DataGarage: Warehousing Massive Performance Data on Commodity Servers

Charles Loboz
Microsoft Corporation
cloboz@microsoft.com

Slawek Smyl
Microsoft Corporation
slsmyl@microsoft.com

Suman Nath
Microsoft Research
sumann@microsoft.com

ABSTRACT

Contemporary datacenters house tens of thousands of servers. The servers are closely monitored for operating conditions and utilizations by collecting their *performance data* (e.g., CPU utilization). In this paper, we show that existing database and file-system solutions are not suitable for warehousing performance data collected from a large number of servers because of the scale and the complexity of performance data. We describe the design and implementation of DataGarage, a performance data warehousing system that we have developed at Microsoft. DataGarage is a hybrid solution that combines benefits of DBMSs, file-systems, and MapReduce systems to address unique challenges of warehousing performance data. We describe how DataGarage allows efficient storage and analysis of years of historical performance data collected from many tens of thousands of servers—on commodity servers. We also report DataGarage's performance with a real dataset and a 32-node, 256-core shared-nothing cluster and our experience of using DataGarage at Microsoft for the last one year.

1. INTRODUCTION

Contemporary datacenters house tens of thousands of servers. Since they are large capital investments for online service providers, the servers are closely monitored for operating conditions and utilizations. Assume that each server in a datacenter is continuously monitored by collecting 500 hardware and software performance counters (e.g., CPU utilization, job queue size). Then, a data center with 100,000 servers yields 50 million concurrent data streams and, with a mere 15-second sampling rate, more than 1TB data a day. While the most recent data is used in real-time monitoring and control, archived historical data is also used for tasks such as capacity planning, workload placement, pattern discovery, and fault diagnostics. Many of these tasks require computing pair-wise correlation, histogram, and first-order trend over last several months [10, 13]. However, due to sheer volume and complexity of the data, archiving it for a long period of time and supporting useful queries on it reasonably fast is extremely challenging.

In this paper we show that traditional data warehousing solutions are suboptimal for *performance data*, the data of performance counters collected from monitored servers. This is primarily due to the scale and the complexity of performance data. For example, one important design goal of performance data warehousing is to reduce storage footprint since an efficient storage solution can reduce storage, operational, and query processing overhead. Prior works have shown two different approaches to organize data as relational tables. In the *wide-table* approach, a single table having one column for each possible counter is used to store data from a large number of heterogenous servers, with null values in the columns that do not exist for a server. In the *narrow-table* approach, data from different servers is stored in a single table as key-value pairs [5]. We show that both these approaches have a high storage overhead, as well as high query execution overhead, for performance data warehousing. This is because different sets of software and hardware counters are monitored in different servers and therefore performance data collected from different servers are highly heterogenous. Another reason why off-the-shelf warehousing solutions are not optimal for performance data is that their data compression techniques, which work well for text or integer data, do not perform well on floating-point performance data. (We discuss the challenges in details in Sections 3 and 4.)

Prior works have also shown two different approaches to general, large-scale data storage and analysis. The first approach, which we call *TableStore*, stores data as relational tables in a parallel DBMS or multiple single-node DBMS (e.g., HadoopDB [2]). Parallel DBMSs process queries with database engines, while HadoopDB processes queries using a combination of database engine and a MapReduce-style query processor [6]. The second approach, which we call *FileStore*, stores data as files or streams in a distributed filesystem and processes queries on it using a MapReduce-like system (such as Hadoop [7] or Dryad [8]). We show in Section 3 that both TableStore and FileStore have very large storage footprints for performance data due to its high heterogeneity. Previous work has shown that a database query engine on top of a TableStore has better query performance and simpler query interface, but it has poor fault tolerance [12]. On the other hand, FileStore has a lower cost, higher data management flexibility, and more robustness during MapReduce query processing, but it has inferior query performance and more complex query interface than a DBMS approach.

In this paper, we describe our attempt to build a DBMS and filesystem hybrid that combines the benefits of TableStore and FileStore for performance data warehousing. We describe design and implementation of DataGarage, a system that we have built at Microsoft to warehouse performance data collected from tens of thousands of servers in Microsoft datacenters. The design of DataGarage has the following key aspects:

1. In contrast to traditional *single wide-table* or *single narrow-table* approaches of organizing data as relational tables, Data-

Proceedings of the VLDB Endowment, Vol. 3, No. 2
Copyright 2010 VLDB Endowment 2150-8097/10/09... $ 10.00.

Garage uses a new approach of using *many* (e.g., tens of thousands) *wide-tables*. Each wide-table contains performance data collected from a single server and is stored as a database file in a format accessible by a light-weight embedded database. Database files are stored in a distributed file system, resulting in a novel DBMS-filesystem hybrid. We show that such a design reduces storage footprint and makes queries compact, simple, and run faster than alternative approaches.

2. DataGarage uses novel floating-point data compression algorithms that use ideas from column-oriented databases.
3. For data analysis, DataGarage accepts SQL queries, pushes them inside many parallel instances of an embedded database, aggregates results along an aggregation hierarchy, and dynamically adapts with faulty or resource-poor nodes (like MapReduce).

Thus, DataGarage combines the storage flexibility and low cost of a file system, compression benefits of a column-store database, performance and simple query interface of a DBMS, and robustness of MapReduce in the same system. To the best of our knowledge, DataGarage is the first large-scale data warehousing system that combines the benefits of these different systems.

In our production prototype of DataGarage, we use Microsoft SQL Server Compact Edition (SSCE) files [14] to store data and SSCE runtime library to execute queries on them. SSCE was originally designed for mobile and embedded systems, and to the best of our knowledge, DataGarage is the first large-scale data analysis system to use SSCE. Our implementation of the DataGarage system is extremely simple—it uses existing NTFS file system, Windows scripting shell, SSCE files and runtime library, and several thousand lines of custom code to glue everything together. We have been using DataGarage for the last one year to archive data from many tens of thousands of servers in Microsoft datacenters.

Many design decisions behind DataGarage were guided by the lessons we learnt from our previous unsuccessful attempts of using existing solutions to warehouse performance data. At a high level, DataGarage and HadoopDB have similarities—they both use TableStore for storage and MapReduce for query processing. However, the fundamental difference between these two systems is that HadoopDB uses a single DBMS system in each node, while DataGarage uses tens of thousands of embedded database files. As we will show later, such finer-grained partitioning of data into many light-weight relational stores significantly reduces storage footprint of heterogenous performance datasets and makes typical DataGarage queries simple, compact, and efficient. We believe that our current design adequately addresses the unique challenges of a typical performance data warehousing system.

In the rest of the paper, we make the following contributions. First, we discuss unique properties of performance data, desired goals of a warehousing system for performance data, and why existing solutions fail to achieve the goals (Sections 2 and 3). Second, we describe design and implementation of DataGarage. We show how it reduces storage footprint by fine-grained data partitioning and compression techniques (Section 4). We also describe how its query engine achieves scalability and fault-tolerance by using a MapReduce-like approach (Section 5). Third, We evaluate DataGarage with a real dataset and a 32-node, 256-core shared nothing cluster (Section 6). Finally, we describe our experience of using DataGarage in Microsoft for the last one year (Section 7).

2. DESIGN RATIONALE

In this section, we describe performance data collection process, properties of performance data, and desired properties of a performance data warehousing system.

2.1 Performance Data Collection

Performance data from a server is collected by a background *monitoring process* that periodically scans selected software and hardware performance counters (e.g., CPU utilization) of the server and stores their values in a file. In effect, daily performance data looks like a wide-table, each row containing counter values collected at a time (as shown in Table 1). A few hundred performance counter values are collected from a typical server.

The number of performance counters and the sampling period are decided based on analysis requirements and resource availability. The more counters one can collect, the more types of analysis he can perform. For example, if one collects only hardware counter data (e.g., processor and disk utilization), he can analyze how much load a server is handling—but he may not precisely know the underlying reason of such load. If he also collects SQL Server usage counters, he can correlate these two types of counters and obtain some insight into why the server is loaded and if something can be done about it. Similarly, the more frequently one collects counter data, the more precise he can be about his analysis—using hourly averages of counter data, one can find which hour has the highest load, using 15-second averages he can also find under which conditions a particular disk is a bottleneck, using 1-second sampling he can further obtain good estimates of disk queue lengths.

In the production deployment of DataGarage, the sampling interval is 15 seconds and the number of collected counters varies from 100 to 1000 among different servers. For some other monitoring scenarios the sampling period may be as high as 2 minutes and the number of counters as low as 10.

Monitoring is relatively cheap for a single server. Our DataGarage monitoring process uses 0.01% of processor time on a standard server and produces 5-100MB of data per server per day. For 100,000 servers this results in a daily flow of over 1TB of data. This sheer volume alone can make the tasks of transferring the data, archiving it for years, and analyzing it extremely challenging.

2.2 Performance Data Characteristics

Performance data collected from a large number of servers has the following unique properties.

Counter sets. Performance data collected from different servers can be highly heterogenous. This is because each server may have a different set of counters due to different numbers of physical and logical disks, network cards, installed applications (SQL Server, IIS, .NET), etc. We have seen around 30,000 different performance counters over all DataGarage monitored servers, while different servers are monitored for different subsets, of size less than 1,000 for most servers, of these counters.

Counter Data. Almost all performance data is floating-point data (with timestamps). Once collected, the data is read-only. Data can often be "dirty"; e.g., due to bugs in the OS or in the data collection process, we have observed dirty values such as $2,000,000$ for the performance counter `%DiskIdleTime`, which is supposed to be within the range $[0, 100]$. Such dirty data must be i) ignored during computing average disk idle time, and ii) retained in the database, as the frequency and scale of such strange values may indicate something unusual in the server.

Query. Queries are relatively infrequent. While most queries involve simple selection, filtering, and aggregation, complex data mining queries (e.g., discovering trends or correlations) are not uncommon. Queries are typically scoped according to a hierarchy of monitored servers (e.g., `hotmail.com` servers within a rack inside a given datacenter). Example queries include comput-

Figure 1: A tabular view of performance data from a server

ServerID	SampledTime	CPUUtil	MemFreeGB	NetworkUtil	dsk0Bytes	dsk1Bytes	···
13153	15:00:00.460	98.2	2.3	47	78231	19000	···
13153	15:00:16.010	97.3	3.4	49	65261	18293	···
13153	15:00:31.610	96.1	3.5	51	46273	23132	···
13153	15:00:46.020	95.2	3.8	48	56271	28193	···
...	...	...	...	...	...	...	...

ing average memory utilization or discovering unusual CPU load of servers within a datacenter or used by an online property (e.g., `hotmail.com`), estimating hardware usage trend for long term capacity planning, correlating one server's behavior with another server's, etc.

2.3 Desired Properties

We now describe the desired properties of a warehousing system designed for handling a massive amount of performance data.

Storage Efficiency. The primary design goal is to reduce the storage footprint as much as possible. As mentioned before, monitoring 100,000 servers produces more than 1TB raw binary data; archiving and backing up this data for years can easily take a petabyte of storage space. Transferring this massive data (e.g., from monitored servers to storage nodes), archiving it, and running queries on it can be extremely expensive and slow. Moreover, if the data is stored on a public cloud computing platform (for flexibility in using more storage and processing on demand), one pays only for what one uses and hence price increases linearly with requisite storage and network bandwidth. This again highlights the importance of reducing storage footprint.

One can envision building a custom cluster solution such as eBay's Teradata that can manage approximately 2.4PB of relational data in a cluster of 72 nodes (two quad-core CPUs, 32GB RAM, 104 300GB disks per node); however, the huge cost of such a solution cannot be justified for archiving performance data because of its relatively light workload and often non-critical usage.

Query Performance and Robustness. The system should be fast in processing complex queries on a large volume of data. A faster system can make a big difference in the amount, quality, and depth of analysis a user can do. A high performance system can also result in cost savings, as it can allow a company to delay an expensive hardware upgrade or to avoid buying additional compute nodes as an application continues to scale. It can also reduce the cost of running queries in a cloud computing platform, where the cost increases linearly with the requisite compute power.

The system should also be able to tolerate faulty or slow nodes. Our desired system will likely be run on a shared-nothing cluster of cheap and unreliable commodity hardware, where the probability of a node failure during query processing is very high. Moreover, it is nearly impossible to get a homogenous performance across a large number of compute nodes (due to heterogeneity in hardware and software configuration, disk fragmentation, etc.) Therefore, it is desirable that the system can run queries even if a small number of storage nodes are unavailable and its query processing time is not adversely affected if a small number of the computing nodes involved in query processing fail or experience slowdown.

Simple and flexible query interface. Average data analysts are not expected to write code for simple and routine queries such as selection/filtering/aggregation; these should be answered using familiar languages such as SQL. More complex queries, which are infrequent, may require loading outputs of simpler queries into business intelligence tools that aid in visualization, query generation, result dash-boarding, and advanced data analysis. Complex queries also may require user defined functions for complex (e.g., data mining) queries that are not easily supported by standard tools; the system should support this as well.

3. PERFORMANCE DATA WAREHOUSING ALTERNATIVES

In this section we first consider two available approaches of general, large-scale data storage and analysis. Then we discuss how they fail to achieve all the aforementioned desirable properties in the context of performance data warehousing.

3.1 Existing Approaches

►**TableStore.** We call TableStore the traditional approach of storing data in standard relational tables, which are partitioned over multiple nodes in a shared nothing cluster. Parallel DBMSs (e.g., DBMS-X) transparently partition data over nodes and give users the illusion of a single-node DBMS. Recently proposed HadoopDB uses multiple single node DBMS. Queries on a TableStore are executed by parallel DBMSs' query processing engines (e.g., in DBMS-X) or by MapReduce-like systems (e.g., in HadoopDB). Existing TableStore systems support standard SQL queries.

►**FileStore.** We call FileStore the alternative data storage approach where data is stored as files or streams in a file system distributed over a large cluster of shared-nothing servers. Executing distributed queries on FileStore using a MapReduce-like system (e.g., Hadoop, Dryad) has got much attention lately. Recent work on this approach has focused on efficiently storing a large collection of unstructured and structured data (e.g., BigTable [5]) in a distributed filesystem, integrating declarative query interfaces to the MapReduce framework (e.g., SCOPE [4], Pig [11]), etc.

3.2 Comparison

We compare the two above approaches in terms of several desirable properties.

• **Storage efficiency.** Both TableStore and FileStore score poorly in terms of storage efficiency for performance data. For TableStore, the inefficiency comes from two factors. First, due to the high heterogeneity of dataset, storing data collected from different servers within a single DBMS can waste a lot of space. We will discuss the issue in more details in Section 4.1. Second, compression schemes available in existing row-oriented database systems do not work well on floating point data. For example, our experiments show that the built-in compression techniques in SQL Server 2008 provides a compression factor of ≈ 2 for real performance data.[1] Such a small compression factor is not sufficient for massive data and does not justify the additional decompression overhead during query processing. On the other hand, FileStore can have comparable or even larger storage footprint than TableStore. Without a schema, a compression algorithm may not be able to take advantage of tempo-

[1]Column-store databases optimized for floating point data may provide a better compression benefit.

ral correlation of data in a single column (e.g., as in column-store databases [1]) and to use lossy compression technique appropriate for certain columns.

• **Query performance.** Previous work has shown that for many different workloads, queries over a TableStore runs significantly faster than those over a FileStore [12]. Query processing systems on FileStore are slower because they need to parse and load data during query time. The overhead would be even more significant for performance data—since performance data from different servers have different schemas, a query (e.g., the Map function in MapReduce) needs to first load and parse the appropriate schema for a file before parsing and loading the file's content. In contrast, a TableStore can model and load the data into tables before query processing. Moreover, query engine over a TableStore can use many performance enhancing mechanisms (e.g., indexing) developed by the database research community over the past few decades.

• **Robustness.** Parallel DBMSs (that run on TableStores) score poorer than MapReduce systems (that typically run on FileStores) in fault tolerance and ability to operate in a heterogenous environment [2, 12]. MapReduce systems exhibit better robustness due to their frequent checkpoint of completed subtasks, dynamic identification of failed or slow nodes and reassignment of their tasks to other live or faster nodes.

• **Query interface.** Database solutions over TableStore have simple query interfaces: they all support SQL and ODBC, and many of them also allow user defined functions. However, typical queries over performance data are scoped hierarchically, which cannot be naturally supported in a pure TableStore. MapReduce also has flexible query interface. Since Map and Reduce functions are written using general purpose language, it is possible for each task to do anything on its input. However, average performance data analysts may find it cumbersome to write code for data loading, Map, and Reduce functions for everyday queries.

• **Cost.** Apart from the limitations discussed above, an off-the-shelf TableStore solution may be overkill for performance data warehousing. A parallel database is very expensive, especially in a cloud environment (e.g., in Microsoft Azure, the database service is $100\times$ more expensive than the storage service for the same storage capacity). A significant part of the cost is due to expensive mechanisms to ensure high data availability, transaction processing with the ACID property, high concurrency, etc. These properties are not essential for a performance data warehouse where data is read-only, queries are infrequent, and weaker data durability/availability guarantee (e.g., that given by a distributed file system) is sufficient. In contrast, FileStores are cheaper to own and manage than DBMSs. A distributed file system allows simple manipulation of files: files can be easily copied or moved across machines for analysis and older files can be easily deleted to reclaim space. A file system provides the flexibility to compress individual files using domain-specific algorithms, to replicate important files to more machines, to access files according to the file system hierarchy, to place related files together, and to place fewer files in machines with less resource or degraded performance.

Discussion. Ideally, a performance data warehousing system should have the best of both these approaches: the storage flexibility and cost of a file system, compression benefits of column-store databases, query processing performance and simple query interface of a DBMS, and robustness of MapReduce. In the following, we describe our attempt to build such a hybrid system.

4. DATAGARAGE

The architecture of DataGarage follows two design principles. First, data is stored in many TableStores and queries are executed using many parallel instances of a database engine. Second, individual TableStores are stored in a distributed file system. Both these principles contribute to reducing storage footprint. In addition, the first principle gives us query execution performance of DBMSs, while the second principle enables us to use MapReduce-style query execution for its scalability and fault-tolerance.

4.1 The Choice of TableStores

The heterogeneity of performance data collected from different server poses a challenge in deciding a suitable TableStore structure. Consider different options of storing heterogenous counter sets collected from different servers inside a TableStore.

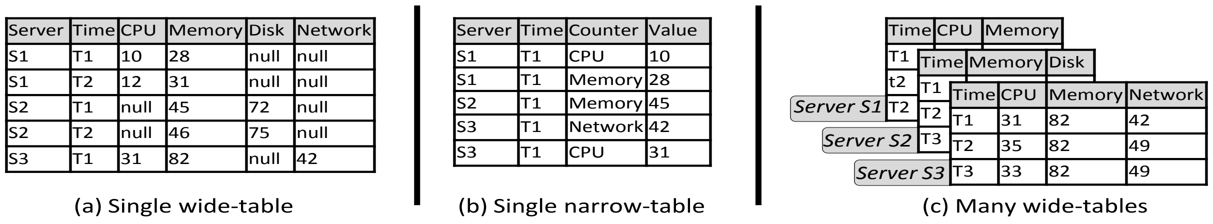
A Wide-table. First consider the *wide-table* option, where data from all servers are stored in a single table, with *one column for each possible counter* across all servers (Figure 2(a)). Then, each row will represent data from a server at a specific timestamp—the counters monitored in that server will have valid values while other counters will have null values. Clearly, such a wide-table will have a large number of columns. In our DataGarage deployment, we have seen around $30,000$ different performance counters from different servers. Hence, a wide-table needs to have that many columns, many more than the maximum number of columns a table can have in many commercial database systems.[2] Even if so many columns can be accommodated, the table will be very sparse, as a small subset of all possible counters are monitored in each server. In our deployment, most servers are monitored for fewer than 1000 counters, and the sets of monitored counters vary across servers. The table will therefore have a very high space overhead.[3]

One option to reduce the space overhead is to create different server types such that all the servers with the same type have the same set of counters. Then, one can create multiple wide-tables, one for each server type. Each wide-table will store data for all servers of the same type. Such an organization will avoid the null entries in the table. Unfortunately, this does not work in practice as a typical data center has too many server types (i.e., most servers are different in terms of *both* their hardware and software counters). Also, rearrangement of logical disks or application mix on a server create new set of counters for the server, making the number of combinations (or, types) simply too big. Moreover, such rearrangements require the server to move from one type to another and its data to span multiple tables over time, complicating the query processing on historical data. Although such rearrangements do not happen frequently for a single server, they become frequent in a population of tens of thousands of servers.

A Narrow-table. Another option to avoid the above problems created by wide-tables is to use a *narrow-table*, with one counter per row (Figure 2(b)). Each column in the original table is translated into multiple data rows of the form (`ServerID`, `Timestamp`, `CounterID`, `Value`). This narrow-table approach allows us to keep data from different servers in one table - even if their counter sets differ. Moreover, since data can be accommodated within a single table, any off-the-shelf DBMS can be used as the TableStore. Before DataGarage, we tried this option for performance data warehousing.

[2]SQL Server 2008 supports at most 1024 columns per table.

[3]This storage overhead can be avoided with SQL Server 2008's *Sparse Columns*, which have an optimized storage for null values. However, this introduces additional query processing overhead and still suffers from the limitation of maximum column count.



Server	Time	CPU	Memory	Disk	Network
S1	T1	10	28	null	null
S1	T2	12	31	null	null
S2	T1	null	45	72	null
S2	T2	null	46	75	null
S3	T1	31	82	null	42

(a) Single wide-table

Server	Time	Counter	Value
S1	T1	CPU	10
S1	T1	Memory	28
S2	T1	Memory	45
S3	T1	Network	42
S3	T1	CPU	31

(b) Single narrow-table

Server S1

Time	CPU	Memory
T1		
t2		
T2		

Server S2

Time	Memory	Disk
T1		
T2		
T3		

Server S3

Time	CPU	Memory	Network
T1	31	82	42
T2	35	82	49
T3	33	82	49

(c) Many wide-tables

Figure 2: Wide- and Narrow-tables

However, this solution has two serious side effects: large storage overhead and limited computability. Since `ServerID` and `TimeStamp` values are replicated in each row, a narrow-table has larger storage footprint than the original binary data. For example, assuming typical server with 200 counters and 15-second sampling interval, the narrow-table solution takes 33MB, which is 7$\times$ higher than the original data size (4.53MB in binary). Then, a one-terabyte disk can hold daily data for only 30,000 servers. Multiple servers are required to hold daily data for 100,000 servers; this moots any attempt to keep historical data for several months.

The narrow-table solution also limits computability. Any query involving multiple counters needs multiple joins on the narrow-table. For example, in Figure 2(b), a query with predicate `CPU>10 AND Memory>20` would involve a join on the `Time` column to link `CPU` and `Memory` attributes from the same sample time. The number of join operations would further increase with the number of counters in the query. This makes a query on a narrow-table significantly longer (in number of lines) and more complicated (often requiring an SQL expert) than an equivalent query on a wide-table. In addition, execution time of such a query is significantly high due to expensive join operations.

►**DataGarage solution.** In DataGarage, we address the shortcomings of above approaches by using *many wide-tables*. In particular, we store data from different servers in separate TableStores (Figure 2(c)). Such a fine-grained data partitioning avoids the overhead of too many columns and the storage overhead due to sparse entries in a single wide-table. It also avoids the space overhead and query complexity of a single narrow-table.

Using one wide-table per server requires maintaining a large number of tables, many more than many off-the-shelf DBMS systems can handle efficiently. We address this using our second design principle of storing the individual TableStores within a distributed file system.

4.2 TableStore-FileSystem Hybrid Storage

To deal with a large number (hundreds of thousands) of TableStores, we store each TableStore as a file in a format accessible by an *embedded database*. In implementing DataGarage, we use SQL Server Compact Edition (SSCE) files [14]. SSCE is an embedded relational database that allows storing an entire database within a single SSCE file (default extension .sdf). An SSCE file can reside in any standard file system and can be accessed for database operations (e.g., update and query) through standard ADO or OLEDB APIs. Accessing an SSCE file requires a lightweight library (Windows DLL file) and does not require installation of any database server application. Each SSCE file encapsulates a fully functioning relational database supporting indices, SQL (and a subset of T-SQL) queries, ACID transactions, referential integrity constraints, encryption, etc.

Storing data within many SSCE files is the key design aspect that gives DataGarage its small storage footprint, the storage simplicity and flexibility of a file system, and query performance of a DBMS. Each SSCE file in DataGarage contains data collected from one

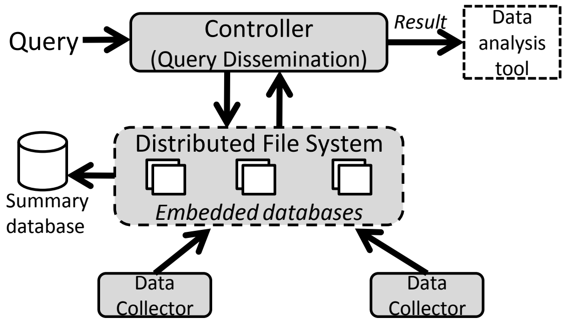


Figure 3: DataGarage Architecture

server over the duration of one day. This allows us to naturally distribute the expensive tasks of loading data into tables and creating appropriate indexes on them among monitored servers. Since data from different days are stored in different files, deleting older data simply requires deleting corresponding files. The files are named and organized in a directory structure that naturally facilitates selecting a subset of files that contain data from servers within a datacenter, and/or a given owner, and/or within a range of dates using regular expressions on file names and paths. For example, assuming that files are organized in a hierarchy of server owners, datacenters, and dates, all data collected from `hotmail.com` servers in the datacenter `DC1` in the month of October, 2009 can be expressed as `hotmail/dc1/*.10-*-2009.sdf`.

Figure 3 shows the architecture of DataGarage. The *Data Collector* is a background process that runs at every monitored server and collects its performance counter data. The set of performance counters and data collection interval are configured by server owners. The raw data collected by collectors are saved in as SSCE files in a distributed file system. A *Summary Database* maintains hourly and daily summaries of data from each server. This enables efficiently running frequent queries on summary data and retaining summary data even when the corresponding raw data is discarded due to storage limitation. The *Controller* takes queries, processes it, and outputs the results in various formats, which can further be pushed to external data analysis tools for additional analysis.

Another advantage of using independent SSCE file for each server is that the owner of a server can independently define its schema (i.e., the set of counters to collect data from) and tune it for appropriate queries (e.g., by defining appropriate indices). The column name for a counter is the same as the counter name reported by the data collector. It is important to note that the same data collector is used in all monitored servers and it uses the same name for the same (or, semantically equivalent) counter across severs. For example, the data collector names the amount of available memory as `TotalMemoryFree`, and hence database files for all servers that have chosen to collect this specific counter will have a column with name `TotalMemoryFree`. Such uniformity in column naming is essential for processing queries over data from multiple servers.

4.3 Reducing Storage Footprint with Compression

As mentioned before, the most important design goal of DataGarage is to reduce the storage footprint and network bandwidth to store performance data (or to increase the amount of performance data within available storage). Our design principle of using many wide-tables already reduces storage footprint compared to alternative approaches. We use data compression to further reduce storage footprint. However, lossless compression techniques available in off-the-shelf DBMSs do not work very well for floating-point performance data. For example, our experiments with real dataset show a compression factor of only two by using the compression techniques in SQL Server 2008. Such a small compression ratio is not sufficient for DataGarage.

To address this, we have developed a suite of compression algorithms that work well for performance data. Since data in DataGarage is stored as individual files, we can use our custom algorithms to compress these files (and automatically decompress them before query processing). To compress each SSCE file, we first extract all its metadata describing its schema, indices, stored procedures, etc., and compress them using standard lossless compression techniques such as Lempel-Ziv. The bulk part of each file is its tables, and they are compressed using the following techniques.

4.3.1 Column-oriented Organization

Following observations from previous works [1, 9], we employ a column-oriented storage in DataGarage: inside each compressed file, we store data from the same table and column together. Since performance data comes from temporally correlated processes, such a column-oriented organization increases data locality and compression factor. This also improves query processing time as only the columns that are accessed by a query can be read off the disk.

4.3.2 Lossless Compression

A typical performance data table contains few timestamp and integer columns and many floating point columns. The effective compression scheme for a column depends on its data type. For example, timestamp data is most effectively compressed with delta encoding followed by a run-length encoding (RLE) of the deltas. Delta encoding is effective due to small sampling periods. Moreover, since a single file contains data from a single server and sampling period (or, delta) is a constant for each server, RLE is very effective to compress such deltas. Integer data is compressed with variable-byte encoding. Specifically, we allow integer values to use a variable number of bytes and encode the number of bytes needed to store each value in the first byte of the representation. This allows small integer values to be encoded in a small number of bytes.

Standard lossless compression techniques, however, are not effective for floating point data due to its unique binary representation. For example, consider the IEEE-754 single precision floating point encoding, the widely used standard for floating point arithmetic. It stores a number in 32 bits: 1 *sign* bit, 8 *exponent* bits, and 23 *fraction* bits. Then, a number has value $v = s \times 2^{e-127} \times m$, where s is +1 if the sign bit is 0 and -1 otherwise, e is the 8-bit number given by the exponent bits, and $m = 1.fraction$ in binary. Since a 32-bit representation can encode only a finite number of values, a given floating point value is mapped to the nearest value representable by the above encoding.

Since floating point data coming from a performance counter changes almost at every sample, techniques such as RLE do not work. Moreover, due to unique binary representation of floating point values, techniques such as delta encoding or dictionary-based compression are not very effective. Finally, a small change in the decimal values can result in a big change in the underlying binary representation. For example, the hexadecimal representations of IEEE-754 encoding of the decimal values 80.89 and 80.9 are `0x42A1C7AE` and `0x42A1CCCC`, respectively. Even though the two numbers are within 0.01% of each other, their binary representations differ in the 37.5% least significant bits. Lossless compression schemes that do not understand the semantics of binary representations of numbers cannot exploit the relative similarity of the two numbers just by looking at their binary representations.

►**Byte-interleaving.** To address the above problem, we observe that *a small change in values results in changes in the lower-order fraction bits only*; the sign bit, the exponent bits, higher-order fraction bits remain the same. Since data in a column represents temporally correlated data from the same server collected relatively frequently, subsequent values show small changes. To exploit this, we use *byte-interleaving* as follows. Given a column of floating point values, we first store the first bytes of all values together, then we store their second bytes together, and so on. Since higher order bytes do not change for small changes, such an organization significantly improves compression factor, even with simple compression techniques such as RLE or dictionary-based compression. In some sense, byte-interleaving is an extreme case of column-oriented organization, where each byte of the binary representation of a floating point value is treated as a separate column.

4.3.3 Lossy Compression

DataGarage supports an optional lossy compression technique. Performance data warehouse can typically tolerate some small (e.g., $< 0.1\%$) loss in accuracy of archived data for following reasons. First, due to their cheap, sampling-based data acquisition process, data collectors often introduce small noise and hence the data is not treated as precise. Second, most of the time the data is analyzed in aggregation and hence a small error in raw data does not significantly affect the accuracy of outputs. On the other hand, tolerating a very small decompression error can result in a very high compression factor, as we show in our evaluation.

An important design decision is to choose the appropriate lossy compression algorithm. Each column in a table is essentially a time-series, and prior work has shown many different lossy compression techniques including DFT, DCT, Wavelet transform, random projection, etc. [13]. Most of these techniques guarantee or minimize average reconstruction error (or, L_2 norm). Such techniques are not suitable for DataGarage since they can lose local spikes in the time series, which are extremely important in applications intended for local anomaly detection. Techniques such as Piecewise Linear/Constant Approximation guarantees worst-case (L_∞) reconstruction error, but there effectiveness in compression comes from smoothness of data [3]. Performance data (e.g., CPU or memory utilization) is rarely smooth and is dominated by frequent jitters and spikes. Our experiments show that using PLA or PCA gives very small compression factor for performance data, and in some cases data cannot be compressed at all.

►**Bit Truncation.** We use a novel IEEE-754 floating point compression algorithm for compressing noisy floating point data with worst-case decompression error. The algorithm is called *bit-truncation*, and is based on the observation that removing a small number of fraction bits from the IEEE-754 representation introduces a small and *bounded* relative error[4] in the reconstructed value. More specifically, we claim that

[4]If a value v is reconstructed as v', the relative reconstruction error is given by $|v - v'|/v$.

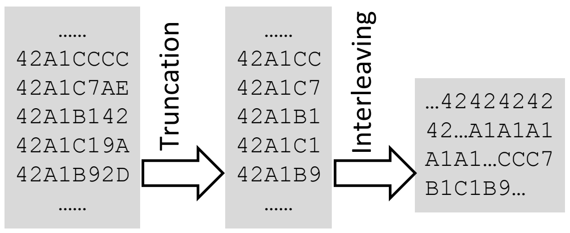


Figure 4: Bit-truncation and byte-interleaving of floating point representations.

CLAIM 1. *Replacing k least significant fraction bits of a IEEE-754 32-bit single precision (or 64-bit double precision) floating point representation with zero bits introduces an relative error of* $\leq \sum_{i=0}^{k-1} 2^{i-23}$ *(or* $\leq \sum_{i=0}^{k-1} 2^{i-52}$*, respectively).*

We omit the proof of the claim for brevity. Quantitatively, removing 8 and 16 lowest order bits of a 32-bit single precision representation result in relative errors of only $\leq 6.1 \times 10^{-5}$ and ≤ 0.16 respectively. For 64-bit double precision representation, the effect is even more negligible. For example, even after removing the 32 least-significant bits, the relative reconstruction error is guaranteed to be $\leq 1.3 \times 10^{-6}$.

Figure 4 shows how a column of floating point numbers is compressed using truncation and interleaving. First, depending on the maximum tolerable relative error, least significant bits of each number of the column are truncated. This step is done only if a lossy compression is allowed. If the number of truncated bits is not a multiple of 8, remaining bits are packed together into the minimum number of bytes. After truncation, individual bytes are interleaved into stripes. Finally, different stripes are compressed using lossless compression (e.g., RLE or Lempel-Ziv).

4.4 Data Thinning

DataGarage periodically needs to discard existing data from its storage. Such *data thinning* discards high-fidelity raw data; the corresponding aggregate summaries may still be left in the summary database. Data thinning is operationally important as it allows gradual reduction of the size of archived data, and it can happen in many scenarios including the following.

1. Operational restrictions such as storage limitations of the system and privacy considerations of the data can force dropping data older than a certain date.
2. Many servers have days when they are not used much. For such days, it is sufficient to keep only aggregate (hourly) data in the summary database and drop the raw data.
3. Even in heavily used servers, some counters are less important than others—especially after the data is older than a week or a month.

The design choices we have made for DataGarage makes data thinning simple and efficient. Since typical data thinning granularity is multiple of a day and data from one server over a day is stored in a separate file, data thinning in DataGarage does not involve any database operations—it involves simply selecting the target files using regular expression on file names and deleting them. Data thinning by dropping less important columns involves operations inside files; but due to our column oriented organization of the compressed database files, such operations can be done efficiently within compressed files. In contrast, if data were stored in a single parallel DBMS, data thinning could be very expensive as it might involve bulk deletion, index update, and even schema change.

4.5 Schema optimization

In our original design of DataGarage, each database file contained one wide-table called `RawData`, containing data from all counters of a server. However, based on operational experience, we realized that certain queries cannot be efficiently supported on this simple schema. So, we added two additional elements into the schema to facilitate those queries.

Separate tables for multiple-instance counters. We remove all counters with multiple instances from the `RawData` table and put them in separate tables in the same SSCE file. For example servers typically have multiple instances of physical disks, and each physical disk has a set of counters. Therefore, we create a separate table for physical disks with one disk instance per row and one disk-related counter per column. This simplifies certain types of queries. For example, consider the query of computing total disk space in all non-system disks (i.e., disks with instance number higher than 0) in each server. With a separate disk table, this can be expressed simply as `SELECT Sum(AvailableSpace) FROM DiskTable WHERE InstanceID>0`. This would not have been so simple if all disk instances were stored in the `RawData` table as column names `Disk0Bytes`, `Disk1Bytes`, etc., and different servers have different numbers of physical disks (e.g., the `Disk5Bytes` column may be available in some disk tables and unavailable in others). Like physical disks, logical disks, processors, network cards are also kept in separate tables.

Separate instance tables have two additional benefits. First, this helps keeping the number of columns in the `RawData` table less than 1024, the maximum number of columns possible in a table inside SSCE file. Second, queries over instance tables run faster as they need to access tables significantly smaller than the main `RawData` table.

Identification of 'previous' sample time. DataGarage sometimes needs to compare data in temporal order of their collection timestamps. For example, often data analysts are not interested in the absolute value of a counter, but in the change of its values—e.g., How did processor utilization grew from the last time? How many times was CPU utilization over a threshold, excluding the isolated spikes between two low utilization samples? This pattern of comparing data in temporal order occurs in many classes of analysis. Unfortunately, relational databases are inefficient in handling such pattern. To address this, we make the data collector to report the 'previous timestamp' with each sample and store this value with each record in the main table. This allows us to retrieve previous sample of a server by using self-join on timestamp and 'previous timestamp' columns (see an example in Section 5).

5. QUERY PROCESSING

Since data in DataGarage is stored in many small files in a distributed file system, a MapReduce-style query processing system seems natural for DataGarage. We have developed such a system, which is optimized for typical queries in DataGarage.

5.1 DataGarage Queries

A DataGarage query (or, DGQuery in short) runs over a collection of SSCE files and outputs a single SSCE, or Excel, or CSV file containing the result. Encapsulating the output as an SSCE file enables us to pipeline a sequence of DGQueries and to easily use the output in external data processing applications that can directly load SSCE files. A DGQuery has the following general syntax:

```
APPLY <apply_script>
ON <source>
COMBINE <combine_script>
```

The query has three components.

1. The `<apply_script>` is applied to a set of input SSCE files to produce a set of output files (in SSCE, CSV or Excel format). The script is applied to multiple input files in parallel.

2. The `ON` statement of a DGQuery specifies a set of input SSCE files for the query to operate on. The set of files can be expressed with a regular expression on filesystem path or with a text file containing an explicit list of files. The `source` can also be another DGQuery, in which case, output of one query acts as input for another. This enables hierarchical data aggregation by recursively applying one DGQuery on the output of another DGQuery.

3. The `<combine_script>` is applied to a set of SSCE files (outputs of the Apply scripts) to produce a single output file (SSCE, CSV, or Excel format).

Note that the same `<apply_script>` is applied to many SSCE files with potentially different schemas. With a global catalog of what counters are monitored at which server, DataGarage can perform compile-time validation of whether the counters mentioned in the `<apply_script>` are present in all input SSCE files. During run time, DataGarage ignores the input SSCE files that do not contain all counters in the `<apply_script>` and includes the count of such ignored SSCE files with query results as a completeness indication.

To illustrate, we here give a few simple example queries in DataGarage. In practice, DataGarage queries are more complicated as they involve more counters and predicates.

• **Query1.** *Find 10 servers with maximum average memory usage among all* `hotmail.com` *servers in the datacenter* `DC1` *in the month of October 2009. Consider only samples with nontrivial cpu and disk utilization (*`CPUUsage < 0.2` *and* `DiskIdle <0.02`*).*

```
APPLY "Select ServerID, Avg(MemUsage) as AvgMem
      From RawData
      Where CPUUsage<0.2 AND DiskIdle<0.02
      Group by ServerID"
ON //hotmail/dc1/*.10-*-2009.sdf
COMBINE "Select Top 10 ServerID, AvgMem
      from ApplyResult
      Order by AvgMem Desc"
```

The Apply script above computes average memory usage of all servers. The `ON` statement scopes the query to the appropriate set of SSCE files. The example shows how DataGarage uses file system hierarchy to define hierarchical scope of a query. Finally, the Combine script computes the top 10 servers based on average memory usage. (The table `ApplyResult` in the Combine script above is a virtual table that contains all data output by the Apply script.)

• **Query 2.** *Compute sequences of 15-minute average CPU usage of all servers.* The Apply script is as follows (we omit the Combine script as it simply concatenates outputs of the Apply script).

```
Select ServerID, _Mins15Time; as Mins15,
       Avg(CPUUsage) as AvgCPUUsage
From RawData
Group by _Mins15Time; order by _Mins15Time;
```

The keyword `_Mins15Time;` denotes a predefined macro that produces the 15-minute interval of a sample time.

• **Query 3.** *Compute disk response time for non-trivial situations in the system.* Computing this accurately is tricky since disk response time is affected by disk paging and frequently we observe isolated peaks of counter values. For example, the time series from the "% Disk Busy" counter (`pctDiskBusy`) may look like: $\ldots, 0, 0, 6, 0, 0, \ldots, 0, 3, 8, 7, 2, 0, 0, \ldots$. We must be careful not to include the response time for utilization 6, as it is a momentary aberration. So, to obtain better estimate of disk response time, we want compute the response times only in situations when (i) `pctDiskBusy` really nontrivial, e.g. $> 5\%$, (ii) the previous sample had nontrivial `pctDiskBusy`, e.g., $> 1\%$, and (iii) there is no significant paging. This can be expressed using the following Apply script.

```
Select r.serverID, r.sampleTime,
       r.pctDiskBusy, r.diskResponseTime
From RawData as r
Join RawData as rprev
       on r.prevSampleTime = rprev.sampleTime
Where r.pctDiskBusy > 5 and rprev.pctDiskBusy > 1
       and r.paging < 10 and rprev.paging < 10
```

Note that our wide-table approach makes the above queries compact and relatively simple. All of them would be significantly longer and complicated if data were organized as a narrow-table.

5.2 Query Execution

A DGQuery is executed in two phases: an Apply phase when the `<apply_script>` is applied on input files and a Combine phase when the `<combine_script>` is applied on the outputs of the apply phase. The Controller module of DataGarage performs the necessary synchronization between these two phases. At a high level, the Apply and the Combine phase resemble the Map and the Reduce phase of the MapReduce framework.

To see how a DGQuery is executed, consider a simple scenario where the input files are given as a regular expression on filesystem path and the query is run on a single machine. The Controller first enumerates all the input files (using the filesystem directory structure). Then it starts the Apply phase, where it invokes multiple parallel Apply threads, each of which processes the sequence of input files and serializes the results to temporary SSCE files. To process a compressed SSCE file, the Apply thread decompresses the relevant columns from the file and applies the `<apply_script>` inside an embedded database. After all the apply threads terminate, multiple temporary SSCE files, each containing the intermediate results of applying the `<apply_script>` to one input database file, reside in the file system. Since the same `<apply_script>` runs on all input files, the intermediate files are in fact horizontal partitions of a larger database tables. Finally, the Controller starts the *combine phase*, where it constructs a single SSCE file, with a virtual table called `ApplyResult` by concatenating data from all intermediate files and applies the `<combine_script>` on the combined file to produce the final result.

A DGQuery can also run on multiple machines, as shown in Figure 5. In that case, the Controller is configured with a list of

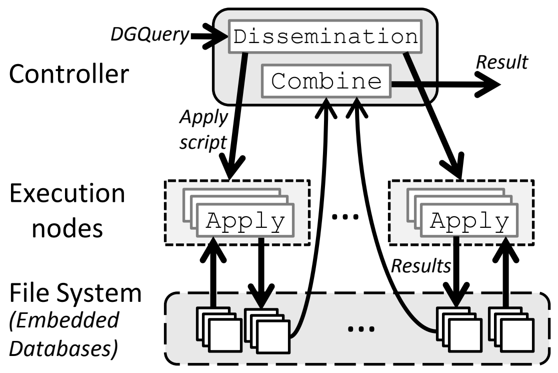


Figure 5: Query Execution in DataGarage

execution nodes, each of which has access to the filesystem storing DataGarage data. To run a query, the Dissemination module of the Controller partitions the input database file names and sends the partitions to available execution nodes. If explicit locations of input files are known, execution nodes are chosen as close as possible to the file sources. Each execution node then runs the `<apply_script>` on its portion of the database files. The outputs of the apply phase are written to temporary SSCE files in the distributed filesystem. Finally, the controller runs the `<combine_script>` on the intermediate results.

In principle, the combine phase with decomposable functions can be made parallel as well, e.g., by running the combine function along a tree hierarchy. However, we have found that the combine phase in a typical DataGarage query, such as aggregation, filtering, anomaly detection, etc. needs to deal with a relatively small amount of data and hence running the combine phase in a single execution node is sufficient.

5.3 Robustness

DataGarage uses several techniques to deal with faulty or slow nodes. The underlying file system uses replication, and hence data is available during query processing even if a small number of storage nodes are down. To cope with faulty execution nodes during query processing, the Controller monitors liveness and progress of each execution node. Liveness is monitored by periodic heartbeat messages, while progress of each node is monitored by examining the number of temporary intermediate files it has produced so far. If a node fails during the Apply phase, the controller determines the list of input files yet to be processed by the node and distributes the processing of these remaining files among other live execution nodes (by simply sending them additional lists of files to process). Thus a query does not need to be restarted from the beginning due to the failure of an execution node. Moreover, only the unfinished portion of the task at the faulty node needs to be redistributed, thanks to the small granularity of inputs to each task.

DataGarage copes with heterogenous nodes by using two techniques. First, during query dissemination, the Controller assigns less work (i.e., fewer input files to process) to nodes that are known to be slower. However, seemingly homogenous machines with similar tasks can perform very differently in practice. For example, two similar machines can process the same query in different speeds if they have different degrees of disk fragmentations or if one accesses data from its own physical rack in the datacenter but the other accesses data from a far away rack. To avoid a slow node from becoming the bottleneck, whenever a fast node completes its share of the Apply task, it starts working on the remaining task of the slowest node. To make this happen, the Controller node creates a list of the input files the slow node is yet to process and sends the second half of the list to the faster node. Thus, some tasks of slower nodes may be replicated in faster nodes, and the Apply phase finishes when all the files have been processed by at least one execution node.

Like many MapReduce systems, the Controller remains the single point of failures. However, by using a node with good hardware and software configuration as the Controller, the probability of its failure during processing of a query can be made very small. If further reliability of the Controller is desired, two (or more) Controller nodes can be used where the secondary Controller can take the control after the primary one fails. Note that since the results of the Apply phase are persisted to the file system, failure of one Controller does not require running the Apply phase again—the new Controller can simply start with the intermediate results in the filesystem.

6. EXPERIMENTS

In this section, we evaluate DataGarage with a real workload and a shared-nothing cluster.

Dataset. We use performance data collected over one day from 34,946 servers in several Microsoft datacenters. Thus, the data is archived as 34,946 SSCE files in a distributed file system. The total size of the dataset is around 220GB. The minimum, maximum, average, std. deviation, and median of the file sizes are 20KB, 11.2MB, 6.4MB, 5.4MB, and 2.1MB respectively. The high standard deviation of file sizes implies high heterogeneity of counter data sets collected from different servers.

Computing nodes. We use a Windows High Performance Computing (HPC) cluster of 32 2.5GHz nodes, each having 8 cores and 16GB RAM. The execution granularity in the cluster is a core, and hence the cluster allows us to use up to 248 cores in parallel in 31 nodes (except the head node of the cluster). The head node of the cluster is used as the DataGarage Controller node, which schedules Apply tasks on other nodes. The Combine tasks are executed at the Controller node.

Queries. We use the three queries mentioned in Section 5 in our evaluation. The queries exercise different aspects of query execution. **Query1** has a nontrivial Combine script (Combine phases in other queries simply concatenate outputs of Apply scripts). **Query2** has more I/O overhead than **Query1**, as its Apply script produces and writes to disk a larger output. **Query3**, in addition to having a large intermediate results, involves a self join and hence is computationally more expensive than the other queries.

6.1 Compression

We first evaluate the most important aspect of DataGarage: its storage efficiency. The storage efficiency comes from two factors. The first one is its organizing data in many wide-tables. On our dataset, this approach reduces storage footprint by 7× compared to the narrow-table approach mentioned in Section 3. The second factor contributing to DataGarage's storage efficiency is its data compression techniques. To show the benefit, we compare DataGarage's compression and decompression algorithms, which we denote as `DGZip` and `DGUnzip` respectively, with popular `Zip` and `Unzip` algorithms.

Figure 6 and Table 1 show the distribution of compression factors achieved by different algorithms. We use `DGZip` with three configurations: `DGZip` denotes lossless compression, while `DGZip(0.16)` and `DGZip(0.00006)` denote lossy compression with *maximum*

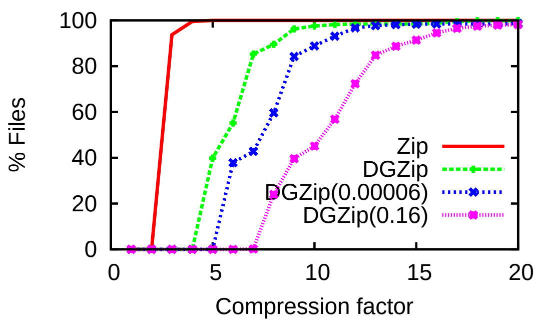


Figure 6: Cumulative distribution of compression factors

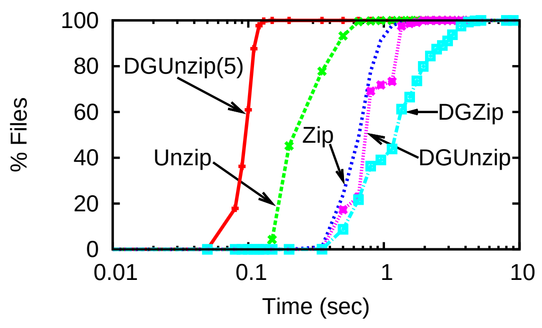


Figure 7: Cumulative distribution of compression and decompression time

Table 1: Compression factor

Compression Scheme	*Compression factor*			
	Min	Max	Average	Std. Dev
`DGZip`	4.03	21.7	5.9	1.99
`DGZip(0.00006)`	5.1	28.25	7.6	2.5
`DGZip(0.16)`	6.8	41.74	10.8	3.6
`Zip`	2.1	4.7	2.5	0.4

Table 2: Compression/decompression time

Compression Scheme	Time (sec)			
	Min	Max	Avg.	Std. Dev.
`Zip`	0.11	3.32	0.67	0.2
`Unzip`	0.08	1.98	0.25	0.12
`DGZip`	0.09	5.03	1.3	0.8
`DGUnzip`	0.27	1.97	0.72	0.22
`DGUnzip(5)`	0.05	0.19	0.1	0.014

relative decompression error of 0.16 and 0.00006 respectively. As shown, `Zip` provides very little compression for our dataset (the average compression factor is 2.5).[5] In contrast, `DGZip` achieves an average compression factor of 5.9, more than 2× higher than `Zip`'s compression factor. The high compression factor comes from column-oriented organization and byte-stripping technique used by `DGZip`. The compression factor further increases with lossy compression. As shown, even with a very small relative decompression error of 0.00006, `DGZip` can provide a compression factor of 7.6, a 3× improvement over `Zip`.

The high compression factor of `DGZip` comes at the cost of its higher compression/decompression time compared to `Zip/UnZip`. Figure 7 and Table 2 show the distribution of compression and decompression times of different algorithms. The compression time of `DGZip` is independent of the decompression error, and hence we report the time of lossless compression only. However, since `DGUnzip` allows efficiently decompressing only few selected columns from a table, its decompression time depends on the number of columns to decompress. In Figure 7 and Table 2, we consider two configurations: `DGUnzip` that decompresses the entire database, and `DGUnzip(5)` that decompresses only 5 columns (corresponding to 5 performance counters) from a table. The results show that `DGZip` and `DGUnzip` are $\approx$ 2-3× slower than `Zip` and `UnZip`. High latency of `DGZip` is tolerable, as data is compressed only once, during collection. With an average compression time of 1.3 seconds, `DGZip` on a 8-core machine can compress data from 100,000 servers in less than 6 hours. However, reducing decompression latency is important as data is decompressed on the fly during query processing. Fortunately, even though `DGUnzip` is expensive, most queries are run over a relatively small number of columns, and using `DGUnzip` to decompress only the relevant columns from a compressed SSCE file is very fast. As shown in the figures, `DGUnzip(5)` is 60% faster than `UnZip`, which decompresses the entire file even if only a few columns are required for query processing. Another advantage of `DGUnzip`'s column-oriented decompression is that the decompression time is independent of the total number of columns in the table, as shown by the very small variance of decompression times of `DGUnzip(5)`.

[5]With SQL Server 2008's row- and page-compression techniques, we found a compression factor of $\approx$ 2 for our dataset.

6.2 Query processing

► **Scalability of different queries.** To understand how data analysis on DataGarage scales with the number of query execution nodes, we run all three queries on our 32-node, 256-core Windows HPC cluster. All queries run on the entire dataset, and we vary the number of query execution cores. The cores are evenly distributed among 31 non-head nodes of the cluster. We report average completion time of five executions of the queries.

Figure 8 shows the total execution time of different queries as a function of the number of execution cores. Even though the absolute execution time depends of processing power and I/O bandwidth of execution nodes, the figure makes a few general points. First, **Query1** is the fastest. **Query2** is slower due to its additional I/O overhead for writing larger intermediate results. **Query3** is the slowest as it involves, in addition to its high I/O overhead for writing larger intermediate results, an expensive join operation. This also highlights a performance problem with a narrow-table approach, where every query having multiple performance counters (even **Query1** and **Query2**) would involve multiple join operations, making the queries run extremely slow. In contrast, most common aggregation queries can be expressed without any join in our wide-table approach, making the performance of **Query1** and **Query2** representative of typical DataGarage queries.

Second, for all queries, the execution latency decreases almost linearly with the number of execution cores. For **Query1** and **Query2**, the scaling becomes sublinear after 62 cores as I/O becomes the bottleneck when multiple parallel instances of a query run on the same node. In contrast, **Query3** does not see such behavior, as CPU is the bottleneck for the query. In general, the overall execution time of a typical non-join query is dominated by I/O cost. In the above experiments, each node had a peak disk bandwidth of only 6MB/sec. In our experiments, both **Query1** and **Query2** consumed $< 5\%$ CPU time per core and disk idle time approached zero when more than two cores per node were running queries (which

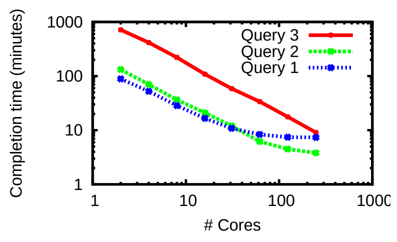


Figure 8: Query completion time

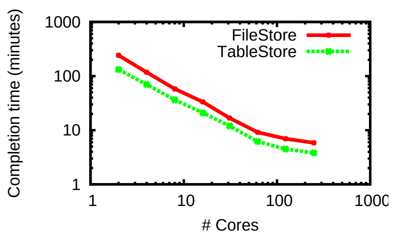


Figure 9: Query2 completion time

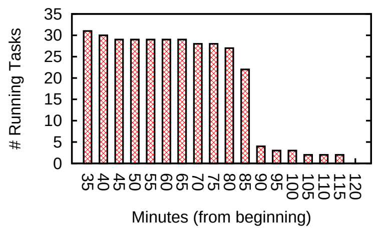


Figure 10: Effect of stragglers

explains the sub-linear scaling in Figure 8 after 62 cores). In a separate experiment with nodes configured with much faster disks (up to 92MB/sec peak bandwidth), we observed a linear decrease in execution time even with 8 cores per node.

Finally, **Query1** has a smaller slope that other two queries. This is due to a higher overhead of **Query1**'s Combine phase, which cannot be parallelized.

► **Comparison with a FileStore.** We also compared DataGarage with a pure FileStore-based solution. We consider a hypothetical MapReduce-style execution, where input data is read from a binary file and the Apply script (i.e., the Map function) parses and loads the data during query time. Figure 9 shows the execution times for these two approaches for **Query2**. As shown, the query runs almost $2\times$ faster than on TableStore than in FileStore. This highlights the benefits of preloading data into tables and pushing queries inside databases

6.3 Heterogeneity and fault tolerance

Heterogeneity Tolerance. Even though we used a cluster of nodes with similar hardware and software configurations and allocated similar amount of tasks (in terms of the number of database files to process) to each node, surprisingly, we observed that some nodes finished execution of their tasks much faster than others. To illustrate, consider an experiment where we executed **Query1** in 31 cores in 31 nodes. Figure 10 shows the number of nodes still executing their assigned tasks over the entire duration of the execution of the Apply phase of **Query1**. As shown, two nodes finished execution within the first 45 minutes, all of the remaining but four finished execution within 85 minutes, and 2 nodes took more than 100 minutes to finish execution. After closer examination of the slower nodes (that took more than 85 minutes to execute), we identified two reasons behind their running slow. First, even though all nodes were given the same number of input files, slower nodes had larger average file sizes than faster nodes. This is possible since our input files have a large variance in size as the number of performance counters monitored in different servers vary a lot. Second, the slower nodes had slower disk operations due to disk fragmentation. More specifically, slower nodes and faster nodes had $> 40\%$ and $< 5\%$ of their files fragmented, respectively. This caused slower nodes to have 15% less disk throughput than faster nodes. Since the Combine phase starts after all Apply tasks (including the ones in the slowest node) finish, this considerably increases the overall query execution time.

As mentioned in Section 5.3, DataGarage schedules unfinished portion of a slower node's task in a faster node after the faster node has finished execution of its own tasks. For example, in the above scenario, after the fastest node finishes executing its own task, the Controller examines the progress of remaining nodes (by looking at how many output files they have generated). Then, it assigns half the input files of the slowest node to the fastest node. In addition, it writes in a file the list of input files the fastest node has started working on, so that the slowest node can ignore them. This simple technique significantly improves the overall execution time. When running **Query1** on 31 nodes, we observed a reduction of 25% in the overall execution time (from $\approx$ 112 minutes to $\approx$ 82 minutes).

Fault Tolerance. To test fault tolerance of DataGarage's query execution, we executed **Query1** on 10 nodes, with one Apply task on each node. Then we terminated one node after it has completed 50% of its task. As mentioned in Section 5.3, when the Controller node detects failure of a node due to absence of periodic heartbeat messages, it redistributes the *remaining* task of the failed node to other available nodes. Since the other nodes now have more to do, the overall execution time increases.

We observed that, as a result of the above failure, the overall execution time increased by 7.6%. Note that since DataGarage assigns tasks at the granularity of a file, only the unfinished portion of the faulty node's task need to redistribute. Therefore, the overall slowdown depends on when a node fails. The more a node processes before it fails, the less is the additional tasks for other nodes. Our experiments show that if a node fails after 75% and 90% completion of its task, the overall slowdown becomes 4.8% and 3.1%. We also simulated a HadoopDB-like policy of distributing the whole task of the faulty node to other nodes, and observed a slowdown of 13.2%. This again highlights the advantage of small input granularity of DataGarage.

7. OPERATIONAL EXPERIENCE

We have been using a production prototype of DataGarage for last one year to archive data collected from many tens of thousands of servers in Microsoft datacenters. We here discuss some of the lessons we have learnt over this time.

Performance data warehousing is mostly about storage and computability and our compressed, wide-table storage has been a key to DataGarage's success. Before designing DataGarage, we made an attempt to use narrow-tables. The decision was natural because it supports heterogeneous sets of counters and can be stored inside any off-the-shelf DBMS. However, we soon realized that such a design severely limits the amount of data we can archive as well as the type of computations we can perform. As mentioned before, a narrow-table has a high storage overhead. Compression algorithms perform poorly too as data loses locality in a narrow-table. As a specific example, with narrow-table, we could store 30,000 server-days worth of data in a single 1TB disk. In contrast, with our compressed wide-table scheme, DataGarage can archive 1,000,000 to 3,000,000 server-days worth of data on the same amount of storage. In many situations, a significant portion of all DataGarage data can be stored in one or two storage servers, which significantly reduces operational overhead of the system.

Narrow-tables also limit computability. A typical query on multiple counters involves multiple self-joins on the table, making the query long and error-prone and extremely slow to run. For example, a narrow-table equivalent of the example Query3 in Section 5 requires tens of lines in SQL and runs orders of magnitude slower than the same query on a wide-table. Moving to wide-table gave DataGarage a significant benefit in terms of storage footprint and computability.

We also experienced several unanticipated benefits of storing data as SSCE files in a file system. First, we could easily scavenge available storage from additional machines that were originally not intended for DataGarage warehousing. Whenever we discover some available storage in a machine, possibly used for some other purpose, we use it for storing our SSCE files (the Controller node remembers the name of the new machine). Had we used a pure DBMS approach for data archival, this wouldn't have been such easy since we had to statically allocate space on these new machines and to connect them to the main database server. Second, SSCE files simplify the data backup problem as a backed-up SSCE file can be accessed in the same way the original SSCE file is accessed. In contrast, to access backup data from a DBMS, the data must first be loaded into a DBMS, which can be extremely slow for a large dataset.

A practical lesson we learnt from DataGarage is that it is important to keep the system as simple as possible and to use as many existing proven tools as possible. The DataGarage system is extremely simple—it uses existing file systems, scripting shell, SSCE files, Windows SQL Server Compact Edition runtime library, and a several thousand lines of custom code to glue everything together. The outputs of a DGQuery can be a SSCE file, a SQL Server database files, a Microsoft Excel file, or a CSV file—so that the output can be used by another DGQuery or be loaded into SQL Server, Excel, MatLab or R. We also found that having visibility of query execution is extremely useful to detect and identify effects of node failure. For example, since our Apply phase writes output results as separate files, just by looking at the number of output files at different execution nodes help us to easily deal with faulty or slow nodes. Another lesson we learnt is that it is important to delay adding new features until it is clear how the new features will be used. For example, even though, in principle, it is possible to parallelize Combine phase for certain functions, we delayed such implementation and later found that the Combine phase typically deals with a small amount of data and hence running it on a single node is sufficient.

There appears to be a natural fit between DataGarage and cloud computing, and we can use DataGarage at various stages in a cloud computing platform for its attractive pay-as-you-go model. For example, we can use the cloud (e.g., Microsoft Azure or Amazon EC2) to store data only (e.g., SSCE files). Data can then be downloaded on demand for processing. To execute expensive queries or to avoid expensive data transfer, we can use the cloud for executing our MapReduce-style query engine as well. Thus, we can seamlessly move DataGarage to a cloud computing platform without any significant change in our design and implementation. Our design decision of storing data as files, rather than as tables inside a DBMS, again shows its worth: DataGarage on the cloud will use a file storage service only, which is much cheaper than a cloud DBMS service. For example, Windows Azure (which supports file storage and program execution) is $100\times$ cheaper than Microsoft SQL Azure (which provides a relational database solution) for the same storage capacity.

8. CONCLUSION

We described the design and implementation of DataGarage, a performance data warehousing system that we have developed at Microsoft. DataGarage is a hybrid solution that combines benefits of DBMSs, file-systems, and MapReduce systems to address the unique requirements of warehousing performance data. We described how DataGarage allows efficient storage and analysis of years of historical performance data collected from many tens of thousands of servers—on commodity servers. Our experience of using DataGarage at Microsoft for the last one year shows significant performance and operational advantage over alternative approaches.

Acknowledgements. We would like to thank the Microsoft SQL Server Compact Edition team, specifically Murali Krishnaprasad, Manikyam Bavandla and Imran Siddique, for technical support related to SSCE files, Anu Engineer for helping on the first Windows Azure version of DataGarage, Julian Watts for his help in implementing the initial version of the DataGarage system, and Jie Liu for useful inputs on compression algorithms.

9. REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *ACM SIGMOD*, 2006.

[2] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. In *VLDB*, 2009.

[3] C. Buragohain, N. Shrivastava, and S. Suri. Space efficient streaming algorithms for the maximum error histogram. In *ICDE*, 2007.

[4] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2), 2008.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Usenix OSDI*, 2006.

[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Usenix OSDI*, 2004.

[7] Hadoop. `http://hadoop.apache.org`.

[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[9] S. Khoshafian, G. P. Copeland, T. Jagodis, H. Boral, and P. Valduriez. A query processing strategy for the decomposed storage model. In *ICDE*, 1987.

[10] A. Mueen, S. Nath, and J. Liu. Fast approximate correlation for massive time-series data. In *ACM SIGMOD*, 2010.

[11] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *ACM SIGMOD*, 2008.

[12] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *ACM SIGMOD*, 2009.

[13] G. Reeves, J. Liu, S. Nath, and F. Zhao. Managing massive time series streams with multiscale compressed trickles. In *VLDB*, 2009.

[14] P. Seshadri and P. Garrett. SQLServer for Windows CE - A Database Engine for Mobile and Embedded Platforms. In *ICDE*, 2000.