DiNoDB: Efficient Large-Scale Raw Data Analytics

Yongchao Tian
Eurecom
yongchao.tian@eurecom.fr

Ioannis Alagiannis
EPFL
ioannis.alagiannis@epfl.ch

Erietta Liarou
EPFL
erietta.liarou@epfl.ch

Anastasia Ailamaki
EPFL
anastasia.ailamaki@epfl.ch

Pietro Michiardi
Eurecom
pietro.michiardi@eurecom.fr

Marko Vukolić
Eurecom
marko.vukolic@eurecom.fr

ABSTRACT

Modern big data workflows, found in e.g., machine learning use cases, often involve iterations of cycles of batch analytics and interactive analytics on temporary data. Whereas batch analytics solutions for large volumes of raw data are well established (e.g., Hadoop, MapReduce), state-of-the-art interactive analytics solutions (e.g., distributed shared nothing RDBMSs) require data loading and/or transformation phase, which is inherently expensive for temporary data.

In this paper, we propose a novel scalable distributed solution for *in-situ* data analytics, that offers both scalable batch and interactive data analytics on *raw* data, hence avoiding the loading phase bottleneck of RDBMSs. Our system combines a MapReduce based platform with the recently proposed NoDB paradigm, which optimizes traditional centralized RDBMSs for in-situ queries of raw files. We revisit the NoDB's centralized design and scale it out supporting multiple clients and data processing nodes to produce a new distributed data analytics system we call Distributed NoDB (DiNoDB). DiNoDB leverages MapReduce batch queries to produce critical pieces of metadata (e.g., distributed positional maps and vertical indices) to speed up interactive queries without the overheads of the data loading and data movement phases allowing users to quickly and efficiently exploit their data.

Our experimental analysis demonstrates that DiNoDB significantly reduces the data-to-query latency with respect to comparable state-of-the-art distributed query engines, like Shark, Hive and HadoopDB.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: System—*Query processing*

Keywords

Distributed database; In situ query; positional map file

1. INTRODUCTION

In recent years, modern large-scale data analysis systems have flourished. For example, systems such as Hadoop and Spark [12, 4] focus on issues related to fault-tolerance and expose a simple yet elegant parallel programming model that hides the complexities of synchronization. Moreover, the batch-oriented nature of such system has been complemented by additional components (e.g., [5, 20]) that offer (near) real-time analytics on data streams. The communion of these two approaches is now commonly known as the "Lambda Architecture" (LA) [16]. The LA is split into three layers, *a)* the batch layer for managing and pre-processing append-only set of raw data, *b)* the serving layer that indexes the batch views so that we can support ad-hoc queries with low latency and *c)* the speed layer that handles all low latency requirements using fast and incremental algorithms over recent data only.

Armed with new systems to store and process data, users' needs recently grew in complexity as well. For instance, modern data analysis involves operations that go beyond extract-transform-load (ETL) or aggregation workloads often employing statistical learning techniques to, e.g., build data models and cluster similar data.

As an illustrative use case, we take the perspective of a user (e.g., a data scientist) focusing on a data clustering problem. In such a scenario, our user typically faces the following issues: *i)* clustering algorithms (e.g., k-means [14], DBSCAN [11]) require parameter tuning and an appropriate distance function; and *ii)* computing clustering "quality" typically requires a trial-and-error process whereby test data is assigned to clusters and only domain-knowledge can be used to discern a good clustering from a bad one. In practice, even such a simple scenario illustrates a typical "development" workflow which involves: a *batch processing phase* (e.g., running k-means), an *interactive query phase on temporary data* (i.e., on data interesting in relatively short periods of time), and several iterations of both phases until algorithms are properly tuned and final results meet users' expectations.

Our system, called DiNoDB, explicitly tackles such "development" workflows. Unlike current approaches, which generally require a long and costly data loading phase that considerably increases the data-to-query latency, DiNoDB allows querying raw data in-situ, and exposes a standard SQL interface to the user making query analysis easier and effectively removing one of the main operational bottlenecks of data analysis. In particular, DiNoDB can be seen as a distributed instantiation of the NoDB paradigm [3], which is

specifically designed with in-situ interactive queries in mind (we briefly revisit NoDB [3] in Section 2). The main principle of DiNoDB is avoiding the traditional loading phase on temporary data; indeed, the traditional data loading phase makes sense when the workload (data and queries) is stable and critical in the long term. However, since data loading may include creating indexes, serialization and parsing overheads to truly accelerate query processing, it is reasonable to question its validity when working on temporary data.

The key design idea behind DiNoDB is that of shifting the part of the burden of a traditional load operation to the batch processing phase of a "development" workflow. While batch data processing takes place, DiNoDB piggybacks the creation of distributed positional maps and vertical index files; such auxiliary files (DiNoDB metadata) are subsequently used by a cluster of DiNoDB nodes to improve the performance of interactive user queries on the temporary data. Interactive queries operate directly on raw data files produced by the batch processing phase, which are stored, for example, on a distributed file system such as HDFS [15], or directly in memory. Our experimental evaluation indicate that, for such workloads, DiNoDB systematically outperforms current solutions, including HadoopDB [1], Hive [13] and Shark [18].

In summary, our main contributions in this paper include:

- The design of the DiNoDB architecture, which colocates with the Hadoop MapReduce framework. DiNoDB is a distributed instantiation of the NoDB paradigm, and provides efficient, in-situ SQL-based querying capabilities;
- A system performance evaluation and comparative analysis of DiNoDB versus state-of-the-art systems including HadoopDB, Hive and Shark.

The rest of the paper is organized as follows. In Section 2, we give a short introduction to NoDB. The DiNoDB architecture and its detailed components information are covered in Section 3. Section 4 presents our experimental evaluation demonstrating the performance of DiNoDB. We briefly overview related work in Section 5 and conclude in Section 6.

2. BACKGROUND ON NODB

NoDB [3] is a database systems design paradigm that proposes the conceptual approach for building database systems tailored to querying raw files (i.e., *in-situ*). As a centralized instantiation of the paradigm, [3] presents a variant of PostgreSQL called PostgresRaw. The main advantage of PostgresRaw, compared to other traditional DBMS, is that it avoids the data loading phase and proceeds directly to the querying phase once data becomes available. In a traditional DBMS, loading the whole data inside the database is unavoidable, even if few attributes in a table are needed; the loading process however might take hours (or more) for large amounts of data. NoDB on the other hand, adopts in-situ querying instead of loading and preparing the data for queries. To speed up query executions on raw data files, PostgresRaw works in two directions: it minimizes raw data access cost and reduces raw data access.

To minimize raw data access cost, PostgresRaw tokenizes only necessary attributes and parses only qualified tuples. PostgresRaw builds an auxiliary structure called *positional map*, which contains relative positions of attributes in a line, and updates it during the queries. With positional maps, once processed attributes can be later obtained directly without scanning the entire data records. If requested attributes are omitted in the positional map, a nearby attribute position can be used to navigate faster to the requested attributes. Furthermore, to reduce raw data accesses, PostgresRaw also contains a data cache that temporarily holds the previously accessed data.

The most expensive query for PostgresRaw is the first query for which PostgresRaw needs to scan the whole raw file because it does not have any auxiliary positional map or cache that would be used to speed-up the query. However, even the first PostgresRaw query has lower latency than the loading phase in other traditional DBMSs, which involves the scanning of the entire file [3]. As the positional map and cache grow, after several queries, query latency in PostgresRaw will be comparable to or sometimes even better than that of traditional DBMSs. Overall, PostgresRaw often outperforms classical DBMSs in latency of raw data queries.

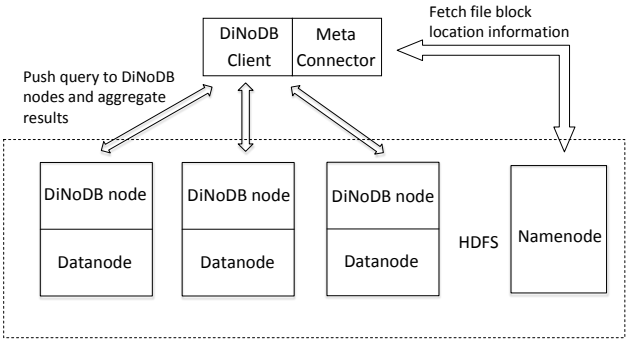


Figure 1: DiNoDB architecture

3. DISTRIBUTED NODB (DiNoDB)

In this section, we present Distributed NoDB (DiNoDB) in details. DiNoDB is designed for interactive query analytics on large volumes of raw, unloaded data, as well as to be integrated into batch processing frameworks such as the MapReduce/Hadoop framework.

3.1 High-level design

At a high level (see Figure 1), DiNoDB consists of a set of PostgresRaw instances (used for interactive queries) combined with the Hadoop framework (used for batch processing). Our DiNoDB prototype is inspired by HadoopDB [1]: it orchestrates PostgresRaw instances using the Hadoop framework itself, relying on existing "connectors" that plug into PostgresSQL. As a result of such a high-level design, raw data that is to be queried in DiNoDB resides in the Hadoop Distributed File System (HDFS)[15]. DiNoDB ensures data locality by co-locating *DiNoDB nodes* (described in Section 3.4) with HDFS data nodes, as shown in Figure 1.

Next, we describe in more detail the DiNoDB pre-processing (batch processing) phase, as well as DiNoDB components, namely DiNoDB clients and DiNoDB nodes.

3.2 Pre-processing (batch processing) phase

The batch processing phase in a "development" workflow typically involves the execution of (sophisticated) analytics

algorithms. This phase is accomplished with one or more MapReduce jobs, whereby output data is written to HDFS by reducers in a key-value format. DiNoDB leverages the batch processing phase as a *pre-processing* phase for future interactive queries.

Namely, DiNoDB piggybacks the generation of auxiliary metadata, including *positional maps* and a *vertical index* to the batch processing phase. Complementary to positional maps, vertical indices in DiNoDB allow users to specify one attribute as a key attribute. An entry in the vertical index has two fields for each record: the key attribute value and the record row offset value. More details about vertical indices can be found in Sec 3.4. DiNoDB metadata is created with user defined functions (UDFs) executed by reducers.This guarantees the co-location of raw data files and their associated DiNoDB metadata, since reducers always first write their output on the local machine in which they execute.

In addition to metadata generation, DiNoDB capitalizes on data pre-processing to store output data in-memory: this can be done implicitly or explicitly. In the first case, raw output data is stored in the file system cache of DiNoDB nodes (i.e., HDFS DataNodes), which uses an approximate form of an LRU eviction policy: recently written output data reside in memory. Alternatively output data can be explicitly stored in RAM, using the `ramfs` file system as an additional mount point for HDFS.[1] With explicit in-memory data caching, DiNoDB avoids the drawbacks of the implicit file system caching scheme in multi-tenant environments where multi-tenancy might cause cache evictions.

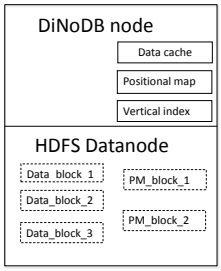
As we demonstrate in Section 4, the combination of metadata and in-memory data storage substantially contributes to DiNoDB query execution performance.

3.3 DiNoDB clients

A DiNoDB client serves as entry point for DiNoDB interactive queries: it provides a standard shell command interface, hiding the network layout and the distributed system architecture from users. As such, applications can use DiNoDB just like a traditional DBMS.

DiNoDB clients accept application requests, and communicate with DiNoDB nodes. When a DiNoDB client receives a query, it fetches related metadata for the "tables" (raw files) indicated in the query, using the *MetaConnector* module. The MetaConnector (see Figure 1) is a proxy between DiNoDB and the HDFS NameNode, and is responsible for retrieving HDFS metadata information like partitions and block locations of raw data files. Using this metadata, the MetaConnector guides DiNoDB clients to query DiNoDB nodes that hold raw data files relevant to user queries. Additionally, the MetaConnector remotely configures DiNoDB nodes so that they can build the mapping between "tables" and the related blocks, including all data file blocks, positional map blocks and vertical index blocks. In summary, the anatomy of a query execution is as follows: (i) using the MetaConnector, a DiNoDB client learns the location of every raw file blocks and pushes the query to the respective DiNoDB nodes; (ii) DiNoDB nodes process the query in parallel; and finally, (iii) the DiNoDB client aggregates the result.

[1]This technique has been independently considered for inclusion in a recent patch to HDFS [8].



(a) DiNoDB node internals

	Key attribute	Row offset
Row 0	Value_0	Offset_0
Row 1	Value_1	Offset_1
Row 2	Value_2	Offset_2
...	...	...
Row n	Value_n	Offset_n

(b) DiNoDB vertical index

Figure 2: DiNoDB node

Note that, since DiNoDB nodes are co-located with the HDFS DataNodes, DiNoDB inherits fault-tolerance from HDFS replication. If a DiNoDB client detects a failure of a DiNoDB node, or upon the expiration of a timeout on DiNoDB node's responses, the DiNoDB client will issue the same query to another DiNoDB node holding a replica of the target HDFS blocks.

3.4 DiNoDB nodes

DiNoDB nodes instantiate customized PostgresRaw databases which execute user queries, and are co-located with HDFS DataNodes (see Figure 2(a)). DiNoDB nodes leverage positional map files generated in the DiNoDB pre-processing phase (see Section 3.2) when executing queries. DiNoDB strives at keeping metadata small: indeed, in some cases, a positional map may contain the position of all "attributes" present in a raw data file, which could cause metadata to be as large as the data to query, thus nullifying its benefit. To do so, DiNoDB uses a *sampling* technique to store only a few attributes, depending on a user-provided sampling rate. From the query latency perspective, an approximate positional map still provides tangible benefits: when a query "hits" an attribute in the sampled map, its benefits are clear; otherwise, DiNoDB nodes use sampled attributes as anchor points, to proceed with a sequential scan of nearby attributes to satisfy a query.

When the nature of interactive queries is known in advance, DiNoDB users can additionally specify a single *key attribute*, which is used to create a vertical index file (see Figure 2(b)). Vertical index generation (just like that of positional maps) is piggybacked to the batch-processing phase. When available, DiNoDB nodes load vertical index files, which contain the key attribute value and its row offset in the raw data file. This technique contributes to improved system performance: if a user query "hits" the key attribute, DiNoDB nodes perform an index scan instead of full sequential scan of the raw data, which considerably saves disk reads.

Note that both positional map and vertical index files are loaded by a DiNoDB node during the first query execution. As our performance evaluation shows (see Section 4), the metadata load time is considerably small, when compared to the execution time of the first query: as such, only the first query represents a bottleneck for DiNoDB (a cost that the NoDB paradigm also has to bear).

Optimizations in DiNoDB nodes. In the vanilla PostgresRaw [3] implementation, a "table" maps to a single raw

data file. Since the HDFS files are instead split into multiple blocks, DiNoDB nodes use a customized PostgresRaw instance, which features a new file reader that can map a "table" to a list of raw data file blocks. In addition, the vanilla PostgresRaw implementation is a multiple-process server, which forks a new process for each new client session, with individual metadata and data cache per process. Instead, PostgresRaw nodes place metadata and data cache in shared memory, such that user queries – which are sent through the DiNoDB client – can benefit from them across multiple sessions.

An additional feature of DiNoDB nodes is that they access raw data files bypassing the HDFS client API. This design choice provides increased performance by avoiding the overheads of a Java-based interface. More importantly, as discussed in Section 3.2, DiNoDB users can selectively indicate whether raw data files are placed on disk or in memory. Hence, DiNoDB nodes can seamlessly benefit from the native file system cache, or from a memory-backed file system to dramatically decrease query times.

4. EVALUATION

Next, we present an experimental analysis of DiNoDB. First, in Section 4.1 we study the performance of DiNoDB in the context of interactive queries and compare DiNoDB to Shark [18] (version 0.8.0), Hive [13] (version 0.9.0) and PostgreSQL-backed HadoopDB [1]. Then, in Section 4.2 we evaluate the benefits of pre-generating DiNoDB metadata (i.e., positional maps and vertical indices) as well as the benefits of the in-memory cache, that is populated in the pre-processing phase (see Sec. 3.2). Finally, in Section 4.3, we compare the performance of DiNoDB to that of other systems using a dataset that does not fit in memory. All experiments are conducted on a cluster with 6 machines, with 4 cores, 16 GB RAM and 1 Gbps network interface each. The underlying distributed file system is HDFS, configured with one NameNode and five DataNodes.

4.1 Comparative performance evaluation

In this section, we proceed with a comparative analysis using two kinds of query templates: random and key attribute queries. The raw data file we use in our experiments contains $3.6 * 10^7$ tuples. Each tuple has 150 attributes, which are all integers distributed randomly in the range $[0\text{-}10^9)$. The size of the raw file is 50 GB. The dataset is stored in RAM, using the `ramfs` file system available on each machine.

Additional details are as follows. DiNoDB uses a 2.2 GB positional map (generated with a 1/10 sampling rate) and a 720 MB vertical index file. HadoopDB uses PostgreSQL as a local database, co-located with HDFS DataNodes. To minimize loading time in HadoopDB, we load bulk data into PostgreSQL directly without using HadoopDB's Data Loader which requires to repartition data. Finally, we evaluate Shark by both loading the dataset in its internal caching mechanism before issuing queries (that we label *Shark_cached*), and using the default, disk-based mechanism, (that we call *Shark*).

Random queries. A sequence of five SELECT SQL queries are executed in all systems. Each query targets one random attribute with 1‰ selectivity. The result is shown in Figure 3. Hive has the worst performance since each query repeatedly scans the whole data files. It spends more than

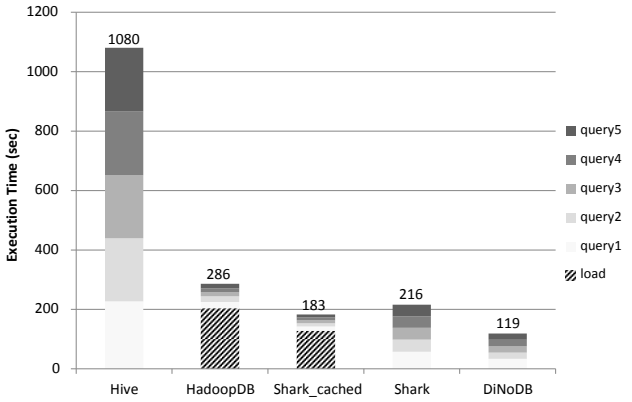


Figure 3: DiNoDB vs. other distributed systems: Random query

200 seconds for each query. HadoopDB has a very short query latency, but it only happens after 200 seconds loading time. Shark_cached enjoys faster loading than HadoopDB because it achieves a maximum degree of parallelism to load data into memory at the aggregated throughput of the CPU [18]. HadoopDB could also support a high degree of parallelism to load the data, but that would require data repartitioning on each datanode, which would require extra effort. On the other hand, when queries are executed in Shark in situ, i.e., without caching/loading, then each query latency is about 43 seconds. Finally, DiNoDB achieves the best performance with an average query latency of 24 seconds. This can be explained by the fact that DiNoDB avoids explicitly the loading phase and leverages positional maps to speed-up queries on raw files.

Key attribute based queries. In this experiment, we measure the effect of vertical indices on the overall query latency. To this end, we execute range queries on the key attribute. The result is shown in Figure 4. Hive supports index scan and needs more than 400 seconds to build the index. However, we register a query latency that is worse than Hive with a sequential scan (depicted in Figure 4). In HadoopDB, building an index on our dataset requires about 19 seconds. Overall, HadoopDB requires 223 seconds before any query can be executed, although each query has less than 10 seconds latency overall. Shark does not have any indexing mechanism, so the performance we register is similar to our experiments with random queries. DiNoDB achieves the best performance: the average query latency is about 11 seconds, which is less than half of the latency for random queries.

Discussion. In our current prototype a DiNoDB node only uses a single process with one thread. In contrast, Shark can use all available cpus on loading and processing data.[2] Hence, Shark_cached has low per-query latency (after loading) which makes it appealing when there are many queries

[2]A more "fair" comparison between DiNoDB and Shark would involve restricting Shark to using only a single core; such a "single-core" Shark in our experience suffers a threefold (3x) latency increase compared to the performance reported in Figures 3 and 4.

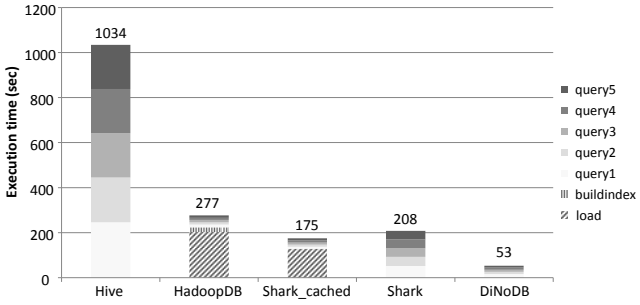


Figure 4: DiNoDB VS. other Distributed systems: Key value query

(e.g., when data of interest is not temporary). Moreover, since DiNoDB is coupled to the Hadoop framework, it suffers from the its job scheduling overhead, which is about 7 seconds per query. In our future work, we aim at eliminating these sources of overhead, although globally, DiNoDB outperforms all other systems for the "development" workloads we target in this work.

4.2 Benefits of pre-processing

In this experiment, we evaluate the benefits of the DiNoDB pre-processing phase (Sec. 3.2), where we generate DiNoDB metadata. Indeed, one major difference between a vanilla PostgresRaw instance and DiNoDB is that the latter *loads* positional map and vertical index files *before* executing the first query. As such, what we really measure is the overhead related to the metadata loading phase. Note that in the following we use a 1/10 sampling rate to generate a DiNoDB positional map.

We run the next experiments in a single-machine deployment: thus, we compare the performance of a vanilla PostgresRaw instance to that of a single DiNoDB node. In both cases, the two systems execute the same sequence of queries on a 10GB raw file. All the queries are `SQL SELECT` statements, which have the selection condition in the `WHERE` clause and randomly project one attribute.

Additionally, we also study the impact of raw data being in memory or on disk. We do this using vanilla PostgresRaw, which allow us to isolate the effects of metadata from the benefit of a faster storage tier. We denote the two cases by *PostgresRaw_ramfs* and *PostgresRaw_disk*, respectively. Figure 5 shows our results.

We can see that for *PostgresRaw_disk*, the first query has a latency of about 143 seconds: in this case, the disk is the bottleneck. The first query in *PostgresRaw_ramfs* takes about 53 seconds. Instead, a DiNoDB node executes the first query in 13 seconds, of which 0.5 seconds are related to loading metadata files. For all subsequent queries, all three systems have similar performance, because data reside in memory. Note that more queries imply richer positional maps, and hence lower latencies.

In summary, the experiments shown in this Section validate the design choices in DiNoDB: if possible, raw data should reside in ram, to avoid paying the price of slow disk access speed. Additionally, piggybacking the generation of metadata files in the pre-processing phase, which are then loaded in memory before query execution only marginally affects the overall query latency.

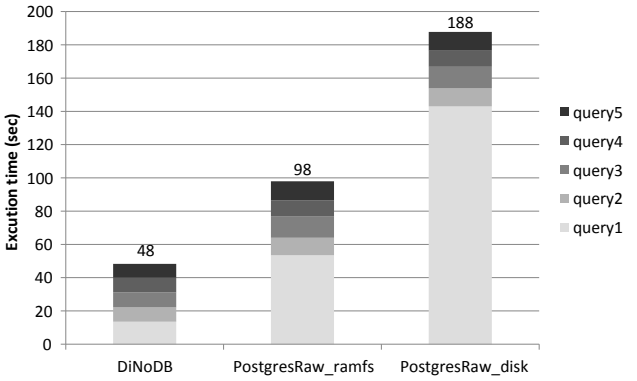


Figure 5: Benefits of pre-processing

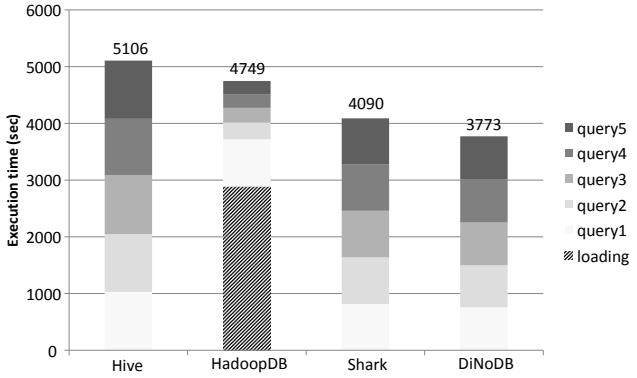


Figure 6: 250 GB dataset experiment

4.3 Large dataset test

We now study the performance of different systems when the dataset does not fit in memory. We use a 250 GB raw data file that has been generated similarly to the raw file in Section Section 4.1 (albeit with more tuples), and the same random queries defined in Section 4.1. The results are shown in Figure 6, where we omit latencies for *Shark_cached*, for obvious reasons.

HadoopDB achieves the smallest query latencies, at the cost of a 50 minutes loading phase. Hive, Shark and DiNoDB have similar performance because they are all bound by disk I/O. Overall, DiNoDB has the smallest aggregate query latency. Clearly, DiNoDB faces a "tipping point": when the number of queries on temporary data is small enough, aggregate query latencies are in its favor; the situation changes when the number of queries exceed a threshold. A better characterization of this threshold is part of our future work.

5. RELATED WORK

Several research works and commercial products complement the batch processing nature of Hadoop/MapReduce [12, 7] with systems to query large-scale data at interactive speed using a SQL-like interface. Examples of such systems include HadoopDB [1], Vertica [17], Hive [13] or Impala [6]. These systems require data to be loaded before queries can be executed: in workloads for which data-to-query time matters, for example due to the ephemeral nature of the data at

hand, the overheads due to the load phase, crucially impact query performance. In [2] the authors propose the concept of "invisible loading" for HadoopDB as a technique to reduce the data-to-query time; with invisible loading, the loading to the underlying DBMS happens progressively. In contrast to such systems, DiNoDB avoids data loading and is tailored for querying raw data files leveraging positional maps and vertical indices.

Shark [18] presents an alternative design: it relies on a novel distributed shared memory abstraction [19] to perform most computations in memory while offering fine-grained fault tolerance. Shark builds on Hive [13] to translate SQL-like queries to execution plans running on the Spark system [4], hence marrying batch and interactive data analysis. To achieve short query times, Shark requires data to be resident in memory: as such, the output of batch data processing – which is generally materialized to disk for failure tolerance – need to be first loaded in RAM. As our experimental results indicate, such data "caching" can be a costly operation.

PostgresRaw [3] is a centralized DBMS that avoids data loading and transformation prior to queries. As we detailed in the previous sections, DiNoDB leverages PostgresRaw as a building block, effectively distributing and scaling-out PostgresRaw, while integrating it with the Hadoop batch processing framework. Unlike PostgresRaw, DiNoDB does not generate positional maps on the fly, but leverages the Hadoop batch processing phase to pre-generate positional maps and vertical indices to speed up the query execution.

Finally, DiNoDB shares some similarities with several research work that focuses on improving Hadoop performance. For example, Hadoop++ [9] modifies the data format to include a Trojan Index so that it can avoid full file sequential scan. Furthermore, CoHadoop [10] co-locates related data files in the same set of nodes so that a future join task can be done locally without transfering data in network. However, neither Hadoop++ nor CoHadoop are specifically tuned for interactive raw data analytics, like DiNoDB is.

6. CONCLUSION

We proposed a design of DiNoDB, a distributed database system tuned for interactive queries on raw data files generated by large-scale batch processing frameworks such as Hadoop/MapReduce. Our preliminary evaluation suggest that DiNoDB has very promising performance targeting iterative machine learning data processing, where few interactive queries are performed in between iterative batch processing jobs. In such use cases, we showed that DiNoDB outperforms stat-of-the-art systems, including Apache Hive, HadoopDB and Shark.

In future work, we aim at further improving the performance of DiNoDB, notably by removing the overhead related to Hadoop orchestration framework we use (currently HadoopDB), as well to evaluate our system on a wider set of datasets and workloads.

Acknowledgements. This work is partially supported by the EU project BigFoot (FP7-ICT-223850).

7. REFERENCES

[1] A. Abouzeid and et. al. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2009.

[2] A. Abouzied and et. al. Invisible loading: Access-driven data transfer from raw files into database systems. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, 2013.

[3] I. Alagiannis and et. al. Nodb: efficient query execution on raw data files. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, 2012.

[4] Apache Spark. Webpage. `http://spark.apache.org/`.

[5] Apache Storm. Webpage. `http://storm.incubator.apache.org/`.

[6] Cloudera Impala. Webpage. `http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html/`.

[7] J. Dean and et. al. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, 2004.

[8] Discardable Distributed Memory: Supporting Memory Storage in HDFS. Webpage. `http://hortonworks.com/blog/ddm/`.

[9] J. Dittrich and et. al. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow.*, 2010.

[10] M. Y. Eltabakh and et. al. Cohadoop: Flexible data placement and its exploitation in hadoop. *Proc. VLDB Endow.*, 2011.

[11] M. Ester and et. al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proc. of the 2nd International Conference on Knowledge Discovery and Data Mining*, 1996.

[12] Hadoop. Webpage. `http://hadoop.apache.org/`.

[13] Hive. Webpage. `http://hive.apache.org/`.

[14] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. of 5th Berkeley Symposium on Mathematical Statistics and Probability*, 1967.

[15] K. Shvachko and et. al. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, 2010.

[16] The Lambda Architecture. Webpage. `http://lambda-architecture.net/`.

[17] Vertica. Webpage. `http://www.vertica.com/`.

[18] R. S. Xin and et. al. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, 2013.

[19] M. Zaharia and et. al. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

[20] M. Zaharia and et. al. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.