Big Data: A Small Introduction

Aidan Hogan

July 18, 2014

The Value of Data

Soho, London, August 1854: seven years before the discovery of germs by Louis Pasteur, people were dying in their hundreds from a mysterious disease called *cholera*. The wisdom of the time was that cholera was caused by *miasma*: something bad in the air—a foul fog of disease that would naturally build up in heavily populated areas. But John Snow, a physician working in London, was sceptical of this theory—and so he set out to find a better one.

Snow began by surveying those affected by cholera in the area, marking in a journal their names, gender, ages, locations, date of illness, date of death, daily habits, and so forth, applying various statistical and analytical techniques over the data. From his survey, he mapped out all of the Soho cholera cases, per Figure 1. Each dark rectangle indicates a case of cholera at that location and each stack represents a household or a workplace. Using a Voronoi diagram, it became clear that the cases of cholera were clustered around the water pump on the intersection of Broad Street and Cambridge Street; people living closer to another pump were not affected. Snow convinced the authorities to remove the handle from the pump. New cases of cholera ceased.

Even though the microscopes of the day could not see the physical cause of disease swimming in the water, 616 deaths and 8 days later, Snow's data had found the cause: an infected well dug close to a cess-pit of sewage. This was a revolutionary finding: cholera was not something in the air, but rather something in the water.

As computer scientists, we sometimes forget the value of data. Like a locksmith might consider keys as pieces of metal that have to be cut and sold, we tend to consider data as something that has to be stored or parsed, something that moves from input to output: composed of bytes in a machine, terminals in a grammar. It is only with a sense of the importance of data to other fields – and the ability to refine knowledge from it – that the recent and unusual hype around "*Big Data*" can be understood.

John Snow understood the value of data. His work in Soho, 1854, set the precedent for the field of *epidemiology*: the analysis of data to extract the patterns of causality of illness and disease in populated areas, encompassing public health screening, clinical trials, the cause and spread of infectious diseases, outbreak investigation and simulation, and so forth. Epidemiology's success stories include, amongst others, the eradication of smallpox, the isolation of polio to localised areas, and a marked reduction in cases of malaria and cholera.

Of course the value of data is appreciated not only in the field of epidemiology, or the field of genetics, or the field of medicine, or of observational astronomy, or of experimental physics, or of climatology or oceanography or geology or ecology or sociology, or even of science or enterprise. Data take centre stage in many scientific, commercial and social aspects of life. Like the work of

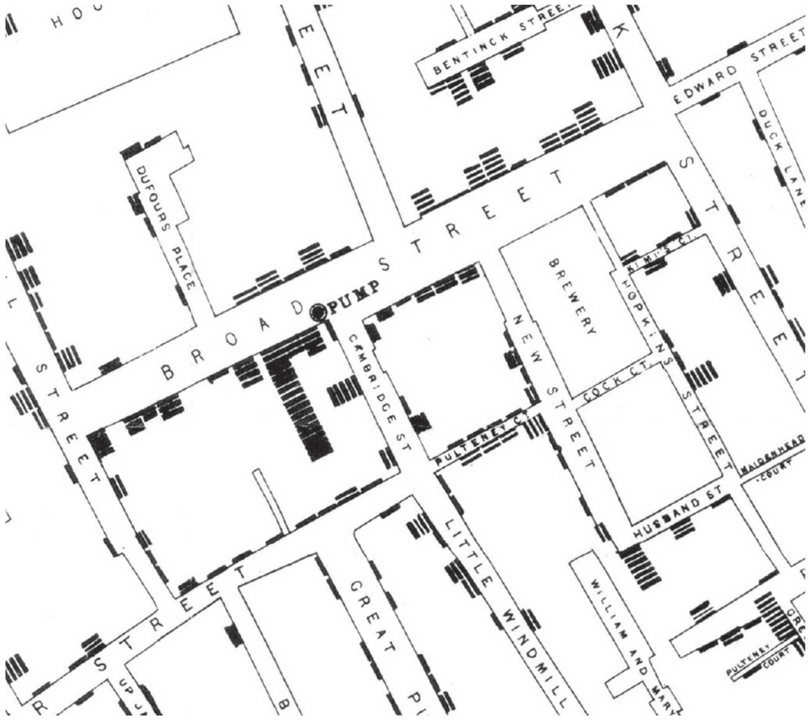


Figure 1: Part of Snow's map of cholera cases in Soho, 1854

Snow, common methodologies exist in all fields that craft data: data collection, curation, hypothesis generation, statistical testing, visualisation, etc. Unlike the work of Snow, computers are now available to collect, manage and process data with levels of scale and efficiency that Snow could scarcely have imagined.

But still it seems that as society's capacity to *capture* more and more data about the world around us continues to grow, conventional computational techniques are not enough to make sense of the result.

Big Data

Big Data, at its core, is a cross-disciplinary idea: having more data than you can make sense out of. Big Data is a call for us computer scientists to once again provide even better methods to crunch even more diverse, even more complex, even more dynamic, even more fine-grained, even *larger* data.

It is also not difficult to see why so many computer scientists feel (quietly?) dismissive of the Big Data buzz. In our field, core topics – like databases, logic, the Web, software engineering, machine learning, etc. – are founded in technologies with a rich pedigree. By contrast, "Big Data" is pretty fluffy. However, it is difficult to remain entirely dismissive once one sees headlines such as the $200 million investment by the US government into a series of national projects called the "Big Data Initiative", or the £30 million investment – by the UK government and a private philanthropist –

into a “Big Data Institute” at Oxford to work on epidemiology and drug discovery; or similar such news stories.

So what is “Big Data”? The most canonical (but still quite fluffy) definition of Big Data is as referring to any data-intensive scenario where the *v*olume, *v*elocity and/or *v*ariety of the data make it difficult to process using “conventional data management techniques”. These are known as the three ‘v’s (in English at least) and most of the emphasis thus far has been on the challenges of volume and (to a lesser extent) velocity.

New Breeds of Database

The traditional answer to working with large amounts of structured data has always been simple: *use a relational database*. If the volume or velocity of your data precluded using a relational database, then you were either (i) a company like Facebook where your team of highly-paid engineers could figure out a custom solution, or (ii) out of luck. If you face a similar problem these days, then you have what is called a “Big Data” problem.

The realisation that the relational database management system (RDBMS) is not – in the words of Stonebraker [10] – a “one size fits all” solution took many years, but the database landscape has now been demonopolised under the banner of “Big Data”. Figure 2 provides a broad perspective of this modern database landscape with a selection of the main systems in each family: systems with larger font-sizes are considered more prominent in terms of adoption.[1]

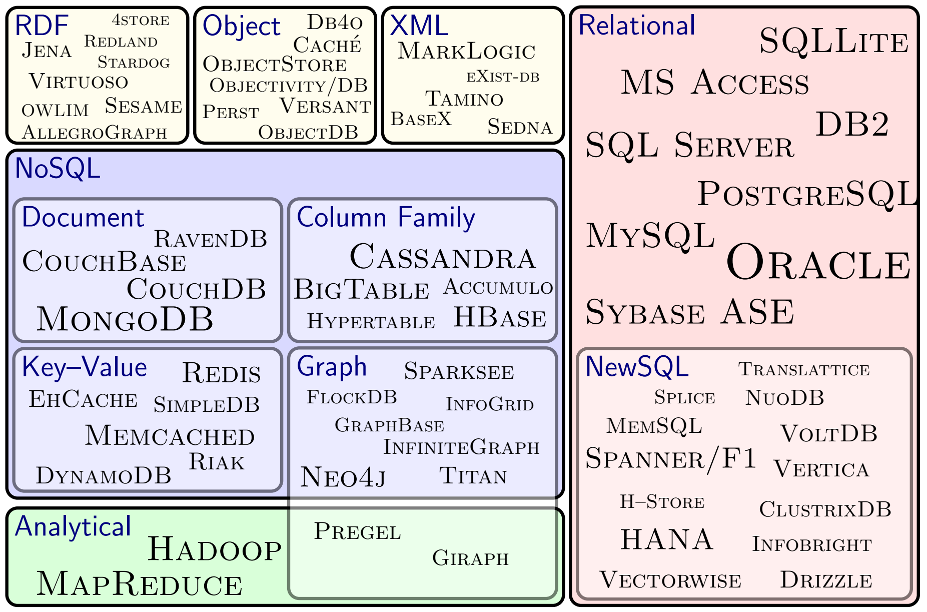


Figure 2: Modern database system landscape

[1]Specifically, we rely heavily on `http://db-engines.com/en/ranking` to judge system popularity, which itself aggregates mentions of systems in web-pages, Google searches, LinkedIn profiles, job postings, technical Q&A sites, Twitter, and so forth. In order of largest to smallest, the font sizes in the figure refer to positions 1, 2–20, 21–50, 51–100, 101–150, 150+, resp. in the list.

In the top right-hand corner, we see familiar traditional relational database systems presented with large font-sizes. These are still the most prevalent technology, offering the security of transactional guarantees (ACID) and a powerful query language (SQL). However, these features come at a cost . . .

Taking a modern open-source in-memory database (Shore) and a standard benchmark (TPC-C), Harizopoulos [5] showed that while the off-the-shelf system could perform 640 transactions per second, by turning off all features relating to transactions and persistence such as logging, latching, locking and buffer management, the same system could perform 12,700 transactions per second. The authors thus estimated that the database was only spending 6.8% of the time performing "useful work" [5]. Of course the latter system could no longer provide ACID guarantees, and the milage for other systems may vary, but in scenarios where such requirements can be relaxed, the core message of this anecdotal experiment is that huge performance and scalability benefits can be gained by minimising administrative work for transactions.

Following this reasoning, and the old idiom "necessity is the mother of invention", a new wave of NoSQL (Not Only SQL) databases emerged from the "Web 2.0" era where companies like Facebook, Google, Amazon, Twitter, etc., faced an unprecedented volume and velocity of user-contributed data. Many of these NoSQL systems were inspired by white-papers emerging from companies like Google and Amazon [1, 4] describing the (often lightweight) alternatives to relational databases that they had developed to meet these new challenges.

NoSQL: Not Only SQL

The aim of NoSQL stores is to enable high levels of horizontal scale (scaling out across multiple machines) and high throughput by simplifying the database model, query language and safety guarantees offered. By relaxing ACID and SQL requirements, and exploiting distribution, these systems claim to enable levels of scale and performance unattainable by traditional relational databases.

With respect to data models, as per Figure 2, four main categories of NoSQL system have emerged:

Key–Value: stores based on a simple associative array/map. Such systems allow for lookups on a single key, returning a given value that may follow a custom structure. Values are often subject to versioning rather than overwriting. Emphasis is placed on distribution and replication, typically following consistent hashing schemes [6]. Many of the systems in this family are inspired by the details of Amazon's Dynamo system [4], which were published in 2007.

Document: stores based on a key–value scheme, but where values refer to complex "documents" over which built-in functions can be executed. One of the main stores in this family is MongoDB, where values are akin to JSON documents.

Column-Family: stores that implement a loose form of the relational model over a key–value abstraction, where keys are akin to primary keys and values are multi-dimensional and conform to a flexible tabular schema. Key–values with similar dimensions are organised into column-families, akin to relational tables. Versioning is typically applied on a "cell"-level. Likewise tables are typically sorted by key, allowing range-queries on index prefixes. Stores in this family are typically inspired by the design of Google's BigTable system [1], details of which were published in 2008.

Graph: stores that implement adjacencies into their indexes such that traversing from one data node to another does not require another index lookup, but rather a pointer traversal. Regular-expression–like path languages allow for transitive navigation of data nodes. One of the most prominent systems in this family is Neo4J.

Each store typically supports a lightweight custom query language, which range from key/version lookups for key–value stores, to keys with embedded JSON/XML expressions for document stores, to a very limited and thus efficient form of SQL for column-family stores, to languages with path expressions for graph stores. As a trade-off for losing SQL, developers must often implement features such as joins, aggregates and transactions in the application code.

In terms of the guarantees a distributed database can offer, the CAP theorem states that a system cannot guarantee consistency (global agreement on state/data), availability (every request is serviced) and partition tolerance (functionality even if messages are lost) all at the same time. In a NoSQL setting, partition tolerance is a key objective since data may reside on hundreds or thousands of machines, increasing the likelihood of failures.

Thereafter, systems find different trade-offs between availability and consistency: a key notion is eventual consistency whereby machines can converge towards global agreement *after* a request has been acknowledged, translating into higher availability and reduced messages, but at the cost of consistency. Thus if you are Amazon, for example, under eventual consistency your users may see one-day-old data about product ratings but your system will not reject new ratings due to message failures, and eventually all operational machines will see these new ratings. On the other end of the spectrum, consensus protocols – such as two-phase commits, three-phase commits or Lamport's PAXOS algorithm [9] – incur high communication costs, less scalability of writes across machines and may imply more frequent downtimes (lower availability) but can ensure stronger notions of consistency.

The result of these performance-enhancing trade-offs is the now widespread use of NoSQL stores in data-intensive scenarios, most prominently in Web companies such as Google, Facebook, Twitter, etc., but also in many other areas: for example, document-centric stores such as MongoDB and CouchDB have been deployed at CERN to manage the vast amount of aggregated data produced by the particle detectors in the Large Hadron Collider [7].

NewSQL: Not Only NoSQL

NoSQL stores have been the subject of considerable criticism in terms of being overused and over-hyped. Though these types of stores undoubtedly have valid use-cases, not everyone faces the same data-intensive challenges as Facebook or CERN. Their naive use may imply an unnecessary risk of data loss or inconsistency. Weakened safety guarantees and lower-level query languages push more responsibility on the application developer to ensure that the database remains stable and the performance of more complex queries acceptable. To quote a recent Google white-paper on the "Spanner" database:

> We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions [2].

Although the features of traditional relational databases that NoSQL stores dispense with are expensive, they are important for many (though perhaps not all) applications: they were originally implemented and added to databases for a good reason.

The NewSQL family of databases (depicted in Figure 2) has recently emerged to strike a compromise between the features of traditional relational databases and the performance of NoSQL stores. NewSQL systems aim to support SQL and uphold ACID but at greater levels of performance and scale than traditional databases (at least in fixed scenarios). The main approach to NewSQL is to design databases from scratch that exploit modern computer architectures, leveraging multiple cores, large main memory capacity, GPUs, or shared-nothing clusters. Likewise, many NewSQL stores opt for column-oriented indexing schemes that enable more efficient aggregation and filtering over the values of entire columns than traditional row-oriented stores.

However, databases are not designed for large-scale data analytics, but rather for executing live queries. Likewise, many forms of off-line analytics do not require the expense of building and maintaining persistent indexes.

Distributed Analytical Frameworks

In Figure 2, we include the Analytical category for distributed data-processing frameworks: though they are not strictly speaking databases, they offer an alternative to databases for analytics.

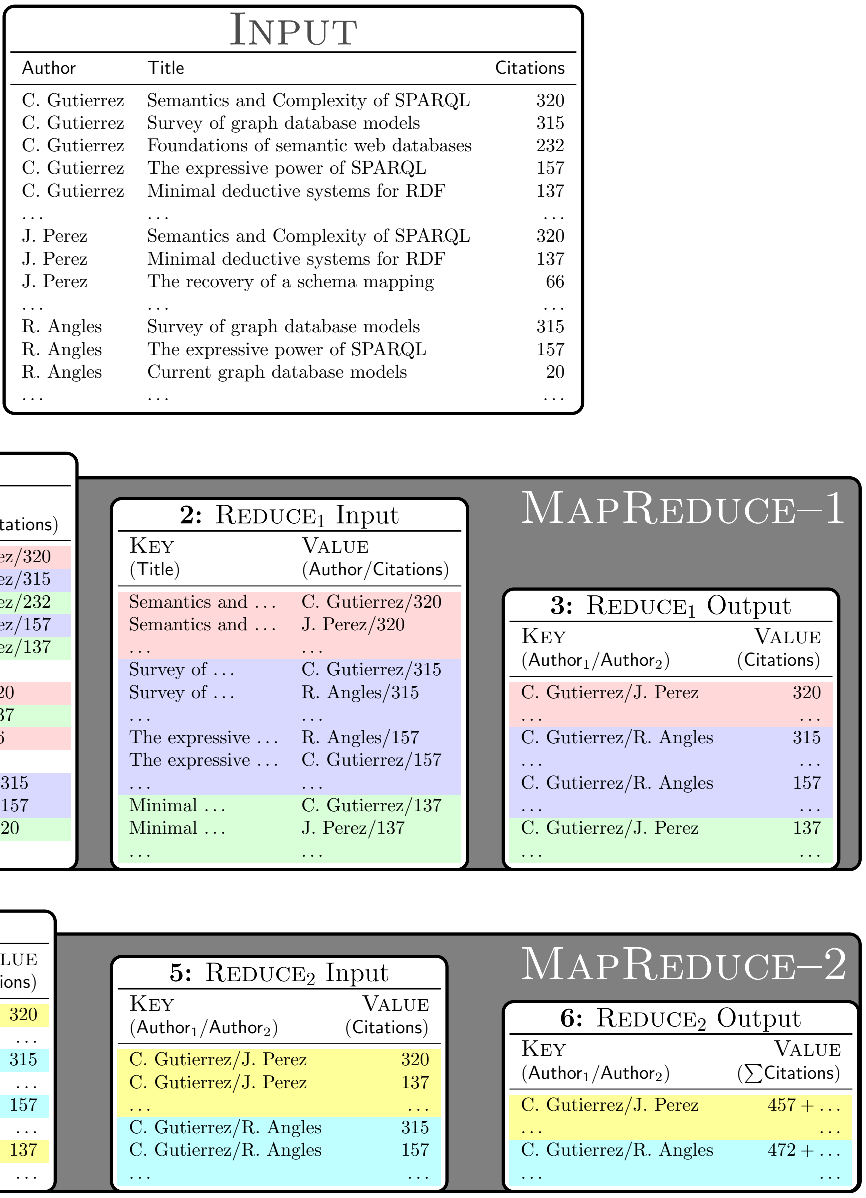
Google first proposed the MapReduce framework in 2004 [3] as an abstraction for executing batch-processing tasks over large-scale flat data (unindexed/raw files) in a distributed setting. Many different types of distributed processing tasks (as run by Google, for example) require common elements: partitioning the data over the machines, supporting fail-safes in case of machine failures, running the processing on each machine, merging the results from each machine into a final result, etc. Thus, the goal of MapReduce is to offer an interface that abstracts these common elements and allows for higher-level development of distributed processing code.

Since MapReduce is considered one of the core Big Data technologies, we'll take a moment to get the gist of how it operates with the example illustrated in Figure 3. The input data are sorted by author, and include papers and citations (note the repetition of paper titles and citations for each author). We can consider this table to be a flat file (e.g., a CSV or TSV file) and to be very large: "frillions" of rows. Now we'd like to see who are the most productive pairs of co-authors in our table: we want to see how many citations pairs of co-authors have counting only those papers they have co-authored together. And we have lots of machines to do this with.

MapReduce performs aggregation/joins in batch by sorting data. Taking a trivial example, since the raw input data are sorted by author name, we can easily get the total citation count for each author, having to store in memory only the current author and their running count, outputting this pair when the author changes. If we consider a distributed setting, aside from the sorting, we have to ensure that all papers for an author end up on one machine.

Returning to our original non-trivial task of finding productive co-author pairs, MapReduce consists of two main phases: MAP and REDUCE. we first need to figure out pairs of co-authors. To do this, we need to do a distributed join on paper title/citations.

MAP$_1$: For each tuple $A_i = (\mathsf{Author}_i, \mathsf{Title}_i, \mathsf{Citation}_i)$ from the input data, create key–value pairs of the following form: $(\mathsf{Title}_i, <\mathsf{Author}_i, \mathsf{Citations}_i>)$, where the key is Title_i and the value is $<\mathsf{Author}_i, \mathsf{Citations}_i>$.



INPUT

Author	Title	Citations
C. Gutierrez	Semantics and Complexity of SPARQL	320
C. Gutierrez	Survey of graph database models	315
C. Gutierrez	Foundations of semantic web databases	232
C. Gutierrez	The expressive power of SPARQL	157
C. Gutierrez	Minimal deductive systems for RDF	137
...	...	...
J. Perez	Semantics and Complexity of SPARQL	320
J. Perez	Minimal deductive systems for RDF	137
J. Perez	The recovery of a schema mapping	66
...	...	...
R. Angles	Survey of graph database models	315
R. Angles	The expressive power of SPARQL	157
R. Angles	Current graph database models	20
...	...	...

MAPREDUCE–1

1: MAP$_1$ Output

KEY (Title)	VALUE (Author/Citations)
Semantics and ...	C. Gutierrez/320
Survey of ...	C. Gutierrez/315
Foundations of ...	C. Gutierrez/232
The expressive ...	C. Gutierrez/157
Minimal ...	C. Gutierrez/137
...	...
Semantics and ...	J. Perez/320
Minimal ...	J. Perez/137
The recovery ...	J. Perez/66
...	...
Survey of ...	R. Angles/315
The expressive ...	R. Angles/157
Current graph ...	R. Angles/20
...	...

2: REDUCE$_1$ Input

KEY (Title)	VALUE (Author/Citations)
Semantics and ...	C. Gutierrez/320
Semantics and ...	J. Perez/320
...	...
Survey of ...	C. Gutierrez/315
Survey of ...	R. Angles/315
...	...
The expressive ...	R. Angles/157
The expressive ...	C. Gutierrez/157
...	...
Minimal ...	C. Gutierrez/137
Minimal ...	J. Perez/137
...	...

3: REDUCE$_1$ Output

KEY (Author$_1$/Author$_2$)	VALUE (Citations)
C. Gutierrez/J. Perez	320
...	...
C. Gutierrez/R. Angles	315
...	...
C. Gutierrez/R. Angles	157
...	...
C. Gutierrez/J. Perez	137
...	...

MAPREDUCE–2

4: MAP$_2$ Output

KEY (Author$_1$/Author$_2$)	VALUE (Citations)
C. Gutierrez/J. Perez	320
...	...
C. Gutierrez/R. Angles	315
...	...
C. Gutierrez/R. Angles	157
...	...
C. Gutierrez/J. Perez	137
...	...

5: REDUCE$_2$ Input

KEY (Author$_1$/Author$_2$)	VALUE (Citations)
C. Gutierrez/J. Perez	320
C. Gutierrez/J. Perez	137
...	...
C. Gutierrez/R. Angles	315
C. Gutierrez/R. Angles	157
...	...

6: REDUCE$_2$ Output

KEY (Author$_1$/Author$_2$)	VALUE ($\sum$Citations)
C. Gutierrez/J. Perez	$457 + \ldots$
...	...
C. Gutierrez/R. Angles	$472 + \ldots$
...	...

Figure 3: MapReduce Example: Find the citations for each pair of co-authors (including only the papers they have co-authored). Row shades indicate partitioning over different machines.

MapReduce will assign key–value pairs with the same key to the same machine (e.g., using a hash function on the key). That machine will sort the pairs assigned to it by the MAP according to their key. Partitioning and sorting are usually taken care of by the framework, meaning that the developer need not worry about what data goes to which physical machine. However, the default partitioning methods and sorting orders can be overriden if necessary.

Now each machine has its own bag of key–value pairs sorted by Title, with all authors for each paper locally available. The next phase is the reduce phase:

REDUCE$_1$: In preparation for the reduce, the sort phase will group the key–value pairs (emitted from MAP$_1$) by title: $(\mathsf{Title}_i, \{<\mathsf{Author}_{i,1}, \mathsf{Citations}_i>), \ldots, <\mathsf{Author}_{i,n}, \mathsf{Citations}_i>\})$. For each such group, REDUCE will output a set of tuples representing each unique pair of co-authors: $\{(<\mathsf{Author}_{i,j}, \mathsf{Author}_{i,k}>, \mathsf{Citations}_i) \mid 1 \leq j < k \leq n\}$.[2]

The result of the reduce phase is a bag of unique pairs of co-authors and their citations for each paper (we no longer need the name of the paper). Our final task is to add up the total citations per co-author pair. In the previous stage, the data were sorted/joined on the paper title, so now we need to perform another MAP/REDUCE phase.

MAP$_2$: For each tuple $(<\mathsf{Author}_x, \mathsf{Author}_y>, \mathsf{Citations}_z)$ from the intermediate data, map the same key–value pair.

This map does not transform the values, but merely ensures that the same co-author pairs end up sorted on the same machine for the final reduce phase.

REDUCE$_2$: With groups $(<\mathsf{Author}_x, \mathsf{Author}_y>, \{\mathsf{Citations}_1, \ldots, \mathsf{Citations}_n\})$, sum the citations in each bag and produce a single output pair: $(<\mathsf{Author}_x, \mathsf{Author}_y>, \sum_{z=1}^{n} \mathsf{Citations}_z)$.

We now have our final result. If we wanted to extract top-k results, we could pass MapReduce a descending sort order and map total citations as key.

In terms of coding effort, the developer must implement the map phases and the reduce phases. The MapReduce framework itself will take care of load-balancing, fault tolerance, and so forth. With a cluster of machines running the MapReduce framework, tasks can be run by multiple users in parallel where the framework will aim to make the best of the resources available. Likewise MapReduce jobs are portable: jobs can be run on any cluster of machines so long as the framework is installed (and the data available).

MapReduce enables scale but, of course, it does not ensure scale: intractable tasks will still be intractable on multiple machines. In our example, REDUCE$_1$ produces a quadratic number of co-author pairs per paper ($\frac{n(n-1)}{2}$), but since author-lists are generally short, we can expect the REDUCE$_1$ to perform near-linearly. With the assumption that tasks can be decomposed into such MAP/REDUCE phases, and that these phases are not computationally expensive or increase data volumes too much, the framework offers a convenient and powerful abstraction for those wishing to work with Big Data.

Unfortunately, the MapReduce system itself is proprietary and kept closed by Google. However, the Apache Hadoop open source project offers a mature implementation of the MapReduce framework and is in widespread use.

[2]We assume $\mathsf{Author}_x < \mathsf{Author}_y$ if and only if $x < y$, etc., to ensure consistent pairs across papers.

In Figure 2, one may note two other systems in the intersection of Graph and Analytical: Pregel and Giraph. These are both MapReduce style frameworks designed specifically for processing graphs. Pregel was first introduced by Google in 2009 [8] for distributed computation on very large graphs, where Apache Giraph is an open source implementation of the same ideas (built on top of Hadoop). In brief, the core computational model of Pregel/Giraph is a message passing system between vertices in a graph. Computation is organised into iterations. A vertex may read messages sent to it in the previous iteration, may change its (possibly continuous) state, and may forward messages to other vertices for the next iteration. Messages can be forwarded to any vertex with a locally known id, but typically these are the linked vertices. These computational frameworks offer an intuitive abstraction for applying graph-based analytics – for running tasks such as shortest path, computing connected components, clustering, centrality measures such as PageRank, etc. – while transparently distributing the computation: handling load balancing, efficient batching of messages and fault tolerance.

Other models: Object, XML, RDF

With respect to Figure 2, we are thus left with three (relatively traditional) database families in the top left corner: Object, XML and RDF.

Object databases allow for storing data in the form of objects as familiar from object-oriented software. The primary motivation for such databases is to offer the option of persistence for software objects in a native format. Typically object databases tailor towards a fixed set of object oriented programming languages, offering the possibility for tight integration between the execution environment and the database. Object databases typically support a query language to search for objects with certain fields or member values; likewise pointers are used between objects to avoid the need for joins. Systems may support various other features, including versioning, constraints, triggers, etc.

XML databases refer specifically to native XML databases, where data are stored in an XML-like data structure. Given the relative popularity of XML as a model for data exchange and its use on the Web (e.g., XHTML, RSS and ATOM feeds, SOAP messages, etc.), XML databases store such data in a native format that enables them to be queried directly through languages such as XQuery, XPath or XSLT.

RDF databases focus on providing query functionality over data represented in the core Semantic Web data model. Such stores typically implement optimised storage schemes designed for RDF and offer query functionality using the W3C-recommended SPARQL query language. Furthermore, engines may implement features related to reasoning, typically with respect to the RDFS and/or OWL standards.

Though these three families of databases have received significant attention in the research literature over the past decade(s), as can be seen from Figure 2, they largely remain a niche technology when considered as part of the broader database landscape.

Variety?

We have talked a lot about volume – e.g., scaling out across multiple machines – and velocity – e.g., increases write throughput by relaxing consistency guarantees. However, we have not spoken much about the third challenge of Big Data: "variety". Whereas the problems of volume and velocity tie in with performance and can be approached from an engineering perspective by composing existing techniques – such as sharding, replication, distributed sorts, consistent hashing, compression, batching, bloom filters, Merkle trees, column-oriented indexing, etc. – in the design of a mature system, the problem of variety raises questions of a more conceptual nature.

Arguably, the NoSQL family of databases offers some advantages with respect to traditional relational databases when it comes to processing diverse datasets: by relaxing the relational model, data can be stored in a "quick and dirty" non-normalised form. Although this may save time with respect to schema management, and is more flexible for loading incomplete or otherwise jagged data, employing simpler data models once again pushes a further burden onto the application layer, where developers must ensure that data remain consistent and that indexes are present to efficiently cover the expected workload of queries.

Likewise variety is partially supported by the range of database types available today. For example, one may use an XML database to store XML data, use a relational database to store relational data, use a graph database for graph-structured data, and so forth. Multiple databases can thus be composed to tackle the variety problem; however, again the burden is put on the application developers to connect the databases and to ensure that data remains consistent across the various systems.

In general, the problem of variety goes much deeper than the shape or the incompleteness of the data. Variety is also far from being a new problem: various sub-areas of Computer Science have come across the hurdle of "data integration", be it combining multiple databases into one, or combining documents from millions of Web sources, or combining source-code repositories from various projects, or ...

Data Are Evolving

It seems that the only practical approach to solving the variety bottleneck in Big Data is to rethink what we mean by "data". From Snow's days of paper and ink, data have evolved through Morse code, punch cards, binary formats, ASCII, markup, and so forth. The evolution of data has always been towards improved machine readability. Much like it would be considered unfeasible to do a count of the occurrences of the word "broccoli" in all books currently shelved in a well-stocked library (and trivial to do so for an electronic text file), there are many data-intensive tasks considered unfeasible by today's standard simply because our notion of data does not enable them.

Currently, we have a plethora of data models and data syntaxes available that allow information to be structured and parsed. However, for the most part, machines still need data-specific code to interpret said data. For example, browsers are built specifically to interpret HTML-related data.

In terms of the evolution of data, it seems that the only natural way forward is to make explicit the semantics of the data being understood, such that machines can, with increasing depth, process the meaning of data. Machines still cannot learn from natural language and still cannot adapt well to new problems; hence semantics need to be made explicit as part of the data to help overcome true variety in structured data: i.e., the ability to incorporate unforeseen datasets.

In terms of making semantics explicit, it can start by simply using global identifiers for items described in the data such that a machine can know, for example, that the "Boston" described in one dataset is the same "Boston" as described in another dataset (e.g., using URIs as identifiers); now the machine can run joins over the two datasets. Even better if the machine can know that the "Boston" in question is the music band, not the U.S. city (e.g., using a classification system); now the machine can return results about "Boston" for queries about music bands. Even better still if the machine can know that music bands have members, and typically release albums and ... (e.g., through a domain ontology).

Big Semantic Data?

Many of these principles have been well-explored by the Semantic Web community; in this author's biased opinion, techniques from the Semantic Web could yet have a large role to play in the area of Big Data, particularly towards tackling diversity. The Semantic Web has long been frowned upon by many as a fruitless academic exercise, doomed to failure due to any number of concerns. And there are grains of truth in these concerns. Many of the techniques proposed in the area of ontologies and deductive reasoning methods are not well suited for scenarios involving lots of data, and certainly not for scenarios involving lots of messy (Web) data. However, the challenges are not insurmountable: a little semantics can go a long way.

Recently companies such as Google, Facebook, Yahoo!, Microsoft, etc., are starting to use parts of Semantic Web technologies to power new applications. For example, in June 2011, Bing, Google, and Yahoo! announced the schema.org ontology for webmasters to make structured data available on their site. Google has used semantic knowledge-bases in the construction of their Knowledge-Graph application. Facebook's Open Graph protocol – trying to create a decentralised connected network of data – uses RDFa: a Semantic Web standard. More recently, Google announced support for semantic markup in GMail using the JSON-LD syntax. In such case, these major companies have turned to Semantic Web techniques to build applications over huge sets of diverse sources coming from millions of arbitrary providers. And this trend of using semantics to tackle diversity looks set to continue.

Conclusion

Big Data is a call for Computer Scientists to investigate methods to handle data with more volume, more velocity and more variety. Though relational databases have served as a reliable work-horse for many years, people are beginning to realise that new alternatives are needed to face the upcoming challenges posed by the emerging flood of data.

The main Big Data technologies that have emerged thus far primarily focus on the challenges of data volume and velocity. NoSQL stores boast new levels of scale by distributing data management over multiple machines, by relaxing traditional ACID guarantees, and by supporting query languages more lightweight than SQL. NewSQL aim to seek a balance by supporting ACID/SQL as per traditional relational databases while trying to compete with the performance and scale of NoSQL systems. Likewise, distributed processing frameworks such as MapReduce offer a convenient abstraction for clients wishing to perform large-scale analytical tasks.

However, variety remains an open problem. In particular, the goal of being able to build applications that can robustly discover and incorporate new sources of data remains elusive. To overcome this hurdle, we may need to rethink our conceptualisation of data. Versus the days of John Snow, nowadays we appreciate the convenience of working with data serialised in an electronic format that can be parsed by a machine. Likewise, having data with explicit semantics may become the norm in the decades to come.

References

[1] 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. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[2] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-Distributed Database. In *OSDI*, pages 261–264, 2012.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key–value store. In *SOSP*, pages 205–220, 2007.

[5] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD Conference*, pages 981–992, 2008.

[6] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *STOC*, pages 654–663, 1997.

[7] V. Kuznetsov, D. Evans, and S. Metson. The CMS data aggregation system. In *ICCS*, number 1, pages 1535–1543, 2010.

[8] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146, 2010.

[9] M. C. Pease, R. E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[10] M. Stonebraker. One size fits all: an idea whose time has come and gone. *Commun. ACM*, 51(12):76, 2008.