Motivation

Huge amounts of data...

- New York Stock Exchange generates about 1TB of new trade data per day
- Facebook hosts approx. 10 billion photos ($\sim$ 1PB)
- Internet Archive stores around 2PB of data (with a growing rate of 20TB/month)
- Large Hadron Collider (LHC) will produce around 15PB of data per year

...entail huge challenges:

- Secure, redundant storage?
- Data analysis/handling for $>$ 1TB?
- Bottleneck: I/O!

$\rightarrow$ Cloud Computing to the rescue!

What is Cloud Computing?

Anonymous

[. . .] Unfortunately the marketing guys got hold of the term before the technicians had known what Cloud Computing is [. . .]

Larry Ellison (CEO of Oracle on Cloud Computing)

We've redefined cloud computing to include everything we currently do. So it has already achieved dominance in the industry. I can't think of anything that isn't cloud computing [. . .] The computer industry is the only industry that is more fashion-driven than women's fashion. [. . .] I have no idea what anybody is talking about. I mean it is really just complete gibberish [. . .]

Cloud Computing - Definitions

Peter Mell and Tim Grance (National Institute of Standards and Technology)

Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

C.Baun, M.Kunze, J.Nimis, S.Tai (Cloud Computing, Springer 2009)

Building on compute and storage virtualization, and leveraging the modern Web, virtualization, and leveraging the modern Web, Cloud Computing provides scalable, network-centric, abstracted IT infrastructure, platforms, and applications as on-demand services that are billed by consumption.

Cloud Computing Overview

Cloud Computing Essentials

- Pay-as-you-go pricing model
- Build upon virtualization of computing resources
- Minimal management on the user part
- Available and configurable on-demand $\rightarrow$ High flexibility!

Cloud Computing Flavors

- SaaS (Software-as-a-Service): WAN-enabled application services (e.g., Google Apps)
- PaaS (Platform-as-a-Service): Foundational elements to develop new applications (e.g., Google Application Engine)
- IaaS (Infrastructure-as-a-Service). Providing computational and storage infrastructure in a centralized, location-transparent service (e.g., Amazon EC^2)

Cloud Computing Applications

Success Stories

- Animoto[1], a service which automatically generates videos from pictures, achieved a scale up from 50 instances to 3,500 instances in three days
- Computation to take scanned images from over 60 years online (approx. 4 TB of data) took a weekend on Amazon's EC[2] using S3 storage service.
- Mogulus[2] streams 120,000 live TV channels without any own hardware expect laptops.
  $\rightarrow$ According to CEO IaaS is reason for being in business.

[1] `http://animoto.com/`
[2] `http://livestream.com/?referrer=mogulus`

Cloud Computing Evaluation

Benefits

- System scales dynamically, no need to optimize for worst case
- Low total cost of ownership, only operative cost
- Facilitates innovations by low entry barriers & time from idea to system
- Generally: Reduces risks for service users

Open Issues and Limitations

- Legal restrictions, e.g. banking sector
- Security and privacy
- Possible vendor lock-in because of missing standardized APIs
- Generally: What applications work in the cloud? Multiplayer games?

→ Limitations = Open research problems!

MapReduce: A Programming Paradigm for the Cloud

Mission Statement

Make large scale data processing ($>$ 1TB) across 100 - 1,000 CPUs easy

Key Characteristics

- Automatic parallelization and distribution of workload
- Built-in fault tolerance
- Available status and monitoring tools
- Clean abstraction for programmers
- Borrows ideas from functional programming

$\rightarrow$ Open source implementation: Hadoop @ `http://hadoop.apache.org/`

Applications of MapReduce

Industrial Applications

- Computation of Google's PageRank
- Data Mining on large search logs
- Computation of inverted indexes

Research Applications

- HadoopDB (Infrastructure: MapReduce, Nodes: Relational DBMS) [1]
- Distributed RDFS reasoning [8]
- XXX with MapReduce

Review of Functional Programming

Observations

- Functional operations do no modify data structures, they create new ones
- Original data still exists in unmodified form
  → No synchronization required, if multiple operations are executed, since original data state is still available
- Data flows are implicit in program design
  → For new data that should be kept, a new name has to be assigned
- Order of operations does not matter
  → Each operation has its own copy

Functional Programming Examples

Example I: Function Currying

```
- fun f(a,b,c,d) = a  + b + c + d;
val f = fn : int * int * int * int -> int
- f(1,2,3,4);
val it = 10 : int
```

$\rightarrow +(1,2,3,4) = 10$

```
- fun f a b c d = a + b + c + d;
val f = fn : int -> int -> int -> int -> int
```

$\rightarrow +(a,+(b,+(c,d)))$

```
- f 1;
val it = fn : int -> int -> int -> int
```

$\rightarrow +(1,+(b,+(c,d)))$

Functional Programming Examples

Example I: Function Currying cont'ed

```
- it 2;
val it = fn : int -> int -> int
```

$\rightarrow +(1, +(2, +(c, d)))$

```
- it 3;
val it = fn : int -> int
```

$\rightarrow +(1, +(2, +(3, d)))$

```
- it 4;
val it = 10 : int
```

$$\left(\begin{array}{cc} +(1, +(2, +(3, 4))) & = \\ +(1, +(2, 7)) & = \\ +(1, 9) & = \\ 10 & \end{array}\right)$$

Functions on Lists

Example II: *reverse* **and** *append*

```
- fun reverse nil = nil |
      reverse(h::t) = reverse(t) @ h::nil ;
val reverse = fn : 'a list -> 'a list
- fun append x nil = x::nil |
      append x lst = reverse(x::(reverse lst));
val append = fn : 'a -> 'a list -> 'a list
- append 5;
val it = fn : int list -> int list
- it [1,2,3,4];
val it = [1,2,3,4,5] : int list
```

Functions on Lists cont'ed

Example III: *doublist*

```
- fun doublist nil = nil |
      doublist(h::t) = 2 * h::(doublist t);
val doublist = fn : int list -> int list
- doublist [1,2,3,4];
val it = [2,4,6,8] : int list
```

Example IV: *inclist*

```
- fun inclist nil = nil | inclist(h::t) = (h+1)::(inclist t);
val inclist = fn : int list -> int list
- inclist [1,2,3,4];
val it = [2,3,4,5] : int list
```

MapReduce

Example V: map

```
- fun map f nil = nil |
      map f (h::t) = (f h)::(map f t);
val map = fn : ('a -> 'b) -> 'a list -> 'b list
```

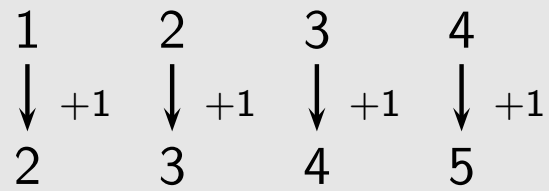
```
- val doublist = map (fn x => 2 * x);
val doublist = fn : int list -> int list
- val inclist = map (fn x => x + 1);
val inclist = fn : int list -> int list
- doublist [1,2,3,4];
val it = [2,4,6,8] : int list
- inclist[1,2,3,4];
val it = [2,3,4,5] : int list
```

MapReduce cont'ed

Parallelism in *map* phase

Computation of f for each element is not dependent on other elements in list.

Example: *inclist*$[1, 2, 3, 4]$



Functions on Lists cont'ed

Example VI: *sum*

```
- fun sum nil = 0 |
      sum(h::t) = h + sum t;
val sum = fn : int list -> int
- sum [1,2,3,4,5];
val it = 15 : int
```

Example VII: *flatten*

```
fun flatten nil = nil |
    flatten (h::t) = h @ flatten t;
val flatten = fn : 'a list list -> 'a list
- flatten [[1,2],[3,4],[5,6,7]];
val it = [1,2,3,4,5,6,7] : int list
```

MapReduce

Example VIII: *reduce*

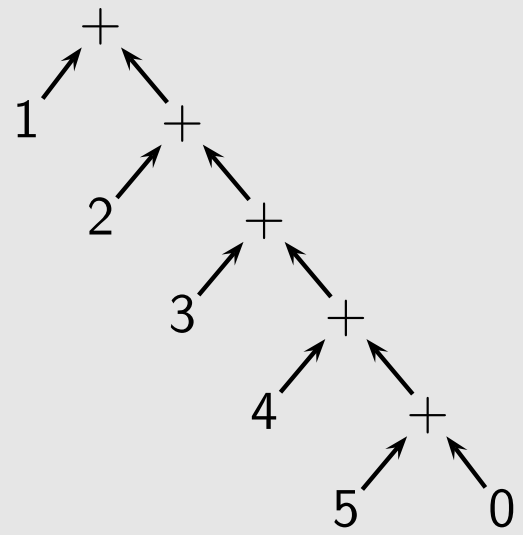
```
fun reduce f b nil = b |
    reduce f b (h::t) = f(h, reduce f b t);
val reduce = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

```
- val sum = reduce (fn(a,b) => a + b) 0;
val sum = fn : int list -> int
- val flatten = (fn xs => reduce (fn(a,b) => a @ b) nil xs);
val flatten = fn : 'a list list -> 'a list
- sum [1,2,3,4,5];
val it = 15 : int
- flatten [[1,2],[3,4],[5,6,7]];
val it = [1,2,3,4,5,6,7] : int list
```

MapReduce cont'ed

Example: *sum* $[1, 2, 3, 4, 5]$

```
- val sum = reduce (fn(a,b) => a + b) 0;
```



MapReduce: Programming Model

map

$map(in\_key, in\_value) \rightarrow (out\_key, intermediate\_value)$

- inputs: records from data source as (*key*, *value*) pairs, e.g. (*filename*, *line*)
- outputs: one or more intermediate values with an output key

reduce

$reduce(out\_key, intermediate\_value\ list) \rightarrow out\_value\ list$

- After *map* phase all intermediate values for one output key are combined together in a list
- *reduce* combines those intermediate values into one or more final values for the same output key

$\rightarrow$ In practice, usually one final value per key

Example Scenario

Analysis of sensor readings

- Given is a large amount of sensor readings that are generated automatically and store the temperature at a specific location (e.g. each ms)
- The data is stored in one huge file ($>$ 1TB) in the Google File System and looks as follows:

```
...
s1 +12
s2 -10
s3 -4
s1 +15
s1 +30
...
```

$\rightarrow$ How can we determine the maximum temperature for each sensor w/ MapReduce?

Solution in Java w/ Hadoop

map

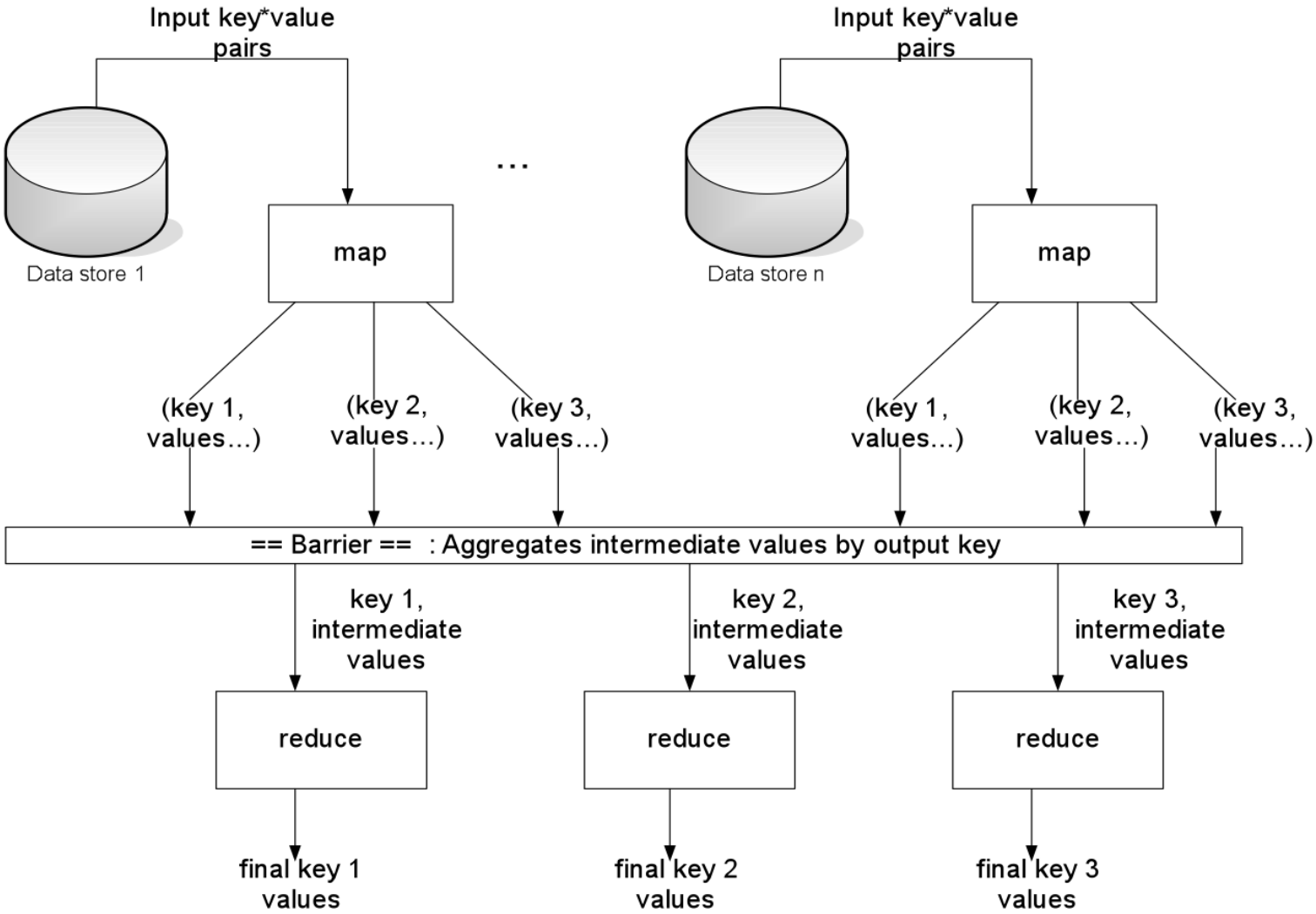
```
//              (in_key,           in_value)
public void map(LongWritable key, Text value,
                Context context) ... {
   String line = value.toString();
   String sensor = line1.substring(0, 2);
   String temperature = line1.substring(3, line1.length());
   int temp = -1;
   if (temperature.startsWith("+")) {
      temp = Integer.parseInt(temperature.substring(1,
                              temperature.length()));
   }
   else {
      temp = Integer.parseInt(temperature);
   }
   //           (out_key,          intermediate_value)
   context.write(new Text(sensor), new IntWritable(temp));
}
```

Solution in Java w/ Hadoop cont'ed

reduce

```
//                (out_key,  intermediate_value list)
public void reduce(Text key, Iterator<IntWritable> values,
                   Context context) ... {
   int maxValue = Integer.MIN_VALUE;
   for (IntWritable value : values) {
      maxValue = Math.max(maxValue, value.get());
   }
   // out_value list
   context.write(key, new IntWritable(maxValue));
}
```

System Architecture[3]



[3] Picture originally from [2]

Google File System (GFS) $\sim$ Hadoop File System (HFS)

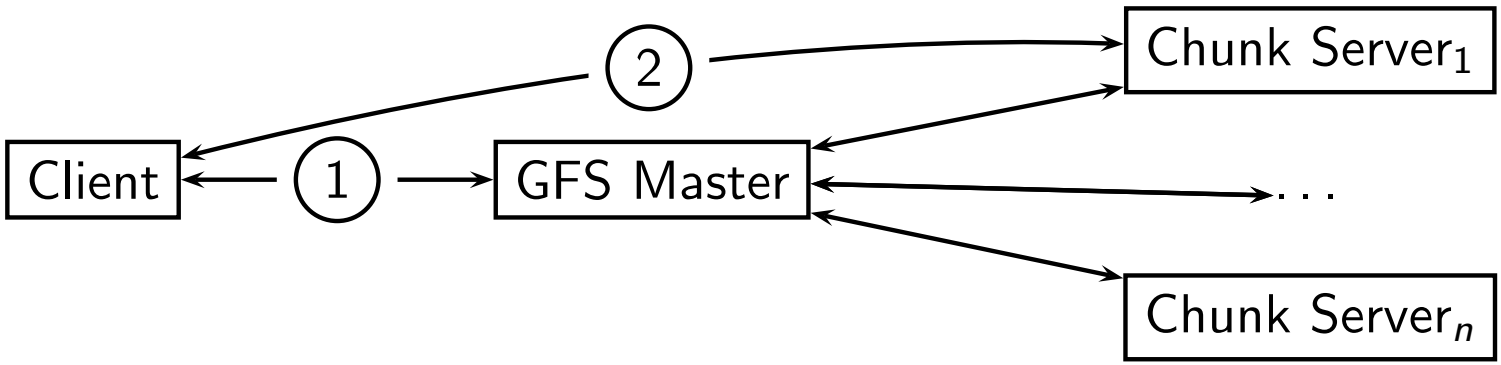
Design Goal

Redundant storage of huge amounts of data on cheap and unreliable machines

Assumptions and (some) consequences

- High component failure rates
  $\rightarrow$ Replicate data on different machines (usually 3 times)
- Modest ($\sim 1,000,000$ files) number of large ($> 1GB$ files)
  $\rightarrow$ Increase chunk size to $64MB$ (on ext3: $4KB$!)
- Files are written once, mostly appended to (perhaps concurrently)
  $\rightarrow$ Add atomic record append operation
- Large in order streaming reads
  $\rightarrow$ No need for data caching on the client side
- High throughput more important than low latency

GFS/HFS Architecture Overview



File mutations

1. "Open" file for writing/appending
2. Writes go directly to single chunk server, which distributes changes

GFS/HFS Architecture Overview cont'ed

Master server duties

- Metadata storage (location of chunk sizes, etc.)
- Namespace management, locking (naming, creating, deleting files)
- Periodic communication with chunk servers
- Chunk creation, re-replication, balancing
- (Lazy) garbage collection: delete log and rename file to hidden name

Problem: Single Point of Failure & I/O Bottleneck

- Need to keep additional shadow master servers to prevent total failure
- Traffic between client and master server is low (only metadata)
- Updates are done on chunk servers directly

MapReduce Key Characteristics revisited

Parallelism

- All *map*s run in parallel
- All *reduce*rs run in parallel

Locality

- Maps are instantiated close to the data (ideally on same machine)
- Normal chunk size: $64MB$ (GFS chunk size)

Fault Tolerance

- Master detects worker failure
- If failure cannot be resolved, partial results will be returned

Optimizations

Bottlenecks

- *reduce* phase cannot start before *map* phase is finished

  → Execute slow-running map tasks redundantly and use result of first finished copy
- Transferring map results can be slow due to limited I/O bandwidth

  → Introduce new function type *combine*, which ...
  - runs on same machine as *map*s, and
  - causes mini-reduce phase to occur before real reduce phase

Optimizations cont'ed

Example

- Maximum temperature of a sensor w/o *combine*:
  - $map_1 : \{(s_1, 12), (s_1, 20), (s_1, 15)\}$
  - $map_2 : \{(s_1, 23), (s_1, 13), (s_1, 30)\}$
  - $\rightarrow reduce : \{(s_1, 12), (s_1, 20), (s_1, 15), (s_1, 23), (s_1, 13), (s_1, 30)\}$
- Maximum temperature of a sensor w/ *combine*:
  - $map_1 : \{(s_1, 12), (s_1, 20), (s_1, 15)\} \rightarrow combine_1 : \{(s_1, 20)\}$
  - $map_2 : \{(s_1, 23), (s_1, 13), (s_1, 30)\} \rightarrow combine_2 : \{(s_1, 30)\}$
  - $\rightarrow reduce : \{(s_1, 20), (s_1, 30)$

MapReduce From A Database Perspective

Relational DBMS vs. MapReduce[4]

	Relational DBMS	**MapReduce**
Data size	Gigabytes	Petabytes
Access	Interactive and batch	Batch
Updates	Read and write many times	Write once, read many times
Structure	Static schema	Dynamic schema
Integrity	High	Low
Scaling	Nonlinear	Linear

[4] Table originally from [9]

MapReduce From A Database Perspective cont'ed

MapReduce: A major step backwards (D. J. DeWitt and M. Stonebraker in The Database Column)

MapReduce may be a good idea for writing certain types of general-purpose computations, but to the database community, it is:

1. a giant step backward in the programming paradigm for large-scale data intensive applications,
2. a sub-optimal implementation, in that it uses brute force instead of indexing,
3. not novel at all – it represents a specific implementation of well known techniques developed nearly 25 years ago,
4. missing most of the features that are routinely included in current DBMS,
5. incompatible with all of the tools DBMS users have come to depend on.

Evaluation of MapReduce

Pros

- No need to take care of low level aspects, such as synchronization, failure of nodes, message passing, etc.
- Scales (more or less) linearly with the number of machines utilized
- Easy programming model, only map and reduce tasks need to be written

Cons

- High startup costs (e.g. allocation of nodes), i.e. badly suited for online, interactive tasks
- Not trivial to write and maintain multi-stage MapReduce jobs
- Not well-suited for heterogeneous data sets, i.e. no native support for joins

Map-Reduce-Merge

Comparison to MapReduce

- Additional merge phase for processing multiple related heterogeneous datasets
- Allows for efficient implementation of joins in parallel environment
- Relationally complete, while keeping the benefits of the MapReduce idea

But. . .

- Still hard to program, since operating on a low level

Pig(Latin)

Motivation

Data flow language for performing ad-hoc analysis of extremely large data sets

Quick Facts

- Originally developed by Yahoo!, now Apache project
- Imperative-style data flow language
- $(NF)^2$ data model supporting bags, tuples, and maps
- Support for user-defined functions (UDF)
- Compiled to MapReduce jobs

Pig(Latin) Example[5]

Given: DB table urls: (url,category, pagerank)

Query: Return for each sufficiently large category, the average pagerank of high-pagerank URLs in that category.

SQL:

```
SELECT category, AVG(pagerank)
FROM urls WHERE pagerank > 0.2
GROUP BY category HAVING COUNT(*) > 1000000
```

PigLatin:

```
good_urls  = FILTER urls BY pagerank > 0.2;
groups     = GROUP good_urls BY category;
big_groups = FILTER groups
             BY COUNT(good_urls) > 1000000;
output     = FOREACH big_groups
             GENERATE category,
                      AVG(good_urls.pagerank);
```

[5] Adapted from [6]

Hive

Motivation

Learning curve for BI users too steep with Pig, which are used to SQL

Quick Facts

- Originally developed at Facebook, now Apache project
- Declarative-style, SQL-like query language
- Data model is divided into three units:
  1. tables $\sim$ relational DB tables
  2. partitions (keys) $\sim$ relational DB indexes (influence data storage)
  3. buckets divide data in each partition based on a hash of some table column
- Data model supports typical primitive types and arrays and maps natively
- Support for user-defined functions (UDF)
- Operates on top of MapReduce & HFS

Hive Example[6]

Given: Hive tables user: (id, gender, age) and page_view: (viewTime, userid, page_url, ip, date)

Query: Return for each user, who visited a URL on March 3rd 2008, the associated page view information, her gender and age

```
INSERT OVERWRITE TABLE pv_users
SELECT pv.*, u.gender, u.age
FROM user u JOIN page_view pv ON (pv.userid = u.id)
WHERE pv.date = '2008-03-03';
```

→ The results are always inserted into a table

[6] Adapted from `http://wiki.apache.org/hadoop/Hive/Tutorial`

Summary

- Cloud Computing lowers entry barriers and allows for massive on-demand scaling of applications with low startup costs
- MapReduce offers a simple abstraction to parallel computation that is applied to real-world data analysis problems
- Near future: Blend of research results from different areas, e.g. a blend of distributed computing with database research (already happening: HadoopDB [1])

MapReduce@DBIS

Master's Theses

- Please contact Alexander Schätzle[7] or Martin Przyjaciel-Zablocki[8]

WS 2013/14

Team project w/ MapReduce

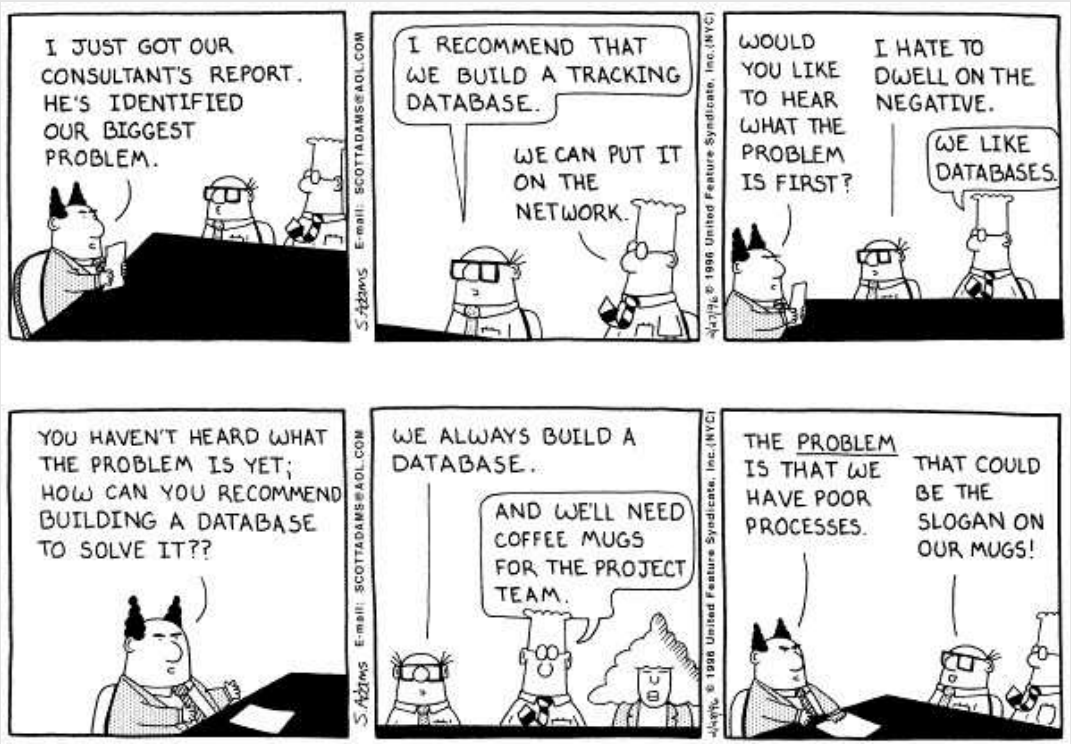
[7] schaetzle@informatik.uni-freiburg.de
[8] zablocki@informatik.uni-freiburg.de

Acknowledgements

List of References

- Section 1: [9, 4]
- Section 2: [4]
- Section 3: [2, 3, 9, 5]
- Section 4: [9, 7]
- Section 5: [10, 6]

Thanks for your attention!



Further Reading I

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz.
HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads.
PVLDB, 2(1):922–933, 2009.

[2] C. Bisciglia, A. Kimball, and S. Michels-Slettvet.
Lecture 2: MapReduce Theory and Implementation.
http://www.slideshare.net/sriprasanna/mapreduce-theory-and-implementation,
2007.

[3] C. Bisciglia, A. Kimball, and S. Michels-Slettvet.
Lecture 3: Distributed Filesystems.
http://www.slideshare.net/sriprasanna/distributed-file-systems-from-google,
2007.

Further Reading II

[4] M. Creeger.
Cloud Computing: An Overview.
ACM Queue, 7(5):2, 2009.

[5] A. Cumming.
A Gentle Introduction to ML.
http://www.dcs.napier.ac.uk/ cs66/course-notes/sml/, 1998.

[6] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins.
Pig Latin: A Not-So-Foreign Language for Data Processing.
In J. T.-L. Wang, editor, SIGMOD Conference, pages 1099–1110. ACM, 2008.

[7] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker.
A Comparison of Approaches to Large-Scale Data Analysis.
In U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, editors, SIGMOD Conference, pages 165–178. ACM, 2009.

Further Reading III

[8] J. Urbani, S. Kotoulas, E. Oren, and F. van Harmelen.
Scalable Distributed Reasoning Using MapReduce.
In A. Bernstein, D. R. Karger, T. Heath, L. Feigenbaum, D. Maynard, E. Motta, and K. Thirunarayan, editors, International Semantic Web Conference, volume 5823 of Lecture Notes in Computer Science, pages 634–649. Springer, 2009.

[9] T. White.
Hadoop: The Definitive Guide.
O'Reilly, 2009.
ISBN: 978-0-596-52197-4.

[10] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. P. Jr.
Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters.
In C. Y. Chan, B. C. Ooi, and A. Zhou, editors, SIGMOD Conference, pages 1029–1040. ACM, 2007.