To appear in IEEE BigData 2014 (3rd International Congress on Big Data)

BigObject Store: In-Place Computing for Interactive Analytics

Yi-Cheng Huang, Wenwey Hseush, Yu-Chun Lai, Michael Fong
MacroData Inc.
{ychuang, wenwey, eugene.lai, michael.fong}@macrodatalab.com

***Abstract*—In order to address real-time analytic problems for big data, we have introduced a technology, *In-Place Computing*, a set of principles for storing and computing big data. It defines an abstract model in which data objects live and work on a flat and infinite address space. The term in-place indicates data is ready for computing in two ways: (1) in-memory and data-centric, in which computations take place where data resides, and (2) well-organized data objects, which have algebraic expressions and operators. Based on these principles, we have developed an in-place computing system, *BigObject Store*, which implements an extended relational model in the sense allowing various shapes of macro objects besides "tables" and meanwhile supporting algebraic operators besides "join". An abstraction, *Macro Data Structure*, is defined and a programming paradigm is used to support data-centric computing. Our experiments show that BigObject delivers promising results on typical analytic tasks.**

***Index Terms*—NoSQL; in-memory database; interactive analytics; big data; in-place computing;**

I. Introduction

The approach of cluster computing together with NoSQL [1], [2] such as MapReduce [3] with document-based/key-value stores [4] has successfully demonstrated a powerful way of storing and processing an unfathomable scale of unstructured/semi-structured data. From a spatial aspect, it assumes that target problems are *breakable* or *dividable* in the way that data can be broken into smaller datasets to be processed in parallel. From a temporal aspect, it practices a *transform-by-copy* paradigm (an analogy to call-by-copy [5]), where a task converts one dataset into another in a completely separate copy and then passes it to the next task as a read-only copy. This read-only property is critical to implement data replication and compression/decompression in a distributed environment. However, when heavy data shuffling among nodes, large data copying or intensive IO overhead is unavoidable, it is not easy to deliver results in a timely manner. Transform-by-copy may lead to a successively copying data. Recent work such as Spark [6] is an effort attempting to reduce IO overhead in cluster computing by sharing in-memory datasets.

NoSQL for unstructured/semi-structured data is best for collecting live, dynamic real-world data, and answering applications such as search or *ad-hoc* query [7]/analysis. On the other hand, structured data, which is organized with schemas and can be easily manipulated with set algebra, is logical for addressing analytic problems such as comparative analysis or in-depth analysis based on sets. In today's business, structured-data-based applications still remain the majority even though the volume of data grows rapidly. Applications include business intelligence (multi-dimensional analysis), interactive (demand/budget) planning, what-if analysis, anomaly detection, recommendation, and various data mining functions.

Unfortunately, in the presence of rapidly growing data, traditional data processing technologies such as relational database management systems (RDBMS) [8] are inadequate for computing those in-depth analytic applications in real time due to the complexity/limitation of SQL (especially costly join operations) and maintaining durability and historical implementation burdens. Even if it comes with a rich set of built-in functions and in-database programming such as *store procedure*, SQL is still a declarative language by nature and is not designed for writing algorithms. When it comes to a sophisticated algorithm, it is required to allocate memory buffers and then query out a large portion of data into the buffers before computing can actually take place. This *compute-centric* model has been deeply rooted in developers for more than fifty years. It leads to poor performance due to bulky data retrieval and is not scalable in the sense that data loaded in memory cannot be larger than the swap space. Other solutions such as online analytical process (OLAP) [9] mandates preprocessing data off-line (i.e., building cubes) to streamline online analysis.

Recent work such as Impala [10] is aimed to support SQL for interactive queries on semi-structured data. The question remains as to how semi-structured data (key-value or document-based datasets) can fit into the relational form (tables). It requires a powerful DDL (Data Definition Language) or a mapping technique to support logical (relational) views on top of semi-structured data. Analyzing structured data and semi-structured data in real time has become a challenging job for data scientists while facing live, dynamic data today.

Our work is to bridge the gap between the above two trends and focus on addressing complex analytic problems other than search or *ad-hoc* query. We first introduce *in-place computing* technology to provide high cost-performance value – delivering computations up to a few billion records within seconds on a commodity computer. An analysis job is said to be interactive or (near) real-time when the elapsed time it takes to compute is within a reasonable waiting time – such as the response time for a web page or a commercial break in television. Besides *scale-out* (employing more servers) and *scale-up*

(adding more memory), in-place computing is thought of as a *scale-in* approach, striving to squeeze the ultimate computing power out of the CPU. It aims to compute data as much and as timely as possible. A distributed computing system can then take advantage of this technology to achieve the same result with fewer nodes and in shorter time.

Based on in-place computing technology, we have implemented *BigObject store*, a data engine capable of storing and computing large data sets structured in a multi-dimensional model for real-time analytic applications. BigObject store adopts an extended relational model in the sense that it allows various shapes of data objects besides tables. Meanwhile it supports algebraic operators besides *join*, and supports operator overloading as well. The extended relational model defines regular data structures (opposed to schema-free data structures) for data collection, which are efficient for implementing algorithms, especially for algorithms based on sets and set algebra.

In the area of very large supply/demand chain planning and monitoring, we have used BigObject store to build applications such as interactive planning, interactive multi-dimensional analysis and real-time exception detection/alert, which are aimed to process a few billion records in seconds. As compared to relational databases in our experiments, BigObject store shows promising results in accelerating performance in two to three orders of magnitude for exception detection cases. And when compared with built-in functions such as `ROLLUP` for group-by aggregation cases, in-place computing outperforms the in-database one by one to two orders of magnitude in speed.

II. In-Place Computing

In-place computing is an unconventional technology in two ways. First, it moves away from the traditional computing model [1] that requires explicit data retrieval to a data-centric computing model in which computations take place where data resides. The idea is, instead of moving big data around, it moves small code to where the data resides for execution. Second, it organizes big data vertically by determining macro objects as the basic functional units and their behaviors in a similar way that macromolecules such as DNA, proteins and lipids serve as the basic functional units for human living cells. The idea is to express and manipulate big data in a high-level (macro-level) way such that the complexity can be easily handled.

In-place computing defines an abstract model in which data objects live (save) and work (compute) in one flat and infinite (address) space. The term *in-place* indicates data (always) ready for computing. It is a perfect world for big data. There is no notion of *storage* and all data is in residence in a boundless virtual memory space. A true in-place computing model is not implementable in the real world where resources are limited, but it is possible to approximate it with today's computer technologies or technologies from other disciplines such as

[1]In some literatures, the traditional computing model is referred to as *compute-centric computing*.

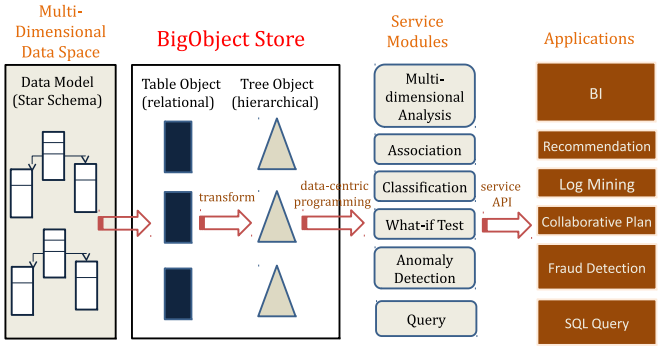


Figure 1. Framework of BigObject Store

biological memory storage. Such an approximated in-place computing system, or *in-place computing system* for simplicity, is a single-space scenario, best for addressing the problems of big data with high degree of interdependency, which will otherwise suffer large data exchange or data replication in a distributed setup. In-place computing does more than just *in-memory computing*, which takes advantage of memory speed. The assumption of one flat and infinite space suggests a new way of designing algorithms by trading space for time.

At the core of in-place computing is the notion of *Macro Data Structure* (MDS), an abstraction to organize collective data elements into macro objects to live and work in a virtual memory space. An MDS looks just like an ordinary data structure in C/C++ or any native programming languages except that it is 1) large enough to hold any big data, 2) persistent, 3) re-locatable and 4) splittable or mergeable. That is, an MDS survives across multiple executions and can be split, merged, copied or moved to a new address location without modifying any internal references, in the same sense that compiled program codes can be relocated or dynamically linked/loaded.

The abstraction of MDS is critical to data-centric computing, where computations take place around data. The programming paradigm based on MDS is referred to as *data-centric programming* or *in-place programming*. At the macro level, it supports expressions and manipulations of multiple MDS's in algebraic semantics. *Transformation* is a special form of manipulation, which converts one MDS to another. At the micro level, it supports programming constructs for traversing or computing data elements inside the MDS, sequentially or in parallel. The abstraction of MDS is somehow similar to RDD (Resilient Distributed Datasets) in Spark, yet different in many ways. Basically, MDS is designed to be long-lived and updateable as part of a data store while RDD is used as an intermediate and read-only result between jobs.

64-bit architecture is the key to implementing MDS in in-place computing systems and making it meaningful and effective. While 2^{32} is quite limited, 2^{64} is considerably infinite in the sense that a 64-bit address space is assumed large enough to hold all the big data that we can see today. Under this assumption, virtual memory with a 64-bit architecture just

creates an illusion of a flat and infinite address space for big data, and it becomes possible to implement an in-place computing system by simply using one 64-bit commodity machine today. In the history of computer science, it is not odd to see the same assumption made in a different situation, where the address space - whether 32-bit, 64-bit or other - is assumed to be large enough to load any program entirely into virtual memory space. As a result, approaches such as paged virtual memory are effective, and deliver fairly good performance for running big codes even with small physical memory. The availability of a 64-bit processor enables the implementation of in-place computing systems.

III. BIGOBJECT STORE

BigObject store is an in-place computing system for multidimensional data domain. Data are organized in *big objects* (a form of MDS), which are in-place in memory and in-place to apply algebraic expressions or algorithms.

The framework of the BigObject store is illustrated in Fig. 1. BigObject takes data in star schema (or snowflake schema) as input and converts to table objects in relational form, which can be transformed to tree objects in hierarchical form. A variety of big objects serve as the building blocks to provide a rich set of services such as multi-dimensional analysis, association, classification, anomaly detection, etc.

In order to implement an efficient computing system, BigObject store exploits three mechanisms: in-memory, transformation, and data-centric programming. In-memory technique provides efficient computing and persistent storage. Transformation mechanism extends the relational model by introducing new kinds of data objects besides tables. Data-centric computing provides a programming framework to run code directly on big objects. In the following subsections, we will describe BigObject store in more details.

A. Multi-dimensional Space

BigObject store organizes data in multi-dimensional space. A dimension is composed of a set of members, each of which may possess certain properties called *attributes*. For example, gender and age are common attributes in *customer* dimension. In addition, it is assumed that each member in a dimension can be identified via an unique key, e.g., *customer id* in *customer* dimension.

Data to be analyzed, such as sales, are called *facts* or *measures*. Multi-dimensional data are naturally defined in relational databases with star schema or snow-flake schema. For instance, Table I and II illustrates two dimensions, *channel* and *product*, and their attributes, *area* and *category*. Table III is a measure table of sales records. Each row indicates the sales amount for a product sold at a store. In this example, *channel.store* and *product.sku* are keys for *channel* and *product* dimension respectively.

B. Big Objects

Both dimensions and measures are organized into table objects in BigObject store, with a MDS similar to tables in relational databases. In our example, data in Table I, II, and III are imported as table objects: *channel*, *product*, and *sales* respectively. Moreover, a hash index is built for each dimension table object based on its key.

TABLE I
A CHANNEL DIMENSION TABLE

channel.store	*channel.area*
S1	NY
S2	NY
S3	CA

TABLE II
A PRODUCT DIMENSION TABLE

product.sku	*product.category*
P1	Food
P2	Food
P3	Cloth

TABLE III
A SALES MEASURE TABLE

channel.store	*product.sku*	sales value
S1	P1	6
S1	P2	5
S1	P3	7
S2	P3	8
S3	P1	4
S3	P2	9

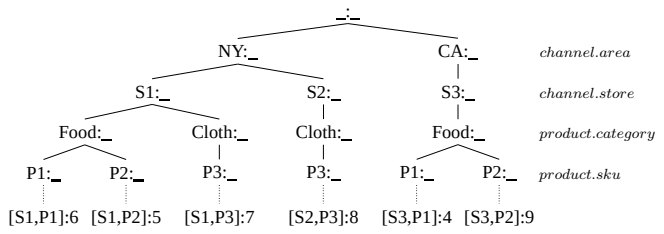


Figure 2. A big object *actual-sales* with hierarchy: *channel.area* $\prec$ *channel.store* $\prec$ *product.category* $\prec$ *product.sku*.

Another kind of objects is called tree objects. A tree object has a hierarchical MDS that is suitable for performing analytical type computation efficiently. Fig. 2 shows a tree object derived from *sales* object, where hierarchical attributes from top to bottom are *channel.area* $\prec$ *channel.store* $\prec$ *product.category* $\prec$ *product.sku*. We use $a_1 \prec a_2$ to denote that the level of attribute a_1 is higher than the level of a_2 in the tree hierarchy.

In general, the hierarchy of a tree object is semantically associated with a meaning or property of data, described by attributes in dimensions. Thus, it represents a nature structure of human thinking process to narrow down a big problem into sub-problems (i.e., divide and conquer).

Each node in a tree object holds a key-value pair of data, denoted as "key:value". The value of a node can be a numerical value, an array, a time series, or even a complex data object such as a `struct` in C/C++. The symbol '_' indicates an unknown or un-initialized value. The value in the leaf nodes can be treated differently than the internal nodes. Each leaf node can either hold a copy of value from the source object or contain a pointer to value in the source object. The dashed lines in Fig 2 simply indicate their source records in *sales* table object for each leaf node.

C. In-Memory

BigObject store is implemented using the memory mapping technique, i.e., each big object is memory-mapped and backed by a file. Thus, it allows programs to access each object's structure and data in memory directly – a way to approximate the flat and infinite memory space for in-place computing.

As a storage unit, the memory space for a big object is automatically saved in the memory-mapped file. As a computing unit, algorithms can be directly applied on the memory structure of a big object transparently without any explicit data retrieval and additional memory allocation.

The beauty of the memory-mapping approach is that a big object is simply a file, in which the content of the object can be preserved even if the system is powered off, or move around to different locations. It can also serve as a paging device when physical memory is exhausted.

More importantly, the size of memory-mapped files can be bigger than the combined size of physical memory and swap space. It is common that application developers need to design out-of-core or external memory algorithms [11] to process data that is too large to fit into the allocatable memory at one time. The memory-mapping technique provides a scalable solution to this problem.

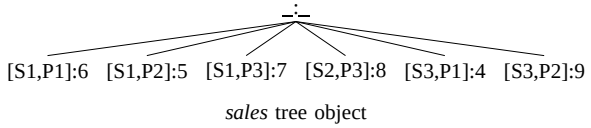
We carefully design the memory layout for each kind of big objects. Our current implementation arranges table objects in a mixed column and row-based manner. To be more precise, attributes are row-based, but attributes and fact data are separated in two different columns. This approach can be extended to allow pure column-oriented tables [12].

For tree object, we structure the memory layout to preserve both sibling locality and descendant locality, where the relevant nodes are byte-to-byte contiguous in memory with no irrelevant nodes mixed in between. By taking advantage of tree semantics, such memory layout helps to reduce cache miss rate and page faults, and thus improves computing performance.

D. Transformation

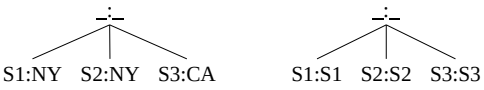
Transformation is a common technique used in science, which converts a problem in a domain into another domain easy to manipulate, such as fourier transform in signal processing, mapper in MapReduce, or transformation in Spark. However, transformation doesn't come free. It always incurs cost, more or less depending on the underlying implementation. The idea of transformation mechanism is to transform data into a data structure suitable for efficient computation, assuming the cost of transformation can be amortized or tolerated.

BigObject store provides a transformation operator, *transformative join* (*trans-join* in short), based on a similar semantics of join from relational algebra [13]. Trans-join transforms a big object into a tree object according to a desired hierarchy. It is a binary operator which takes a big object as the first operand and attributes of dimensions as the second operand, and creates a tree object. A trans-join operation "lifts" the hierarchy of the tree object one level up by grouping the leaf nodes by an attribute. For each group of leaf nodes that share the same attribute value, an intermediate node is created as the new parent node of the group. As a result, a new level of intermediate nodes are inserted between the leaf nodes and the original parent nodes.



sales tree object

Figure 3. A table object represented as a tree object.



(a) *channel.area* (a) *channel.store*

Figure 4. Projection objects representing attributes of *channel* dimension.

For example, the tree object *actual-sales* in Fig. 2 is trans-joined from the *sales* table object in Table III based on hierarchical attributes *channel.area* $\prec$ *channel.store* $\prec$ *product.category* $\prec$ *product.sku*. This process is similar to using `GROUP BY` with `ROLLUP` operator on columns *channel.area*, *channel.store*, *product.category*, and *product.sku* by joining *sales*, *channel*, and *product* tables in SQL. It is noted that trans-join operator differs from the `JOIN/GROUP BY/ROLLLUP` operators in that it physically constructs an object with tree structure rather than outputs a table in relational model [14].

Trans-Join Semantics

From the point of view of modeling trans-join operation, it can be considered as an operator transforming a tree object to another tree object based on a dimension tree object. In fact, each table object can be represented as a tree object. Fig. 3 shows the tree object for *sales* according to Table III.

Furthermore, we can derive a projection tree from a dimension tree based on an attribute by retaining only value related to that attribute, a process similar to the projection operation in relational algebra. For example, Fig. 4 shows two projection tree objects *channel.area* and *channel.store* projcted from *channel* dimension for attribute *area* and *store* respectively.

Trans-join then can be treated as an operator that adding a level of parent nodes above leaves of a tree by grouping leaf nodes with the same parent based on the projection tree. Fig. 5 shows an example of how *sales* trans-joins *channel.area*. The result *sales.area* is lifted-up with a new level *channel.area*. In this example, nodes [S1,P1]:6, [S1:P2]:5, and [S1,P3]:7 match S1:NY and [S2,P3]:8 matches S2:NY in *channel.area*. A parent node NY:_ is added and shared by these nodes. It simply means that all sales records in NY area are grouped together. We use $sales \underset{\text{trans-join}}{\triangle} channel.area$ to denote the above operation. This process is similar to the join operation in relational algebra and that's why we chose the word "join"

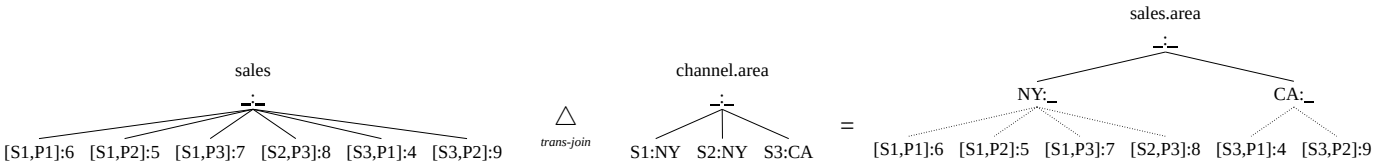


Figure 5. Big object *sales.area* is the result of trans-joining *sales* object with *channel.area* object

```
1:  procedure TRAN-JOIN(s, r, n)        ▷ s: data, r: rank, n: node
2:      while r ≠ ∅ do
3:          a ← first attribute of r
4:          if a is an attribute in dimension D then
5:              for each data record e in s do
6:                  extend e with corresponding value of a in D
7:              end for
8:          end if
9:          partition s based on a
10:         for each partition s_i do
11:             k ← value of a
12:             add n_i as a child of n with key k
13:             Trans-join(s_i, r − 1, n_i)
14:         end for
15:     end while
16: end procedure
```

Figure 6. Trans-join algorithm

to appear in *trans-**join***.

Trans-join can be used to construct a tree object with desired level of hierarchy, which allows us to create the most suitable tree object for analytical tasks. For instance, the tree object in Fig. 2 can be modeled by a sequence of trans-join operations as shown in the following equation:

$$sales \underset{\text{trans-join}}{\triangle} channel.area \underset{\text{trans-join}}{\triangle} channel.city \underset{\text{trans-join}}{\triangle} product.category \underset{\text{trans-join}}{\triangle} product.sku = actual\text{-}sales \quad (1)$$

Trans-Join Algorithm

The pseudo code for trans-join is described in Fig. 6, where s represents the data set from source big object. It recursively constructs a tree object based on the desired hierarchical attributes, which is denoted as rank r.

The hash index for each dimension table is used to efficiently find the corresponding element in a dimension at line 6. For example, to obtain the value of *channel.area* for sales data [S1,P1]:6 (see Fig. 5), trans-join will first search for the record S1:NY in *channel* dimension by using S1 as key via its hash index. Then NY can be retrieved as value of *channel.area*.

Trans-Join Analysis

To support multi-dimensional analysis, trans-join allows a filter such that only the fact data satisfying the specified criteria will be used to construct a tree object. The filter adopts a simplified SQL `WHERE` clause syntax. For example, the following filter will create a tree object with all 'Food' sales in NY.

```
channel.area='NY' AND product.category='Food'
```

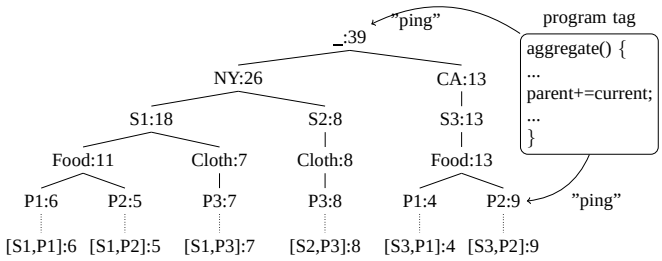


Figure 7. Aggregation using a program tag.

In addition, BigObject store also provides *attribute function* such that the value of an attribute is generated by a function. It provides a rich set of functions for several types of attribute. For example, we can use function *year* to obtain the year of a *birthday* attribute by *birthday.year*, or apply function *age* to find one's age by *birthday.age*.

Attribute functions can be used in a filter to support more flexible analysis. For example, filter

```
date.year >= 2010 AND date.month % 2 = 0
```

will only use fact data every two month after year 2010.

E. Data Centric Programming

One of the key ingredients of in-place computing is data centric computing. Instead of using query languages such as SQL to retrieve necessary data out of databases, BigObject store provides a simple programming framework, which allows (1) a piece of program to attach to an object node for execution and (2) expressing and evaluating an arithmetic expression of big objects. This framework together with compiler optimization not only provides a high-level expressive power, but also delivers good performance. The first feature in the programming framework is realized by a language construct called *program tag*, and the second feature is done by *macro expression*.

Program Tag

A program tag is a piece of program containing statements, which is used to implement a node-level function for a tree object. Once a program tag is defined, it can be attached to nodes of tree objects and then evaluated node-by-node in a tree traversal order, such as depth-first traversal.

For example, to implement aggregation operation, we can "ping" each node in the tree object with a program tag as

```
void discrepancy(BO<int> C, BO<int> A, BO<int> B)
{
    C = (A - B) / A;
}
```

Listing 1. Macro expression

shown in Fig. 7. In this example, each node simply adds its value to its parent's node. During a depth-first traversal starting from the root, each tag program will be executed when the attached node is visited. Program tags can be designed to behave differently in different levels and can be parameterized at invocation. BigObject programming framework provides programming capabilities to let developers design program tags to perform complex computation on each node in a high-level way.

By applying the similar idea of program tag, we have developed program tag class, which includes both variables and codes. When a program tag class c is pined to a tree object T, it creates another tree object T_c. T_c shares the same tree structure as T. The key in each node of T_c is the same as in the corresponding node of T. However, each node in T_c holds an instance of c's variables as value. In other words, when c is pined to T, a c object is created on each node in T_c.

We can define methods in c to accomplish our computing tasks as in object-oriented programming. These methods is able to access c's variables on each node and use them to hold computing results of each node in T_c. Note that T_c is also memory-mapped and therefore can be persistently saved on storage.

Macro Expression

Similar to matrix arithmetic, BigObject programming framework allows users to write and evaluate algebraic expression over big objects, where operators can be either pre-defined or overloaded.

For instance, Listing 1 shows how to calculate discrepancy between two isomorphic tree objects using macro expression. An example of how the expression can be evaluated is shown in Fig. 8, where the macro level arithmetic operators for tree objects are defined as primitive type arithmetic operators applied on every corresponding nodes in each tree.

For efficiency, program tags and macro expressions are translated to C++ code first by a language compiler and then the generated C++ code is further compiled into an executable library by a C++ compiler. Compiled executables are sent to BigObject store. They can be dynamically linked and invoked when needed.

Program tag and macro expression provide an efficient way to manipulate tree objects. It is noted that the similar concept can be easily extended to other kinds of objects, or even among different kinds of objects.

F. Compression Scheme

BigObject store employs a dictionary compression scheme which is commonly used in databases to replace data with smaller codes [15]–[17]. From our experiments, this technique does not only speedup trans-join operator but also reduce the space needed for big objects. The overhead to encode/decode when data are inputted/outputted appears to be marginal.

G. Concurrency Control

Big objects are updatable. In order to address concurrency issues, we adopt a simple coarse-grained read-write lock concurrency control by locking the whole objects when shared data are being updated. To avoid deadlock, a wait/die protocol is implemented [18].

IV. Experiments

We have done some experiments on comparing BigObject store and relational databases. These experiments are simple, but can provide insight into the performance of BigObject store.

Besides BigObject, we selected three relational databases, PostgreSQL (version 9.1), MySQL (version 5.5.31), and a high-performance commercial database (named Z for privacy purpose). Both PostgreSQL and MySQL support durability to maintain data integrity in case of system failure. BigObject, Z, and MySQL have in-memory implementations. MySQL also has a non-in-memory storage engine InnoDB. In addition, Z has an JIT compiler and is able to execute pre-compiled query plans.

Since our goal is to develop an affordable technology, the hardware used for the experiments is a commodity machine with CPU at 2.3 GHz, 64GB of RAM, and 1.0TB of hard disk. The machine runs 64-bit Ubuntu Linux (version 12.10).

The problem used in the experiments is similar to the program described in Listing 1. Assuming in a business, the management wishes to continuously review business performance and to be notified in real time when the discrepancy between *actual sales* and *forecasted sales* is greater than a user defined threshold.

The data used in the experiments share the same dimensions as shown in Table I and II. We generated several data sets representing *actual sales* and *forecasted sales* with various sizes. In preparation of our experiments, tree objects for *actual sales* and *forecasted sales* are built from the table objects by trans-join and have the same hierarchical structure as shown in Figure 2. In addition, indices are built for tables in relational databases for better performance.

Table IV shows the tables sizes in MySQL, PostgreSQL, and the sizes of table objects and tree objects in BigObject store. A table/tree object is smaller than a table in relational databases mainly due to concise record structures and data encoding scheme.

For relational databases, we implemented a C++ program that connects to MySQL, PostgreSQL, and Z, retrieves data using SQL queries, aggregates, and computes discrepancy. For

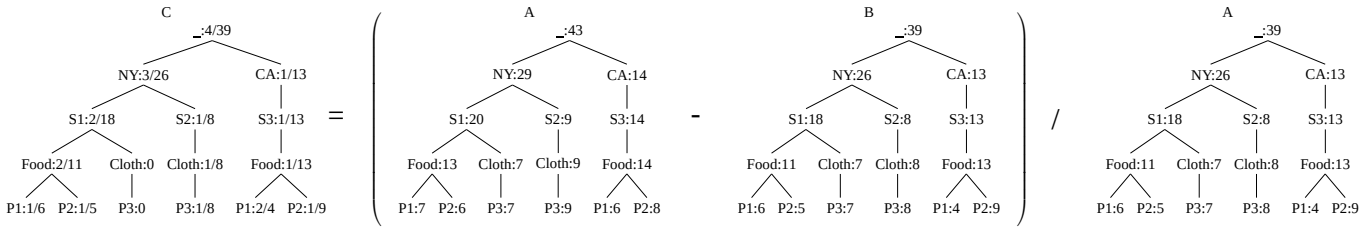


Figure 8. Illustration of evaluation for macro expression C = (A - B) / A.

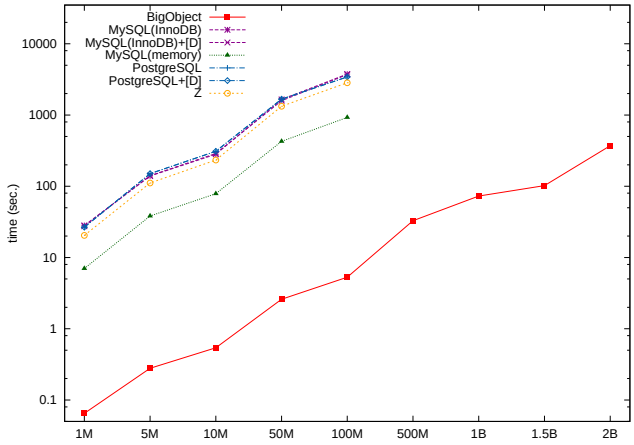


Figure 9. Results of read only computing.

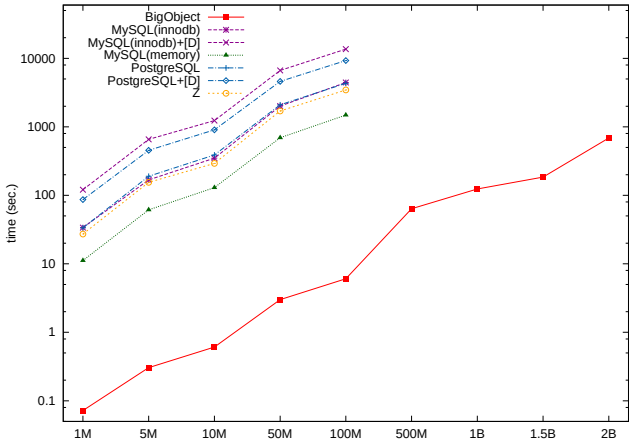


Figure 10. Results of read-write computing.

TABLE IV
STORAGE SIZE FOR DIFFERENT SYSTEMS.

Size	1M	5M	10M	50M	100M
A Tree Object	20MB	97MB	193MB	964MB	1.9GB
A Table Object	20MB	93MB	184MB	1.0GB	2.2GB
PostgreSQL	89MB	471MB	947MB	4.7GB	9.5GB
MySQL	81MB	410MB	822MB	4.1GB	8.3GB

BigObject, we used program tag and macro expressions to achieve the same purpose.

The first set of results is shown in Figure 9 for read-only computing – only query time and computing time are measured. MySQL and PostgreSQL with durability settings are denoted with "+[D]". It can be seen that durability plays a trivial role in performance for read-only computing as expected. Also, by comparing two implementations of MySQL, memory engine and InnoDB (disk-based) engine, we learn that in-memory implementation is about four times faster than non-in-memory implementation. In general, BigObject store outperforms tested relational databases in two orders of magnitude in this typical exception detection task. Our experiments show that, when data size is 100 millions, BigObject takes 5.284 seconds, while MySQL (memory engine) and Z take more than 925 seconds and 2834 seconds respectively, about 175 to 536 times faster. We did not perform experiments on relational databases for data size more than 100 millions due to limited disk size and time.

The second set of results is shown in Figure 10 for read-write computing. We assume less than 1% of data raise exceptions that need to be recorded. We use a tree object created by a program tag to save exceptions in BigObject and extra tables to save exceptions for relational databases. First, it can be seen that MySQL (InnoDB) and postgreSQL with durability settings are about two to three times slower than the ones without durability. Second, in-memory implementation is about three times faster than with in-disk implementation. Our experiments show that, without durability, BigObject takes 6.057 seconds for data size of 100 million, while MySQL (memory engine) and Z take 1480 seconds and 3460 seconds respectively, about 245 to 571 times faster.

Comparing In-database and In-place Computing

We further compare in-database and in-place computing. Specifically, we perform a SQL `GROUP BY` with `ROLLUP` aggregation such that all operations are accomplished in MySQL without data retrieval. The SQL statement in the experiment joins *sales*, *channel*, and *product* tables and groups data by *channel.area*, *channel.store*, *product.category*, and *product.sku*. For BigObject, we perform trans-join and aggregation on the same hierarchical order. Figure 11 shows the time for MySQL and BigObject. The time for BigObject includes both trans-join and aggregation. It can be seen that trans-join operator is quiet efficient compared to `JOIN` and `GROUP BY` operators in MySQL. Overall, BigObject outperforms MySQL

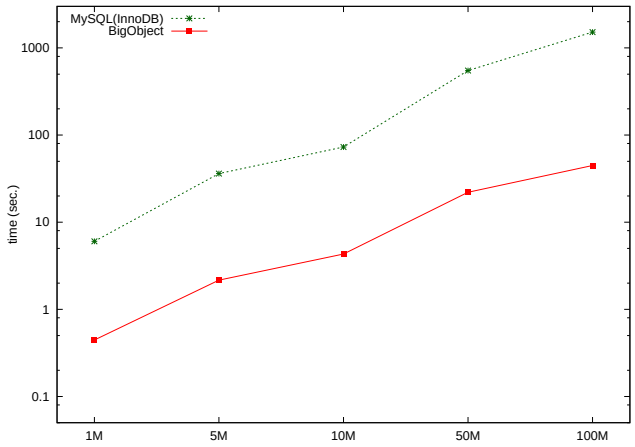


Figure 11. Times for MySQL with JOIN/GROUP BY/ROLLUP and BigObject.

on this specific `JOIN/GROUP BY/ROLLUP` task by at least one order of magnitude.

In summary, these preliminary experiment results demonstrate that BigObject store uses less space in terms of disk storage and outperforms relational databases in one to two orders of magnitude.

V. Conclusion

The in-place computing abstract model creates a perfect world for big data – all data resides in one flat and boundless virtual memory space, ready for execution. 64-bit architecture is the key to implementing in-place computing systems effectively. Based on the principles of in-place computing, we have developed BigObject store, which delivers promising performance results in both speed and scalability. Our experiments show performance acceleration in two to three orders of magnitude in comparison to the traditional data processing approach, and in one to two orders of magnitude in comparison to in-database computing approach.

From our experimental research work, we have learned many things that lead to substantial performance advantages. The abstraction of one flat and boundless virtual memory space supports the ground for organizing big data complexity by trading space for time. The unconventional data-centric computing/programming paradigm provides a new way to avoid big data retrieval and write sophisticated algorithms. MDS (Macro Data Structure) is an abstraction to organize basic functional units as *big objects*, which can be manipulated with macro expressions and algebraic operators in an efficient way. File-backed in-memory computing supports approximation of in-place computing model. BigObject differentiates dimension attributes (from dimension tables) from measures (from fact tables). Related attributes are grouped in a raw-based manner, while measures are arranged in a column-based manner. Data locality and compression scheme improve memory utilization, reduce cache misses and page faults and thus boost performance.

BigObject store has been used to develop applications for real-time multi-dimensional analytics, interactive planning [22], instant recommendation, exception detection/alert, interactive log analysis, and have demonstrated with significant cost-performance value and capability to organize big data.

References

[1] C. Strauch, U.-L. S. Sites, and W. Kriha, "Nosql databases," *URL: http://www. christof-strauch. de/nosqldbs. pdf (07.11. 2012)*, 2011.

[2] R. Cattell, "Scalable sql and nosql data stores," *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12–27, 2011.

[3] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[4] K. Chodorow, *MongoDB: the definitive guide*. " O'Reilly Media, Inc.", 2013.

[5] H. Abelson, *Structure and interpretation of computer programs*. Paul Muljadi, 1996.

[6] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[7] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 330–339, 2010.

[8] C. Date, "An introduction to database," *System, Addison Wesley Publishing*, 2001.

[9] S. Chaudhuri and U. Dayal, "An overview of data warehousing and olap technology," *ACM Sigmod record*, vol. 26, no. 1, pp. 65–74, 1997.

[10] M. Kornacker and J. Erickson, "Cloudera impala: real-time queries in apache hadoop, for real," 2012.

[11] J. S. Vitter, "External memory algorithms and data structures: Dealing with massive data," *ACM Computing surveys (CsUR)*, vol. 33, no. 2, pp. 209–271, 2001.

[12] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil *et al.*, "C-store: a column-oriented dbms," in *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 2005, pp. 553–564.

[13] E. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[14] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data Mining and Knowledge Discovery*, vol. 1, no. 1, pp. 29–53, 1997.

[15] G. Graefe and L. D. Shapiro, "Data compression and database performance," in *Applied Computing, 1991.,[Proceedings of the 1991] Symposium on*. IEEE, 1991, pp. 22–27.

[16] M. A. Roth and S. J. Van Horn, "Database compression," *ACM Sigmod Record*, vol. 22, no. 3, pp. 31–39, 1993.

[17] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006, pp. 671–682.

[18] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts, Windows XP update*. Wiley. com, 2006.

[19] D. Verneer, "execute-in-place," *Memory Card Magazine*, 1991.

[20] M. Stonebraker, J. Becla, D. J. DeWitt, K.-T. Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik, "Requirements for science data bases and scidb." in *CIDR*, vol. 7, 2009, pp. 173–184.

[21] J. Bowie and G. Barnett, "Mumps–an economical and efficient time-sharing system for information management," *Computer Programs in Biomedicine*, vol. 6, no. 1, pp. 11–22, 1976.

[22] W. Hseush, Y.-C. Huang, S.-C. Hsu, and C. Pu, "Real-time collaborative planning with big data: Technical challenges and in-place computing," in *Collaborative Computing: Networking, Applications and Worksharing (Collaboratecom), 2013 9th International Conference Conference on*. IEEE, 2013, pp. 96–104.