2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications

Data Interface All-iN-A-place (DIANA) for Big Data

Frank Zhigang Wang
School of Computing
University of Kent
UK
Contact author:
frankwang@ieee.org

Theo Dimitrakos
British Telecommunication
UK

Na Helian
SCS
University of Hertfordshire
UK

Sining Wu
Xyratex
Havant
UK

Yuhui Deng
University of Jinan
China

Ling Li
School of Computing
University of Kent
UK

Rodric Yates
Hursley Lab.
IBM
UK

Abstract: **"Variety" in Big Data means we have a wide range of data types and sources: e.g. file systems and database systems co-exist for decades as two popular data-accessing interfaces. This work is to unify these two interfaces by presenting a Data Interface All-iN-A-place (DIANA). The first challenge lies in distinguishing structured and un-structured data and diverting them to different underlying platforms. It is demonstrated that a speedup of 5000 in indexing has been achieved at the expense of a slowdown of 100 in extracting attributes. A DIANA-based cloud storage system is constructed for versatile, long distance and large volume big data accessing operations to address "Volume" and "Velocity" in Big Data. It encapsulates a dynamic multi-stream/multi-path engine at the socket level, which conforms to Portable Operating System Interface (POSIX).**

***Keywords:** big data; variety; volume; velocity; file systems; database systems; service-oriented architecture*

I. Introduction

4Vs (volume, velocity, variety and value are four defining properties or dimensions of big data, out of which variety refers to the number of types of data [1]. Based on the above 4Vs model, the challenges of big data management come from all four properties, rather than just the volume and velocity.

File systems and database systems are two main stream platforms in terms of interfacing applications and storage devices. Computers can store information on several different storage media, such as magnetic disks, magnetic tapes, and optical disks. File systems and databases provide a uniform logical view of information storage to abstract from the physical properties of its storage devices.

A file system replies on POSIX (IEEE Std 1003.1-2001) VFS (virtual file system or virtual filesystem switch) to support applications [2]. The purpose of a VFS is to allow client applications to access different types of concrete file systems in a uniform way. A VFS can, for example, be used to access local and network storage devices transparently without the client application noticing the difference. It can be used to bridge the differences in Windows, Mac OS and Unix filesystems, so that applications can access files on local file systems of those types without having to know what type of file system they're accessing. One of the first virtual file system mechanisms in Unix-like systems was introduced by Sun Microsystems in SunOS 2.0 in 1985.

SQL (Structured Query Language) is a standard interactive and programming language for querying and modifying data and managing databases [3]. SQL was adopted as a standard by ANSI in 1986 and ISO in 1987 [4]. The SQL standard has gone through a number of revisions: SQL: 1999 (SQL3) added support for procedural and control-of-flow statements and ISO/IEC 9075-14:2006 defines ways in which SQL can be used in conjunction with XML [4].

Why are these two platforms formed historically? What is the difference between a filesystem and a database?

File-based systems were an early attempt to computerize the manual filing system that we are all familiar with. From the end-user's point of view, file systems proved to be a great improvement over manual systems. Simply speaking, a file is a stream of bytes, which are typically un-structured. An example of a file could be a Text File (a collection of alphanumeric characters that, when put together, form a readable document) or a Bitmap Image File (a collection of bytes that software would then interpret as pixels of an image).

There are a number of problems with file systems [4]:

- Separation, isolation and duplication of data. Owing to the decentralized approach, a file system encourages the uncontrolled separation, isolation and duplication of data.
- Data dependence or Incompatible file formats. The structure of files is embedded in the application programs.
- Fixed queries/proliferation of application programs. File systems are very dependent upon the application developer, who has to write any queries or reports that are required.
- No provision for security or integrity;
- Recovery, in the event of a hardware or software failure, was limited or non-existent.

All the above limitations of file systems can be attributed to two factors: (1) the definition of the data is embedded in the application programs, rather than being stored separately and independently; (2) there is no control over the access and manipulation of data beyond that imposed by the application programs.

To become more effective, a new approach of managing data was required. What emerged were the database systems. A database is both a program to store and organize data, and make it searchable, and the data contained in it. A database holds many tables and each table can hold many records as well as fields. Each table in a database requires one field to be designated as the Primary Key to uniquely identify a record in a table.

Therefore, all databases are files, but not all files are databases. A file or a database table is just a logical storage unit. File systems are easy-to-use, un-structured, OS-resident (easy-to-obtain) and easy-to-maintain. Databases tend to be large but have strong data definition and manipulation capabilities including query, search, sorting and calculation. There two platforms co-exist for decades playing complementary roles: for example file systems are well-used to store un-structured text and image documents whereas databases are designed to handle high transaction throughput such as on-line transaction processing (OLTP) in order-entry, stock control, accounting, banking, financing, etc.

Why not a pure database system? The answers are probably as below:

- Scientific applications are usually based on a POSIX API. Many tools are scripts or compiled programs that might be difficult to modify to use a database.
- Users are accustomed to a POSIX API.
- Databases are good at storing structured data, but most don't store large unstructured data well.

Why filesystems alone aren't a solution? Traditional B+-tree and hashing are not suitable for multidimensional data as they can handle only one dimensional data. Using multiple B+-trees (one per dimension) or space linearization followed by B+-tree indexing are not efficient solutions. We need multidimensional index structures: those that can index data based on multiple dimensions simultaneously, sometimes beyond 10-15 dimensions in modern data-intensive applications like multimedia retrieval (e.g., 64-d color histograms), data mining/OLAP (e.g., 52-d bank data in clustering) and time series/scientific/medical applications (e.g., 20-d Space Shuttle data, 64-d Electrocardiogram data) [5].

II. The DIANA Vision and Uniqueness

2.1 The DIANA vision

As shown in Fig.1, DIANA encapsulates POSIX, SQL and an extensible interface reserved for metadata. In DIANA, file and database operations are unified into a uniform interface. That is to say, DIANA provides uniform access to unstructured data stored in files and tabular data stored in databases.

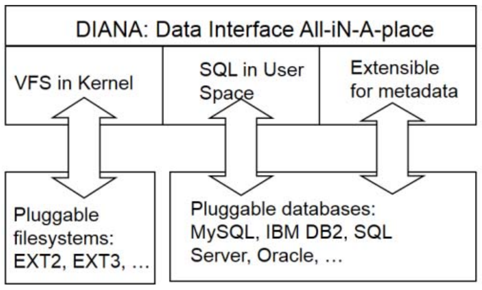
DIANA: Data Interface All-iN-A-place
VFS in Kernel
SQL in User Space
Extensible for metadata
Pluggable filesystems: EXT2, EXT3, …
Pluggable databases: MySQL, IBM DB2, SQL Server, Oracle, …

Fig.1 DIANA encapsulates POSIX VFS and SQL standard interfaces as well as an extensible interface reserved for metadata operations. DIANA provides uniform access to pluggable filesystems and databases.

DINAE has a tighter coupling between files and database tables, than provided by a separate file system and a database. It supports the frequent interactions and great synchronicity between data and metadata. For example, while creating a file, an entry for the new file will be made in the tabular directory. The directory entry records the name of the file and the location in the file system, and possibly other provenance metadata. In many domains provenance increases an object's value [6].

Metadata such as provenance is typically stored in standalone database systems, maintained in parallel with the data to which it refers to. Separating provenance from its data introduces problems such as: ensuring consistency between the provenance and the data, enforcing provenance maintenance, and preserving provenance during backup, restoration, copies, etc [6]. Provenance should ideally be maintained by a unified platform such as DIANA, since provenance is merely meta-data and DIANA is equipped with powerful manipulation capability to query, index and manage meta-data.

DIANA provides the following features:

- DIANA generates system-level metadata automatically. Application-level solutions have to involve users to manually collect metadata. In other words, it delays provenance collection, performing it at user-level by writing it to an external database.
- DIANA provides tight coupling between data and metadata on the system level. Application-level solutions have to involve users to synchronize data and metadata.

- While writing a file, given the name of the file, DIANA searches the tabular directory via its SQL interface to conveniently and quickly find the location of the file. A pointer is provided to the location in memory where the content to be written is kept. To read from a file, again, the directory is searched via SQL for the associated directory entry.

2.2 Related works and our innovation

OGSA-DAI (Data Access and Integration) is perhaps one of the most useful and successful Globus components [7]. Developed in the UK, it provides uniform Web Services interfaces to diverse data resources. These interfaces allow clients not only to "access" data, but also to query, update, transform, and deliver it. In other words, they let you specify some pretty fancy server-side operations [8]. Audit records generated during job execution are stored in a database and can subsequently be retrieved by (authorized) clients. OGSA-DAI is used to create a virtual database from internal audit and accounting databases. The value of the OGSA-DAI abstractions and implementation has been positively evaluated [9]. However DAI is just a universal interface for heterogamous database products.

The PVFS (Parallel Virtual File System) serves as both a platform for parallel I/O research as well as a production file system for the cluster computing community. PVFS supports the UNIX I/O interface and allows existing UNIX I/O programs to use PVFS files without recompiling [10]. The familiar UNIX file tools (ls, cp, rm, etc.) will all operate on PVFS files and directories as well. This is accomplished via a Linux kernel module which is provided as a separate package [10]. In the Sloan Digital Sky Survey or SkyServer project, Carnegie Mellon University and Los Alamos Lab, together with an astronomy community, have added multidimensional extensions on SQLite DB to PVFS [5]. Such a multidimensional filesystem is one which also indexes and allows efficient access to files based on their meta-data tags. Anyway, PVFS is an enhanced file system with a multi-dimensional index extension.

Provenance-Aware Storage System (PASS) originated by Hardvard University is a storage system that automatically collects and maintains provenance or lineage, the complete history or ancestry of an item [6]. PASS manages its provenance database directly in the kernel and extends SQL to support lineage and accuracy information when requested by a user or application. PASS provides useful provenance-aware functionality via the conventional filesystem interface. In short, PASS is a storage system with the functionality not available in today's file systems or provenance management systems.

To our best knowledge, Data Interface All-iN-A-place (DIANA) is the first attempt to unify the two popular interfaces: filesystems and databases. DIANA is expected to provide the advantages of both worlds.

III. DIANA Implementation

To implement DIANA (Fig.1), an interface needs to be designed first, which should include system calls in the form of functions to universally store, index and query all types of data objects, no matter if they are structured, semi-structured or un-structured. A system call is the mechanism used by an application program to request service from the operating system.

On Unix-based and POSIX-based systems, popular system calls are open, read, write, close, wait, exec, fork, exit, and kill.

SQL allows a user to create the database and table (relation) structures; perform basic data management tasks, such as the insertion, modification, and deletion of data from the tables; perform both simple and complex queries.

3.1 DIANA interface design

DIANA encapsulates POSIX VFS and SQL standard interfaces as well as an extensible interface reserved for metadata input/query. DIANA provides uniform access to unstructured data stored in files and tabular data stored in databases.

POSIX consists of both operating system interfaces and shell/utilities. Six basic file operations are provided to create, write, read, reposition, delete, and truncate files. We have identified u_create (open), u_write, u_read and u_delete functions ("u" stands for "universal") in our prototype implementation. As listed in the table, their corresponding functions in SQL are: create table, load/insert/update, select and drop table. Search against a certain criteria has not been defined in POSIX but it could be implemented as a shell command. The corresponding function in SQL is the select function with a "where" clause.

The following operations in DIANA are highlighted:

u_create (dataset_name);

It creates an entry in the Global Multi-dimensional Index Facility (GMDIF) under the current user's account. First, space in the file system or the database must be found for the newly-created object. Second, an entry for the new object must be made in the GMDIF directory. The directory entry records the name of the object and the location in the file system or database, and possibly other information.

u_write (dataset_name, location/object);

It writes an assigned data object (a text, an image, or tabular data with an extension) from the path to a new dataset_name. To write an object, we specify the name of the object and the dataset_name to receive this object. Given dataset_name, DIANA searches the GMDIF directory to find the location of the dataset_name. A pointer is provided to the location in memory where the content to be written is kept. The corresponding metadata is also written to the GMDIF index automatically and transparently.

u_read (location, dataset_name);

It reads an existing dataset_name (a text, an image, or tabular data with an extension) to the location. Again, the GMDIF directory is searched for the associated directory entry.

u_search (dataset_name, 'key1' 'key2'…);

It performs a multi-dimensional search, returning a GMDIF location of one or more dataset_name that matches the provided keys. The keys could be a number of keywords of a text, the current date/time and GPS location information of a photo, and any attribute values of a table, etc.

In principle, DIANA includes not only the above commonly-used operations of VFS and SQL but also all the pure VFS and SQL operations. In other words, it covers nearly all the data operations. It is universal. DIANA is also extensible in terms of reserving an interface for metadata-related operations, such as definition, interaction and synchronization.

3.2 A DIANA Prototype Implementation

Driven by the identified problems of "Variety", "Volume" and "Velocity" in Big Data, we implemented a prototype DIANA (Fig.2) in Linux 2.4.20. The implementation is approximately 5,000 lines of code. This prototype includes the POSIX VFS standard and the SQL standard. The challenge lies in distinguishing structured and un-structured data. For example, text and image data may ideally be processed and stored on files in a less-structured file system environment but the transactions and metadata (including provenance) should be separately operated on tuples within a database framework due to its power in data manipulation.

As shown in Fig.2, DIANA uses a switch to divert un-structured data to a file system and structured data to a database system. This switch distinguishes the extension of an input data object. For example, ".txt", ".doc", "jpg" and ".bmp" are categorized as un-structured data whereas ".sql", ".mdb" are structured data. A conservative policy has been adopted in DIANA, which means an un-recognized object will be treated as an un-structured one. A semi-structured object such as ".html" and ".xml" will also be viewed as an un-structured one. There may be performance degradation with this conservative policy. The overhead will be measured and evaluated in Section 4.

A further advanced switch is being implemented, which can scan the content of an unknown object to accurately distinguish its structure. This is a challenging work taking into consideration that there are enormous types of data objects. Like the above work, a conservative policy is thought to be still needed in case the distinguishing procedure fails.

As shown in Fig.2, Global Multi-Dimensional Index Facility (GMDIF) implemented on MySQL helps an end user find the files or databases he/she needs quickly. Traditional filesystems allow one to access files along a single dimension: that of the filename and path. However, filenames are frequently irrelevant in practice, in which analysis needs to be applied to all data with a certain set of attributes not a certain name. The GMDIF is a multidimensional index that universally locates a desired object across filesystems and databases based on its multiple meta-data tags (attributes).

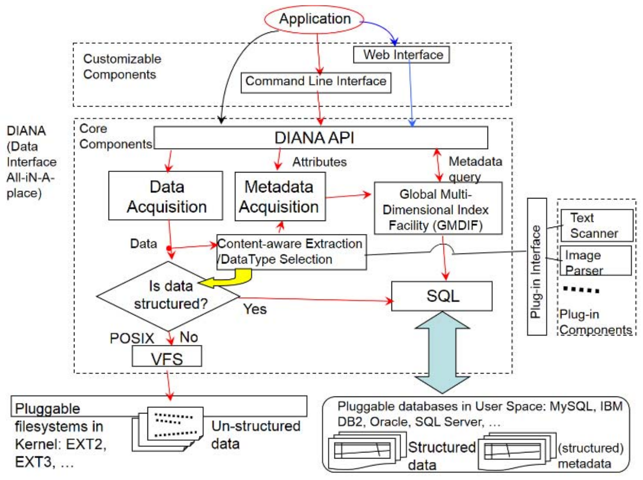

Fig.2 DIANA includes a switch to divert un-structured data to a file system and structured data to a database system. Global Multidimensional Index Facility (GMDIF) on MySQL help an end user find the files or databases he/she needs quickly. A channel is designed to penetrate the boundary between the user space and the kernel for synchronization and consistency purposes via a pair of inter-connected Kernel Demon and User Demon.

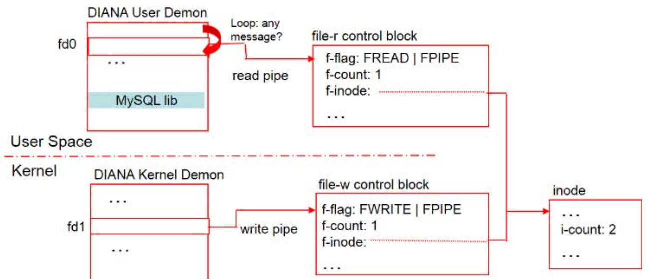

Fig.3 A pipe is used to connect the Kernel Demon and the User Demon, one end of which is written by the Kernel Demon and another end of which is read by the User Demon.

In hybrid filesystem/database DIANA, MySQL is not only used to store structured data objects, but also to index and query metadata referring to all saved objects. This is an embedded solution with low total cost of ownership. All 'normal' metadata (POSIX attributes, file sizes, etc.) are indexed. DIANA also allows application-specific metadata (e.g., the current time/date and the GPS location of a photo) to be added as extended attributes for any object indexed by the GMDIF. Attributes are asynchronously written to GMDIF. Queries are SQL style query strings. Expressiveness limited only by application metadata tags. Clients collate and report results.

The metadata interface is designed to enable that user to input and query these metadata. The interface may also automate the collection of provenance associated with data and their operations, which can be used to further boost the GMDIF.

The challenge also lies in establishing a channel penetrating the boundary between the user space and the kernel for synchronization and consistency purposes. As shown in Fig.2, a kernel-memory module, the DIANA Kernel Demon, acts as a VFS interface. The DIANA server nominates a user-space daemon, the DIANA User Demon, to communicate with the Kernel Demon. The VFS is implemented in the kernel. This implementation conforms naturally to the standard POSIX semantics and provides applications with seamless access to DIANA. A request is linked into the VFS's request queue by a kernel thread and is then swept up in a perpetual loop supported by the above Kernel Demon and the User Demon. A copy of the request is transferred to the user space from within the kernel. It dives repeatedly into the kernel to copy the data, then transmits it in standard SQL code.

In DIANA, the above-mentioned pair of the Kernel Demon and the User Demon is connected by three different message/data passing mechanisms for different considerations. The first mechanism is a pipe, as shown in Fig.3, one end of which is written by the Kernel Demon and another end of which is read by the User Demon. The second mechanism is a message queue, in which each message generated by the User Demon stays until the Kernel Demon reads it. The third is a new mechanism, which we call “Data Window” (Fig.4). The Data Window mechanism exceeds the space limit (32 MB) of the well-used IPC shared memory (in this means we focus our attention on the bulk data transfer). Like the IPC shared memory, the implemented “Data Window” mechanism also avoids copying data between the user space and the kernel space. A tighter coupling between files and database tables, than provided by a separate file system and a database, is easily guaranteed in DIANA, which supports the frequent interactions and great synchronicity between data and metadata.

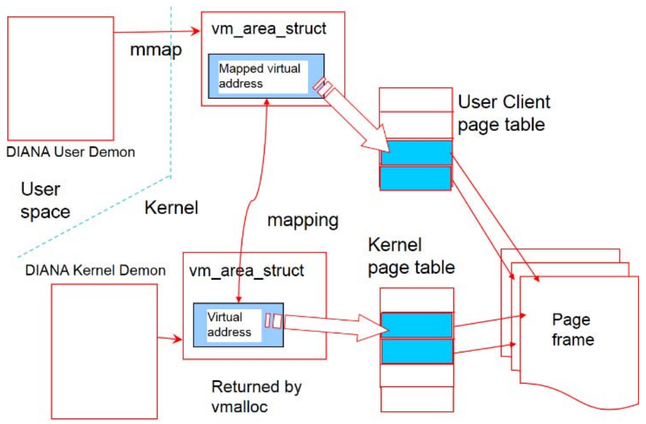

Fig.4 The invented ”Data Window” mechanism breaks the space limit (32 MB) of the well-used IPC shared memory. A driver maps a virtual address to the User Demon’s user space (page table), which allows the User Demon and the Kernel Demon to access some common data structures.

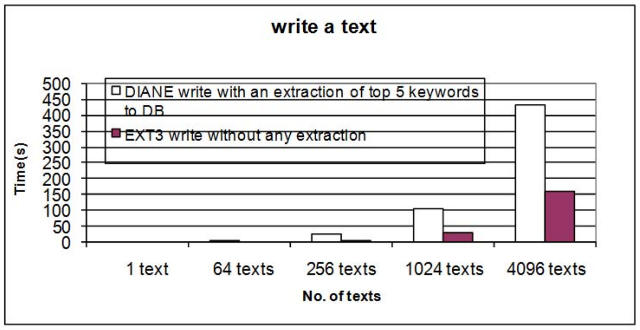

Fig.5 Graph of write time versus the number of texts.

IV. DIANA Evaluation

The purpose of this evaluation was to examine the alpha release of the DIANA code, and to test and compare its performance with that of traditional standards. The local file system was configured as EXT3. We have selected EXT3 as the candidate for comparison for two reasons: 1. EXT is mature and de facto in the Unix/Linux user community; 2. EXT and DIANA-FS can run on precisely the same hardware and OS. They can, in fact, coexist on the same machine and be used simultaneously. Using EXT allowed us to conduct controlled experiments in which the only significant variable was the file system component. The performance differences we observed were due to the design and implementation of the file systems and were not artifacts of hardware, network, or OS variation.

We evaluated our DIANA prototype on a 1 GHz Dell machine with 1024MB of RAM, 80GB of a SATA disk drive, running RedHat 7.3. To quantify the overhead of our system, we took measurements on both a DIANA and a non-DIANA system. We obtain results marked “DIANA” by running our DIANA interface on EXT3FS and MySQL. We obtain non-DIANA results, marked “EXT3”, running on Linux 2.4.20 kernel and EXT3FS.

We will measure the overhead of typical data-accessing operations (u_write, u_read, u_search, etc.). Ten trials are used to generate each data point. In nearly all cases, the standard deviations were less than 5%. Measurements are carried out in a cold cache environment unless stated. To ensure a cold cache, we reformatted the file system on which the experiments took place between test runs. For each file read/write mechanism, we transferred a set of objects numbering from 1 to 4096.

4.1 Text operations

When a new object is written not only the data need to be stored but also the metadata information is stored in the database. The overhead time to extract the top five most frequent keywords from a text document (.txt) of 611,235 Bytes and add them to the database is included in the u_write operation of that document. We have measured the overhead imposed by the DIANA interface. Graph of write time versus the number of texts is shown in Table 1 and Fig.5. Although a slowdown (the reciprocal of speedup) of 1.4 – 17.7 is shown, note that EXT3 does not extract any keywords during a write operation.

Table.1 Write time of in seconds versus the number of texts.

No. of Texts	DIANA write with an extraction of top 5 keywords to DB	EXT3 write without any extraction	Slowdown
1 text	0.172	0.121	1.4
64 texts	6.491	0.366	17.7
256 texts	26.754	2.778	9.6
1024 texts	104.622	30.117	3.5
4096 texts	434.31	159.154	2.7

A comparative behaviour of multi-dimensional indexing in a filesystem needs to be measured. Unfortunately, today's file systems do not support multi-dimensional indexing. When a file is created, an entry in the directory tree is added. The directory entry records the name of the file and other information. We changed the file name format as a concatenation of the selected attributes, as illustrated in Fig.6. The advantage of changing the filename format is that we don not need to modify the directory tree structure in a filesystem. The query time of a multi-dimensional "find" by scanning all the extended filenames in EXT3 is included in Table 2. It takes 242 seconds to generate those 4096 texts' extended filenames in EXT3. The multi-dimensional query time by scanning all saved texts in EXT3 is also included. A speedup of 4800 has been achieved. The overhead of extracting attributes to GMDIF while writing has been paid off.

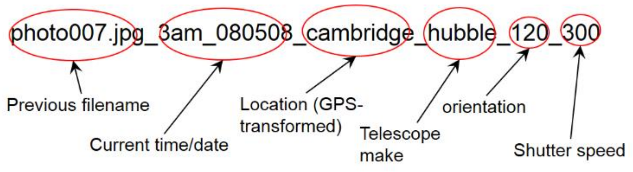
photo007.jpg_3am_080508_cambridge_hubble_120_300
Previous filename
Current time/date
Location (GPS-transformed)
Telescope make
orientation
Shutter speed

Fig.6 A multi-dimensional search in a traditional filesystem can be performed by changing the filename format as a concatenation of the selected attributes.

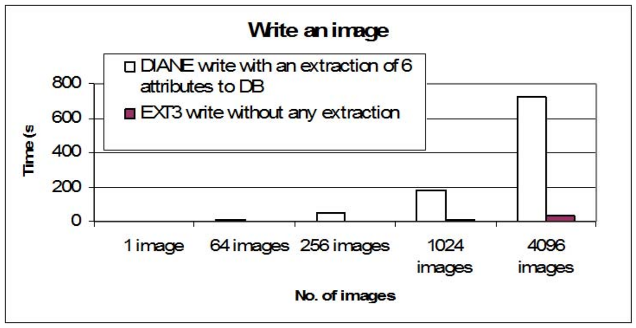

Fig.7 Graph of write time versus the number of images.

Table.2 Search time (s) of 4096 text entries against No. of attributes. The speedup is the time of EXT search (scanning texts) over that of SQL search.

No. of attributes	1	2	3	4	5
EXT search by scanning all the saved texts	202.4	202.5	202.5	202.6	202.6
EXT search by scanning all the extended filenames	0.083	0.084	0.084	0.084	0.084
SQL search in GMDIF	0.042	0.043	0.041	0.045	0.044
Speedup	4819	4707	4937	4498	4600

4.2 Image operations

The overhead time to extract six selected tags (attributes) from the header of a JPG image of 105,542 Bytes and add them to the database is included in the u_write operation of that image. Graph of write time versus the number of images is shown in Table 3 and Fig.7. Although a slowdown of 5-76 is shown, note that EXT3 does not extract any attribute during a write operation.

Table.3 Write time of in seconds versus the number of images.

No. of Images	DIANA write with an extraction of 6 attributes to DB	EXT3 write without any extraction	slowdown
1 image	0.286	0.054	5.3
64 images	10.99	0.202	54.4
256 images	43.598	0.573	76.1
1024 images	176.566	4.817	36.7
4096 images	720.202	29.433	24.5

In the above JPG image files, Exchangeable Image File Format (EXIF) is used to include metadata. EXIF is a specification for the image file format used by digital cameras. The specification uses the existing JPEG, TIFF Rev. 6.0, and RIFF WAV file formats, with the addition of specific metadata tags. An EXIF file header consists of a collection of tagged attribute/value pairs, some of which are provenance. The metadata tags defined in the EXIF standard cover a broad spectrum [11]:

• Date and time information. Digital cameras will record the current date and time and save this in the metadata.

• Camera settings. This includes static information such as the camera model and make, and information that varies with each image such as orientation, aperture, shutter speed, focal length, metering mode, and ISO speed information.

• A thumbnail for previewing the picture on the camera's LCD screen, in file managers, or in photo manipulation software.

• Descriptions and copyright information.

The EXIF format has standard tags for location information. Currently, only very few cameras, such as the Ricoh 500SE, have a built-in GPS receiver and store the location information in the EXIF header when the picture is taken. But GPS data can be added to any digital photograph on a computer, either by correlating the time stamps of the photographs with a GPS record from a hand-held GPS receiver or manually using a map or mapping software. The process of adding geographic information to a photograph is known as geocoding [11].

Whenever such an image file is transformed, additional metadata is added to this header. This approach addresses the challenge of making the metadata and data inseparable, but it introduces other disadvantages. It is expensive to search the attribute space to find objects meeting some criteria. In DIANA, extracting attributes of an image to the Global Multidimensional Index Facility (GMDIF) is expected to find the images quickly.

Similar to Section 4.1, the query time of a multi-dimensional "find" by scanning all the extended filenames in EXT3 is included in Table 4. It takes 638 seconds to generate those 4096 images' extended filenames in EXT3. The multi-dimensional query time by scanning the headers of all saved images in EXT3 is also included. A speedup of 5200 has been achieved. Again, the overhead of extracting attributes to GMDIF while writing has been paid off.

Table.4 Search time (s) of 4096 image entries against No. of attributes. The speedup is the time of EXT search (scanning headers) over that of SQL search.

No. of attributes	1	2	3	4	5
EXT search by scanning the headers of all saved images	621.2	621.3	621.3	621.3	621.4
EXT search by scanning all the extended filenames	0.395	0.395	0.396	0.396	0.396
SQL search in GMDIF	0.118	0.119	0.12	0.122	0.121
Speedup	5265	5221	5177	5093	5135

4.3 Exhaustive Search

We have measured the performance improvement compared with traditional approaches (B+-tree and hashing in filesystems). A Linux-2.4.20 source code tree is used as a searching target. After compiled, this source code tree has 21,777 file entries in total. A shell command "find" is first used search the tree for "Makefile" meeting a criteria of obj_size < 4096 bytes. This tree is then inserted into a MySQL table with 21,777 records (one-inode-per-record). A DIANA search operation is performed to search the tabular tree (the creation time of this tabular tree is 589.625 seconds) for the same object with the same criteria. Query time is shown in Table 5. A speedup of 410 has been achieved. The overhead of extracting attributes to DB while writing has been paid off.

Table.5 Comparison of query time between DIANA search and the shell command "find" against the Linux-2.4.20 source code tree with 21,777 file entries.

DIANA search	Shell find	speedup
0.067s	27.534s	410

We have also measured the dependency of operation overhead in a typical multi-attribute search on the number of attributes. It is observed that the performance of DIANA search behaves much more rapidly than the POSIX interface. Traditional B+-tree and hashing are not suitable for multidimensional data as they can handle only one dimensional data. A DB-based multidimensional index structures can index data based on multiple dimensions simultaneously. We have also increased the number of criteria in the search operation but no additional overhead is observed. This is because either the single-dimensional or multi-dimensional search is performed against a single index table. A simple sequential scan through the entire tabular index to answer the query is even faster than using a multidimensional B+-tree structure.

4.4 Tailoring Operations

The DIANA includes mechanisms for tailoring the input and output streams (typically images, audio or other multimedia objects). This is performed by associating a 'BLOB (binary large object)' datatype with the input and storing a collection of binary data as a single entity in the database. These conversion operations occur on the fly and are conveniently transparent to the user.

As mentioned in Section III, a conservative policy has been adopted in DIANA, which means an un-recognized object will be treated as an un-structured one. On the other hand, the performance may degrade if while writing or reading an un-structured object in a structured environment (a database) by mistake.

We deliberately inserted a JPG image (800x600 pixels, 105,542 Bytes) into a MySQL table in BLOB. A file is a stream of bytes. Every 32 KBytes of that image file are inserted into a record of the created database table used to receive that image. The comparison of writing time between DIANA and EXT3 is summarized in Table 6. DIANA is a universal interface that can process and store any type of data. Thanks to the above-mentioned conservative policy of treating an un-recognized object as an un-structured one, this universality may not result in degradation in performance.

Table.6 Comparison of writing/reading time of an image between DIANA and EXT3.

Operation	DIANA	EXT3	DIANA/EXT
write	0.271s	0.147s	1.84

read	0.031s	0.033s	0.94

4.5 Summary

DIANA provides functionality, unavailable in either a pure filesystem or a pure database, with moderate overhead. A speedup of 5000 in indexing has been achieved at the expense of a slowdown of 100 in metadata extracting. We and our users are satisfied with the performance.

V. Case Study: DIANA/CloudJet

We have constructed a DIANA/CloudJet system, in which a new data communication protocol (CloudJet) is designed for long distance and large volume big data accessing operations to alleviate the large latencies encountered in sharing big data resources in the clouds [12]. CloudJet encapsulates a dynamic multi-stream/multi-path engine at the socket level, which conforms to Portable Operating System Interface (POSIX) and thereby can accelerate any POSIX-compatible applications across IP based networks. A mixture of texts, photos and tables can be stored and indexed universally and efficiently via a graphic interface.

In our practice, service is interpreted as an environment in which an end user is immersed. In other words, service comprises all components except for the end user himself/herself within the framework. As a result, DIANA is featured with not only encapsulation of all resources but also transparent and automatic interactions between data and metadata.

We began the DIANA implementation with the simplest and lowest-level schema that could meet our query needs. In parallel with development of the prototype, we are also extending DIANA into the OS kernel to provide "micro-services" to application programs. Such micro-services can be used by an application program to request a universal DS5 storage space from the OS.

According to our investigations [13][14][15], a large number of applications, either legacy or newly-emerged, demand for file support as well as database support interactively. For example, in a provenance-aware system, the raw data may be processed and stored on files in a less-structured file system environment but, ideally, the provenance (metadata) should be separately operated on tuples within a database framework due to its power in data manipulation [6].

VI. Conclusions and Discussions

Just as the shipping container revolutionized the flow of goods [16], the Data Interface All-iN-A-place (DIANA) revolutionizes the flow of information for big data applications. As generic as a container can hold just about anything, from coffee beans to cellphone components, DIANA attempts to unify the two most popular data-accessing interfaces: filesystems and databases. By sharply cutting costs and enhancing reliability, container-based shipping enormously increased the volume of international trade and made complex supply chains possible. In a similar way, DIANA is expected to be service-oriented and make complex data accesses simple for big data management.

The overhead of extracting metadata from a data object and the performance improvement in typical multi-dimensional searches have been measured. It is shown that a speedup of 5000 in indexing has been achieved at the expense of a slowdown of 100 in extracting attributes, so the new features incur no perceptible cost. Typical big data applications such as very large database (VLDB), data mining, media streaming and office applications can be accelerated up to tenfold in real-world DIANA/CloudJet tests.

VII. Acknowledgments

This work is sponsored by the UK government and European Commission (EC) through an EPSRC/DTI grant (£ 1 million) "Grid-oriented Storage (GOS)", an EPSRC grant (£470k) "Accelerating NFS/CIFS", an EC grant (€1 million) "QuickLinux" and an EC grant (€400k) "EuroAsiaGrid".

References

[1] L. Douglas, "The Importance of 'Big Data': A Definition". Gartner. Retrieved 21 June 2012.

[2] A. Silberschatz, et al, Operating System Concepts, John Wiley & Sons, 2007

[3] SQL, en.wikipedia.org/wiki/SQL, 2008

[4] Connolly, T., "Database Systems: A Practical Approach to Design, Implementation and Management", Addison Wesley Longman, 2007

[5] Milo Polte, Finding the Needles in the Haystack: Multidimensional Extensions to a Distributed Filesystem, 2007

[6] K. Reddy, et al, Provenance-Aware Storage Systems, USENIX Annual Conference, 2006.

[7] Ian Foster, Data Access and Integration, 2006

[8] A. Sanchez. MAPFS-DAI, Future Generation Computer Systems 23 (2007) pp.138-145.

[9] M. Oever. The Use of OGSA-DAI with IBM DB2 Content Manager for Multiplatforms in the eDiaMoND Project. The Future of Grid Data Environments Workshop, GGF10, March 2004.

[10] Parallel Virtual File System, pvfs.org/, 2008

[11] EXIF, www.exif.org/, 2008

[12] Frank Wang, et al, CloudJet4BigData: Streamlining Big Data via an accelerated socket interface, IEEE Big Data, USA, June 2014

[13] In-Kernel Berkeley DB Databases, www.am-utils.org/project-kbdb.html, 2008

[14] Frank Wang, et al, Grid-oriented Storage: A Single-Image, Cross-Domain, High-Bandwidth Architecture, IEEE Transaction on Computers, ISSN: 0018-9340, pp.474-487, Vol.56, No.4, 2007.

[15] Yuhui Deng and Frank Wang, A Heterogeneous Storage Grid Enabled by Grid Service, ACM Operating System Review, ACM SIGOPS, No.1, Vol.41, 2007.

[16] The Container That Changed the World, VIRGINIA POSTRER, March 23, 2006