DOI:10.1145/1536616.1536632

Article development led by acmqueue
queue.acm.org

Scale up your datasets enough and your apps come undone. What are the typical problems and where do the bottlenecks surface?

BY ADAM JACOBS

The Pathologies of Big Data

WHAT IS "BIG DATA" anyway? Gigabytes? Terabytes? Petabytes? A brief personal memory may provide some perspective. In the late 1980s at Columbia University, I had the chance to play around with what at the time was a truly enormous disk: the IBM 3850 MSS (Mass Storage System). The MSS was actually a fully automatic robotic tape library and associated staging disks to make random access, if not exactly instantaneous, at least fully transparent. In Columbia's configuration, it stored a total of around 100GB. It was already on its way out by the time I got my hands on it, but in its heyday, the early- to mid-1980s, it had been used to support access by social scientists to what was unquestionably "big data" at the time: the entire 1980 U.S. Census database.[2]

Presumably, there was no other practical way to provide the researchers with ready access to a dataset that large—at close to $40K per GB,[3] a 100GB disk farm would have been far too expensive, and requiring the operators to manually mount and dismount thousands of 40MB tapes would have slowed progress to a crawl, or at the very least severely limited the kinds of questions that could be asked about the census data.

A database on the order of 100GB would not be considered trivially small even today, although hard drives capable of storing 10 times as much can be had for less than $100 at any computer store. The U.S. Census database included many different datasets of varying sizes, but let's simplify a bit: 100GB is enough to store at least the basic demographic information—age, sex, income, ethnicity, language, religion, housing status, and location, packed in a 128-bit record—for every living human being on the planet. This would create a table of 6.75 billion rows and maybe 10 columns. Should that still be considered "big data?" It depends, of course, on what you're trying to do with it. Certainly, you could *store* it on $10 worth of disk. More importantly, any competent programmer could in a few hours write a simple, unoptimized application on a $500 desktop PC with minimal CPU and RAM that could crunch through that dataset and return answers to simple aggregation queries such as "what is the median age by sex for each country?" with perfectly reasonable performance.

To demonstrate this, I tried it, with fake data of course—namely, a file consisting of 6.75 billion 16-byte records containing uniformly distributed random data (see Figure 1). Since a 7-bit age field allows a maximum of 128 possible values, one bit for sex allows only two (we'll assume there were no NULLs), and eight bits for country allows up to 256 (the UN has 192 member states), we can calculate the

Details of Jason Salavon's 2008 data visualization American Varietal (U.S. Population, by County, 1790–2000), commissioned as part of a site-specific installation for the U.S. Census Bureau. http://salavon.com/

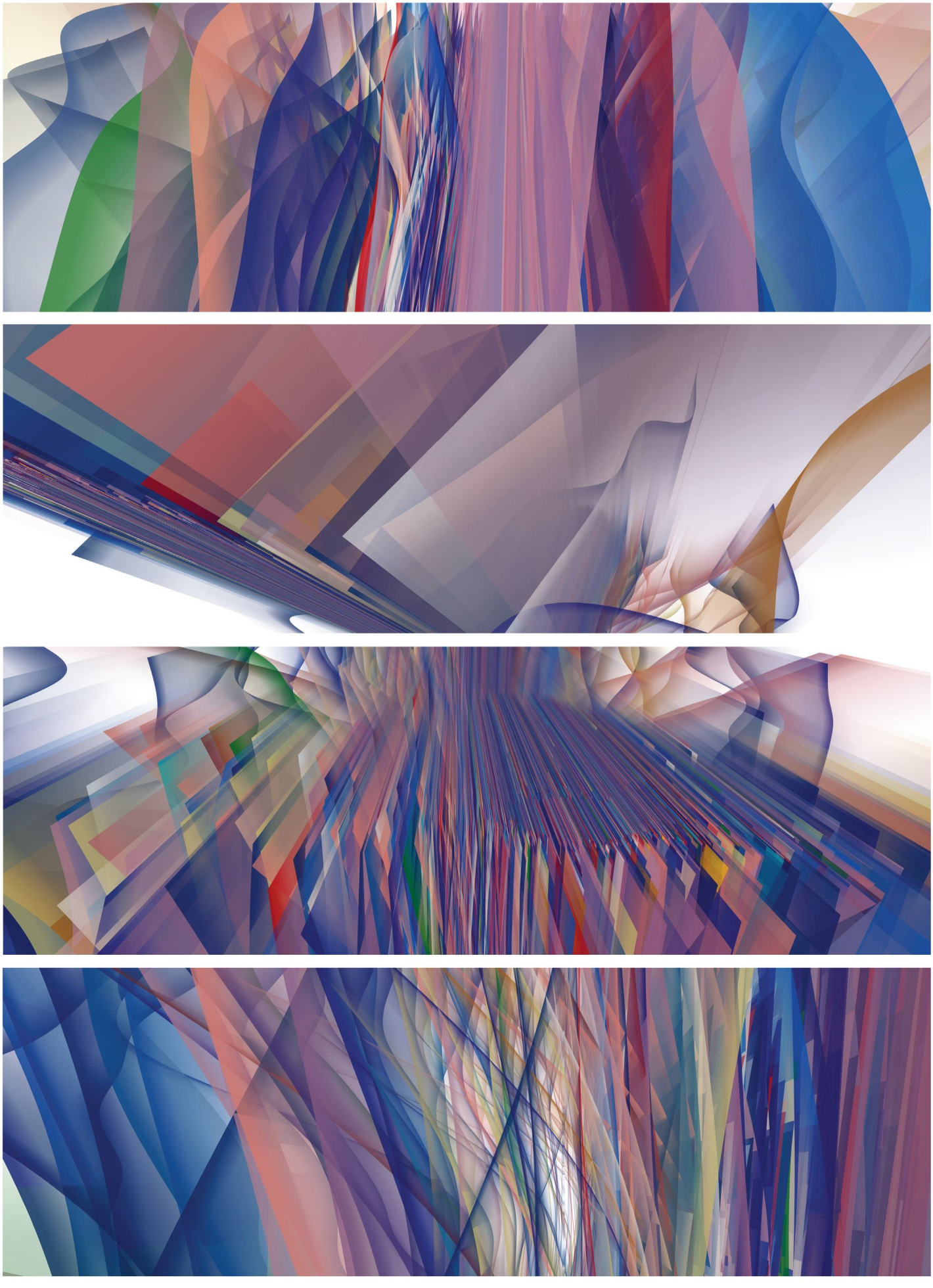
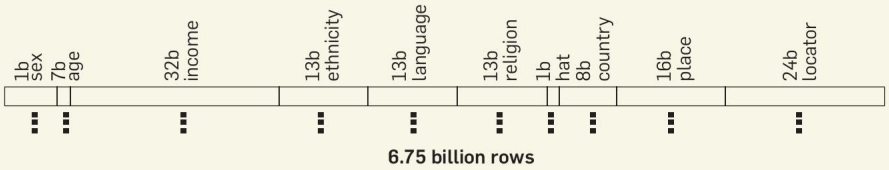


Figure 1. Calculating the median age by sex and country over the entire world population in a matter of minutes.

Record layout:



To find median age by sex and country,

```
int age, sex, country;
int cnt[2][256][128];
int tot,acc;
byte r[16];
fill cnt with 0;
do
   read 16 bytes into r;
   age = r[0] & 01111111b;
   sex = r[1] & 10000000b;
   ctry = r[11] & 11111111b;
   cnt[sex][ctry][age] += 1;
until end of file;
```

then

```
for sex = 0 to 1 do
 for ctry = 0 to 255 do
   output ctry, sex;
   tot = sum9cnt[sex][ctry][age];
   acc = 0;
   for age = 0 to 127 do
     acc += cnt[sex][ctry][age];
     if(acc >= tot/2)
       output age;
       go to next ctry;
     end if;
   next age;
 next ctry;
next sex;
```

median age by using a counting strategy: simply create 65,536 buckets—one for each combination of age, sex, and country—and count how many records fall into each. We find the median age by determining, for each sex and country group, the cumulative count over the 128 age buckets: the median is the bucket where the count reaches half of the total. In my tests, this algorithm was limited primarily by the speed at which data could be fetched from disk: a little over 15 minutes for one pass through the data at a typical 90MB/s sustained read speed,[9] shamefully underutilizing the CPU the whole time.

In fact, our table of "all the people in the world" will fit in the *memory* of a single, $15K Dell server with 128GB RAM. Running off in-memory data, my simple median-age-by-sex-and-country program completed in less than a minute. By such measures, I would hesitate to call this "big data," particularly in a world where a single research site, the LHC (Large Hadron Collider) at CERN (European Organization for Nuclear Research), is expected to produce 150,000 times as much raw data each year.[10]

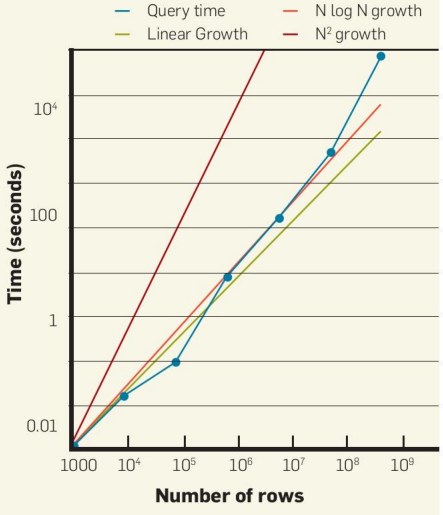
For many commonly used applications, however, our hypothetical 6.75-billion-row dataset would in fact pose a significant challenge. I tried loading my fake 100GB world census into a commonly used enterprise-grade database system (PostgreSQL[6]) running on relatively hefty hardware (an eight-core Mac Pro workstation with 20GB RAM and two terabytes of RAID 0 disk), but had to abort the bulk load process after six hours as the database storage had already reached many times the size of the original binary dataset, and the workstation's disk was nearly full. (Part of this, of course, was a result of the "unpacking" of the data. The original file stored fields bit-packed rather than as distinct integer fields, but subsequent tests revealed that the database was using three to four times as much storage as would be necessary to store each field as a 32-bit integer. This sort of data "inflation" is typical of a traditional RDBMS and shouldn't necessarily be seen as a problem, especially to the extent that it is part of a strategy to improve performance. After all, disk space is relatively cheap.)

I was successfully able to load subsets consisting of up to one billion rows of just three columns: country (8-bits, 256 possible values), age (7-bits, 128 possible values), and sex (one bit, two values). This was only 2% of the raw data, although it ended up consuming more than 40GB in the DBMS. I then tested the following query, essentially the same computation as the left side of Figure 1:

```
SELECT country,age,sex,count(*)
FROM people GROUP BY
country,age,sex;
```

This query ran in a matter of seconds on small subsets of the data, but execution time increased rapidly as the number of rows grew past 1 million (see Figure 2). Applied to the entire billion rows, the query took more than 24 hours, suggesting that PostgreSQL was not scaling gracefully to this big dataset, presumably because of a poor choice of algorithm for the given data and query. Invoking the DBMS's built-in EXPLAIN facility revealed the problem: while the query planner chose a reasonable hash table-based aggregation strategy for small tables, on larger tables it switched to sorting by grouping columns—a viable, if suboptimal strategy given a few million rows, but a very poor one when facing a billion. PostgreSQL tracks statistics such as the minimum and maximum value of each column in a table (and I verified that it had correctly identified the ranges of all three columns), so it could have chosen a hash-table strategy with confidence. It's worth noting, however, that even if the table's statistics had not been known, on a billion rows it would take far less time to do an initial scan and determine

Figure 2. PostgreSQL performance on the query SELECT country,age,sex,count(*) FROM people GROUP BY country,age,sex.



* Curves of linear, linearithmic, and quadratic growth are shown for comparison.

the distributions than to embark on a full-table sort.

PostgreSQL's difficulty here was in analyzing the stored data, not in storing it. The database didn't blink at loading or maintaining a database of a billion records; presumably there would have been no difficulty storing the entire 6.75-billion-row, 10-column table had I had sufficient free disk space.

Here's the big truth about big data in traditional databases: it's easier to get the data in than out. Most DBMSs are designed for efficient transaction processing: adding, updating, searching for, and retrieving small amounts of information in a large database. Data is typically *acquired* in a transactional fashion: imagine a user logging into a retail Web site (account data is retrieved; session information is added to a log), searching for products (product data is searched for and retrieved; more session information is acquired), and making a purchase (details are inserted in an order database; user information is updated). A fair amount of data has been added effortlessly to a database that—if it's a large site that has been in operation for a while—probably already constitutes "big data."

There is no pathology here; this story is repeated in countless ways, every second of the day, all over the world. The trouble comes when we want to take that accumulated data, collected over months or years, and learn something from it—and naturally we want the answer in seconds or minutes! The pathologies of big data are primarily those of analysis. This may be a slightly controversial assertion, but I would argue that transaction processing and data storage are largely solved problems. Short of LHC-scale science, few enterprises generate data at such a rate that acquiring and storing it pose major challenges today.

In business applications, at least, data warehousing is ordinarily regarded as the solution to the database problem (data goes in but doesn't come out). A data warehouse has been classically defined as "a copy of transaction data specifically structured for query and analysis,"[4] and the general approach is commonly understood to be bulk extraction of the data from an operational database, followed by reconstitution in a different database in a form that is more suitable for analytical queries (the so-called "extract, transform, load," or sometimes "extract, load, transform" process). Merely saying, "We will build a data warehouse" is not sufficient when faced with a truly huge accumulation of data.

To understand how to avoid the pathologies of big data, whether in the context of a data warehouse or in the physical or social sciences, we need to consider what really makes it "big."

How must data be structured for query and analysis, and how must analytical databases and tools be designed to handle it efficiently? Big data changes the answers to these questions, as traditional techniques such as RDBMS-based dimensional modeling and cube-based OLAP (online analytical processing) turn out to be either too slow or too limited to support asking the really interesting questions about warehoused data. To understand how to avoid the pathologies of big data, whether in the context of a data warehouse or in the physical or social sciences, we need to consider what really makes it "big."

Dealing with Big Data

Data means "things given" in Latin—although we tend to use it as a mass noun in English, as if it denotes a substance—and ultimately, almost all useful data is given to us either by nature, as a reward for careful observation of physical processes, or by other people, usually inadvertently (consider logs of Web hits or retail transactions, both common sources of big data). As a result, in the real world, data is not just a big set of random numbers; it tends to exhibit predictable characteristics. For one thing, as a rule, the largest *cardinalities* of most datasets—specifically, the number of distinct entities about which observations are made—are small compared with the total number of observations.

This is hardly surprising. Human beings are making the observations, or being observed as the case may be, and there are no more than 6.75 billion of them at the moment, which sets a rather practical upper bound. The objects about which we collect data, if they are of the human world—Web pages, stores, products, accounts, securities, countries, cities, houses, phones, IP addresses—tend

to be fewer in number than the total world population. Even in scientific datasets, a practical limit on cardinalities is often set by such factors as the number of available sensors (a state-of-the-art neurophysiology dataset, for example, might reflect 512 channels of recording[5]) or simply the number of distinct entities that humans have been able to detect and identify (the largest astronomical catalogs, for example, include several hundred million objects[8]).

What makes most big data *big* is repeated observations over time and/or space. The Web log records millions of visits a day to a handful of pages; the cellphone database stores time and location every 15 seconds for each of a few million phones; the retailer has thousands of stores, tens of thousands of products, and millions of customers but logs billions and billions of individual transactions in a year. Scientific measurements are often made at a high time resolution (thousands of samples a second in neurophysiology, far more in particle physics) and really start to get huge when they involve two or three dimensions of space as well; fMRI neuroimaging studies can generate hundreds or even thousands of gigabytes in a single experiment. Imaging in general is the source of some of the biggest big data out there, but the problems of large image data are a topic for an article by themselves; I won't consider them further here.

The fact that most large datasets have inherent temporal or spatial dimensions, or both, is crucial to understanding one important way that big data can cause performance problems, especially when databases are involved. It would seem intuitively obvious that data with a time dimension, for example, should in most cases be stored and processed with at least a partial temporal ordering to preserve locality of reference as much as possible when data is consumed in time order. After all, most nontrivial analyses will involve at the very least an aggregation of observations over one or more contiguous time intervals. One is more likely, for example, to be looking at the purchases of a randomly selected set of customers over a particular time period than of a "contiguous range" of customers (however defined) at a randomly selected set of times.

Here's the big truth about big data in traditional databases: It's easier to get the data in than out.

The point is even clearer when we consider the demands of time-series analysis and forecasting, which aggregate data in an order-dependent manner (for example, cumulative and moving-window functions, lead and lag operators, among others). Such analyses are necessary for answering most of the truly interesting questions about temporal data, broadly: "What happened?" "Why did it happen?" "What's going to happen next?"

The prevailing database model today, however, is the relational database, and this model explicitly ignores the ordering of rows in tables.[1] Database implementations that follow this model, eschewing the idea of an inherent order on tables, will inevitably end up retrieving data in a nonsequential fashion once it grows large enough that it no longer fits in memory. As the total amount of data stored in the database grows, the problem only becomes more significant. *To achieve acceptable performance for highly order-dependent queries on truly large data, one must be willing to consider abandoning the purely relational database model* for one that recognizes the concept of inherent ordering of data down to the implementation level. Fortunately, this point is slowly starting to be recognized in the analytical database sphere.

Not only in databases, but also in application programming in general, big data greatly magnifies the performance impact of suboptimal access patterns. As dataset sizes grow, it becomes increasingly important to choose algorithms that exploit the efficiency of sequential access as much as possible at all stages of processing. Aside from the obvious point that a 10:1 increase in processing time (which could easily result from a high proportion of nonsequential accesses) is far more painful when the units are hours than when they are seconds, increasing data sizes mean that data access becomes less and less efficient. The penalty for inefficient access patterns increases disproportionately as the limits of successive stages of hardware are exhausted: from processor cache to memory, memory to local

disk, and—rarely nowadays!—disk to off-line storage.

On typical server hardware today, completely random memory access on a range much larger than cache size can be an order of magnitude or more slower than purely sequential access, but completely random disk access can be five orders of magnitude slower than sequential access (see Figure 3). Even state-of-the-art solid-state (flash) disks, although they have much lower seek latency than magnetic disks, can differ in speed by roughly four orders of magnitude between random and sequential access patterns. The results for the test shown in Figure 3 are the number of four-byte integer values read per second from a 1-billion-long (4GB) array on disk or in memory; random disk reads are for 10,000 indices chosen at random between one and one billion.

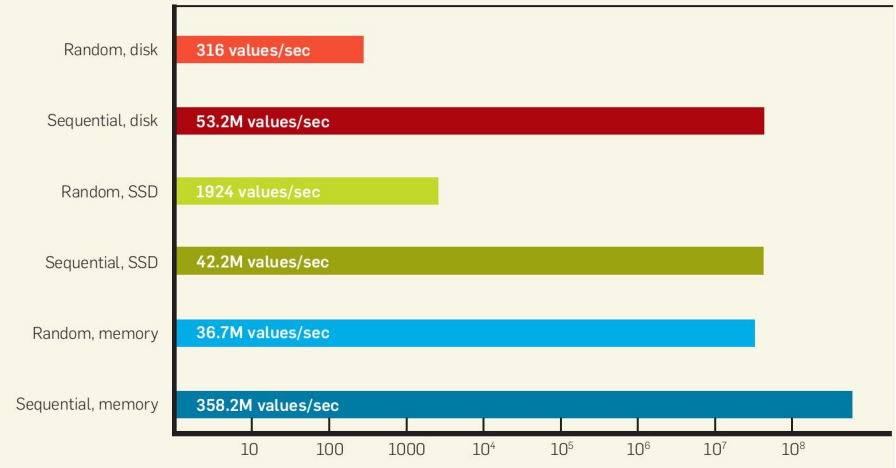
A further point that's widely underappreciated: in modern systems, as demonstrated in the figure, random access to memory is typically slower than sequential access to disk. Note that random reads from disk are more than 150,000 times slower than sequential access; SSD improves on this ratio by less than one order of magnitude. In a very real sense, *all* of the modern forms of storage improve only in degree, not in their essential nature, upon that most venerable and sequential of storage media: the tape.

The huge cost of random access has major implications for analysis of large datasets (whereas it is typically mitigated by various kinds of caching when data sizes are small). Consider, for example, joining large tables that are not both stored and sorted by the join key—say, a series of Web transactions and a list of user/account information. The transaction table has been stored in time order, both because that is the way the data was gathered and because the analysis of interest (tracking navigation paths, say) is inherently temporal. The user table, of course, has no temporal dimension.

As records from the transaction table are consumed in temporal order, accesses to the joined user table will be effectively random—at great cost if the table is large and stored on disk. If sufficient memory is available to hold the user table, performance will be improved by keeping it there. Because random access in RAM is itself expensive, and RAM is a scarce resource that may simply not be available for caching large tables, the best solution when constructing a large database for analytical purposes (for example, in a data warehouse) may, surprisingly, be to build a fully denormalized table—that is, a table including each transaction along with all user information that is relevant to the analysis (as shown in Figure 4).

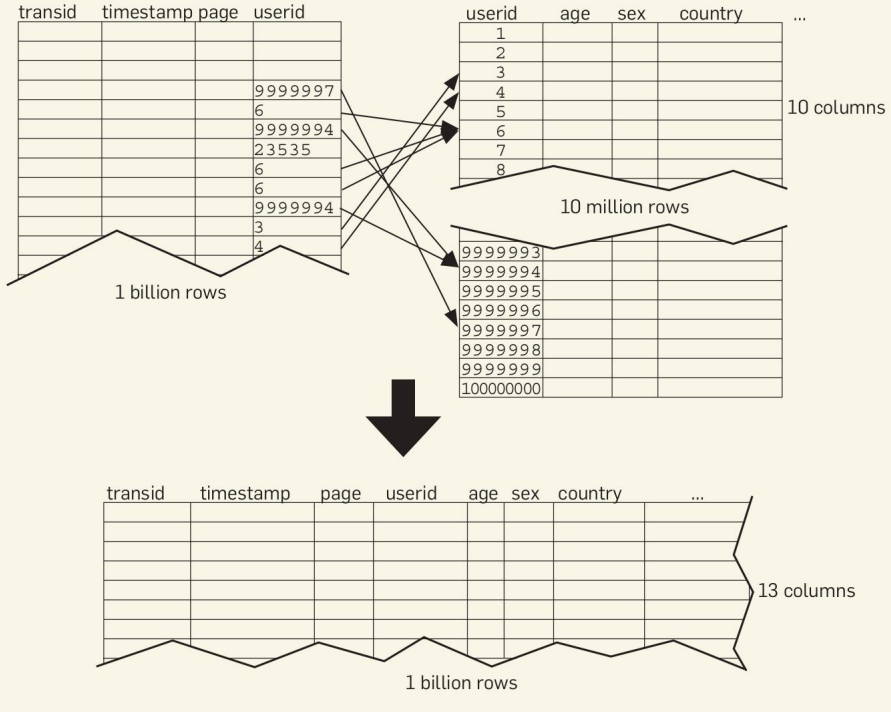
Denormalizing a 10-million-row, 10-column user information table onto a 1-billion-row, four-column transaction table adds substantially to the size of data that must be stored (the denormalized table is more than three times the size of the original tables combined). If data analysis is carried out in timestamp order but requires information from both tables,

Figure 3. Comparing random and sequential access in disk and memory.



* Disk tests were carried out on a freshly booted machine (a Windows 2003 server with 64GB RAM and eight 15,000RPM SAS disks in RAID5 configuration) to eliminate the effect of operating-system disk caching. SSD test used a latest generation Intel high-performance SATA SSD.

Figure 4. Denormalizing a user information table.



then eliminating random look-ups in the user table can improve performance greatly. Although this inevitably requires much more storage and, more importantly, more data to be read from disk in the course of the analysis, the advantage gained by doing all data access in sequential order is often enormous.

Hard Limits

Another major challenge for data analysis is exemplified by applications with hard limits on the size of data they can handle. Here, one is dealing mostly with the end-user analytical applications that constitute the last stage in analysis. Occasionally the limits are relatively arbitrary; consider the 256-column, 65,536-row bound on worksheet size in all versions of Microsoft Excel prior to the most recent one. Such a limit might have seemed reasonable in the days when main RAM was measured in megabytes, but it was clearly obsolete by 2007 when Microsoft updated Excel to accommodate up to 16,384 columns and one million rows. Enough for anyone? Excel is not targeted at users crunching truly huge datasets, but the fact remains that anyone working with a one million-row dataset (a list of customers along with their total purchases for a large chain store, perhaps) is likely to face a two million-row dataset sooner or later, and Excel has placed itself out of the running for the job.

In designing applications to handle ever-increasing amounts of data, developers would do well to remember that hardware specs are improving too, and keep in mind the so-called ZOI (zero-one-infinity) rule, which states that a program should "allow none of foo, one of foo, or any number of foo."[11] That is, limits should not be arbitrary; ideally, one should be able to do as much with software as the hardware platform allows.

Of course, hardware—chiefly memory and CPU limitations—is often a major factor in software limits on dataset size. Many applications are designed to read entire datasets into memory and work with them there; a good example of this is the popular statistical computing environment R.[7] Memory-bound applications naturally exhibit higher performance than disk-bound ones (at least insofar as the data-crunching they carry out advances beyond single-pass, purely sequential processing), but requiring all data to fit in memory means that if you have a dataset larger than your installed RAM, you're out of luck. On most hardware platforms, there's a much harder limit on memory expansion than disk expansion: the motherboard has only so many slots to fill.

Data replicated to improve the efficiency of different kinds of analyses can also provide redundancy against the inevitable node failure.

The problem often goes further than this, however. Like most other aspects of computer hardware, maximum memory capacities increase with time; 32GB is no longer a rare configuration for a desktop workstation, and servers are frequently configured with far more than that. There is no guarantee, however, that a memory-bound application will be able to use all installed RAM. Even under modern 64-bit operating systems, many applications today (for example, R under Windows) have only 32-bit executables and are limited to 4GB address spaces—this often translates into a 2- or 3GB working set limitation.

Finally, even where a 64-bit binary is available—removing the absolute address space limitation—all too often relics from the age of 32-bit code still pervade software, particularly in the use of 32-bit integers to index array elements. Thus, for example, 64-bit versions of R (available for Linux and Mac) use signed 32-bit integers to represent lengths, limiting data frames to at most $2^{31}-1$, or about two billion rows. Even on a 64-bit system with sufficient RAM to hold the data, therefore, a 6.75-billion-row dataset such as the earlier world census example ends up being too big for R to handle.

Distributed Computing as a Strategy for Big Data

Any given computer has a series of absolute and practical limits: memory size, disk size, processor speed, and so on. When one of these limits is exhausted, we lean on the next one, but at a performance cost: an in-memory database is faster than an on-disk one, but a PC with 2GB RAM cannot store a 100GB dataset entirely in memory; a server with 128GB RAM can, but the data may well grow to 200GB before the next generation of servers with

twice the memory slots comes out.

The beauty of today's mainstream computer hardware, though, is that it's cheap and almost infinitely replicable. Today it is much more cost-effective to purchase eight off-the-shelf, "commodity" servers with eight processing cores and 128GB of RAM each than it is to acquire a single system with 64 processors and a terabyte of RAM. Although the absolute numbers will change over time, barring a radical change in computer architectures, the general principle is likely to remain true for the foreseeable future. Thus, it's not surprising that distributed computing is the most successful strategy known for analyzing very large datasets.

Distributing analysis over multiple computers has significant performance costs: even with gigabit and 10-gigabit Ethernet, both bandwidth (sequential access speed) and latency (thus, random access speed) are several orders of magnitude worse than RAM. At the same time, however, the highest-speed local network technologies have now surpassed most locally attached disk systems with respect to bandwidth, and network latency is naturally much lower than disk latency.

As a result, the performance cost of storing and retrieving data on other nodes in a network is comparable to (and in the case of random access, potentially far less than) the cost of using disk. Once a large dataset has been distributed to multiple nodes in this way, however, a huge advantage can be obtained by distributing the *processing* as well—so long as the analysis is amenable to parallel processing.

Much has been and can be said about this topic, but in the context of a distributed large dataset, the criteria are essentially related to those discussed earlier: just as maintaining locality of reference via sequential access is crucial to processes that rely on disk I/O (because disk seeks are expensive), so too, in distributed analysis, processing must include a significant component that is local in the data—that is, does not require simultaneous processing of many disparate parts of the dataset (because communication between the different processing domains is expensive). Fortunately, most real-world data analysis does include such a component. Operations such as searching, counting, partial aggregation, record-wise combinations of multiple fields, and many time-series analyses (if the data is stored in the correct order) can be carried out on each computing node independently.

Furthermore, where communication between nodes is required, it often occurs after data has been extensively aggregated; consider, for example, taking an average of billions of rows of data stored on multiple nodes. Each node is required to communicate only two values—a sum and a count—to the node that produces the final result. Not every aggregation can be computed so simply, as a global aggregation of local sub-aggregations (consider the task of finding a global median, for example, instead of a mean), but many of the important ones can, and there are distributed algorithms for other, more complicated tasks that minimize communication

Figure 5. Two ways to distribute 10 years of sensor data for 1,000 sites over 10 machines.

Node 1

timestamp	sensor	reading
19990101000000	1	
19990101000015	1	
19990101000030	1	
⋮	⋮	⋮
20081231235930	1	
20081231235945	1	
19990101000000	2	
19990101000015	2	
19990101000030	2	
⋮	⋮	⋮
20081231235930	2	
20081231235945	2	
19990101000000	3	
⋮	⋮	⋮
20081231235945	100	

Node 1

timestamp	sensor	reading
19990101000000	1	
19990101000000	2	
19990101000000	3	
⋮	⋮	⋮
19990101000000	1000	
19990101000015	1	
19990101000015	2	
19990101000015	3	
19990101000015	4	
⋮	⋮	⋮
19990101000015	1000	
19990101000030	1	
19990101000030	2	
⋮	⋮	⋮
19991231235945	100	

Node 2

timestamp	sensor	reading
19990101000000	101	
19990101000015	101	
19990101000030	101	
⋮	⋮	⋮
20081231235930	101	
20081231235945	101	
19990101000000	102	
19990101000015	102	
19990101000030	102	
⋮	⋮	⋮
20081231235930	102	
20081231235945	102	
19990101000000	103	
⋮	⋮	⋮
20081231235945	200	

Node 2

timestamp	sensor	reading
20000101000000	1	
20000101000000	2	
20000101000000	3	
⋮	⋮	⋮
20000101000000	1000	
20000101000015	1	
20000101000015	2	
20000101000015	3	
20000101000015	4	
⋮	⋮	⋮
20000101000015	1000	
20000101000030	1	
20000101000030	2	
⋮	⋮	⋮
20001231235945	1000	

Node 10

timestamp	sensor	reading
19990101000000	901	
19990101000015	901	
19990101000030	901	
⋮	⋮	⋮
20081231235930	901	
20081231235945	901	
19990101000000	902	
19990101000015	902	
19990101000030	902	
⋮	⋮	⋮
20081231235930	902	
20081231235945	902	
19990101000000	903	
⋮	⋮	⋮
20081231235945	1000	

Node 10

timestamp	sensor	reading
20080101000000	1	
20080101000000	2	
20080101000000	3	
⋮	⋮	⋮
20080101000000	1000	
20080101000015	1	
20080101000015	2	
20080101000015	3	
20080101000015	4	
⋮	⋮	⋮
20080101000015	1000	
20080101000030	1	
20080101000030	2	
⋮	⋮	⋮
20081231235945	1000	

between nodes.

Naturally, distributed analysis of big data comes with its own set of "gotchas." One of the major problems is nonuniform distribution of work across nodes. Ideally, each node will have the same amount of independent computation to do before results are consolidated across nodes. If this is not the case, then the node with the most work will dictate how long we must wait for the results, and this will obviously be longer than we would have waited had work been distributed uniformly; in the worst case, all the work may be concentrated in a single node and we will get no benefit at all from parallelism.

Whether this is a problem or not will tend to be determined by how the data is distributed across nodes; unfortunately, in many cases this can come into direct conflict with the imperative to distribute data in such a way that processing at each node is local. Consider, for example, a dataset that consists of 10 years of observations collected at 15-second intervals from 1,000 sensor sites. There are more than 20 million observations for each site; and, because the typical analysis would involve time-series calculations—say, looking for unusual values relative to a moving average and standard deviation—we decide to store the data ordered by time for each sensor site (shown in Figure 5), distributed over 10 computing nodes so that each one gets all the observations for 100 sites (a total of two billion observations per node). Unfortunately, this means that whenever we are interested in the results of only one or a few sensors, most of our computing nodes will be totally idle. Whether the rows are clustered by sensor or by time stamp makes a big difference in the degree of parallelism with which different queries will execute.

We could, of course, store the data ordered by time, one year per node, so that each sensor site is represented in each node (we would need some communication between successive nodes at the beginning of the computation to "prime" the time-series calculations). This approach also runs into the difficulty if we suddenly need an intensive analysis of the past year's worth of data. Storing the data *both* ways would provide optimal efficiency for both kinds of analysis—but the larger the dataset, the more likely it is that two copies would be simply too much data for the available hardware resources.

Another important issue with distributed systems is reliability. Just as a four-engine airplane is more likely to experience an engine failure in a given period than a craft with two of the equivalent engines, so too is it 10 times more likely that a cluster of 10 machines will require a service call. Unfortunately, many of the components that get replicated in clusters—power supplies, disks, fans, cabling, and so on—tend to be unreliable. It is, of course, possible to make a cluster arbitrarily resistant to single-node failures, chiefly by replicating data across the nodes. Happily, there is perhaps room for some synergy here: data replicated to improve the efficiency of different kinds of analyses, as noted here, can also provide redundancy against the inevitable node failure. Once again, however, the larger the dataset, the more difficult it is to maintain multiple copies of the data.

A Meta-Definition

I have tried here to provide an overview of a few of the issues that can arise when analyzing big data: the inability of many off-the-shelf packages to scale to large problems; the paramount importance of avoiding suboptimal access patterns as the bulk of processing moves down the storage hierarchy; and replication of data for storage and efficiency in distributed processing. I have not yet answered the question I opened with: What is "big data," anyway?

I will take a stab at a meta-definition: big data should be defined at any point in time as "data whose size forces us to look beyond the tried-and-true methods that are prevalent at that time." In the early 1980s, it was a dataset that was so large that a robotic "tape monkey" was required to swap thousands of tapes in and out. In the 1990s, perhaps, it was any data that transcended the bounds of Microsoft Excel and a desktop PC, requiring serious software on Unix workstations to analyze. Nowadays, it may mean data that is too large to be placed in a relational database and analyzed with the help of a desktop statistics/visualization package—data, perhaps, whose analysis requires massively parallel software running on tens, hundreds, or even thousands of servers.

In any case, as analyses of ever-larger datasets become routine, the definition will continue to shift, but one thing will remain constant: success at the leading edge will be achieved by those developers who can look past the standard, off-the-shelf techniques and understand the true nature of the hardware resources and the full panoply of algorithms that are available to them. C

Related articles on queue.acm.org

Flash Storage Today
Adam Leventhal
http://queue.acm.org/detail.cfm?id=1413262

A Call to Arms
Jim Gray
http://queue.acm.org/detail.cfm?id=1059805

You Don't Know Jack about Disks
Dave Anderson
http://queue.acm.org/detail.cfm?id=864058

References

1. Codd, E.F. A relational model for large shared data banks. *Commun. ACM 13*, 6 (June 1970), 377–387.
2. IBM 3850 Mass Storage System; http://www.columbia.edu/acis/history/mss.html.
3. IBM Archives: IBM 3380 direct access storage device; http://www-03.ibm.com/ibm/history/exhibits/storage/storage_3380.html.
4. Kimball, R. *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses*. John Wiley & Sons, NY, 1996.
5. Litke, A.M. What does the eye tell the brain? Development of a system for the large-scale recording of retinal output activity. *IEEE Transactions on Nuclear Science 51*, 4 (2004), 1434–1440.
6. PostgreSQL: The world's most advanced open source database; http://www.postgresql.org.
7. The R Project for Statistical Computing; http://www.r-project.org.
8. Sloan Digital Sky Survey; http://www.sdss.org.
9. Throughput and Interface Performance. Tom's Winter 2008 Hard Drive Guide; http://www.tomshardware.com/reviews/hdd-terabyte-1tb,2077-11.html.
10. WLCG (Worldwide LHC Computing Grid); http://lcg.web.cern.ch/LCG/public/.
11. Zero-One-Infinity Rule; http://www.catb.org/~esr/jargon/html/Z/Zero-One-Infinity-Rule.html.

Adam Jacobs is senior software engineer at 1010data Inc., where, among other roles, he leads the continuing development of Tenbase, the company's ultra-high-performance analytical database engine. He has more than 10 years of experience with distributed processing of big datasets, starting in his earlier career as a computational neuroscientist at Weill Medical College of Cornell University (where he holds the position of Visiting Fellow) and at UCLA.