BigFoot: Big Data Analytics of Digital Footprints

Project name	BigFoot
Project ID	FP7-ICT-ICT-2011.1.2 Call 8 Project No. 317858
Working Package Number	WP4
Deliverable Number	D.4.4
Document title	Consolidated implementation of the storage layer
Document version	1.0
Author	EPFL, EUR
Date	31/05/2015
Status	*Public*

Revision History

Date	Version	Description	Author
24/03/15	0.1	Initial setup	Pietro Michiardi
24/03/15	0.2	Add content of DiNoDB	Yongchao Tian
24/03/15	0.5	Adding executive summary	Pietro Michiardi
24/03/15	1.0	Final Deliverable Review	Pietro Michiardi

Executive Summary

The purpose of this document is to summarize the research and development activities in the context of the WP4 of the BigFoot project, covering the whole project duration.

First, we summarize the contributions made in the first two years of the project, which cover single-node solutions to the storage layer of the BigFoot stack. Then we focus on the consolidation effort that was made in the last year of the project, and present a novel distributed query engine that builds on the research and development effort made in the first two years of the project.

We conclude with a global summary of the contributions made in WP4 for the whole project duration.

Contents

1 Introduction

Nowadays, increasingly more applications in various domains generate and collect massive amounts of data at a rapid pace. New research fields and applications (e.g., network monitoring, sensor data management, stock market arbitrating, clinical studies etc.) emerge and require broader data analysis functionality to gain deeper insights from the available data.

As data is growing in size at a rapid pace, it makes the analysis an increasingly harder endeavor but at the same time gives birth to new usage scenarios and data analysis opportunities.

In this context of Big Data management, data storage and processing is of paramount importance. In WP4 of the BigFoot project, we address the storage media/heterogeneity, distribution of data storage for scalability and reliability, as well as distributed execution for efficient data analysis.

1.1 Overview of Previous Work

Single-node solutions:. In deliverable D.4.2, we presented results related to the management of meta-data for optimizing query execution on modern SSD hardware. In particular, using real-world datasets provided by GridPocket, we showed how one could save a substantial amount of storage space in SSDs by marginally sacrificing indexing accuracy using BF-Tree. In deliverable D.4.3, we demonstrated the benefit of using the NoDB approach for querying over raw data files using an experimental analysis of real-world datasets provided by our industrial partner Symantec. Together, these two work items provided an optimized single-node query execution platform based on which a distributed query execution engine can be built.

Towards a distributed query engine:. In D.4.2, we discussed distributed data partitioning patterns for optimizing distributed query execution within a data center. Following this, in D.4.3, we expanded our scope from the intra-datacenter work of D.4.2 to cross-data center concerns by presenting techniques to achieve reliable storage across datacenters. Specifically, we introduced, GRAF, an efficient geo-replication protocol for tolerating arbitrary faults beyond crashes in critical metadata. Together, these two work items provided the fundamental background for scaling the single-node work we mentioned previously to large cluster that span data centers.

1.2 Deliverable Goals

The research effort for D.4.4 presents the steps following the work presented in deliverable D.4.2 and D.4.3 considering scale-out query execution over raw data files. In this deliverable, we initially describe a number of applications and use cases, both in the context of BigFoot and others, that motivate marrying the in-situ query processing benefit of NoDB with the large-scale batch-processing capabilities of Hadoop. Subsequently, we introduce, DiNoDB, a distributed interactive query engine that can provide low-latency in-situ querying over raw data stored in large-scale Hadoop clusters.

2 DiNoDB

In recent years, modern large-scale data analysis systems have flourished. For example, systems such as Hadoop and Spark [16, 6] 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 systems has been complemented by additional components (e.g., Storm and Spark streaming [7, 28]) that offer (near) real-time analytics on data streams. The communion of these approaches is now commonly known as the "Lambda Architecture" (LA) [24]. In fact, LA is split into three layers, i) the *batch layer* (based on e.g., Hadoop/Spark) for managing and pre-processing append-only raw data, ii) the *speed layer* (e.g., Storm/Spark streaming) tailored to analytics on recent data only while achieving low latency using fast and incremental algorithms, and iii) the *serving layer* (e.g., Hive [17]/SparkSQL [6]) that exposes the batch views to support ad-hoc queries written in SQL, with low latency.

The problem with such existing large scale analytics systems is twofold. First, combining components (layers) from different stacks, though desirable, raises performance issues and is sometimes not even possible in practice. For example, companies who have expertise in, e.g., Hadoop and traditional SQL-based (distributed) RDBMSs, would arguably like to leverage this expertise and use Hadoop as the batch processing layer and RDBMSs in the serving layer. However, this approach requires an expensive transform/load phase to, e.g., move data from Hadoop's HDFS and load it into a RDBMSs, which might not be possible to be amortized, in particular in scenarios with a narrow processing window.

On the other hand, using integrated large-scale distributed data analytics frameworks (e.g., the Hadoop or Spark stack) raises a second major concern; programming these distributed frameworks is complex and results in code specifically engineered for the framework it runs on. This limits the flexibility of the big-data solutions and rings the "lock-in" bells. Moreover, such integrated data analytics systems often fail to leverage decades of advances in optimizing the performance of (distributed) RDBMSs. In summary, contemporary data scientists face a wide variety of competing approaches that they can adopt for the batch and the serving layer. Nevertheless, these approaches often have strict roles, in many cases ignoring one the other and thus, failing to explore potential benefits from learning from each other.

In this report, we propose the DiNoDB architecture that addresses the above issues. Our approach is based on a modular integration of batch

processing engines (e.g., Hadoop) with a novel distributed, fault-tolerant and scalable interactive query engine for *in-situ* analytics on *raw* data. DiNoDB integrates the batch processing with the serving layer, through a series of plug-in components; these components piggyback the creation of *metadata* (in the form of auxiliary data structures, e.g., positional maps and vertical indexes) during the batch processing phase, that DiNoDB can heavily exploit to speed-up the interactive data analysis of the same raw data files. Our solution effectively brings together the batch processing and the serving layer for big data workflows while avoiding any loading and transformation cost when it comes to ad-hoc interactive query engines with a narrow processing window.

We perform experimental analysis of DiNoDB on both synthetic and real-life datasets that demonstrates that it significantly reduces the data-to-query time and achieves very good performance compared to state-of-the-art distributed query engines, such as Hive, Stado, SparkSQL and Impala. Specifically, DiNoDB can be up to one order of magnitude faster, in terms of aggregate query execution time, than the second best system, Impala, when interactive queries follow (or interleave with) batch processing jobs. In general, DiNoDB is often an order of magnitude (or more) faster than all other remaining systems we analyze, for a variety of use cases.

In summary, our main contributions include:

- The design of DiNoDB, a distributed interactive query engine. DiNoDB leverages modern multi-core architectures and provides efficient, distributed, fault-tolerant and scalable in-situ SQL-based querying capabilities for raw data. As such, DiNoDB is the first distributed and scalable instantiation of the NoDB paradigm [3].

- Proposal and implementation of the plug-in based approach to interfacing batch processing and interactive query serving engines in a data analytics system. Plug-ins generate, during the batch processing phase, metadata that aims to facilitate and expedite subsequent interactive queries.

- Detailed performance evaluation and comparative analysis of DiNoDB versus state-of-the-art systems including Hive, Stado, SparkSQL and Impala, showing the research space of a variety of use cases.

The rest of the DiNoDB section is organized as follows. In Section 2.1, we further motivate our approach and the need for a system such as DiNoDB. In Section 2.2, we describe the architecture of DiNoDB. Section 2.3

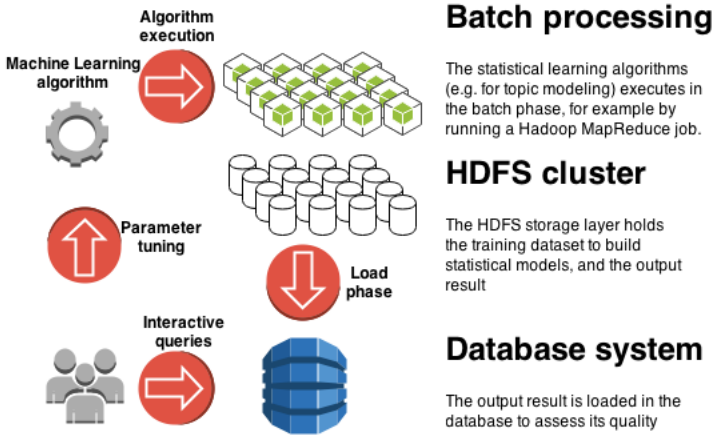


Figure 1: Machine learning use case.

overviews related work. Section 3 concludes the DiNoDB section. We give our experimental results based on both synthetic and real-life datasets in D.2.4.3.

2.1 Application and Use Cases

In this section, we overview some of the contemporary uses cases in which batch processing precedes an interactive query serving layer in the data analytics flows. These use cases include machine learning (Section 2.1.1) and data exploration (Section 2.1.2). For each of these use cases we discuss: i) how better communication between the batch processing and the serving layer that DiNoDB brings may help, and ii) the applicability of our *raw data* analytics approach.

2.1.1 Machine learning

In the first use case we take the perspective of a user (e.g., a data scientist) focusing on a complex data clustering problem. Specifically, we consider the task of learning *topic models* [8], which amounts to automatically and jointly clustering words into "topics", and documents into mixtures of topics. Simply stated, a topic model is a hierarchical Bayesian model that associates with each document a probability distribution over "topics", which are in turn distributions over words. Thus, the output of a topic modeling data analysis can be thought of as a (possibly very large) matrix of probabilities:

each row represents a document, each column a topic, and the value of a cell indicates the probability for a document to cover a particular topic.

In such a scenario, depicted in Figure 1, the user typically faces the following issues: i) topic modeling algorithms (e.g., Collapsed Variational Bayes (CVB) [23]) require parameter tuning, such as selecting an appropriate number of topics, the number of unique features to consider, distribution smoothing factors, and many more; and ii) computing "modeling quality" typically requires a trial-and-error process whereby only domain-knowledge can be used to discern a good clustering from a bad one. In practice, such a scenario illustrates a typical "development" workflow which requires: a *batch processing phase* (e.g., running CVB), 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.

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-insight time, DiNoDB allows querying raw data in-situ, and exposes a standard SQL interface to the user. This simplifies query analysis and reveals the main advantage of DiNoDB in this use case, that is the removal of the *temporary data* loading phase, which today represents one of the main operational bottlenecks in data analysis. Indeed, the traditional data loading phase makes sense when the workload (i.e., data and queries) is stable in the long term. However, since data loading may include creating indexes, serialization and parsing overheads,it is reasonable to question its validity when working with temporary data, as in our machine learning use case.

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 (see Section 2.2 for details) 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 on a distributed file system such as HDFS [20], or directly in memory.

2.1.2 Data exploration

In this section, we discuss another prominent use case which is another important motivation for our work. Here we consider a user involved in a preliminary, yet often fundamental and time-consuming, *data exploration*

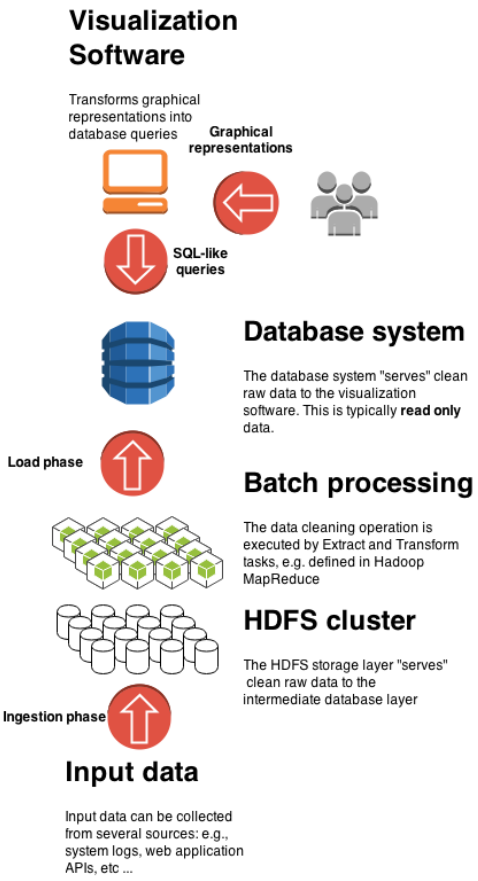


Figure 2: A typical data exploration architecture.

task. Typically, the user collects data from different sources (e.g., an operational system, a public API) and stores it on a distributed file system such as HDFS for subsequent processing. However, before any useful processing can happen, data needs to be "cleaned" and studied in detail.

Data exploration generally requires visualization tools, that assist users in their preliminary investigation by presenting the salient features of the raw data. Current state-of-the-art architectures for data exploration can be summarized as in Figure 2. A *batch processing phase* ingests "dirty" data to produce useful raw data; such raw data is then loaded into a database system that supports an *interactive query phase*, whereby a visualization software (e.g., Tableau Software) translates user-defined graphical representations into a series of queries that the database system executes. Such queries

typically "reduce" raw data into aggregates, by filtering, selecting subsets satisfying predicates and by taking representative samples (e.g., by focusing on top-k elements).

In the scenario depicted above, DiNoDB drastically reduces the data-to-insight time, by allowing visualization software to directly interact with the raw representation of data, without paying the cost of the load phase that traditional database systems require. In addition, the metadata that DiNoDB generates by piggybacking on the batch processing phase (while data is "cleaned"), substantially improves query performance, making it a sensible approach for applications where interactivity matters.

2.2 DiNoDB architecture and implementation

In this section, we present the high level design architecture of DiNoDB in detail. DiNoDB is designed to provide a modular integration of batch processing frameworks such as the MapReduce/Hadoop framework with a distributed solution for in-situ data analytics on large volumes of raw data files. Additionally, we present how the DiNoDB plug-ins allow us to connect the batch processing and the interactive query serving phase in DiNoDB. Finally, we show how the scalable interactive query engine of DiNoDB, tailored to querying *temporary* data on large scale without any initialization overhead, leverages the metadata generated during the batch processing phase.

In the remaining of this section we assume that raw data processed during the batch processing phase might be stored in a variety of file formats (e.g., textual, hierarchical, binary). On the other hand, the *temporary* output data files produced by the batch processing phase and used in the query serving phase are in a structured textual data format (e.g., comma-separated value files).

2.2.1 High-level design

The batch processing phase (e.g., in the machine learning and data exploration use cases outlined previously) typically involves the execution of (sophisticated) analysis algorithms. This phase might include one or more Hadoop jobs, whereby output data is written to HDFS as key-value pairs.

The key idea behind DiNoDB is to leverage batch processing as a *pre-processing* phase for future interactive queries. Namely, DiNoDB features *plug-ins* that are executed *together with* the batch processing phase. These DiNoDB plug-ins piggyback the generation of metadata that is used by

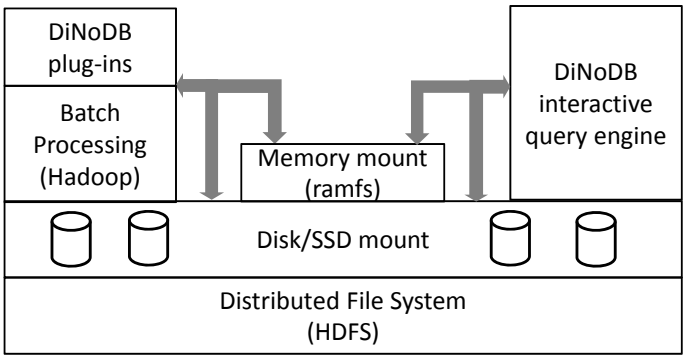


Figure 3: High level architecture of DiNoDB.

DiNoDB to speed up later interactive queries. We further detail DiNoDB plug-ins and metadata generation in Section 2.2.2.

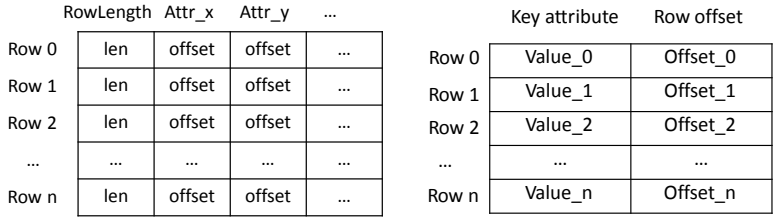
In addition to the metadata generation, DiNoDB capitalizes on data preprocessing by keeping output data from Hadoop jobs in-memory. To be more specific, we configure Hadoop to store output data and plug-in metadata in RAM, using the `ramfs` file system as an additional mount point for HDFS.[1] The DiNoDB prototype supports both `ramfs` and disk mount points for HDFS, a design choice that allows supporting queries on data that cannot fit in RAM. The high-level design of DiNoDB is depicted in Figure 3.

Both the output data and the plug-in metadata are consumed by the DiNoDB interactive query engine. As we detail in Section 2.2.3, the DiNoDB interactive query engine is a massively parallel processing engine that orchestrates several DiNoDB nodes. Each DiNoDB node is an optimized instance of PostgresRaw [3], a variant of PostgreSQL tailored to querying temporary data files produced in the batch processing phase. To ensure high performance and low query execution times, we co-locate DiNoDB nodes to HDFS DataNodes, where the two share data through HDFS, and in particular through its in-memory, `ramfs` mount.

2.2.2 DiNoDB plug-ins

DiNoDB piggybacks the generation of auxiliary metadata to the batch processing phase using the DiNoDB plug-ins. In this section, we introduce what metadata can be generated by DiNoDB plug-ins, and outline their

[1]This technique has been independently considered for inclusion in a recent patch to HDFS [12] and in a recent in-memory HDFS alternative called Tachyon [19].



(a) DiNoDB positional maps (b) DiNoDB vertical index

Figure 4: Metadata generated by DiNoDB plug-ins.

implementation. Our current prototype generates three kinds of metadata: *positional maps* [3], *vertical indexes* and *statistics*.

Positional maps. The positional maps are data structures that NoDB [3] uses to optimize in-situ querying. Contrary to a traditional database index, a positional map indexes the structure of the file and not the actual data. A positional map in DiNoDB (shown in Figure 4(a)) contains relative positions of attributes in a data record, as well as the length of each row in the data file. During query processing the information contained in the positional map can be used to jump to the exact position or as close as possible to an attribute, significantly reducing the *cost of tokenizing and parsing* when a data record is accessed.

To keep the size of the generated positional map relatively small to the size of a data file, DiNoDB uses a *uniform sampling* technique to store positions only for a subset of the attributes in a file. The sampling rate is provided as input from the user. Algorithm 1 shows the algorithm for generating the approximate positional map. An approximate positional map can still provide tangible benefits. If the requested attribute is not part of the sampled positional map, a nearby attribute position is used to navigate faster to the requested attribute without significant overhead. In D.2.4.3, we show the effect of different sampling rates in the query execution performance.

Vertical indexes. The positional map can reduce the CPU processing cost associated with parsing and tokenizing data; however, it still relies on accessing the whole file (through a full table-scan plan). To provide the performance benefit of an index-based access plan, DiNoDB allows users to specify one or more *key attributes* for which vertical indexes are simultaneously created in the batch processing phase. Such vertical indexes can be used to quickly search and retrieve data without having to perform a full scan on the temporary output file. Figure 4(b) shows the structure of a vertical index. An entry in the vertical index has two fields for each record of

Input corresponds to the output tuples of the batch processing phase;
Input : key=k, value=$[attr_1, attr_2, \cdots, attr_n]$
Output: Batch output: *data*, Positional map: *pm*

Initialize *record*, as an empty string
for $attr \in [k, attr_1, attr_2, \cdots, attr_n]$ **do**
  if *attr is sampled* **then**
    position ← `getLength` (record) ;
    positions.`append` (position) ;
  end
  record ← record + ',' + attr ;
end
data.`write` (record);
pm.`write` (positions);

Algorithm 1: Positional map generation.

the temporary output file: the key attribute value and the record row offset value. As such, every key attribute value is associated with a particular row offset in the data file, which DiNoDB nodes use to quickly access a specific row of a file.

Statistics. Modern database systems rely on statistics to choose efficient query execution plans. Query optimization requires knowledge about the nature of processed data (e.g., skew in the distribution of values per attribute, number of unique values per attribute) that helps ordering operators such as joins and selections; however, such statistics are available only after loading the data or after a pre-processing phase. DiNoDB plug-ins can generate statistics of the output data generated in the batch processing phase, by computing the number of records and the number of distinct values for specific attributes. To achieve this, DiNoDB plug-ins use the near-optimal probabilistic counting algorithm HyperLogLog [15]. Statistics on attribute cardinality are used by DiNoDB to improve the quality of the query plans for complex queries, e.g., involving join operations.

Plug-in implementation

DiNoDB plug-ins are designed to be a non-intrusive mechanism, that can be seamlessly integrated into systems supporting Hadoop I/O classes, such as Hadoop MapReduce and Apache Spark. In addition, users are requested to express their requirements in terms of metadata using simple configuration instructions that complement the user code of the batch processing phase.

To use DiNoDB plug-ins, users need to replace the vanilla Hadoop *OutputFormat* class by a new module called *DiNoDBOutputFormat*. Our prototype supports the *TextOutputFormat* sub-class, which allows DiNoDB to operate on textual data formats. Specifically, the *DiNoDBTextOutputFormat* module, implements a new *DiNoDBArrayWritable* class which is used to generate both the temporary output data and its associated metadata. DiNoDB plug-ins are activated at the end of the batch processing phase (during output generation). Metadata generators iterate over output tuples to compute the types of metadata described above, while finalizing the write operation of the output data. For example, to generate positional maps, DiNoDB plug-ins use the positional map generator in the "iterator" implementing Algorithm 1: for each output tuple (key/value pair generated by a reducer or a mapper), the positional map generator uses it to generate attribute positions, and output both the unmodified output tuple and the positional map for this tuple.

DiNoDB plug-ins are configured by passing a configuration file to each batch Hadoop job. Users specify which metadata to generate and indicate parameters, such as the sampling rate to use for the generation of positional maps and the key attributes for the generation of vertical indexes. The current DiNoDB prototype supports three kinds of metadata but our approach can be extended with additional plugins to enrich the system functionality.

2.2.3 DiNoDB interactive query engine

At a high level (see Figure 5), the DiNoDB interactive query engine consists of a set of DiNoDB nodes, orchestrated using a massively parallel processing (MPP) framework. In our prototype implementation, we use the Stado MPP framework [22], which nicely integrates PostgreSQL-based database engines. DiNoDB ensures data locality by co-locating DiNoDB nodes with HDFS DataNodes.

In the following paragraphs, we first describe the DiNoDB client (Section 2.2.3) and the DiNoDB nodes 2.2.3. Then, we describe how DiNoDB achieves fault-tolerance (Section 2.2.3).

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 the users. As such, applications can use DiNoDB just like a traditional DBMS.

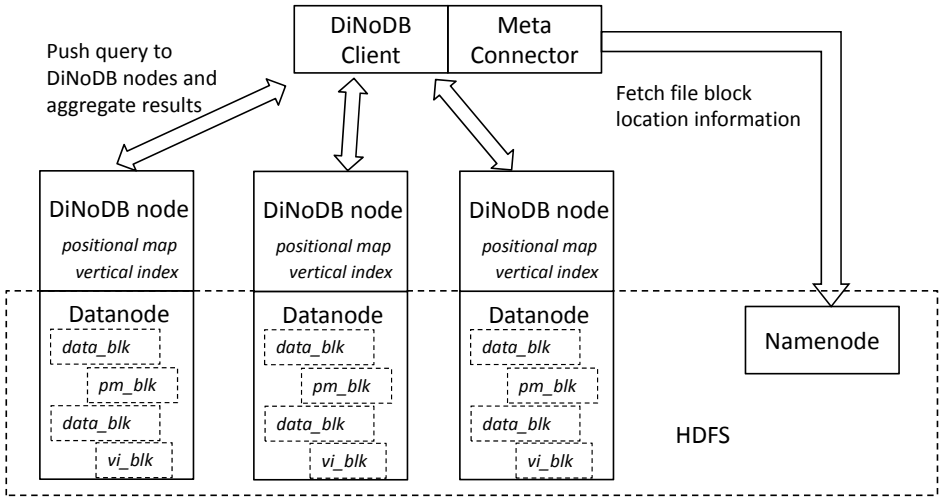


Figure 5: Architecture of the DiNoDB interactive query engine.

DiNoDB clients accept application requests (queries), and communicate with DiNoDB nodes. When a DiNoDB client receives a query, it fetches the metadata for the "tables" (output files of the batch phase) indicated in the query, using the *MetaConnector* module. The MetaConnector (see Figure 5) operates as 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 HDFS metadata, the MetaConnector guides the DiNoDB clients to query the DiNoDB nodes that hold raw data files relevant to the user queries. Additionally, the MetaConnector remotely configures DiNoDB nodes so that they can build the mapping between "tables" and the related HDFS 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) the DiNoDB nodes process the query in parallel; and finally, iii) the DiNoDB client aggregates the result.

Note that, since the 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. We will discuss DiNoDB fault-tolerance in more details.

DiNoDB nodes

The DiNoDB nodes are based on PostgresRaw [3], a query engine optimized for in-situ querying. In the following, we first briefly recall how PostgresRaw differs from native PostgreSQL and then explain all the details behind the DiNoDB nodes, and differences with respect to PostgresRaw.

PostgresRaw. PostgresRaw is a centralized instantiation of the NoDB paradigm [3], and is a variant of PostgreSQL that avoids the data loading phase and executes queries directly on data files. Conventional DBMS are designed with the assumption of loading the data into the database before querying, even if only a few attributes in a table are needed. The loading process is a significant investment both in terms of time and computing resources and might take hours for large amounts of data. PostgresRaw on the other hand, adopts in-situ querying instead of loading and preparing the data for queries. PostgresRaw builds positional maps on-the-fly, during query processing, which are used to speed up later queries.

From PostgresRaw to DiNoDB node. DiNoDB nodes instantiate customized PostgresRaw databases which execute user queries, and are co-located with HDFS DataNodes. 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 new file reader mechanism that can access data on HDFS and maps a "table" to a list of 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 in *shared memory*, such that user queries – which are sent through the DiNoDB client – can benefit from them across multiple sessions.

DiNoDB nodes can take advantage of the fact that data is naturally partitioned into HDFS blocks to leverage modern multi-core processors. Hence, data and the associated metadata can be easily accessed by multiple instances of PostgresRaw, to allow node level parallelism.

Additionally, DiNoDB nodes can access raw data bypassing the HDFS client API. This design choice boosts performance by avoiding the overheads of a Java-based interface. More importantly, DiNoDB users can selectively indicate whether raw data files are placed on disk or in memory. Hence, DiNoDB nodes can seamlessly benefit from a memory-backed file system to dramatically improve query execution times.

Exploiting metadata. DiNoDB nodes leverage positional map files generated in the DiNoDB pre-processing phase when executing queries. In

the case of the approximate positional maps, DiNoDB nodes use the sampled attributes as anchor points, to retrieve nearby attributes within the same row, required to satisfy a query. When vertical indexes are available, DiNoDB nodes use them to speed up queries with low selectivity, by employing an index-based access plan as a replacement for a full sequential scan. Both the positional map and the vertical index files are loaded by a DiNoDB node when the first query requires them. As our performance evaluation shows (see D.2.4.3), the metadata loading time is almost negligible, when compared to the execution time of the query.

Fault tolerance

In a nutshell, the key idea behind DiNoDB fault tolerance is to exploit HDFS n-way replication, in which every HDFS block on a given node is replicated to n-1 other nodes. As DiNoDB nodes co-locate with HDFS DataNodes, user queries can be directed to multiple nodes: in case one node is not available, DiNoDB clients automatically forward queries to other nodes with replicas of the data. By virtue of pro-active request redirection, DiNoDB can address the issues related to latency-tail tolerance [10].

However, the default HDFS replication mechanism is not suited for DiNoDB: indeed, HDFS does not guarantee data and metadata generated by DiNoDB plug-ins to be replicated on the same nodes. For example, some blocks assigned to DataNode D_1 may be replicated across DataNodes D_2 and D_3, whereas other blocks assigned to D_1 might be replicated differently, e.g., on DataNodes D_4 and D_5.

We thus proceed with the design of a new replication mechanism for HDFS that is tailored to DiNoDB, which achieves two objectives: (i) it co-locates data blocks with the corresponding metadata blocks and (ii) it allows selecting different "storage levels" for replicas, to save on cluster resources. To address data/metadata co-location, our DiNoDB prototype implements a *per-node* n-way replication. Every block assigned to a given DataNode D_i is systematically replicated across the same DataNodes D_j and D_k. We are aware that, on the long run, such simple approach may lead the HDFS subsystem to be poorly balanced. An alternative approach that we are currently considering is to create a new Hadoop Output Format that, similarly to Apache Parquet [5], supports "containers" that include data and metadata. Finally, DiNoDB supports different storage levels across replicas: as such, a "primary" replica can be tagged to be stored in the HDFS `ramfs` mount point, while "secondary" replicas are instead stored in an HDFS disk-based mount point [18].

2.3 Related work

Several research works and commercial products complement the batch processing nature of Hadoop/MapReduce [16, 11] with systems to query large-scale data at interactive speed using a SQL-like interface. Examples of such systems include HadoopDB [1], Vertica [25] or Hive [17]. 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. Such files are built in DiNoDB with a lightweight piggybacking mechanism for workloads involving a preliminary data processing phase such as machine learning and data exploration use cases.

Shark [26] presents an alternative design: it relies on a novel distributed shared memory abstraction called Resilient Distributed Datasets (RDDs) [27] to perform most computations in memory while offering fine-grained fault tolerance. Shark builds on Hive [17] to translate SQL-like queries to execution plans running on the Spark system [6], hence marrying batch and interactive data analysis. Recently, SparkSQL [21] was announced as the Shark replacement in the Spark stack, as the new SQL engine for Spark designed from ground-up. The main characteristic of SparkSQL is that, just like Shark, it works with Spark's RDDs. Hence, both SparkSQL and Shark require a variant of data loading, to transform the raw HDFS data file and bring it into the RDD representation, in order to benefit fully from the Sparks in-memory low-latency processing. In contrast, DiNoDB achieves low latency while working on raw files, entirely avoiding data loading.

Impala [9] is a state-of-the-art massively parallel processing (MPP) SQL query engine that runs in Hadoop. As such, Impala is probably the closest system to DiNoDB. However, while collocated with Hadoop, Impala does not leverage the possible synergy between batch processing of Hadoop and analytics power of a MPP SQL query engine. In this paper, we exactly propose such a synergy, and build DiNoDB to validate our approach. Namely, in DiNoDB, unlike in Impala, batch processing is used to generate metadata (positional maps and vertical indexes) that helps expedite SQL queries. We demonstrated in this paper that this synergy between batch processing and query engines, brings a lot of benefits and considerably reduces latency.

PostgresRaw [3] is a centralized DBMS that avoids data loading and transformation prior to queries. DiNoDB leverages PostgresRaw as a building block to obtain DiNoDB nodes, which are to be seen as upgraded version of PostgresRaw (see Section 2.2.3 for detailed comparison between DiNoDB nodes and PostgresRaw). A critical difference between PostgresRaw and DiNoDB system is that, DiNoDB is a distributed, massively parallel system for large-scale data analytics integrated with Hadoop batch processing framework, whereas PostgresRaw is a centralized database.

Finally, DiNoDB shares some similarities with several research work that focuses on improving Hadoop performance. For example, Hadoop++ [13] modifies the data format to include a Trojan Index so that it can avoid full file sequential scan. Furthermore, CoHadoop [14] 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.

To conclude, we compare our current version of DiNoDB with a preliminary version, presented in [4]. The preliminary version of DiNoDB used HadoopDB to orchestrate DiNoDB nodes, which revealed very costly due to inherent overhead of the HadoopDB framework when it comes to interactive queries. Therefore, we replaced HadoopDB with a massively parallel architecture, but had to come address fault tolerance of such an architecture. Moreover, our current version of DiNoDB introduces the support for multi-cores, speeding up DiNoDB system further. As a result, our current version of DiNoDB outperforms state-of-the-art analytics systems such as SparkSQL and Impala, which was not the case with our preliminary version.

2.4 Conclusion

Parallel data processing systems such as Hadoop MapReduce have received increasing attention from the industry, for their promise to analyze any amount or kind of data. However, with such systems, users had to move and load the data produced in the batch-processing phase into a fast relational database, to achieve interactive-speed data manipulation. The specter of "leaving something important behind" related to data movement and adaptation has led academics and the industry to address the problem of designing new systems to overcome the limitations of the batch-oriented nature of Hadoop MapReduce, that would expose a standard, interactive way to manipulate data, while being fully integrated in a growing ecosystem of data processing tools.

In this work we presented the architecture of DiNoDB: a distributed sys-

tem tuned for interactive queries on raw data files generated by large-scale batch-processing frameworks. As shown by our extensive experimental evaluation that we performed with a prototype implementation, DiNoDB offers performance superior to the current state-of-the-art, thanks to its modular design based on a plug-in mechanism that attach to Hadoop MapReduce and piggybacks the creation of auxiliary metadata required for interactive-speed query performance. In addition, such plug-in mechanisms allow DiNoDB to seamlessly integrate with existing frameworks and distributed storage systems. Our experimental evaluation, that we do on both synthetic and real-world datasets, highlights the key benefits of DiNoDB in a number of prominent use cases, *i.e.,* a machine learning "development" workload and data exploration.

Our current research agenda include work to extend the key concepts of DiNoDB plug-ins in various ways. First, we plan to address the integration of such plug-ins, and as a consequence of the DiNoDB system, to other frameworks. For example, by design, DiNoDB plug-ins can be used when the batch processing phase is performed on an Apache Spark cluster: in this case, plug-in configuration could be improved by extending the SparkSQL high-level language to support the specification of positional maps and vertical indexes. Similarly, we plan to extend the Apache Pig system and the Pig Latin language to support (and instruct) DiNoDB plugins. Finally, we will focus on adding support for Apache Parquet files in DiNoDB.

3 Conclusion

In WP4, we designed, implemented, and evaluated the distributed data storage management subsystem for the BigFoot stack. Our proposed BigFoot storage layer is scalable, as it builds upon the foundations of distributed file systems and distributed database systems, reliable, as it has no single points of failure, provides high performance, as it exploits storage media heterogeneity to improve application workloads, and is fully integrated with rest of the BigFoot stack. In order to achieve all the aforementioned goals, we designed and implemented novel solutions to solve performance, scalability, and reliability problems that arise at several granularities ranging from a single node in a data center to multiple data centers.

This deliverable contains a series of Appendices that provide installation instructions and links to source or binary code of our software deliverables.

Summary of Contributions

Below we summarize the research contributions made by WP4 over the course of the BigFoot project to highlight how we advanced state-of-the-art in both database and systems communities.

- **Single Node storage design**
  - We proposed a novel data indexing scheme to account for SSD/flash vs. HDD heterogeneity tailored for timestamped workloads called Bloom Filter Tree (BF-Tree). Using real-world datasets provided by GridPocket, we showed how one could save a substantial amount of storage space in SSDs by marginally sacrificing indexing accuracy using BF-Tree.
  - We adapted the NoDB processing model on a single node for efficient raw data processing in the context and the new requirements of the BigFoot project. We showed how we large-scale datasets, like those provided by Symantec, could be queried directly in raw format at a performance comparable with traditional databases, thereby reducing the query-to-security-insight time significantly.
- **Scalable raw data stores for datacenters (DC)**
  - We designed, implemented, and evaluated DiNoDB, a novel distributed interactive query engine to meet the big-data scalability requirements of the BigFoot project. By using NoDB as

the single-node backend database, DiNoDB extends the in-situ query processing benefits of NoDB to the batch-processing-based Hadoop world. By co-partitioning data with HDFS DataNodes and collocating NoDB engine together with HDFS Data Nodes, DiNoDB brings efficient raw data querying capabilities close to the location of data.

- We showed how DiNoDB accelerates query performance for BigFoot workloads by seamlessly integrating metadata generation with the batch-processing phase.

- **Scalable and reliable storage across datacenters (xDC)**

  - We presented GRAF, an efficient geo-replication protocol to protect critical metadata against datacenter-level failures. We showed how GRAF can tolerate arbitrary faults beyond crashes when they do not simultaneously co-exist with network partitions among correct replicas. We also compared GRAF with state-of-the-art BFT protocols like Zyzzyva, PBFT and aWAN-optimized version of Paxos across the Amazon EC2 datacenters and showed that GRAF significantly outperforms PBFT and Zyzzyva and performs almost as good as the optimized Paxos.

References

[1] Azza Abouzeid et al. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In *VLDB*, 2009.

[2] Azza Abouzied et al. Invisible loading: Access-driven data transfer from raw files into database systems. In *EDBT*, 2013.

[3] Ioannis Alagiannis et al. NoDB: efficient query execution on raw data files. In *SIGMOD*, 2012.

[4] Anonymous. Anonymized for double-blind reviewing.

[5] Apache Parquet. Webpage. `http://parquet.incubator.apache.org/`.

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

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

[8] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3:993–1022, March 2003.

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

[10] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2), February 2013.

[11] Jeffrey Dean et al. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX OSDI*, 2004.

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

[13] Jens Dittrich et al. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). In *VLDB*, 2010.

[14] Mohamed Y. Eltabakh et al. CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop. In *VLDB*, 2011.

[15] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *DMTCS Proceedings*, 0(1), 2008.

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

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

[18] KR Krish, Ali Anwar, and Ali R Butt. hatS: A Heterogeneity-Aware Tiered Storage for Hadoop. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 502–511. IEEE, 2014.

[19] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *ACM SOCC*, 2014.

[20] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *IEEE MSST*, 2010.

[21] SparkSQL. Webpage. `https://spark.apache.org/sql/`.

[22] Stado. Webpage. `https://launchpad.net/stado`.

[23] Yee W Teh, David Newman, and Max Welling. A Collapsed Variational Bayesian Inference Algorithm for Latent Dirichlet Allocation. In *Advances in neural information processing systems*, 2006.

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

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

[26] Reynold S. Xin et al. Shark: SQL and Rich Analytics at Scale. In *ACM SIGMOD*, 2013.

[27] Matei Zaharia et al. Spark: Cluster computing with working sets. In *in Proc. of USENIX HotCloud*, 2010.

[28] Matei Zaharia et al. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *ACM SOSP*, 2013.

A The GRAF Protocol

The source code of GRAF, which is described in detail in Section **??**, is included as a software deliverable and can be downloaded from following `http://www.eurecom.fr/~lius/bigfoot-graf.html`.

B BF-Tree: Approximate Tree Indexing

In this section we provide details on how to compile, configure and run the source code used to perform the experiments related to the BF-Tree. In order to run tests with the BF-Tree prototype:

1. update the indexIO.config file in order to have:
   – in the first line the path to an HDD file to be used for index IO.
   – in the second line the path to an SSD file to be used for index IO.
   – in the third line the max size in number of 4K pages of this file.

2. compile the using the appropriate flags:
   -D_HAS_DUMMY_IO has IO for the index (to run experiment with the index on storage).
   -D_USE_DIRECT for using direct IO (to make sure all IOs hit the storage).
   -DPAYLOAD_SIZE=XXX for giving the payload size of the generated data.

3. Usage: ./BF-Tree <function_options> <options>

   <function_options>: (choose only one)
   -g <NoTuples> (generate synthetic worokload)
   -L <load_from_TPCH> (load from TPCH files)
   -G <load_from_GRIDP> (load from GRIDP files)
   -e <filename> (print the info of a data file)
   -c <NoPages> (run detailed comparison using B+-Tree and BF-Tree)
   -s <NoPages> (scan data)

   <options>:
   -l <file_location> (data file location, not needed for -e)
   -f <false_positive> (use one of the four -f/-F/-E/-X to give false positive configurations)
   -F <multiple_false_positives> (use the format: min_fp: step:max_fp, where max_fp=min_fp*$step^k$)
   -E <multiple_false_positives> (use the format: exp_of_min_fp: step_of_exp: exp_of_max_fp, where max_fp=$10^e xp$_of_max_fp)
   -X <multiple_false_positives> (use the format: exp_of_min_fp: num_of_levels: exp_of_max_fp)
   -p (enable prompts for interactive execution/debugging)
   -v <verbosity> (0:NO, 1: YES)

-n <min_value_for_exp> (def:0, Give minimum size of exp - in order to create results with no hits)
-m <max_tuples_for_exp> (Give file size for experiments in # tuples)
-S <emulate_index_storage> (0:NO, 1:SSD, 2:HDD)
-I <index_storage_latency> (simlated Index IO cost in us)
-w (use warm caches - only leaves cause IO)
-h (If set use 5 hash functions, else 3.)
-r <repetitions_of_exp>
-a <column> (Select indexed column 0-1, 0 for RID, 1 for data, synthetic, TPCH, or GRIDP)
-H <id> (0: hide nothing, 1: show only BPT, 2: show only BFT)
-T (use TPCH date generator, used when running experiment with TPCH)
-K <filename> (use file for possible keys)
-R <hit_rate> (GRIDP queries hit Rate, used to select hit rate for GRIDP data)

4. In order to load TPCH data to a binary file the format of the input text file needs to contain rows of three dates (orderdate, commitdate, shipdate) ordered by orderdate.

5. In order to load GRIDP data to a binary file the format of the input text file needs to contain rows of five values (id, date, value, type, sensorid) ordered by id.

6. The GRIDP keys file (used with -K) contains rows each with a timestamp in the format YYYYMMDDHHmmSS, e.g., 20110805134947

C DiNoDB

The DiNoDB software deliverable comes as a virtual machine image that can be seamlessly deployed by the BigFoot Analytics-as-a-Service module.

Please refer to Deliverable D.5.4 to understand how Apache OpenStack Sahara is used to download and deploy DiNoDB.

For reference, the link to the virtual machine image embedding DiNoDB is available below:

`https://github.com/bigfootproject/sahara-image-elements`