Data-Intensive Computing: Massive Data Processing

DIC Systems

- Google MapReduce
  - Yahoo Hadoop/PIG
  - Data parallel computing
- IBM Research System S
  - InfosphereStream product
  - Continuous data stream processing
- Microsoft Dryad/Dryad LINQ
  - DAG processing
  - Some SQL query support

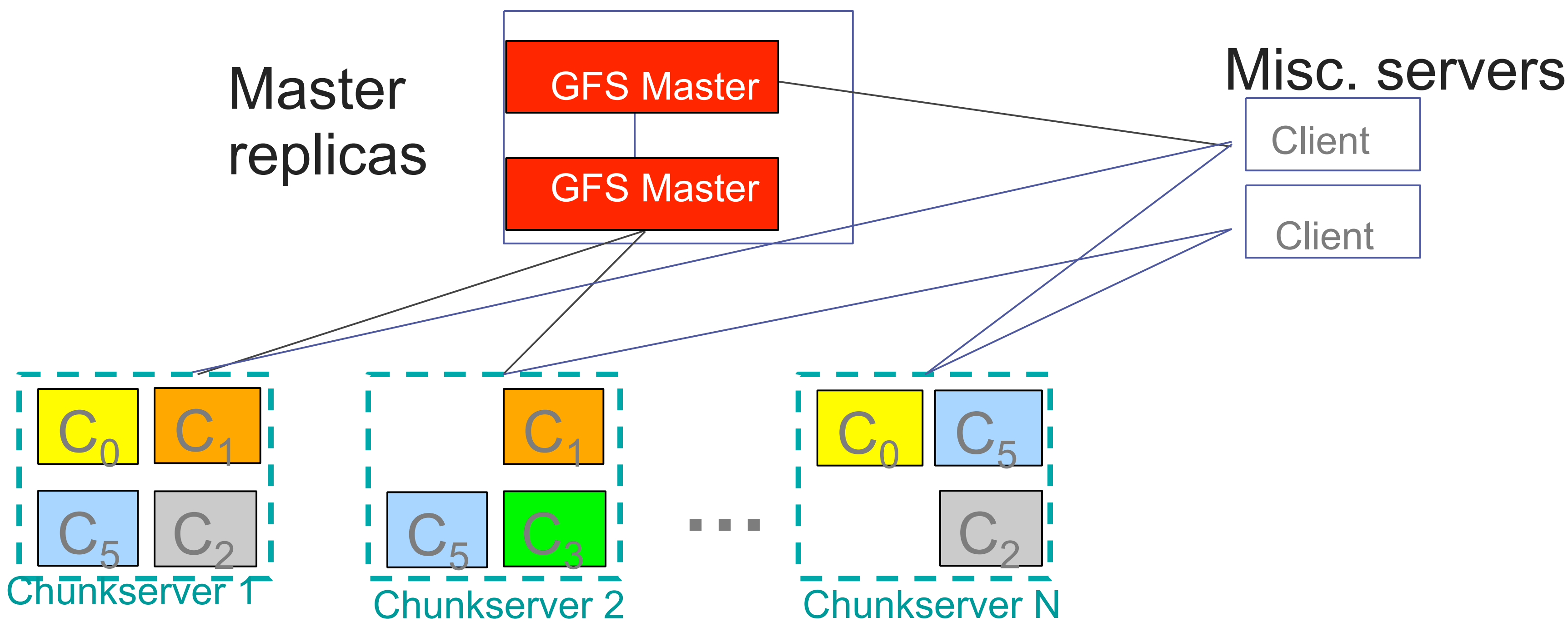
The Building Blocks of DIC at Google

- Distributed file systems: GFS
- Distributed storage: BigTable
- Job scheduler: the workqueue
- Parallel computation: MapReduce
- Distributed lock server: chubby

GFS: The Google File System

- Reliable distributed storage system for petabyte scale filesystems.
- Data kept in 64-megabyte "chunks" stored on disks spread across thousands of machines
- Each chunk replicated, usually 3 times, on different machines so that GFS can recover seamlessly from disk or machine failure.
- A GFS cluster consists of a single *master server*, multiple *chunkservers*, and is accessed by multiple *clients*.

GFS: The Google File System



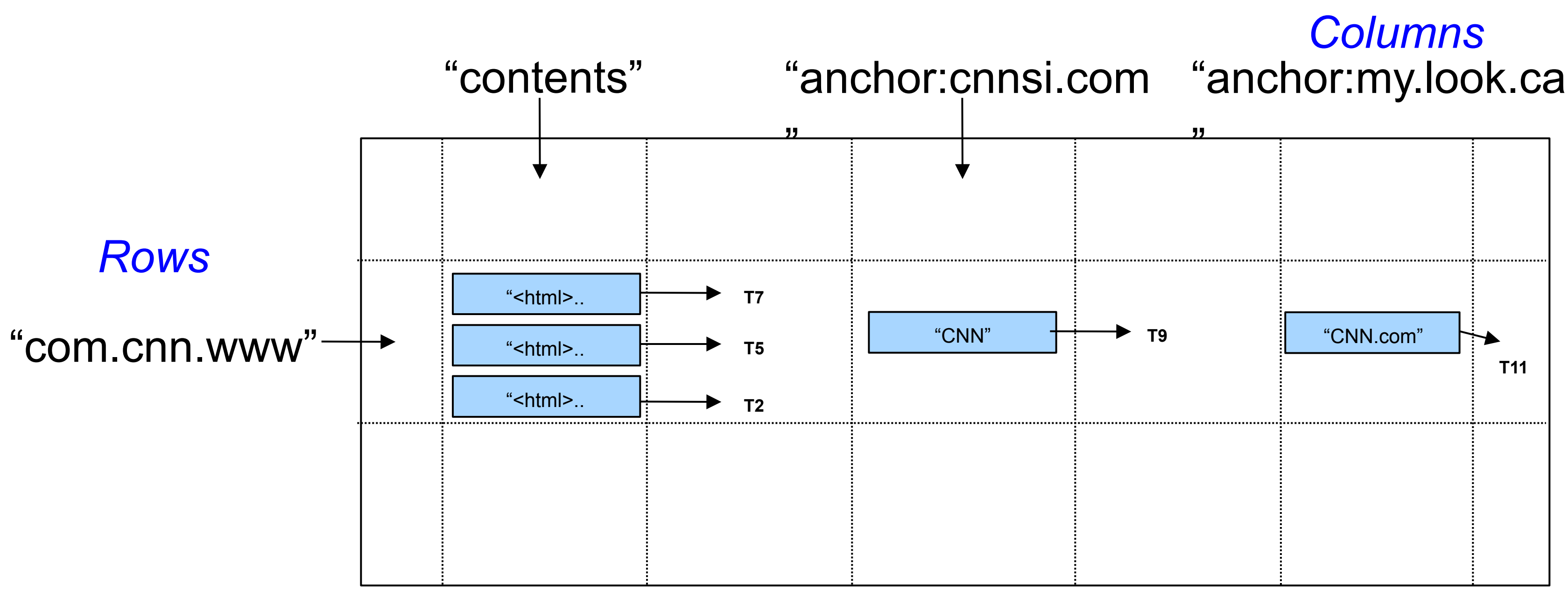
- Master manages metadata
- Data transfers happen directly between clients/chunkservers
- Files broken into chunks (typically 64 MB)
- Chunks triplicated across three machines for safety

BigTable

- A distributed storage system for managing structured data
  - Designed to scale to a very large size: petabytes of data across thousands of commodity servers.
- Built on top of GFS
- Used by more than 60 Google products and projects
  - Google Earth, Google Finance, Orkut, …

Basic Data Model

- Triple (row, column, timestamp) -> keys for lookup, insert, and delete API
- Arbitrary “columns” on a row-by-row basis
  - Column “family:qualifier”: Family is heavyweight, qualifier lightweight
  - Column-oriented physical store: rows are sparse!



Rows

- Name is an arbitrary string.
  - Access to data in a row is atomic.
  - Row creation is implicit upon storing data.
  - Transactions within a row
- Rows ordered lexicographically
  - Rows close together lexicographically usually on one or a small number of machines.
- Does not support relational model
  - No table wide integrity constants
  - No multirow transactions

MapReduce

- A parallel programming model and an associated implementation for processing and generating large data sets.
- A user specified **map** function processes a key/value pair to generate a set of intermediate key/value pairs.
- A user specified **reduce** function merges all intermediate values associated with the same intermediate key.
- Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines.

Motivation

- Large-Scale Data Processing
  - Want to use 1000s of CPUs
    - But don’t want hassle of ***managing*** things
- MapReduce runtime provides
  - Automatic parallelization & distribution
  - Fault tolerance
  - I/O scheduling
  - Monitoring & status updates

Map/Reduce

- Map/Reduce
  - Programming model from Lisp
  - (and other functional languages)
- Many problems can be phrased this way
- Easy to distribute across nodes
- Failure/retry semantics

Map in Lisp (Scheme)

Unary operator

- (map ***f list [list$_2$ list$_3$ ...]***)

Binary operator

- (map square ‘(1 2 3 4))
  – (1 4 9 16)
- (reduce + ‘(1 4 9 16))
  – 30
- (reduce + (map square (map – l$_1$ l$_2$))))

Map/Reduce at Google

- map(key, val) is run on each item in set
  - emits new-key / new-val pairs
- reduce(key, vals) is run for each unique key emitted by map()
  - emits final output

count words in docs

– Input consists of (url, contents) pairs

– map(key=url, val=contents):
  - For each word *w* in contents, emit (w, "1")

– reduce(key=word, values=uniq_counts):
  - Sum all "1"s in values list
  - Emit result "(word, sum)"

Count, Illustrated

map(key=url, val=contents):

For each word *w* in contents, emit (w, “1”)

reduce(key=word, values=uniq_counts):

Sum all “1”s in values list

Emit result “(word, sum)”

see bob throw
see spot run

see	1
bob	1
run1	
see	1
spot	1
throw	1

bob	1
Run	1
see	2
spot	1
throw	1

Grep

- Input consists of (url+offset, single line)
- map(key=url+offset, val=line):
  - If contents matches regexp, emit (line, “1”)

- reduce(key=line, values=uniq_counts):
  - Don’t do anything; just emit line

Reverse Web-Link Graph

- Map
  - For each URL linking to target, …
  - Output <target, source> pairs
- Reduce
  - Concatenate list of all source URLs
  - Outputs: <target, ***list*** (source)> pairs

Implementation Overview

Typical cluster:

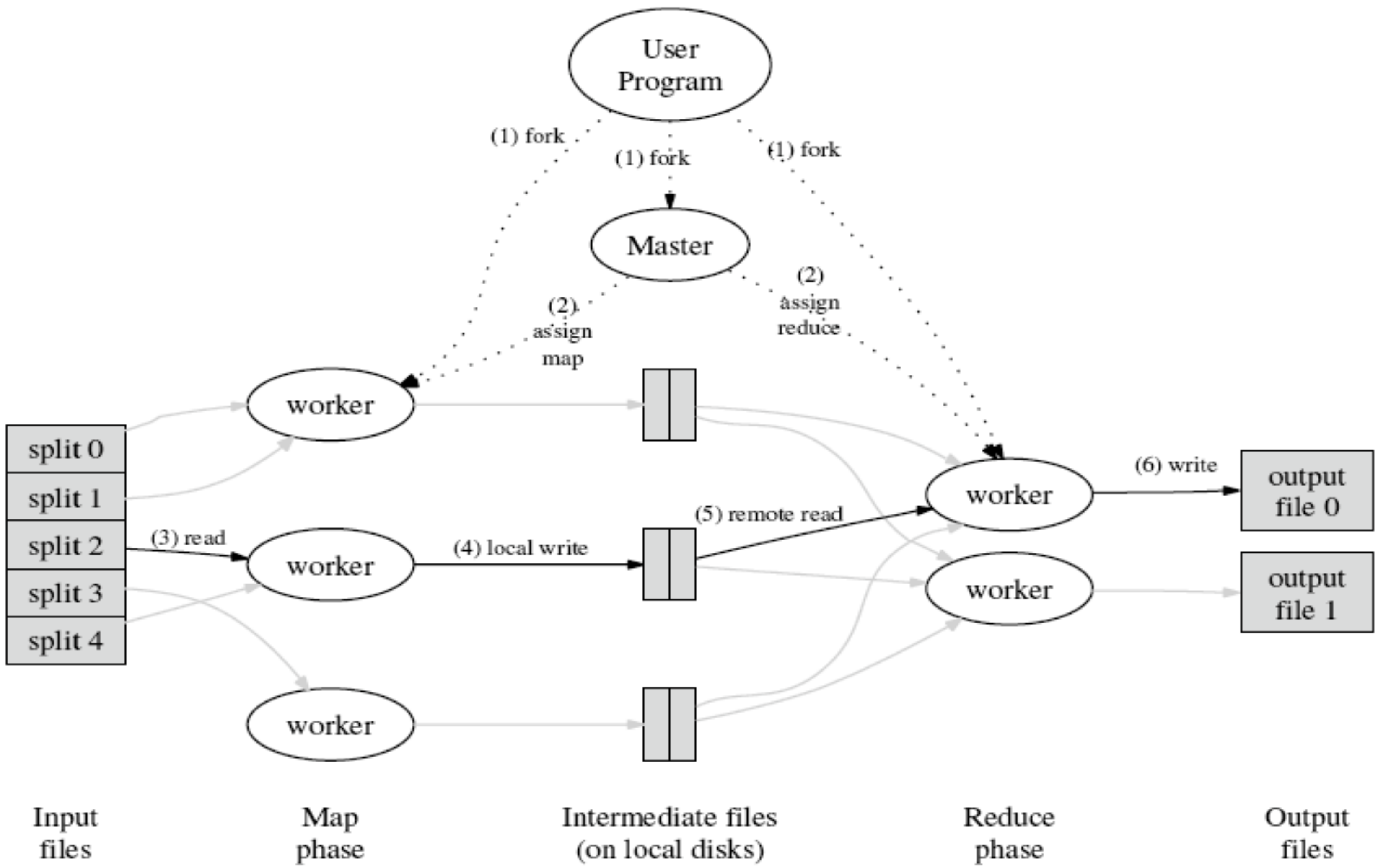
- 100s/1000s of 2-CPU x86 machines, 2-4 GB of memory
- Limited bisection bandwidth
- Storage is on local IDE disks
- GFS: distributed file system manages data
- Job scheduling system: jobs made up of tasks, scheduler assigns tasks to machines

Implementation is a C++ library linked into user programs

MapReduce Runtime System

- How is this distributed?
  - Partition input key/value pairs into chunks, run map() tasks in parallel
  - After all map()s are complete, consolidate all emitted values for each unique emitted key
  - Partition space of output map keys, and run reduce() in parallel
- If map() or reduce() fails, reexecute!

Distributed Execution



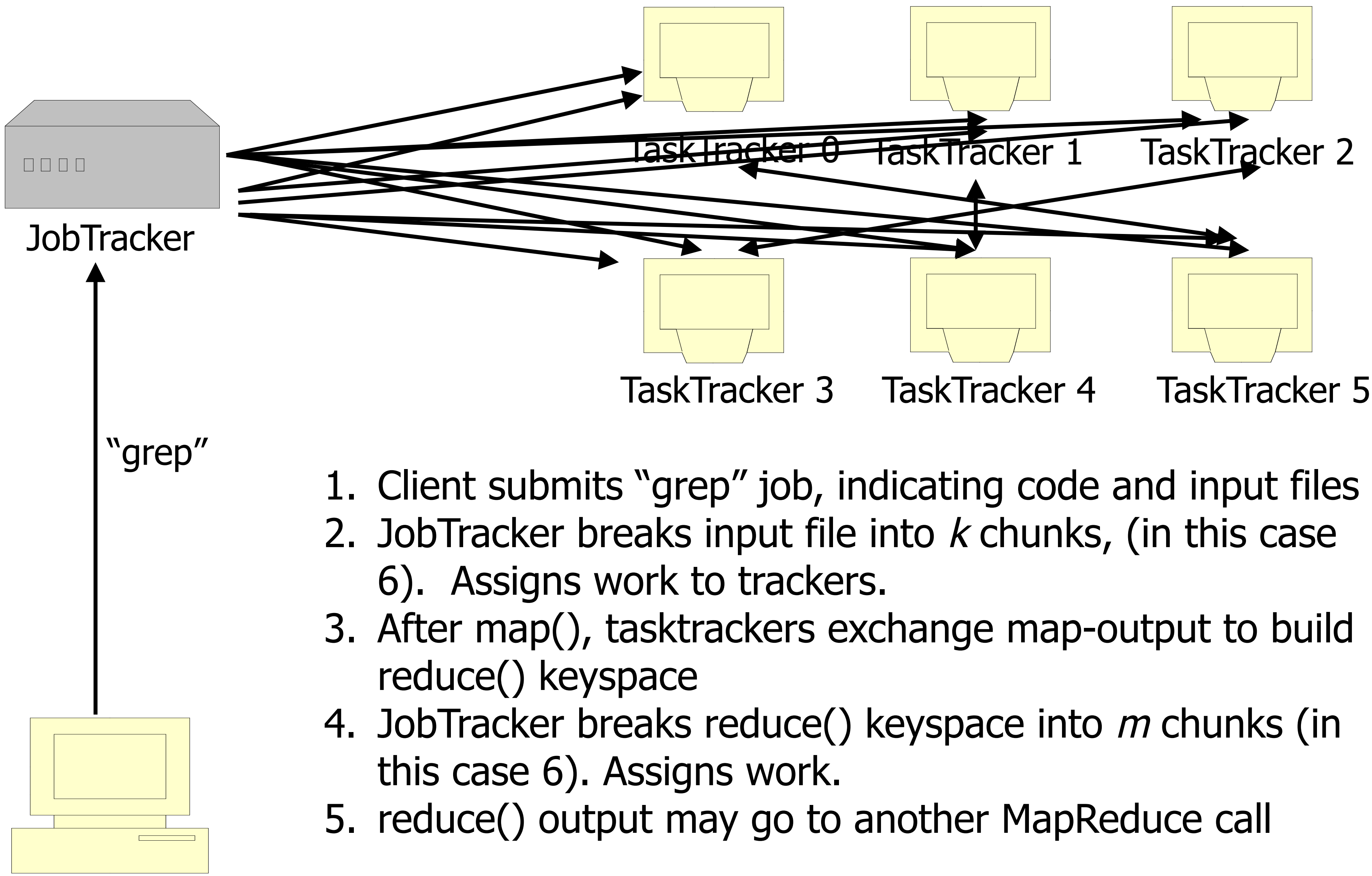
Example: Count word occurrences

```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
    EmitIntermediate(w, "1");

reduce(String output_key, Iterator
  intermediate_values):
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
 Emit(AsString(result));
```

Example vs. Actual Source Code

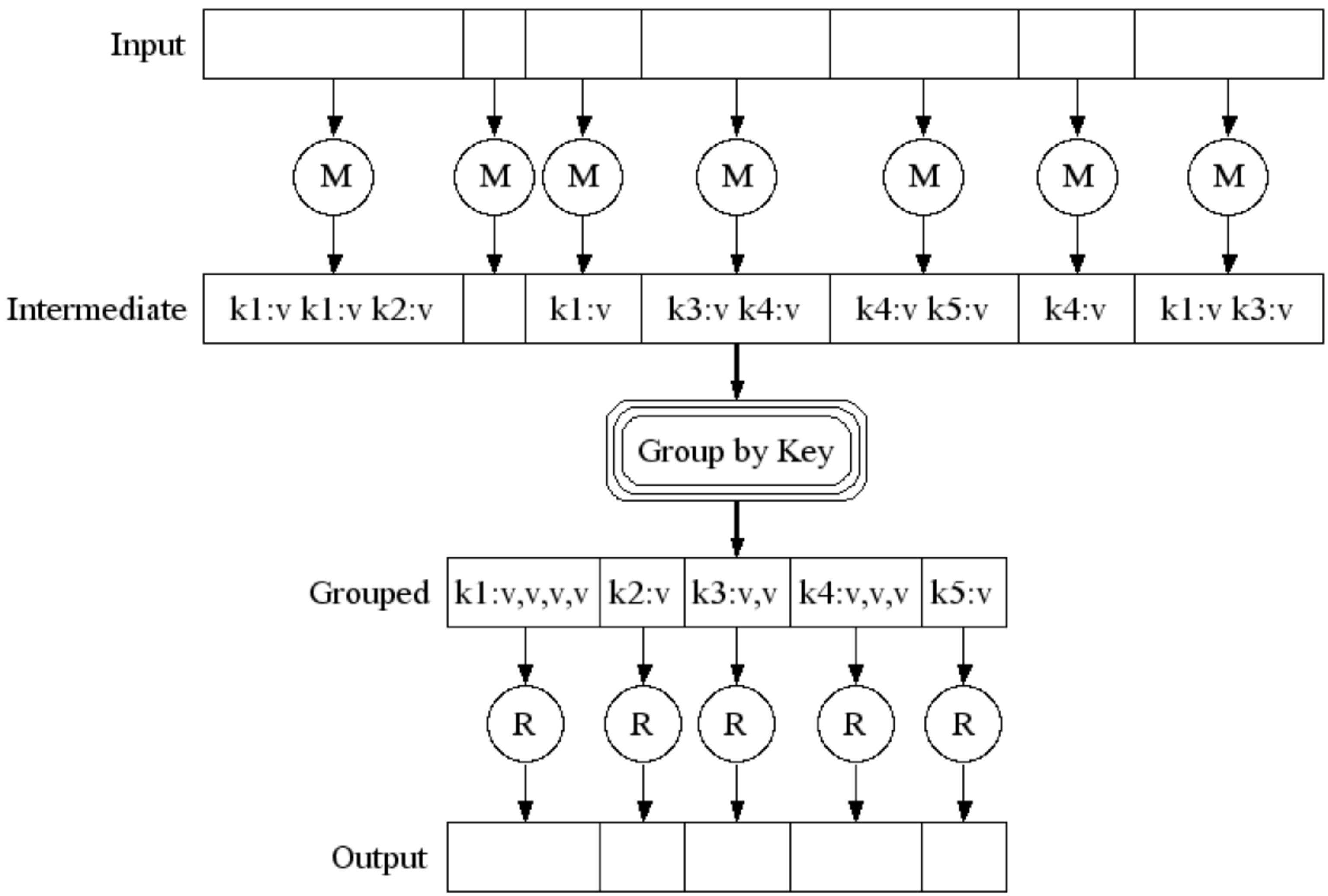
- Example is written in pseudo-code
- Actual implementation is in C++, using a MapReduce library
- Bindings for Python and Java exist via interfaces
- True code is somewhat more involved (defines how the input key/values are divided up and accessed, etc.)

Job Processing



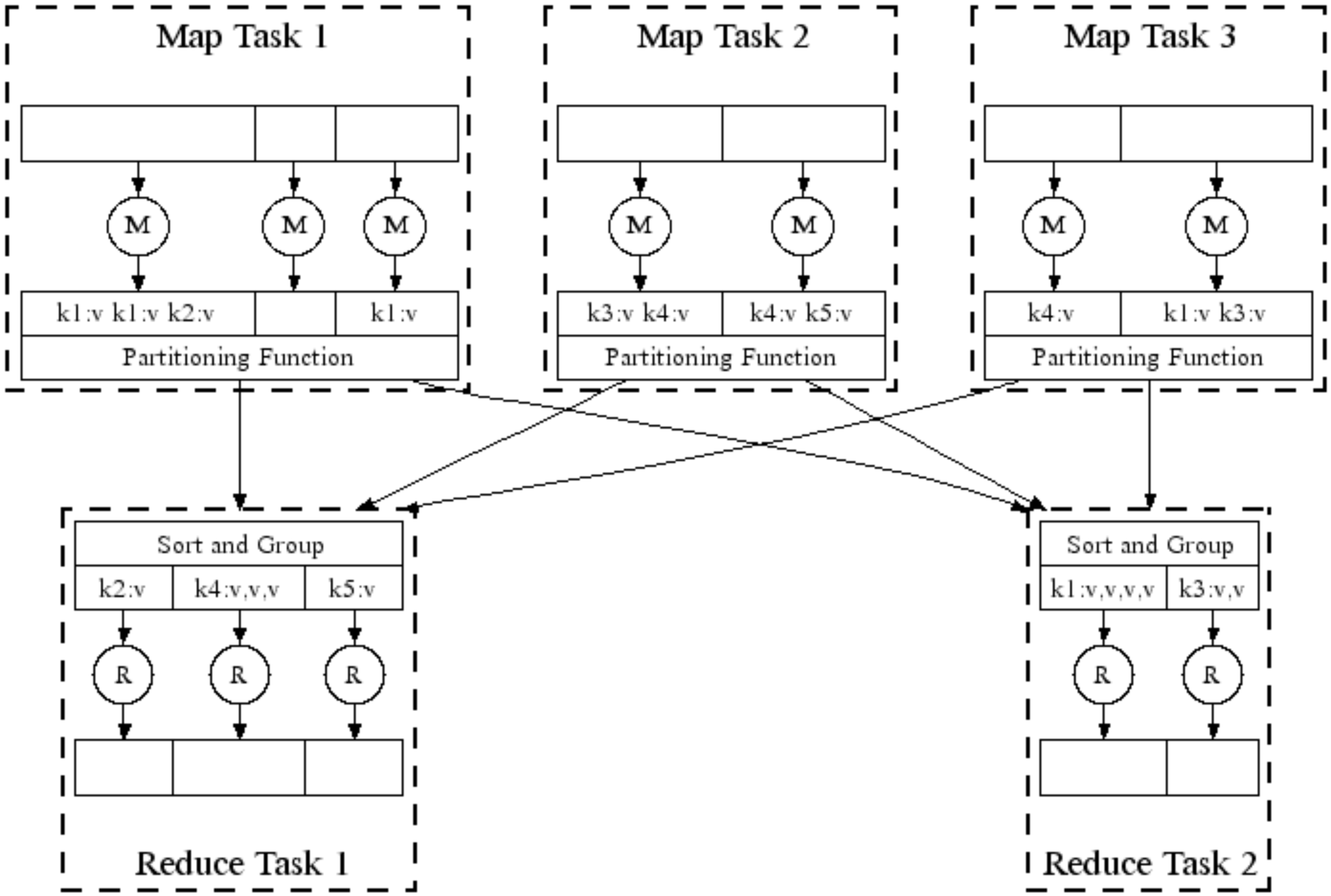
1. Client submits "grep" job, indicating code and input files
2. JobTracker breaks input file into *k* chunks, (in this case 6). Assigns work to trackers.
3. After map(), tasktrackers exchange map-output to build reduce() keyspace
4. JobTracker breaks reduce() keyspace into *m* chunks (in this case 6). Assigns work.
5. reduce() output may go to another MapReduce call

Execution



Input
M
M
M
M
M
M
M
Intermediate
k1:v k1:v k2:v
k1:v
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Group by Key
Grouped
k1:v,v,v,v
k2:v
k3:v,v
k4:v,v,v
k5:v
R
R
R
R
R
Output

Parallel Execution



Fault Tolerance

Handled via re-execution

- Detect failure via periodic heartbeats
- Re-execute completed + in-progress *map* tasks (why?)
- Re-execute in progress *reduce* tasks (why?)
- Task completion committed through master

Robust: lost 1600/1800 machines once → finished ok

Refinement: Redundant Execution

Slow workers significantly delay completion time

- Other jobs consuming resources on machine
- Bad disks w/ soft errors transfer data slowly
- Weird things: processor caches disabled (!!)

Solution: Near end of phase, spawn backup tasks

- Whichever one finishes first "wins"

Dramatically shortens job completion time

Refinement: Locality Optimization

- Master scheduling policy
  - Ask GFS for locations of replicas of input file blocks
  - Map tasks typically split into 64MB (GFS block size)
  - Map tasks scheduled so GFS input block replica are on same machine or same rack
- Effect
  - Thousands of machines read input at local disk speed
    - Without this, rack switches limit read rate

Refinement
Skipping Bad Records

- Map/Reduce functions sometimes fail for particular inputs
  - Best solution is to debug & fix
    - Not always possible ~ third-party source libraries
  - On segmentation fault:
    - Send UDP packet to master from signal handler
    - Include sequence number of record being processed
  - If master sees two failures for same record:
    - Next worker is told to skip the record

Performance

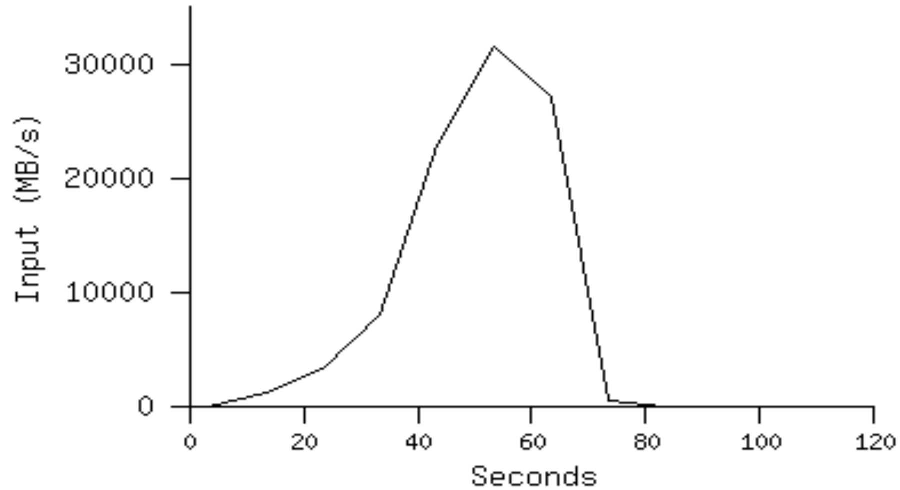
Tests run on cluster of 1800 machines:

- 4 GB of memory
- Dual-processor 2 GHz Xeons with Hyperthreading
- Dual 160 GB IDE disks
- Gigabit Ethernet per machine
- Bisection bandwidth approximately 100 Gbps

Two benchmarks:

MR_GrepScan	1010 100-byte records to extract records matching a rare pattern (92K matching records)
MR_SortSort	1010 100-byte records (modeled after TeraSort benchmark)

MR_Grep

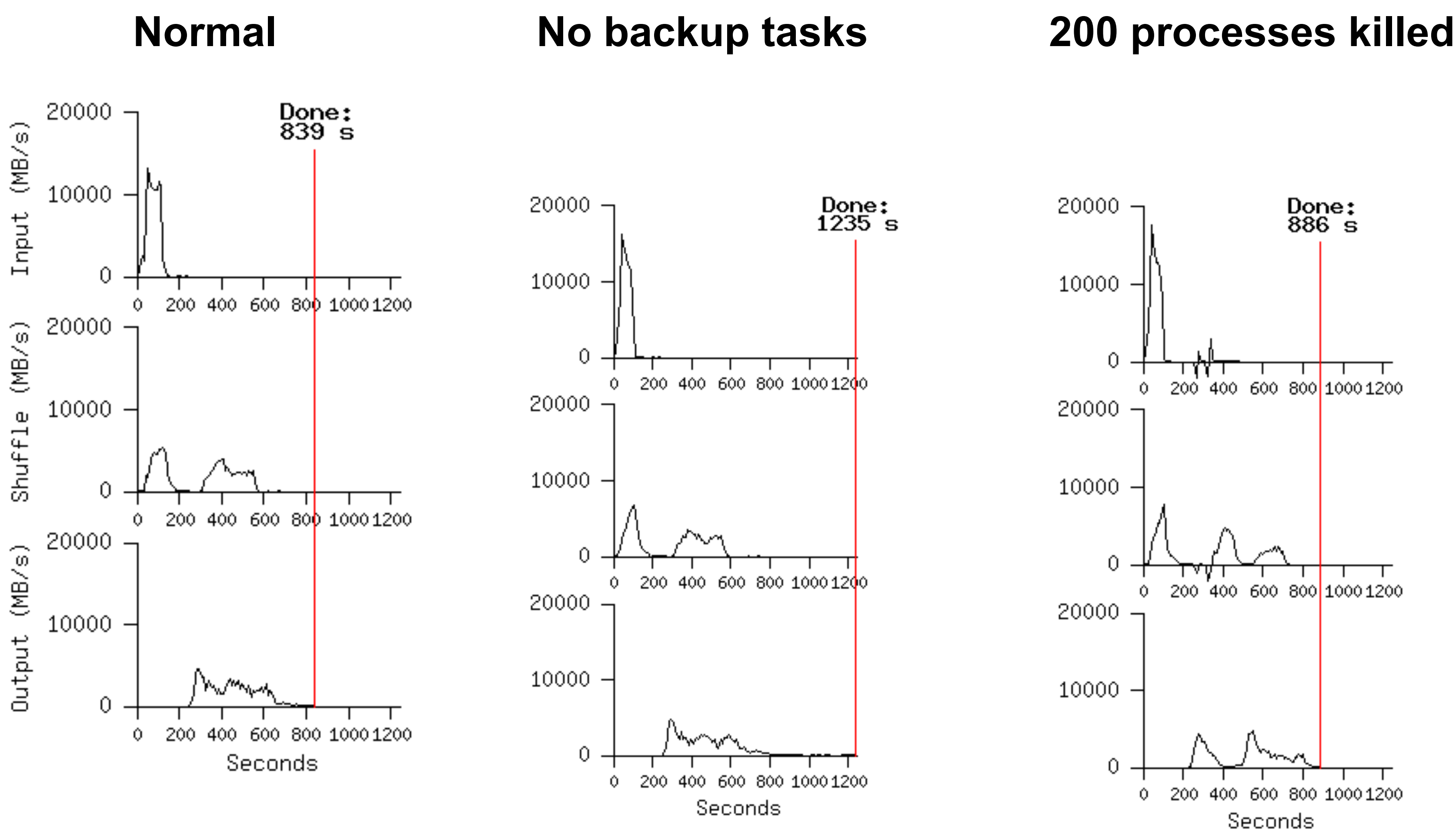


Locality optimization helps:

- 1800 machines read 1 TB at peak ~31GB/s
- W/out this, rack switches would limit to 10 GB/s

Startup overhead is significant for short jobs

MR_Sort

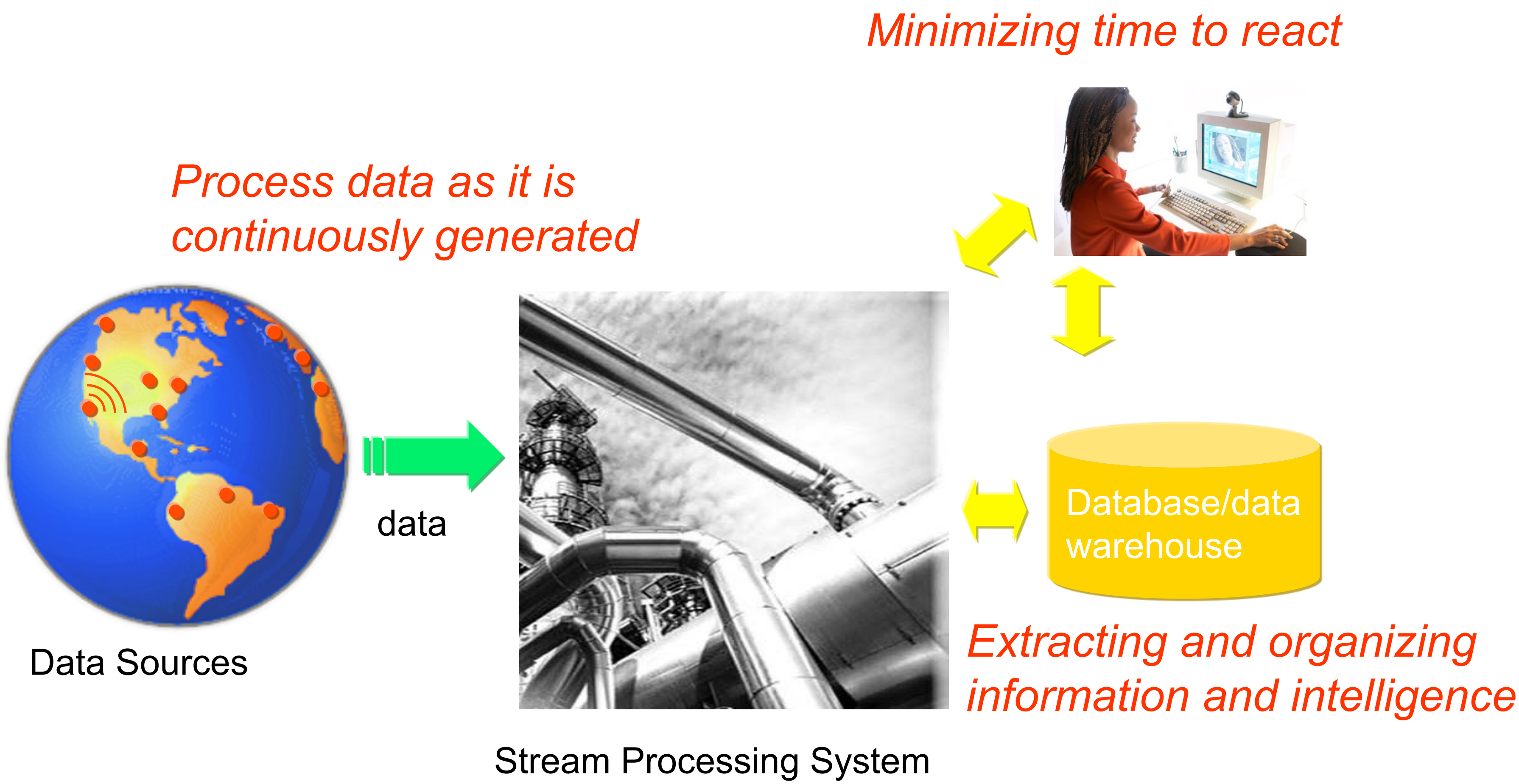


- Backup tasks reduce job completion time a lot!
- System deals well with failures

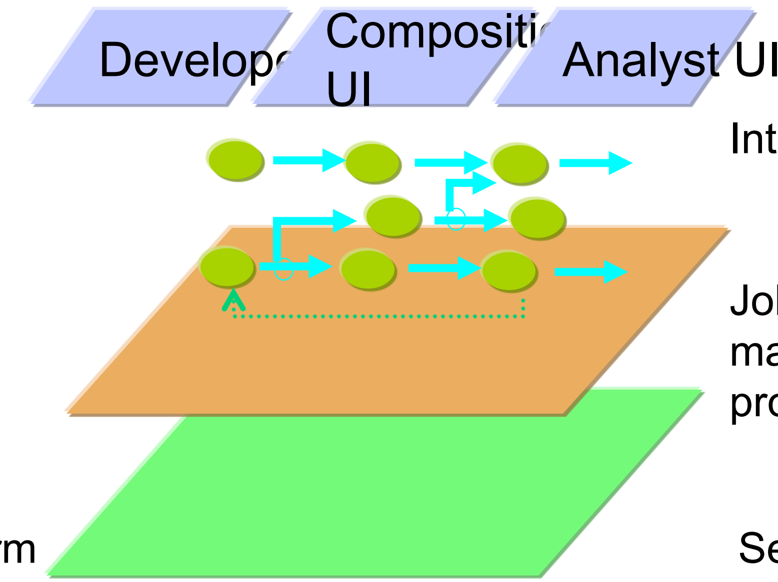
MapReduce Summary

- MapReduce has proven to be a useful distributed programming abstraction
- Greatly simplifies large-scale data-intensive computing
- Functional programming paradigm can be applied to many data analysis applications
- Fun to use: focus on problem, let library deal with messy details

What is Stream Processing?



What Makes a Stream Processing System?



Tooling

Application — Interconnection of operators

Runtime Environment — Job management, resource management, content routing, programming model, object store

Hardware Platform — Servers, networks, storage, operating system, file system

Stream Processing System

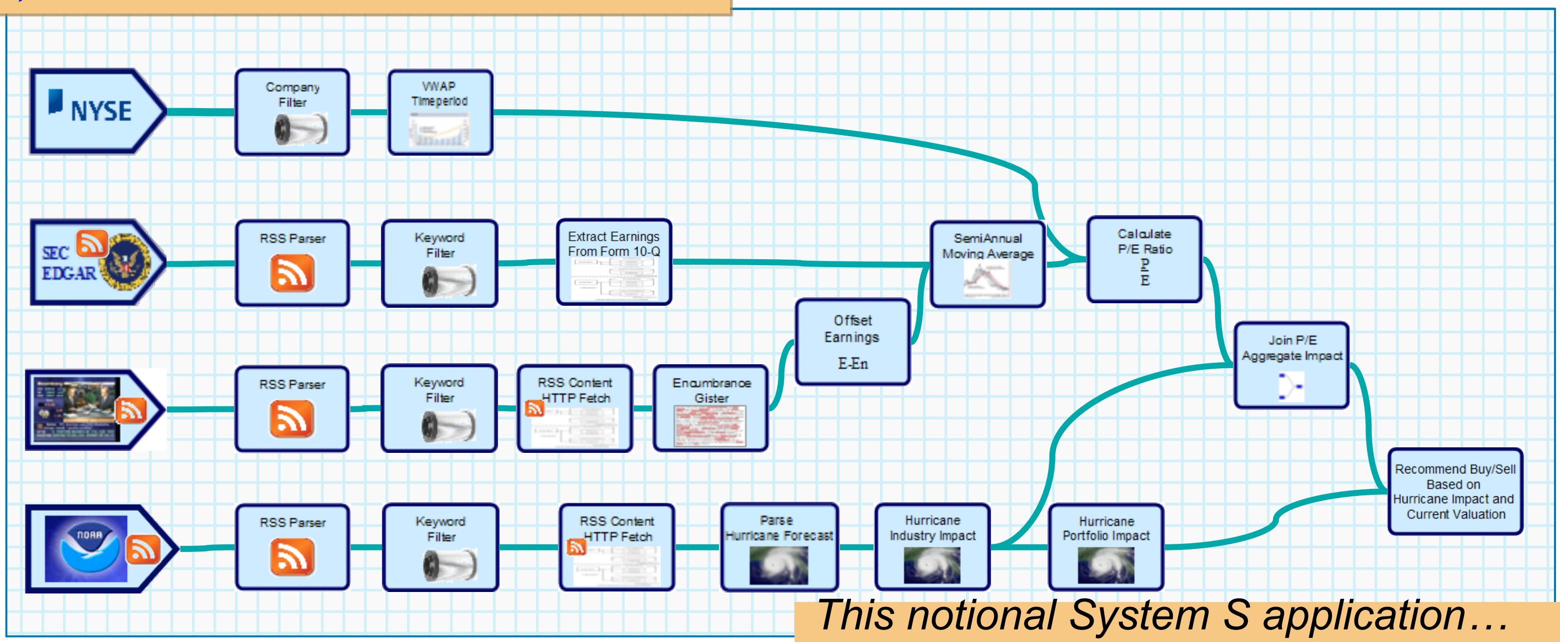
System S Stream Processing

- New *stream* computing paradigm
- Pull information from anywhere in real time
- Ultra-low latency, ultra-high throughput
- Scalable

System S: A Closer Look

System S continually adapts to new inputs, new modalities

Analytics may be a combination of provided and user-developed/legacy operators



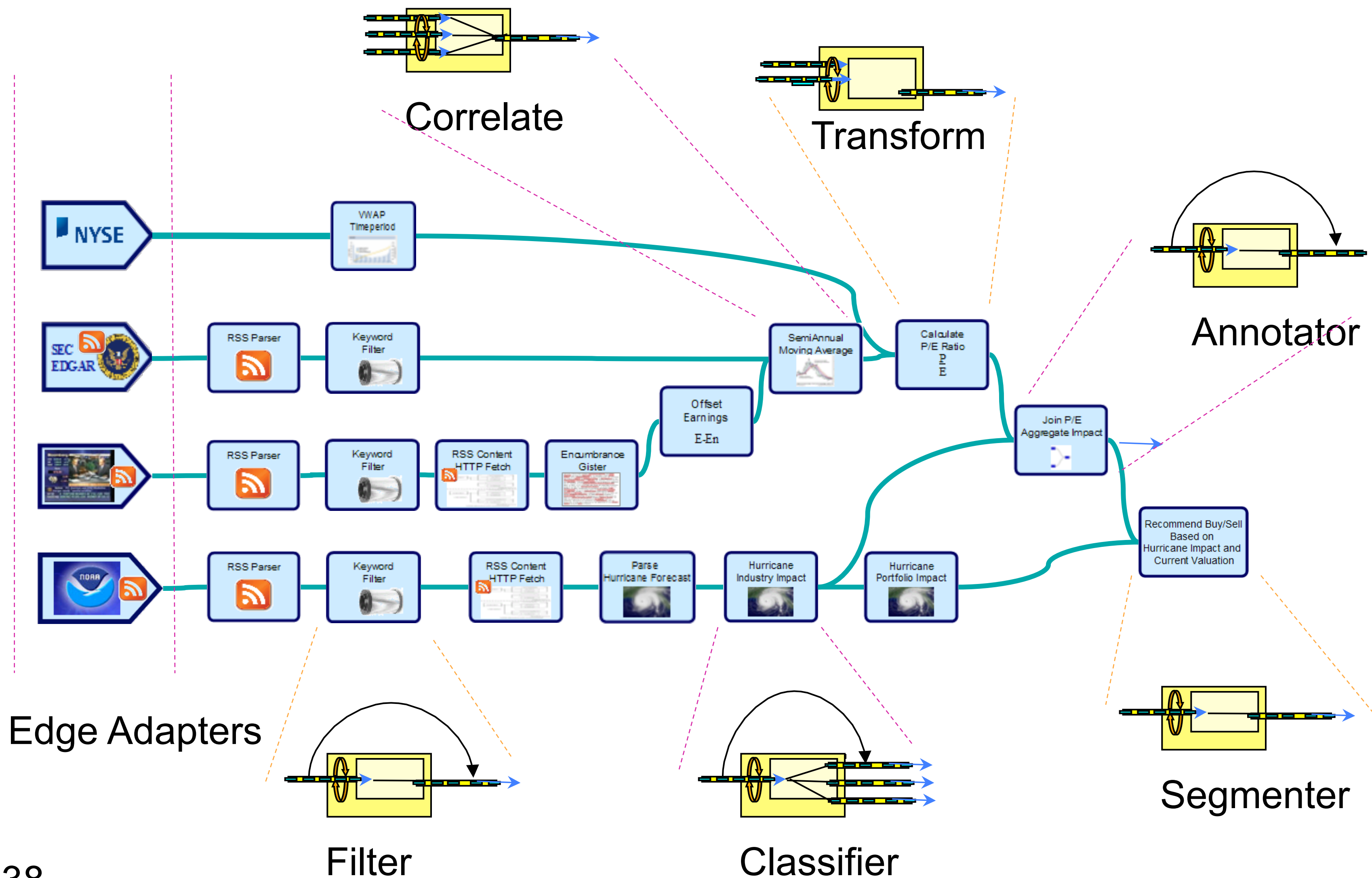
This notional System S application…

- *Calculates VWAP*
- *Calculates P/E, based earnings from Edgar*
- *Refines earnings based on encumbrances identified in newsfeeds*

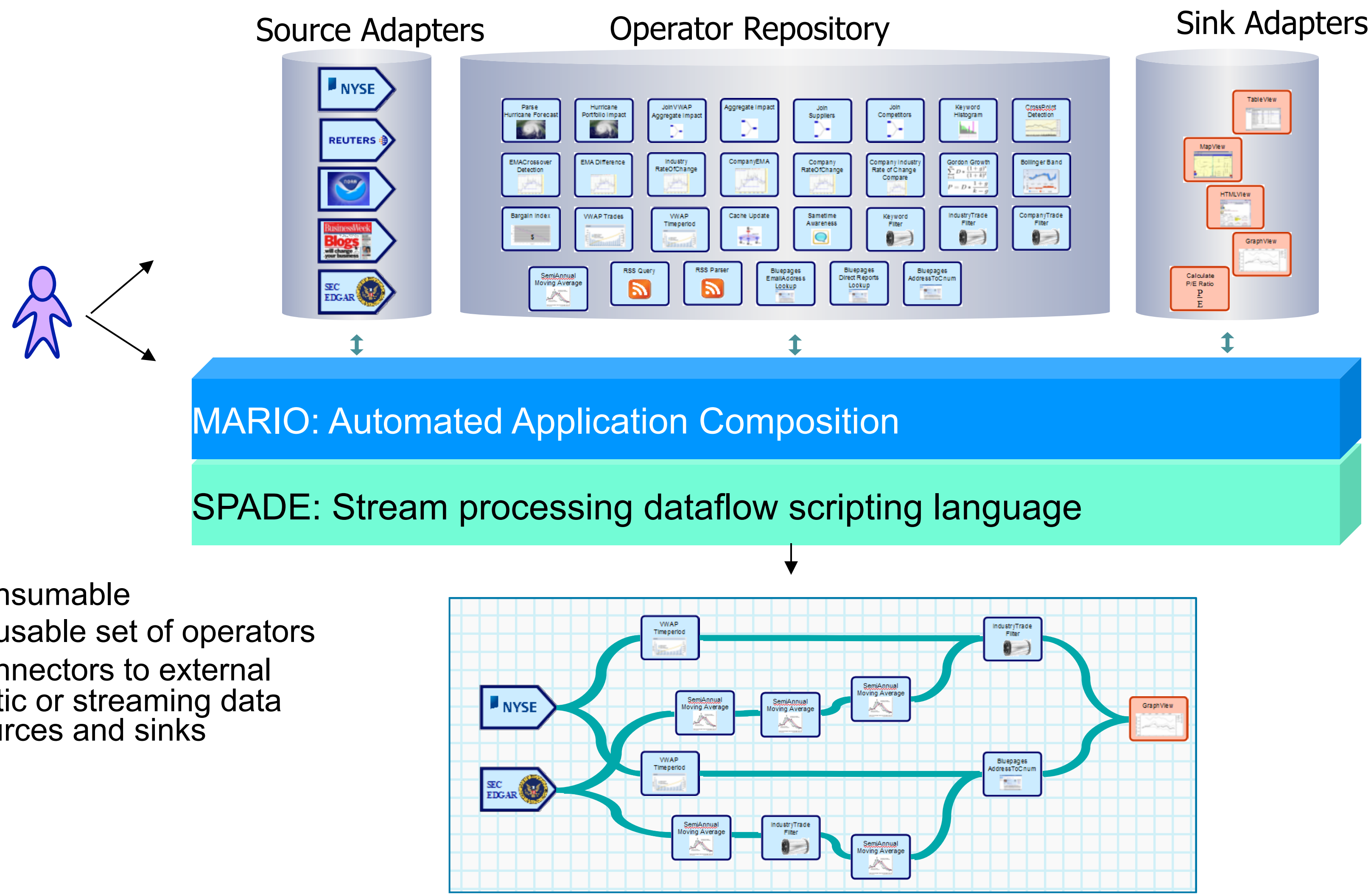
System S applications can seamlessly process structured (event) and unstructured data

SPADE Building Blocks

Classifiers, Annotators, Correlators, Filters, Aggregators



Application Programming



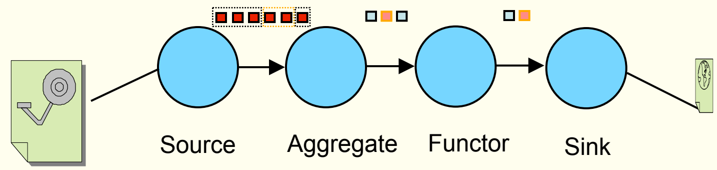
Platform Optimized Compilation

- Consumable
- Reusable set of operators
- Connectors to external static or streaming data sources and sinks

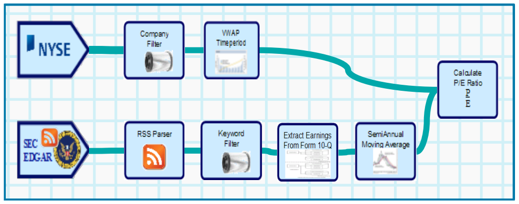
SPADE

- SPADE (*Stream Processing Application Declarative Engine*) is an intermediate language for streaming applications.
  - Simplifies design of applications used by System S
  - Hides complexities of
    - manipulating data streams (e.g., contains generic language support for data types and building block operations)
    - fanning out applications to distributed heterogeneous nodes
    - transporting data through diverse computer infrastructures (ingesting external data, routing intermediate results, looping in feedback, branching, outputing the results, ...)

```
[Application]
SourceSink trace

[Typedefs]
typespace sourcesink

typedef id_t Integer
typedef timestamp_t Long

[Program]
// virtual schema declaration
vstream Sensor (id : id_t, location : Double, light : Float, temperature : Float, timestamp :
timestamp_t)

// a source stream is generated by a Source operator – in this case tuples come from an input file
stream SenSource ( schemaFor(Sensor) )
        := Source( ) [ "file:///SenSource.dat" ] {}

// this intermediate stream is produced by an Aggregate operator, using the SenSource stream as
input
stream SenAggregator ( schemaFor(Sensor) )
        := Aggregate( SenSource <count(100),count(1)> ) [ id . location ]
                    { Any(id), Any(location), Max(light), Min(temperature), Avg(timestamp) }

// this intermediate stream is produced by a functor operator
stream SenFunctor ( id: Integer, location: Double, message: String )
        := Functor( SenAggregator ) [ log(temperature,2.0)>6.0 ]
                  { id, location, "Node "+toString(id)+ " at location "+toString(location) }

// result management is done by a sink operator – in this case produced tuples are sent to a socket
Null := Sink( SenFunctor ) [ "cudp://192.168.0.144:5500/" ] {}
```

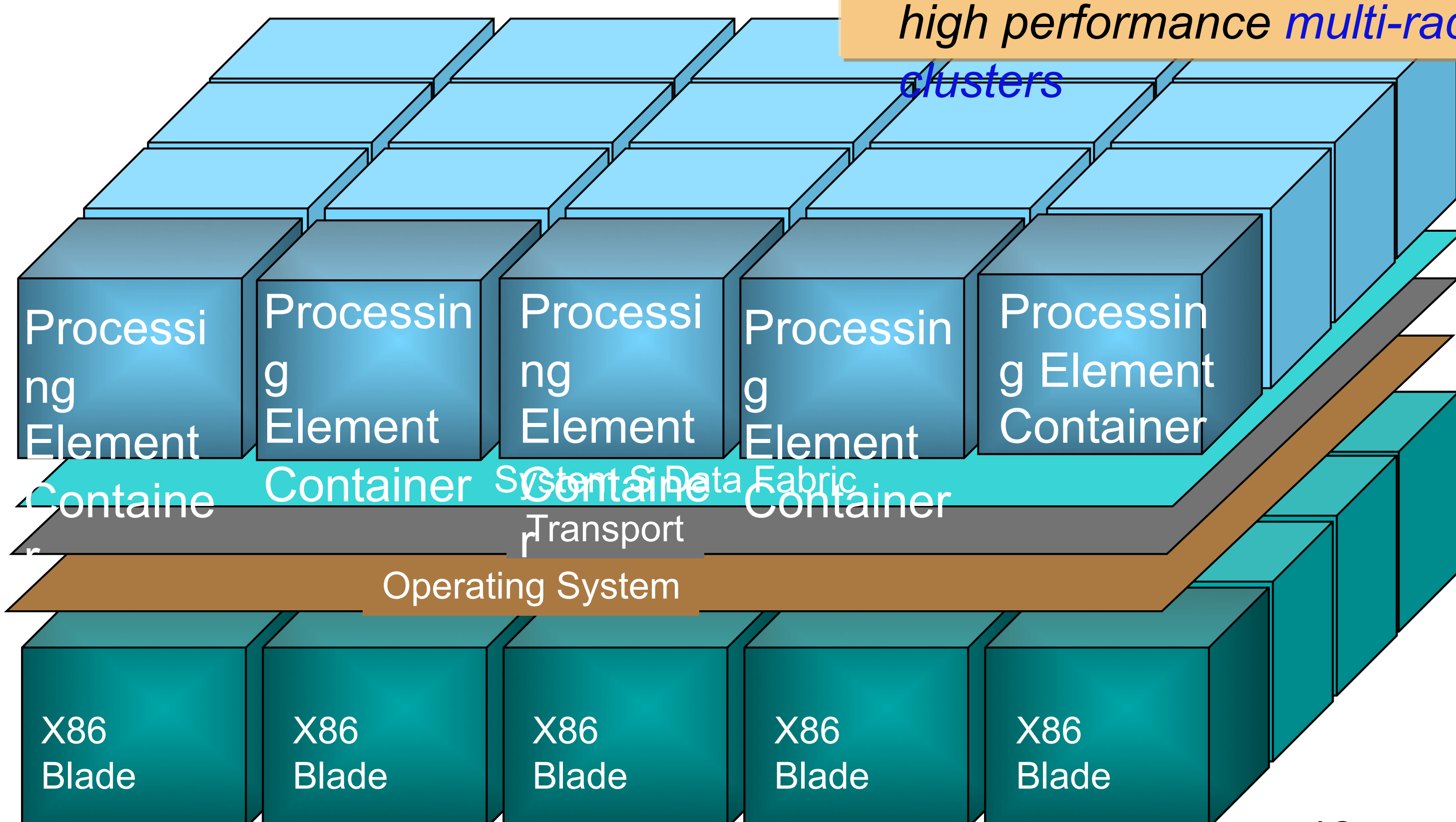


System S Runtime Services



Optimizing scheduler assigns operators to processing nodes, and continually manages resource allocation

Runs on commodity hardware – from single node to blade centers to high performance multi-rack clusters

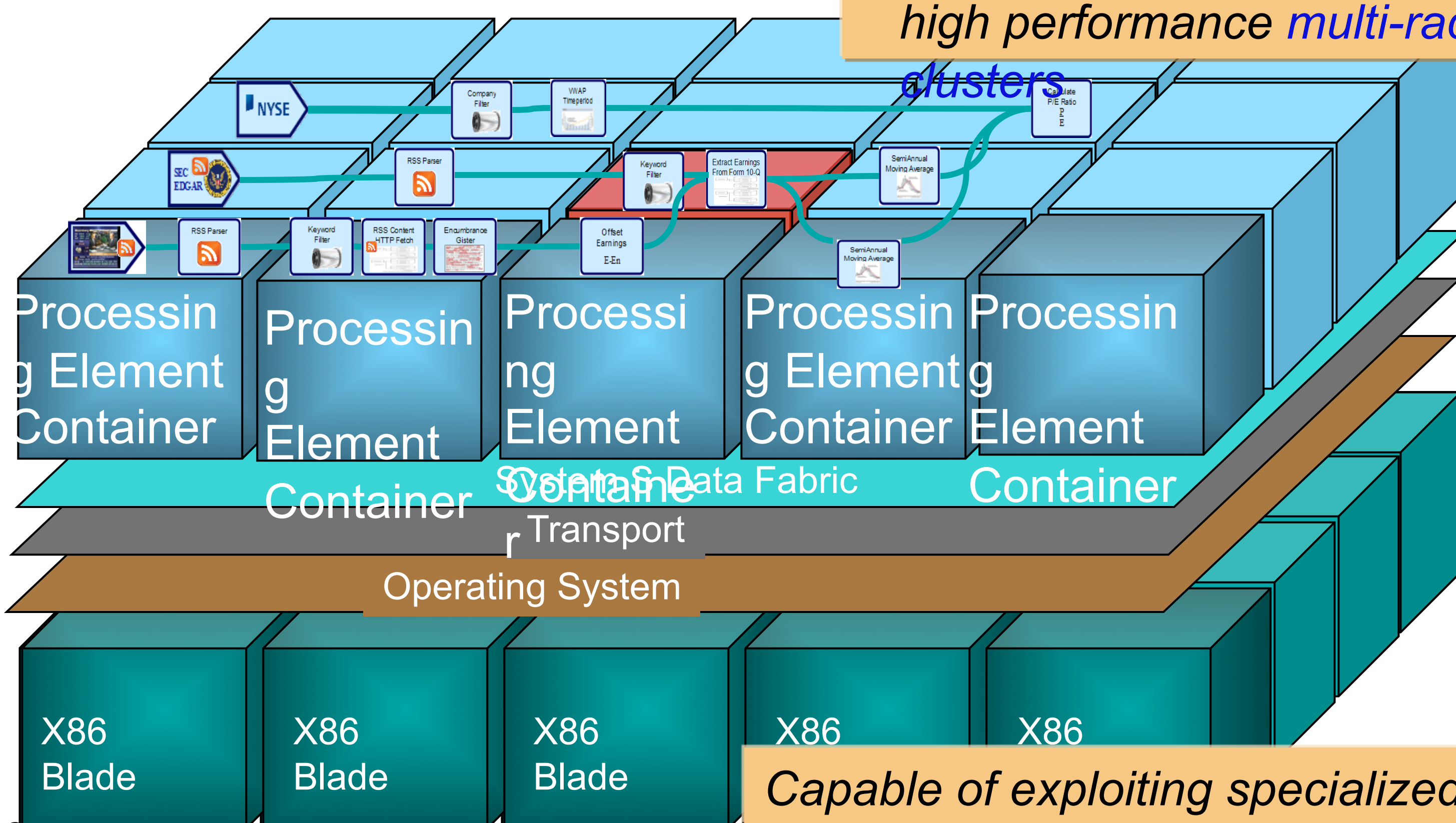


System S Runtime Services

Optimizing scheduler assigns operators to processing nodes, and continually manages resource allocation

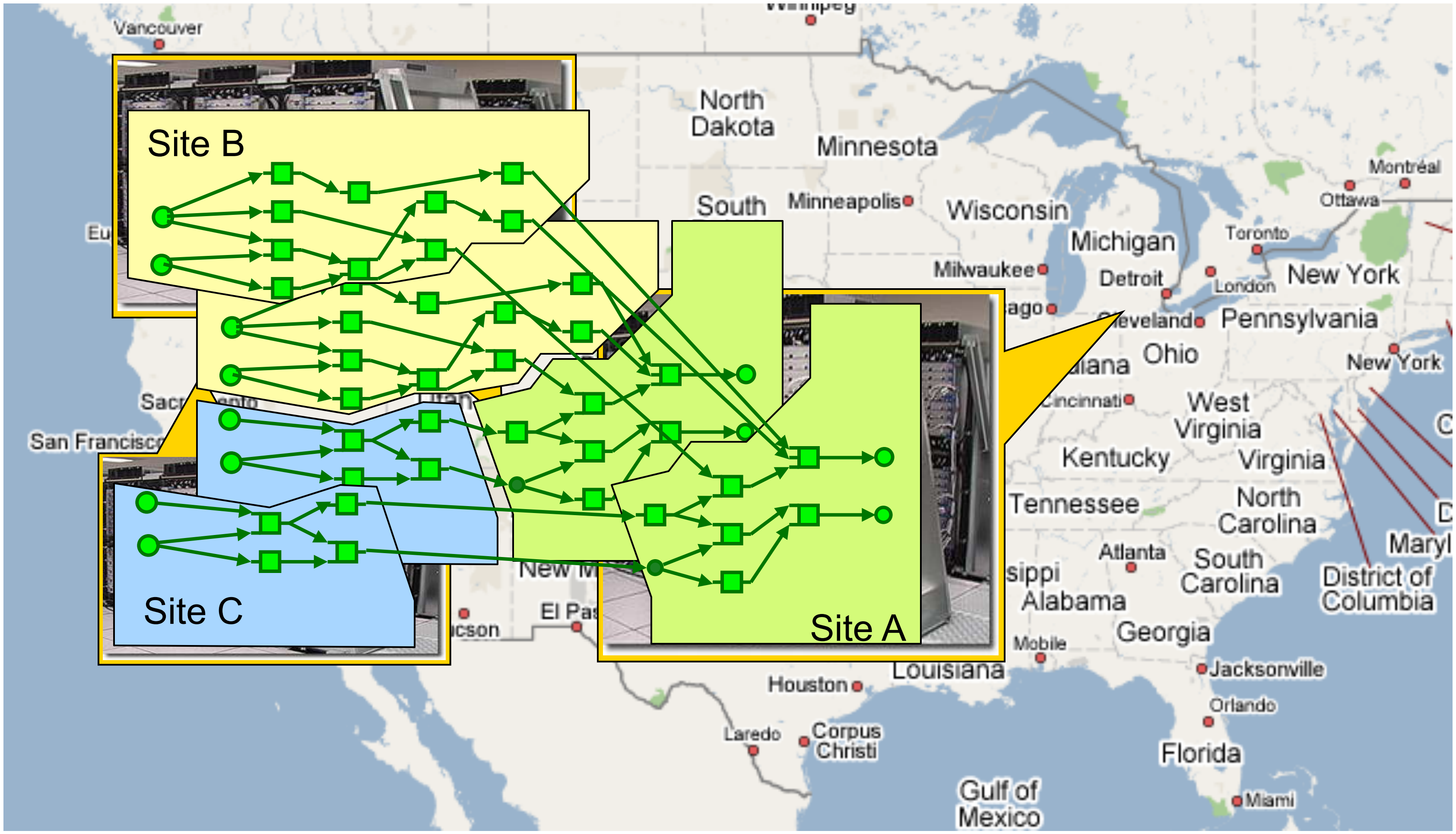
Adapts to changes in resources, workload, data rates

Runs on commodity hardware – from single node to blade centers to high performance multi-rack clusters



Capable of exploiting specialized hardware

Distributed operation



Summary

Simplified Processing Flow Graph