Big Data for Conventional Programmers

Big Data - Not a Big Deal

by Efraim Moscovich, Senior Principal Software Architect, CA Technologies

Processing large volumes of data has been around for decades (such as in weather, astronomy, and energy applications). It required specialized and expensive hardware (supercomputers), software, and developers with distinct programming and analytical skills.

As the push for “Big Data” collection and analysis becomes more prevalent in the general business community, there is an increased demand for systems and language environments that can run on commodity, inexpensive hardware and software that can be programmed and operated by programmers and analysts with average, mainstream skills.

This article examines several languages and tools that were designed for Big Data processing, and discusses the present and desirable attributes of data-intensive programming languages, in particular, as they are related to ease of use, data abstraction, data flow and data transformations.

Introduction

What is “Big Data”?

To paraphrase a section in McKinsey Global Institute’s Big Data report, “Big Data” refers to datasets whose sizes are beyond the ability of typical software tools to capture, store, manage, and analyze them.¹

The definition is intentionally imprecise. Big Data is not defined in precise terabytes or petabytes since it can change as the technology advances, and by sector or type of data.

In addition to the elusive size of the data, the Big Data term implies methodologies and tools for processing and analyzing the data to produce useful results that cannot be inferred or calculated using other methods in an efficient manner.

From personal experience in Capacity Management, I used to write programs to read, summarize, and analyze large mainframe datasets that span years’ worth of performance data stored on tapes (SMF records). The programs would run for days and produce a few reports.

That was certainly “large data” processing. The language in use, SAS, lacked many important attributes for data-intensive computing, such as parallel and distributed processing, but it did the job on the expected volume and in the expected time.

The ubiquitous use of Google Search, which the users perceive to scan terabytes of data just for them, and produce results instantly, raised the expectations level to a much higher degree. If a teenager can easily search for her favorite band and get all relevant information in an instant, why can’t a corporate analyst do the same with the company’s mountains of data?

Problem statement: How to process “Big Data” using the current state-of-the-art technology in agreed upon time and budget?

About the author:

Efraim Moscovich is a Senior Principal Software Architect in the Office of the CTO at CA Technologies.

He has over 25 years of experience in IT and Software Development in various capacities.

Efraim’s areas of expertise include virtualization and cloud computing, automation, event management/complex event processing, internationalization and localization, performance management, Windows internals, clustering and high-availability, large scale software architecture, continuous integration, automated testing, scripting languages, and diagnostics techniques.

He is an active participant in OASIS TOSCA technical committee and DMTF Cloud Management working group.

Prior to joining CA Technologies, Efraim worked on large scale performance management and capacity planning projects at various IT departments.

Efraim has a M.Sc. in Computer Science from New Jersey Institute of Technology.
The Road to Hadoop

Even though CPUs became faster and faster (Moore's law²), the speed of accessing data on disks or volumes was still orders of magnitude slower than the CPUs. Programs that needed to process large amounts of data didn’t benefit much from the added speed (referred to as I/O-bound programs). Modern languages support multi-programming concepts such as multi-threading implemented as libraries (e.g., POSIX pthreads) for C++, or built-in in Java. Some specialized languages support co-routines or parallel collection processing.

To take advantage of servers with multiple CPUs and multiple cores, parallel programming models can be used to reduce the time it takes to process a certain load of computation or data manipulation.

Task Parallelism

In general, certain segments of the program that are ‘not sequential’ are broken down into multiple chunks that can run simultaneously on different CPUs on the same machine, thus reducing the total time it takes to run the program. The program can be designed by the programmer to use parallelism explicitly, by the compiler based on program source analysis, or by the compiler with the aid of hints provided by the programmer. Task parallelism is suitable for many cases, but in general, it is considered complex, especially when the programmers have to use it explicitly.

Issues and Limitations

Generic parallel computing greatly increases coding complexity since the programmer has to deal with the additional overhead associated with it.

Examples:

- **Coordination of concurrent tasks**: Extra code is needed to spawn multiple tasks, wait for their completion, wait until one signals to proceed, and pack/send/unpack data between tasks.
- **Parallelization of algorithms**: Creating parallel versions of algorithms is not straightforward. For example, see “Parallelization of Quicksort”³.
- **Shared memory locking and synchronization**: Shared memory that is used by multiple threads has to be carefully managed and explicitly protected by locks or semaphores to avoid data corruption.

In addition, hardware that supports massive parallelism is specialized, complex and expensive. In many cases the hardware and software are designed specifically for one project or target audience. For example, see IBM Blue Gene/Q⁴.

Data Parallelism

The Data Parallel programming model is more suitable for scenarios where the exact same operations can be applied on multiple, independent data elements such as records, files, documents, and web pages.

Turning sequential processing into parallel:

```c
// Sequential version
foreach (var item in collection) {
    Process(item);
}

// Parallel equivalent
Parallel.foreach( collection, Process(item) );
```

Figure 1 Sequential to Parallel
Simplistically, we want the programming language and its runtime to support a method of running loop iterations in parallel and to handle the distribution of the processing functions with the correct item on any number of available processors automatically.

**SIMD**

On specialized hardware this is called Single Instruction, Multiple Data (SIMD) – the same instruction is performed on multiple data at lockstep (for example, incrementing all the entries in an array simultaneously).

SIMD hardware is not widespread, and the number of operations that can be done in parallel is relatively small (<100) and fixed. Also, SIMD requires vector processors which are expensive.

**Parallel SQL**

Since standard SQL is the de facto language for data manipulation across multiple platforms it is natural to expect that, perhaps, a form of parallel SQL can solve the problem.

Indeed, SQL database servers from Oracle and Microsoft have allowed some form of parallel execution of queries and bulk operations for years. However, it is complex (think nested outer and inner joins) when done explicitly by the programmer, or limited mostly to parallel execution on the same node.

For many computing domains such as batch processing of log and text files for web crawling and page analysis, the computations on the data are relatively simple. These techniques use operations such as reading, filtering, extraction, counting, reordering, matching and similar. However, when the input data is very large these computations have to be distributed across thousands of machines in order to finish the job in a reasonable amount of time.

To achieve this, a large amount of code has to be written to distribute the programs and data to multiple nodes, parallelize the computations, and handle coordination of load-balancing and failures. The actual ‘useful’ code is relatively small and is obscured by the amount of boiler-plate ‘overhead’ code.

Jeffrey Dean and Sanjay Ghemawat from Google tried to simplify this by creating MapReduce.

**Distributed Data Parallelism**

**MapReduce**

MapReduce was inspired by the Map and Reduce operators in functional languages such as LISP. Logically, MapReduce can be reduced (no pun intended) to the following:

- Treat the data as a set of \(<\text{Key}, \text{Value}\rangle\) pairs
- Input readers read the data and generate \(<\text{Key}, \text{Value}\rangle\) pairs
- The user provides two functions, Map and Reduce, that are called by the runtime
- Map:
  - Take a list of \(<\text{Key}, \text{Value}\rangle\) pairs, process them, and output a list of new \(<\text{Key}, \text{Value}\rangle\) pairs
  - Each pair is processed in parallel
- Reduce:
  - Take all of the values associated with the same \(<\text{Key}\rangle\)
  - Process and output a new set of values for this Key
  - Each reducer can be processed in parallel
To illustrate MapReduce, let’s consider the problem of producing a summary of word occurrences in a large collection of text documents.

The programmer has to write two functions:

1. The Map function will iterate over the contents of a single doc and produce (word, count) pairs.

2. The Reduce function will combine the separate counts for each word and produce a single result.

The MapReduce library and runtime handles all the rest: managing the nodes where the computations take place, ‘serving’ the documents to the ‘map’ operations, collecting and combining intermediate results, and running the reduce operations on the combined list.

```
Map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

Reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Figure 2 Map, Reduce Pseudocode

The MapReduce programming model has several advantages:

- The conceptual model is simple to understand: just two basic operations
- It is generic enough for expressing many practical problems that deal with data analysis
- The implementation hides the messy details of parallelization and fault recovery
- It scales well on thousands of commodity machines and terabytes of data
- It can be incorporated into many procedural and scripting languages.
- Automates data distribution & result aggregation
- Restricts the ways data can interact to eliminate locks (no shared state = no locks!)

Despite the simplicity of the concepts, MapReduce has several disadvantages:

- Using MapReduce with a conventional programming language such as Java, C++, or Python is not simple. The programmer has to code even simple operations
- One-input, two-phase data flow is rigid, and hard to adapt to other applications
- Opaque nature of the map and reduce functions impedes optimization

Despite the shortcomings, MapReduce makes a large subset of distributed problems easier to code. MapReduce programming has gained in popularity after it was implemented by Apache in the Hadoop project.

**Hadoop**

Hadoop MapReduce is a programming model and software framework for writ-
ing applications that can process vast amounts of data in parallel on large clusters of compute nodes.

Originally developed at Yahoo! by Doug Cutting and named after his son’s elephant toy, Hadoop source was eventually contributed to Apache.

Hadoop is composed of a distributed file system (HDFS), and a MapReduce engine.

The engine consists of one JobTracker, to which client applications submit MapReduce jobs, and multiple TaskTrackers. The JobTracker pushes work out to non-busy TaskTracker nodes in the cluster, while trying to keep the work as close to the data as possible. The TaskTrackers runs the Map/Reduce and collects the results.

Facebook claims to have the largest Hadoop cluster of 30 PB of storage.

Despite its success, Hadoop is yet to be widely adopted because of its steep learning curve and difficulty in debugging and monitoring submitted jobs.

**Data-Parallel Languages**

Hadoop has extension libraries for the most popular languages such as Java and Python. These libraries simplify many of the MapReduce chores, but some complexity and verbosity remains.

To simplify MapReduce even further, a few specialized languages were developed.

**PIG and Pig Latin**

The Pig Latin language is a hybrid between a high-level declarative query language, such as SQL and a low-level procedural language, such as C++ or Java.

Writing a Pig Latin program is similar to specifying a query execution plan (i.e., data flow graph) which makes it easier for programmers to understand the data flow and control its execution.

The language is flexible and has a fully nested data model. It enables the programmer to write programs that operate over input files without mandating a schema and it can also be extended with user defined functions (UDF).
A Pig Latin program is compiled into a sequence of MapReduce jobs which are then executed by the Pig framework.

Operations such as ‘load’, ‘for each’, ‘filter’ can be implemented as map functions.

Operations such as ‘group’, ‘store’ can be implemented as reduce functions.

The following is an example of a PIG Latin program to analyze log files from a web site, and find the top 10 most visited pages in each category (such as News, Sports, Cooking, etc.).

```pig
visits = load '/data/visits' as (user, url, time);
gVisits = group visits by url;
visitCounts = foreach gVisits generate url, count(urlVisits);
urlInfo = load '/data/urlInfo' as (url, category, pRank);
visitCounts = join visitCounts by url, urlInfo by url;
gCategories = group visitCounts by category;
topUrls = foreach gCategories generate top(visitCounts, 10);
Store topUrls into '/data/topUrls';
```

Figure 4 Sample PIG Program

**Hive and HiveQL**

Hive is a data warehouse system built on Hadoop for providing data services such as summarization, ad-hoc queries, and analysis. The source code was initially developed by Facebook, but was later contributed to the Apache Foundation.

Notable users of Hive include Netflix and Facebook.

HiveQL is a declarative language, similar to SQL, that it is used to create programs to run on Hive. It supports most SQL features, and custom MapReduce scripts. The HiveQL compiler translates HiveQL statements into a directed acyclic graph of MapReduce jobs, which are submitted to Hadoop for execution.

```hive
FROM docs
MAP text USING 'python wc_map.py' AS (word, count)
CLUSTER BY word
) words
REDUCE word, count USING `python wc_reduce.py`
```

Figure 5 HiveQL Example

Hive Components:

- Parser (based on antlr): translates HiveQL into Abstract Syntax Tree (AST)
- Semantic analyzer: translates AST into DAG (directed acyclic graph) of MapReduce tasks
  - Logical plan generator: converts AST into operator trees
  - Optimizer (logical rewrite): rewrites the operator trees
  - Physical plan generator: generates MapReduce tasks from operator trees
- Execution libraries:
  - Operator implementations, user defined functions (UDF)
  - Serialization & object inspector, meta-store
  - File format & record reader
Hive and HiveQL can be thought of as a SQL engine on top of MapReduce and a distributed file system.

It supports data partitioning based on columnar values where one or more partition columns may be specified:

```
CREATE TABLE food (id INT, msg STRING)
PARTITIONED BY (dt STRING);
```

It then creates a subdirectory for each value of the partition column. For example:

```
/user/hive/warehouse/food/dt=2010-01-20/
/user/hive/warehouse/food/dt=2010-01-21/
... 
```

Hive and HiveQL prove to be useful for ad-hoc querying of very large datasets. The similarity to SQL makes it easier for mainstream programmers and analysts to use.

**Sawzall**

Sawzall (a brand name of a reciprocating saw) is a domain specific, high level language and interpreter that runs on top of MapReduce. It was developed by Google for parallel analysis of very large data sets like logs files.

As opposed to other procedural languages, a Sawzall program is designed to process one record at a time. It does not preserve any state (values of variables) between records, and is well suited for execution as the map phase of a MapReduce.

The Sawzall runtime creates many instances of the program on multiple nodes and serves them one record at a time. Each program instance reads and analyzes its input record, and outputs zero or more intermediate values of integers, strings, key-value pairs, tuples, and so on. for each input record.

The output is combined with values from other program instances/records by the Sawzall runtime.

The intermediate values are sent to other nodes that collate and reduce the intermediate values to create the final results.

A final step collects, collates, and formats the final results, to be viewed or stored in a single file.

The following program outputs the count, sum, and square toot of numbers (floating point) stored one per record in a collection of files.

```
count: table sum of int;
total: table sum of float;
sum_of_squares: table sum of float;
x: float = input;
emit count <- 1;
emit total <- x;
emit sum_of_squares <- x * x;
```

**Figure 6 Example of a Sawzall program**

The constrained model that Sawzall provides (one-record-at-a-time) has proven valuable. Despite the fact that some problems, such as database Joins, poorly fit the model, most of the processing done on large data sets fits well. The benefit
gained in notation, convenience, and expressiveness has made Sawzall a popular programming language for this domain.

**Comparison**

<table>
<thead>
<tr>
<th></th>
<th>Pig Latin</th>
<th>HiveQL</th>
<th>Sawzall</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Language</strong></td>
<td>Semi-declarative</td>
<td>Declarative</td>
<td>Procedural</td>
</tr>
<tr>
<td><strong>Schemas</strong></td>
<td>Yes (implicit)</td>
<td>Yes (explicit)</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Nesting</strong></td>
<td>Containers</td>
<td>Full</td>
<td>Containers</td>
</tr>
<tr>
<td><strong>User-defined functions</strong></td>
<td>Yes (Java)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Custom serialization/deserialization</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Joins</strong></td>
<td>Yes+</td>
<td>Yes (equality)</td>
<td>No</td>
</tr>
<tr>
<td><strong>MapReduce steps</strong></td>
<td>Multiple</td>
<td>Multiple</td>
<td>Single</td>
</tr>
</tbody>
</table>

The strength of these languages lies at the simplicity of the dataflow programming model.

**Beyond Hadoop**

The MapReduce programming model, supporting systems, and related languages proved very useful for solving many problems related to analyzing very large datasets especially as they relate to web (such as analyzing web pages and log files). Despite its popularity MapReduce & Hadoop has several shortcomings.

What’s wrong with MapReduce and Hadoop?

- Only Map then Reduce – that’s all. Applications that update databases or generate data cannot be adapted easily to use MapReduce.
- Reduce write to replicated storage
- Complex jobs pipeline and multiple stages - the data is passes through several stages
- Poor fault tolerance between stages
- Map assumes data is always available – cannot wait for data to be generated
- Output of Reduce: two network copies, three disks – extra space/devices are needed
- “Join” combines inputs of different types - redundant type conversion
- “Split” produces outputs of different types (redundant type conversion)
- Many problems can be solved by MapReduce, but
  - Ugly to program
  - Hard to avoid performance penalty
  - Some merge joins are very expensive
  - Is not suitable for non-batch applications

To solve complex problems that cannot be reduced easily to Map & Reduce jobs, other solutions have been developed.

**Dryad and DryadLINQ**

Dryad (a tree-nymph in Greek mythology) is a general-purpose execution environment for distributed, data-parallel applications developed at Microsoft Research. 

*Dryad is a general-purpose execution environment for distributed, data-parallel applications.*
DryadLINQ is a system and a set of language extensions that enable a new programming model for large scale distributed computing. DryadLINQ allows developers to implement Dryad applications in managed code (such as C++, C#, and VisualBasic) by using an extended version of the LINQ programming model and API.

A Dryad application is composed of computational “vertices” with communication “channels” to form a dataflow graph. Dryad runs the application by executing the vertices of this graph on a set of available computers, communicating via files, TCP pipes, and shared memory channels.

The vertices, created by the programmer, are simple and are usually written as sequential programs with no thread creation or locking. Dryad schedules the vertices to run simultaneously on multiple CPUs on single or multiple computers.

The job manager (JM) consults the name server (NS) to discover the list of available computers. It maintains the job graph and schedules running vertices (V) as computers become available using the daemon (D) as a proxy. Vertices exchange data through files, TCP pipes, or shared-memory channels. The shaded bar indicates the vertices in the job that are currently running.

<table>
<thead>
<tr>
<th>Dryad</th>
<th>MapReduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layers</td>
<td>Execution layer</td>
</tr>
<tr>
<td>Job</td>
<td>Arbitrary DAG</td>
</tr>
<tr>
<td>Plugin policies</td>
<td>Many plug-in policies</td>
</tr>
<tr>
<td>Program</td>
<td>Graph generated</td>
</tr>
<tr>
<td>Complexity</td>
<td>Complex (many features)</td>
</tr>
<tr>
<td>Maturity</td>
<td>New (&lt;2 years)</td>
</tr>
<tr>
<td>Deployment</td>
<td>Small</td>
</tr>
<tr>
<td>Distribution</td>
<td>Internal (Microsoft)</td>
</tr>
</tbody>
</table>

Description of a Dryad Job as Directed Acyclic Graph

Dryad is a research project and is used internally at Microsoft by various projects but is not available either as open source or commercially.
Domain Specific Solutions

After trying tools like MapReduce and Hadoop for batch processing, some companies found that they are just not powerful enough. Hadoop is great for batch processing tasks that are "embarrassingly parallel"\(^\text{13}\), but many of the difficult Big Data tasks confronting companies today are much more complex than that. They can involve complex joins, ACID (atomicity, consistency, isolation, durability) requirements, real-time requirements, huge graph traversal, interactive analysis, online transaction processing or the need for continuous incremental updates.

In these cases, Hadoop is unable to provide the levels of throughput and responsiveness required by these applications.

For specific domains, a few specialized environments and languages have emerged.

Back to SQL

SQL has been around more than 30 years. To overcome scalability and performance issues, a new generation of products called NoSQL (Not Only SQL) has emerged. They claim to achieve extreme scalability and very high performance through relaxing or eliminating transaction support and moving back to a low-level DBMS interface, thereby eliminating SQL. NoSQL is good for certain applications that don’t require transactions, but is not good for transactions requiring complex joins or ACID requirements.

These SQL engines (sometimes called ‘NewSQL’) were designed to maintain the old SQL benefits yet provide very high level of scalability and performance. These systems run on clusters of commodity machines and can scale out transparently. For large scale online transaction processing systems that require short response time, NewSQL systems may be the answer.

VoltDB\(^\text{14}\) and Clusterix\(^\text{15}\) are examples of such systems.
**MPI and OpenMPI**

A Message Passing Interface (MPI) has been used on supercomputers since the early 1990s. The open source version OpenMPI\(^\text{16}\) which merged various implementations into one standard is used today by the top 500 supercomputers.

MPI was designed to provide access to advanced parallel hardware and support large number of processors, clusters, and large global memories (single or partitioned).

MPI is suitable for applications (such as modeling, simulation, and fluid dynamics) that require complex algorithms on Big Data in which processors communicate directly at very high speed in order to deliver the needed performance.

MPI has language bindings for C, C++, FORTRAN, Java and others, and is used heavily within the scientific community.

**BSP and Pregel**

The Bulk Synchronous Parallel (BSP\(^\text{17}\)) is an abstract computer model for running parallel algorithms. A BSP computer consists of a collection of processors, each with its own memory. Access to its own memory is fast, while access to remote memory is slower (but access time is fixed).

All BSP programs execute in a series of steps:
1. Concurrent computation on local processors using local memory
2. Communication between all processors
3. Synchronization where a processor waits for the others to complete their communications

Applications that require an analysis of complex, very-large-scale dynamic graphs (billions of nodes, billions of edges) such as social graphs, location graphs, learning and discovery, and network optimization require a different model than MapReduce.

Since MapReduce was found to be inefficient for this type of work, Google’s Pregel\(^\text{18}\) architecture uses a BSP model to enable highly efficient graph computing of ‘internet scale’.

In Pregel, programs are expressed as a sequence of iterations. In each iteration, a vertex can, independently of other vertices, receive messages sent to it in the previous iteration, send messages to other vertices, modify its own and its outgoing edges’ states, and mutate the graph’s topology.

Since many applications require graph traversal and processing, the BSP model needs to available as well.

**Dremel and BigQuery**

Dremel (a brand of power tools) is a scalable, interactive ad-hoc query system for analysis of large, read-only nested data. Per Google reports, the language is capable of running aggregation queries over trillion-row tables in seconds. The system scales to thousands of CPUs and petabytes of data, and has thousands of users at Google. The system is based on a columnar storage representation for nested records.

Dremel\(^\text{19}\) provides a high-level, SQL-like language to express ad hoc queries. It executes queries natively without translating them into MapReduce jobs first.

BigQuery\(^\text{20}\) is a commercial service from Google that is based on Dremel. BigQuery is a web service/API that lets one perform interactive analysis of massive
datasets—up to billions of rows.

BigQuery can be viewed as a SQL engine on a very large, mostly read-only, distributed and clustered database.

**X10**

X10 is a language that is an evolution of ‘Java for concurrency’. It is highly scalable, runs on heterogeneous environments and provides functionality similar to MPI.

Developed at IBM Research, X10 is designed specifically for parallel programming using the partitioned global address space (PGAS) model. All computation is divided among a set of places, each of which holds some data and hosts one or more activities that operate on the data.

The X10 language features:

- Ability to specify fine-grained concurrency (threads, fork/join, GPU, MPI, parallel code segments)
- Ability to distribute computation across large-scale clusters
- Ability to represent heterogeneity at the language level
- Simple programming model for computation offload
- Modern OO language features (build libraries/frameworks)
- Support for writing deadlock-free and determinate code
- Interoperability with Java

X10 has some interesting features, in particular its built-in concurrency and distributed computing models and it should be considered for Big Data processing in the areas of scientific computing and business analytics.

**A Language for Big Data**

Based on the various languages or language extensions that were reviewed in this article, a programming language and runtime for Big Data processing will require several attributes to make it easy to write such programs by programmers with average skills.

1. Programming models & features
   - Data parallel (implicit, hints, explicit)
   - MapReduce
   - Dataflow (on Directed Acyclic Graph)
   - BSP for graph processing
   - Lock-free
2. Runtime:
   - Framework handles the partitioning, distribution, scheduling of work, monitoring, and collection of results
   - Dynamically allocate nodes based on the data size and requested response time (SLA)
   - Distribution over large number of heterogeneous nodes each node with multiple cores
   - Shared nothing architecture
   - Commodity (cheap) hardware
   - Massive parallelism (1000s of nodes with 100s of cores)
3. Language support
   - Similarity to common programming languages and programming model
   - Extensions to Java or C++
   - SQL like query language
4. Data abstraction
a. Data schema as part of the language
b. Data-level API (Select, Insert, Update, Delete)

5. Data sources
a. Multiple data sources
b. Local and distributed file systems
c. Parallel file system
d. Structured (relational, object oriented)
e. Semi-structured (XML, JSON)
f. Unstructured (text, email)
g. Key-value databases (such as Memcachedb)
h. Column databases (BigTable)
i. Document databases (Apache CouchDB)
j. Graph databases (Neo4j)

6. Static & dynamic data (quasi real time)

7. Batch and online processing

8. Data filtering and data scrubbing

9. Data normalization

10. Built-in core operations (similar to SQL)
    a. Sort
    b. Filter
c. Split
d. Group
e. Union
f. Transform
g. Top
h. Join
i. Expand

11. Statistical analysis operators

12. Visual design

Conclusion

Problem: huge amounts of data are produced and accumulated daily, but large-scale processing of that data on commodity computers is difficult → Big Data is difficult.

We have lots of resources (1000s of cheap PCs), but they are very hard to utilize.
We have clusters with over 10k cores, but it is hard to program 10k concurrent threads.
We 1000s of storage devices, but some may break daily.
We have petabytes of storage, but deployment and management is a big headache.
We have petabytes of data, but analyzing it is difficult.
We have a lot of programming skills, but using them for Big Data processing is not simple.

To solve these problems various solutions have been proposed and partially developed. Most solutions require imposing a few constraints on the problem domain. The most successful one is the MapReduce/Hadoop programming model and runtime system which limits the domain to read-only datasets that can be read, split, and analyzed using Map and Reduce operators.

To overcome some of the limitations of MapReduce, new languages or language extensions such as Pig Latin, HiveQL, and Sawzall were developed. They provide
a simpler programming model and environment for creating programs to analyze large amounts of data on clusters of commodity computers.

Beyond MapReduce solutions, old SQL engines were reengineered, and traditionally supercomputing languages such as X10 were extended to support Big Data as well.

These new and more powerful languages and runtimes will make Big Data processing not such a big deal.

**References**

1. MGI - Big data: The next frontier for innovation, competition, and productivity. http://www.mckinsey.com/Insights/MGI/Research/Technology_and_Innovation/Big_data_The_next_frontier_for_innovation
3. ParaSort: David M. W. Powers, Parallelized Quicksort and Radixsort with Optimal Speedup
5. Oracle documentation: http://docs.oracle.com/cd/B10501_01/server.920/a96524/c20paral.htm
6. J. Dean and S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters
15. Clusterix: http://www.clustrix.com/
17. BSP: http://en.wikipedia.org/wiki/Bulk_Synchronous_Parallel
20. BigQuery: https://developers.google.com/bigquery/
21. X10: http://x10-lang.org/

**NOTICES**

Copyright © 2012 CA. All rights reserved. All trademarks, trade names, service marks and logos referenced herein belong to their respective companies.

The information in this publication could include typographical errors or technical inaccuracies, and CA, Inc. (“CA”) and the authors assume no responsibility for its accuracy or completeness. The statements and opinions expressed in this publication are those of the authors and are not necessarily those of CA.

Certain information in this publication may outline CA’s general product direction. However, CA may make modifications to any CA product, software program, service, method or procedure described in this publication at any time without notice, and the development, release and timing of any features or functionality described in this publication remain at CA’s sole discretion. CA will support only the referenced products in accordance with (i) the documentation and specifications provided with the referenced product, and (ii) CA’s then-current maintenance and support policy for the referenced product. Notwithstanding anything in this publication to the contrary, this publication shall not: (i) constitute product documentation or specifications under any existing or future written license agreement or services agreement relating to any CA software product, or be subject to any warranty set forth in any such written agreement; (ii) serve to affect the rights and/or obligations of CA or its licensees under any existing or future written license agreement or services agreement relating to any CA software product; or (iii) serve to amend any product documentation or specifications for any CA software product.

Any reference in this publication to third-party products and websites is provided for convenience only and shall not serve as the authors’ or CA’s endorsement of such products or websites. Your use of such products, websites, any information regarding such products or any materials provided with such products or on such websites shall be at your own risk.

To the extent permitted by applicable law, the content of this publication is provided "AS IS" without warranty of any kind, including, without limitation, any implied warranties of merchantability, fitness for a particular purpose, or non-infringement. In no event will the authors or CA be liable for any loss or damage, direct or indirect, arising from or related to the use of this publication, including, without limitation, lost profits, lost investment, business interruption, goodwill or lost data, even if expressly advised in advance of the possibility of such damages. Neither the content of this publication nor any software product or service referenced herein serves as a substitute for your compliance with any laws (including but not limited to any act, statute, regulation, rule, directive, standard, policy, administrative order, executive order, and so on (collectively, “Laws”) referenced herein or otherwise or any contract obligations with any third parties. You should consult with competent legal counsel regarding any such laws or contract obligations.