registers.

```
SELECT *
FROM R1,R3,
     (SELECT R2.z,COUNT(*)
      FROM R2
      WHERE R2.y=3
      GROUP BY R2.z) R2
WHERE R1.x=7 AND
      R1.a=R3.b AND
      R2.z=R3.c
```

(a)



(b)

Fig. 4: (a) Example Query (b) Execution Plan

This type of execution leads to increased data locality compared to the iterator model, as the pipelining operators are performed over data residing in CPU registers. In addition, as the control flow using this model is simplified, there are also benefits from better code locality.

As far as the control flow is concerned, one can notice that it is no longer operator-centric, but data-centric, as effort is made to have all operations of the query plan applied over data in place (i.e., residing in registers). Regarding the actual execution of the algebraic plan, it is translated into imperative code that is compiled and executed.

Although the operators of the plan conceptually offer an interface native in push-based execution (a *produce()* and *consume()* function), the actual code that is generated does not respect these operator boundaries. This means that parts of an operator's functionality may end up being executed at different times, based on which parts of the operator's logic is included in a single pipelining step. For example, in the case of Figure 4b, the scanning of R_1, the subsequent selection and the building of the hashtable for the execution of the $\bowtie_{a=b}$ will take place in a single step, while the probing of the hashtable will occur in a pipelined fashion as the last step of the query.

C. Code generation

The code generation method to be followed needs to result in portable code and provide fine-grained access to the underlying machine specifics (e.g., ability to handle registers), without penalizing performance by requiring significant time to compile the query. For these reasons, while initially experimenting with C++ code, it was decided to use the Low Level Virtual Machine (LLVM) optimizing compiler framework [10] to generate assembler code.

LLVM is utilized to produce code corresponding to the query logic responsible for accessing and processing tuples in bulk, as this is the part constituting the critical path of the execution. More complex logic, such as the usage of indexing structures or allocation of memory, are still expressed in C++ code. The reason for this is that it is not practical to express all the complex logic and data structures of a DBMS in low-level assembler code. Even if it were possible, it could lead to extensive growth of the code generated, thus increasing compilation time. In any case, this combination causes no issues, as LLVM and C++ methods can be combined.

Regarding performance gains from the generated code, as the majority of the work taking place includes tight loops over tuples, good prefetching and accurate branch prediction are possible. In addition, needed attributes are loaded as late as possible, to avoid "polluting" the registers available with unneeded values, except if these values will be needed in the critical path. In a more general optimization attempt, the code being generated contains much fewer branches than the ones encountered in other database systems. Even for the branches present in the code, effort is made so that they are as "prediction-friendly" as possible, to reduce branch mispredictions occurring in the critical path. Finally, an attempt is made to "incorporate the advantages" of block-oriented query processing, namely the vectorized execution capabilities, while avoiding the materialization penalties. Specifically, if there is available space in the registers for a "block" of values to reside, SIMD instructions can be used over it to speed up execution.

D. Performance

The authors performed a microbenchmark comparing an interpreted and a compiled version of the iterator model, as well

as block-wise tuple processing, with data-centric execution. A simple selection query was used, altering the number of filtered columns between executions. The results, depicted in Figure 5, back the authors' claim for data-centric execution.

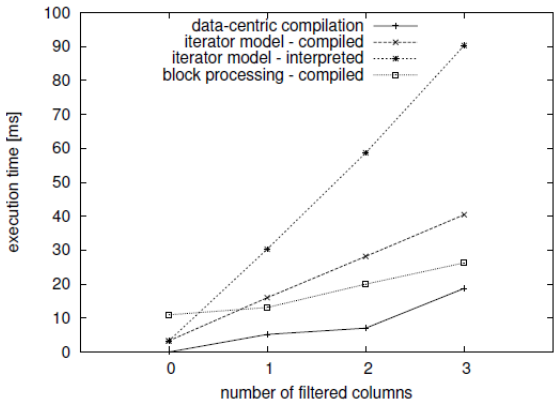


Fig. 5: Performance for cascading selections

In a larger-scale experiment, the authors compared their approach, implemented in the Hyper main-memory DBMS, against a commercial DBMS, and the MonetDB and Vectorwise column stores. The first 5 queries of the TPC-H benchmark are used for the experiments. As data used were main-memory-resident, the main competitors of the examined system were the two column stores. HyPer outperformed the other systems in all queries, the reason according to the authors being mainly the register-based computations performed by HyPer. Another reason is the branch- and cache-friendly nature of the generated code, which was verified by comparing the number of branches and cache misses of HyPer and MonetDB.

E. Summary

By using code generation techniques to gain this fine-grained control over query execution, one can gain significant performance benefits. Actually, as indicated by the microbenchmarks conducted by [14], code compilation and vectorization techniques can be used in conjunction in the context of a query plan to get a synergistic effect. In addition, as we briefly mention in Section V, code generation can be applied not only to gain performance benefits, but to also have a query engine adapting to the underlying data, thus enabling queries over data of heterogeneous formats. Nevertheless, while code generation can offer these performance and functionality opportunities, there is an obvious trade-off with productivity, as implementing complex logic in a low-level language is not a trivial task. The authors of this work recognize this issue, and propose only implementing the parts of the engine operating over bulk tuples in low-level code.

IV. Hierarchical data analysis on the cloud

Both works discussed so far have introduced techniques applied on a single node. However, the constantly increasing amounts of data often render single-node query execution prohibitive for performance reasons. In the case of analytical workloads, there are examples in which petabytes of data need to be scanned to answer a query. Besides the size factor, data to be queried are not necessarily relational; they are instead heterogeneous in their formats, often having a nested structure (e.g., XML and JSON data).

A recent system used by Google to perform interactive analysis over read-only nested data is Dremel [11]. Dremel is used over very large deployments of nodes, and is compatible with the various storage layers of Google (i.e., GFS). It can be considered as a complement of the Map Reduce execution model [5], as MR-based solutions are targetted towards batch-oriented execution, whereas Dremel was designed for interactive, ad-hoc analysis.

Dremel introduces a new storage format that enables representation of nested data in a columnar manner. The motivation for this is that as far as analytical, read-only workloads are concerned, columnar execution has generally been the model of choice [15]. On another note, by basing its architecture on a multi-level serving tree, its basic aim is optimizing the execution of aggregation queries.

A. Columnar storage of nested records

Google uses a strongly-typed data model, named protocol buffers [1], for the serialization of structured data. The authors indicate that a straightforward way to store protocol buffers, and nested data in general, would be by using a record-wise representation as the one depicted in Figure 6a. Instead, by opting for a columnar striped representation and storing values of a nested field contiguously (e.g., all values of nested field A.B.C are stored together), one can read all values of a nested field without reading unneeded neighboring values (e.g., sibling values). In addition, data columns are easier to compress, as values to be compressed are from the same domain. The columnar representation suggested is depicted in Figure 6b.

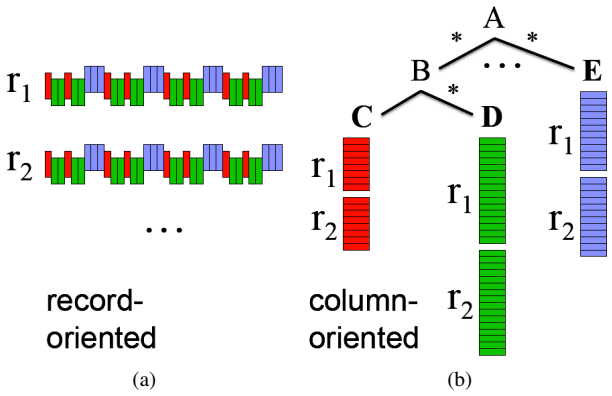


Fig. 6: (a) Record-wise representation (b) Column-wise representation

The columnar representation of nested records needs to be lossless, allowing the reconstruction of the original records. To this end, every column value is assigned a *repetition level* and a *definition level* . The former is used to identify the level in which a "repeated" field occurred, while the latter is used to specify whether a nested record is missing or is explicitly defined. By using definition levels to identify missing values,

there is no need to explicitly store NULL values, thus benefitting in case of sparse datasets. For additional space savings, field values in the columns are compressed.

In order to populate these columns, the original records are recursively traversed, and repetition and definition levels are assigned to all the field values, even if one of them is actually missing. Each value is provided to a field writer, which in turn writes it out. This writer is picked out of a tree of field writers, its structure matching that of the original dataset schema, which is responsible to populate the columns corresponding to every atomic type of the original dataset.

As for the re-assembly of the original records, a finite state machine (FSM) is created, based on which fields of the current data instance have been requested by the user. Each state node of the machine corresponds to an atomic field of the data, while the state transitions are labeled with the appropriate repetition levels. Once created, the FSM is traversed once for each record value.

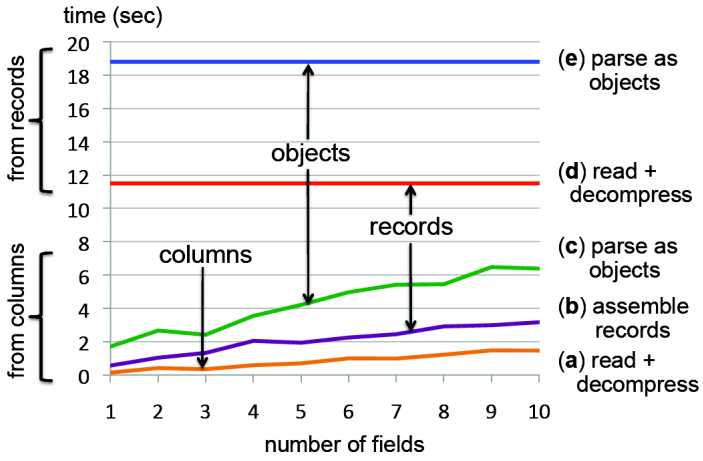


Fig. 7: Performance breakdown of record-wise and column-wise approach

The results of an experiment demonstrating the benefits of the columnar storage model for analytical workloads are depicted in Figure 7. 1GB of data are stored twice, using a record-wise and a column-wise layout each time. They are then read, decompressed, re-assembled into records (in the case of columnar data) and parsed into C++ objects. The benefits observed for the case of columnar storage are due to the selective loading of only the necessary fields. If more fields were to be requested for retrieval, the benefits would be less significant.

B. Query Execution in Dremel

Dremel uses a SQL-like language for querying purposes. The input each expression accepts is one or multiple nested tables, and its output is again a nested table. Queries reaching the system are served using a a multi-level serving tree, as depicted on the left side of Figure 8. When the root server receives a query, it reads the metadata of the relevant tables and routes the query to the next levels of the serving tree. The same process if followed until the query reaches the leaf

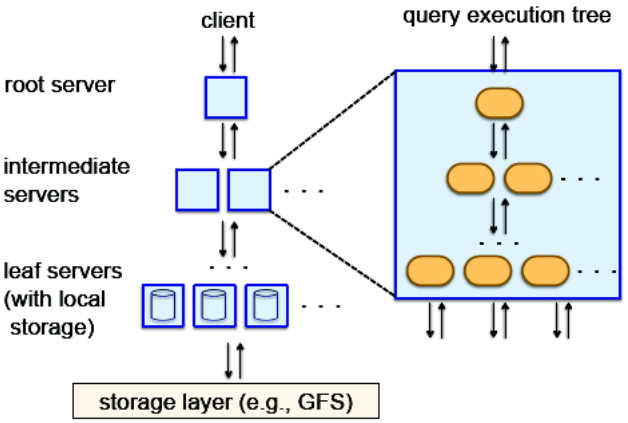


Fig. 8: Architecture of Dremel, and intra-node execution

servers of the architecture, which retrieve the actual data from the underlying storage layer.

While the language of Dremel provides support for constructs such as nested subqueries, top-k expressions and joins, the architecture described is focusing on optimizing aggregation queries. An example would be that of query `SELECT A, COUNT(B) FROM T GROUP BY A`. Once the root server identifies which are the horizontal partitions of the table T, the query would be rewritten and forwarded to all of them. The partial aggregation results would be aggregated in parallel by the intermediate servers, and the root server would produce a final result.

Regarding actual execution, each node of the architecture has an internal execution tree, as depicted on the right side of Figure 8. When evaluating a project-select-aggregate query, a set of iterators perform a coordinated scan the input columns (i.e., they are traversed in lockstep). Any sets of values satisfying the selection criteria are forwarded to the upper layers of the serving tree.

As for multi-user scenarios, Dremel uses a query dispatcher, scheduling queries based on their priorities and the overall system load. Fault tolerance mechanisms are also employed, re-scheduling sub-tasks of a query to different servers when their processing time exceeds a threshold. As in traditional Map Reduce approaches [5], three-way replication is used to facilitate these transitions. These replicas of the sub-partitions of a table are also used to enable serving requests for sub-tables whose primary copy is already in use. Finally, in cases that not completely accurate results are acceptable, Dremel allows returning a result before scanning the entire relations. For example, by scanning 98% of the needed data, it is possible to speed up execution, as the “straggler” nodes that delay the result do not have to be waited for.

C. Summary

Dremel is a successful example of how hierarchical data can be processed using a query engine that is also suitable for analytical workloads, a combination which contributes to our aim for an efficient query engine operating over heterogeneous data. Its ability to scale to thousands of machines enables it to provide solutions to the problems caused by data volumes.

Nevertheless, the fact that the data it processes need to already be converted in its supported columnar format can hinder its use over in situ data. In addition, its requirement to traverse striped tables in a synchronized manner at multiple steps of query evaluation does not allow it to fully exploit a significant benefit of columnar execution strategies, namely vectorized execution.

V. Research Proposal: An Adaptive Query Service for in Situ Querying

Efficiently handling the ever-increasing amount of data produced and overcoming their heterogeneity will be a task of great importance in the upcoming years, given the rate with which data are produced. A novel approach is needed with urgency, or users risk losing the ability to leverage their hard-earned data.

The works covered so far presented necessary building blocks towards our vision. As the NoDB paradigm suggests, raw data need to become a first-class citizen in a DBMS. As also suggested by Dremel, multiple types of data need to be handled, not being restricted to relational formats. Cloud-scale deployments are required for processing, both due to the excessive amounts of data present, but also so that data can be queried in their original locations. Still, single-node performance cannot be overlooked.

Our vision is turning database management into a lightweight service. DBMS are instantiated by queries over the raw data, manipulating and exchanging data independently of its physical location and its representation. Whenever a user queries a dataset, the DBMS learns about the data so that future queries transparently exploit this knowledge for improved data access times.

However, we consider that for such a service to become possible, the engine must not only adapt to the query workload; it also needs to adapt to the underlying data format and query it in place. To achieve this, we are influenced by the code generation techniques presented. We therefore propose using a general query engine which, based on the underlying data format, dynamically generates the access paths for different file formats. In contrast to PostgresRAW, which implemented appropriate access paths for CSV files, such an engine can offer transparent data integration, as a user query can be translated internally to a physical query plan whose access paths actually target different file types. The same code generation techniques can be applied in the rest of the query tree, thus optimizing it appropriately for specific query instances.

Regarding the entire tree, we consider that all operators of an in situ query engine need to take into consideration the fact they are applied over raw data, and avoid all early decisions regarding data conversions. In constrast, while the authors of [2] make the case for selective tokenizing and parsing, these techniques are only applied at the scan operators of the query tree. From our side, we identify general cases in which we can process requested fields without ever converting them to their intended datatype. For example, in the case of numeric attributes, selection operators can be applied over the raw data by using byte comparisons. Structural information such as field length can also be utilized to reduce comparisons. Essentially, the only mandatory condition for us to convert a raw value is whether the user requested a projection of the column it corresponds to. Therefore, we keep conversion costs to the minimum.

The lightweight service described could manifest itself in the form of a simple library. This library could be dispatched to multiple machines, offering a data- and query-adaptive database service over the cloud. While MapReduce-based solutions also allow users to query data directly, avoiding data loading, their single node performance is generally poor, and they are intended for batch-oriented processing. As for "next-generation" cloud-based solutions that allow queries over hierarchical data, as in the case of Dremel, the execution model followed does not exploit modern hardware to a significant degree, or they require data to be stored in a specific format.

All in all, we envision a database service aiming to abolish all static, early decisions that would impose data loading. This service will be able to adapt to the format of the files to be queried, to the queries themselves, and to the query workload as a whole. The price it will pay will be for the minimum necessary amount of data that need to be processed.

References

[1] Protocol buffers: Developer guide. available at https://developers.google.com/protocol-buffers/docs/overview.

[2] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. In *SIGMOD*, 2012.

[3] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct. *PVLDB*, 2(2), 2009.

[4] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.

[5] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.

[6] G. Graefe and W.J. McKenna. The Volcano optimizer generator: extensibility and efficient search. In *Data Engineering, 1993. Proceedings. Ninth International Conference on*, 1993.

[7] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? In *CIDR*, 2011.

[8] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database Cracking. In *CIDR*, 2007.

[9] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[10] Chris Lattner and Vikram S. Adve. Llvm: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, pages 75–88, 2004.

[11] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. *PVLDB*, 3(1), 2010.

[12] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9).

[13] Sriram Padmanabhan, Timothy Malkemus, Ramesh C. Agarwal, and Anant Jhingran. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *ICDE*, 2001.

[14] Juliusz Sompolski, Marcin Zukowski, and Peter A. Boncz. Vectorization vs. compilation in query execution. In *DaMoN*, pages 33–40, 2011.

[15] Michael Stonebraker. Technical perspective - One size fits all: an idea whose time has come and gone. *Commun. ACM*, 51(12), 2008.

[16] Marcin Zukowski, Niels Nes, and Peter Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *Proceedings of the 4th international workshop on Data management on new hardware*, DaMoN '08, 2008.