Dataflow Programming: a scalable data-centric approach to parallelism

EMBEDDABLE DATA MANAGEMENT AND AGILE INTEGRATION

PERVASIVE

Agenda

- Background
- Dataflow Overview
  - Introduction
  - Design patterns
  - Dataflow and actors
- DataRush Introduction
  - Composition and execution models
  - Benchmarks

Background

- Work on DataRush platform
  - Dataflow based engine
  - Scalable, high throughput data processing
  - Focus on data preparation and deep analytics
- Pervasive Software
  - Mature software company focused on embedded data management and integration
  - Located in Austin, TX
  - Thousands of customers worldwide

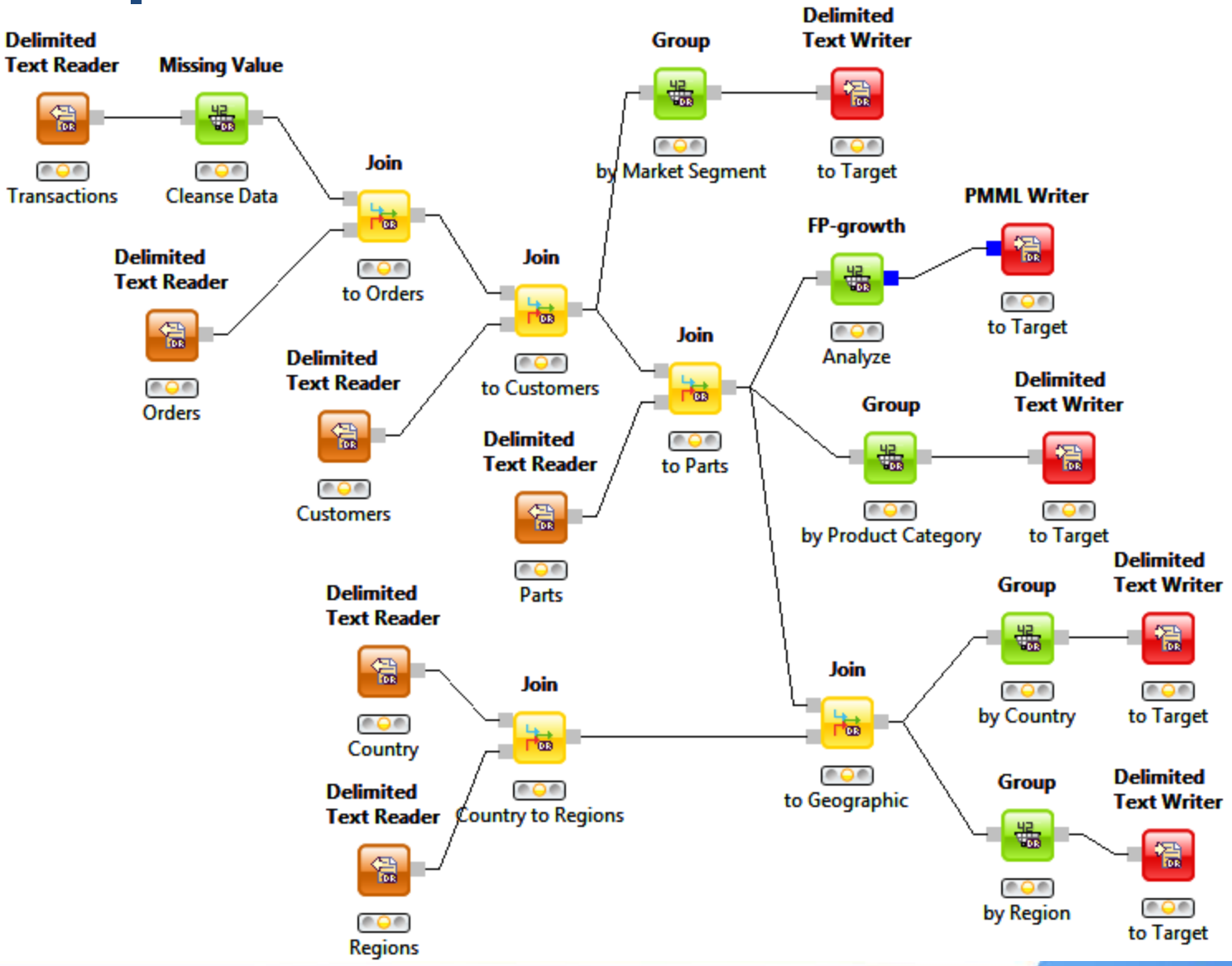
H/W support for parallelism

- Instruction level
- Multicore (process, thread)
- Multicore + I/O (compute and data)
- Virtualization (concurrency)
- Multi-node (clusters)
- Massively multi-node (datacenter as a computer)

Dataflow is

- Based on operators that provide a specific function (nodes)
- Data queues (edges) connecting operators
- Composition of directed, acyclic graphs (DAG)
  - Operators connected via queues
  - A graph instance represents a “program” or “application”
- Flow control
- Scheduling to prevent dead locks
- Focused on data parallelism

Example



Dataflow goodness

- Concepts are easy to grasp
- Abstracts parallelism details
- Simple to express
  - Composition based
- Shared nothing, message passing
  - Simplified programming model
- Immutability of flows
- Limits side effects
- Functional style

Dataflow and big data

- Pipelining
  - Pipeline task based parallelism
  - Overlap I/O and computation
  - Can help optimize processor cache
  - Whole application approach
- Data scalable
  - Virtually unlimited data size capacity
  - Supports iterative data access
- Exploits multicore
  - Scalable
  - High data throughput
- Extendible to multi-node

Parallel design patterns

- Embarrassingly parallel
- Replicable
- Pipeline
- Divide and conquer
- Recursive data

Dataflow and actors

- Actors in the sense of Erlang & Scala
- Commonality
  - Shared nothing architecture
  - Functional style of programming
  - Easy to grasp
  - Easy to extend
  - Semantics fit well with distributed computing
  - Supports either reactor or active models

Dataflow and actors

- Dataflow
  - Flow control
  - Static composition (binding)
  - Data coherency and ordering
  - Deadlock detection/handling
  - Usually strongly typed
  - Great for data parallelism
- Actors
  - Immutability not guaranteed
  - Ordering not guaranteed
  - Not necessarily optimized for large data flows
  - Great for task parallelism

DataRush implementation

- DataRush implements dataflow
  - Based on Kahn process networks
  - Parks algorithm for deadlock detection (with patented modifications)
  - Usable by JVM-based languages (Java, Scala, JPython, JRuby, …)
  - Dataflow engine
  - Extensive standard library of reusable operators
  - API's for composition and execution

DataRush composition

- Application graph
  - High level container (composition context)
  - Add operators using *add()* method
  - Compose using *compile()*
  - Execute using *run()* or *start()*
- Operator
  - Lives during graph composition
  - Composite in nature
  - Linked using flows
- Flows
  - Represent data connections between operators
  - Loosely typed
  - Not live (no data transfer methods)

DataRush composition

```
ApplicationGraph app = GraphFactory.newApplicationGraph();

ReadDelimitedTextProperties rdprops = …

RecordFlow leftFlow = app.add(new ReadDelimitedText("UnitPriceSorted.txt",
    rdprops), "readLeft").getOutput();

RecordFlow rightFlow = app.add(new ReadDelimitedText("UnitSalesSorted.txt",
    rdprops), "readRight").getOutput();

String[] keyNames = { "PRODUCT_ID", "CHANNEL_NAME" };
RecordFlow joinedFlow = app.add(new JoinSortedRows(leftFlow, rightFlow,
    FULL_OUTER, keyNames)).getOutput();

app.add(new WriteDelimitedText(joinedFlow, "output.txt",
    WriteMode.OVERWRITE), "write");

app.run();
```

Create a new graph

Add file reader

Add file reader

Add a join operator

Add a file writer

Synchronously run the graph

Data partitioning

- Partitioners
  - Round robin
  - Hash
  - Event
  - Range
- Un-partitioners
  - Round robin (ordered)
  - Merge (unordered)
- Scenarios
  - Scatter
  - Scatter-gather combined
  - Gather
  - For each (pipeline)

```
ApplicationGraph g = GraphFactory.newApplicationGraph("applyFunction");

GenerateRandomProperties props = new GenerateRandomProperties(22295, 0.25);
ScalarFlow data = g.add(new GenerateRandom(TokenTypeConstant.DOUBLE, 1000000,
    props).getOutput();

ScalarFlow result = partition(g, data, PartitionSchemes.rr(4), new ScalarPipeline() {
    @Override
    public ScalarFlow composePipeline(CompositionContext ctx, ScalarFlow flow,
                    PartitionInstanceInfo partInfo) {
        int partID = partInfo.getPartitionID();
        ScalarFlow output = ctx.add(
              new ReplaceNulls(ctx, flow, 0.0D), "replaceNulls_" + partID).getOutput();
        return ctx.add(
              new AddValue(ctx, output, 3.141D), "addValue_" + partID).getOutput();
    }
});

g.add(new LogRows(result));
g.run();
```

Create a new graph

Generate data

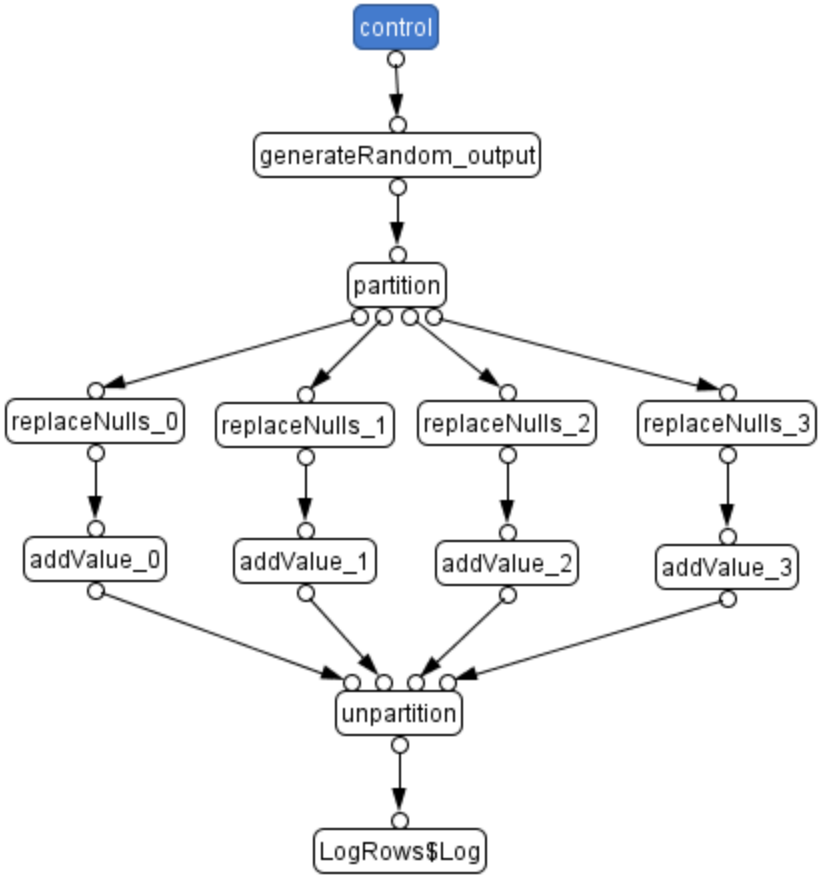
Partition the data using round robin

Compose partitioned pipeline

Each partitions flow will be round robin unpartitioned

Use the results

Partitioning data – resultant graph



DataRush execution

- Process
  - Worker function
  - Executes at runtime
  - Active actor (backed by thread)
- Queues
  - Data transfer channel
  - Single writer, multiple reader
- Ports
  - End points of queues
  - Strongly typed
  - Scalar Java types
  - Record (composite) type

DataRush execution

- No feedback loops
- Data iteration is supported
- Sub-graphs supported (running a graph from a graph)
- Execution Steps
  - Composition invoked
  - Flows are realized as queues
  - Ports exposed on queues to processes
  - Processes are instantiated
  - Threads created for processes and started
  - Deadlock monitoring
  - Stats exposed via JMX and Mbeans
  - Cleanup

Process example

```
public class IsNullProcess extends DataflowProcess {
    private final GenericInput input;
    private final BooleanOutput output;

    public IsNotNull(CompositionContext ctx, RecordFlow input) {
        super(ctx);
        this.input = newInput(input);
        this.output = newBooleanOutput();
    }

    public ScalarFlow getOutput() {
        return getFlow(output);
    }

    public void execute() {
        while (input.stepNext()) {
                output.push(input.isNull());
        }
        output.pushEndOfData();
    }
}
```

Extends DataflowProcess

Declares ports

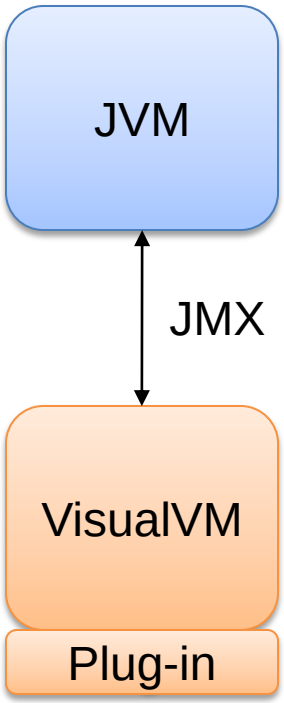
Instantiates ports

Accessor for output port

Execution method:
- Steps input
- Pushes to output
- Closes output

Profiling

- Run-time statistics
  - Collected on graphs, queues and processes
  - Exposed via JMX
  - Serializable for post-execution viewing
- Extending VisualVM
  - Graphical JMX Console ships with the JDK
  - DataRush plug-in
  - Connect to running VM
    - Dynamically view stats
    - Look for hotspots
    - Take snapshots
  - Statically view serialized snapshot



Java VisualVM

File Applications View Tools Window Help

Applications

- Local
  - VisualVM
  - <Unknown Applic
    - app
- Remote
- Snapshots

Start Page | <Unknown Application> (pid 5104)

Overview | Monitor | Threads | Profiler | app - Structure | app - Statistics | app - Hotspots

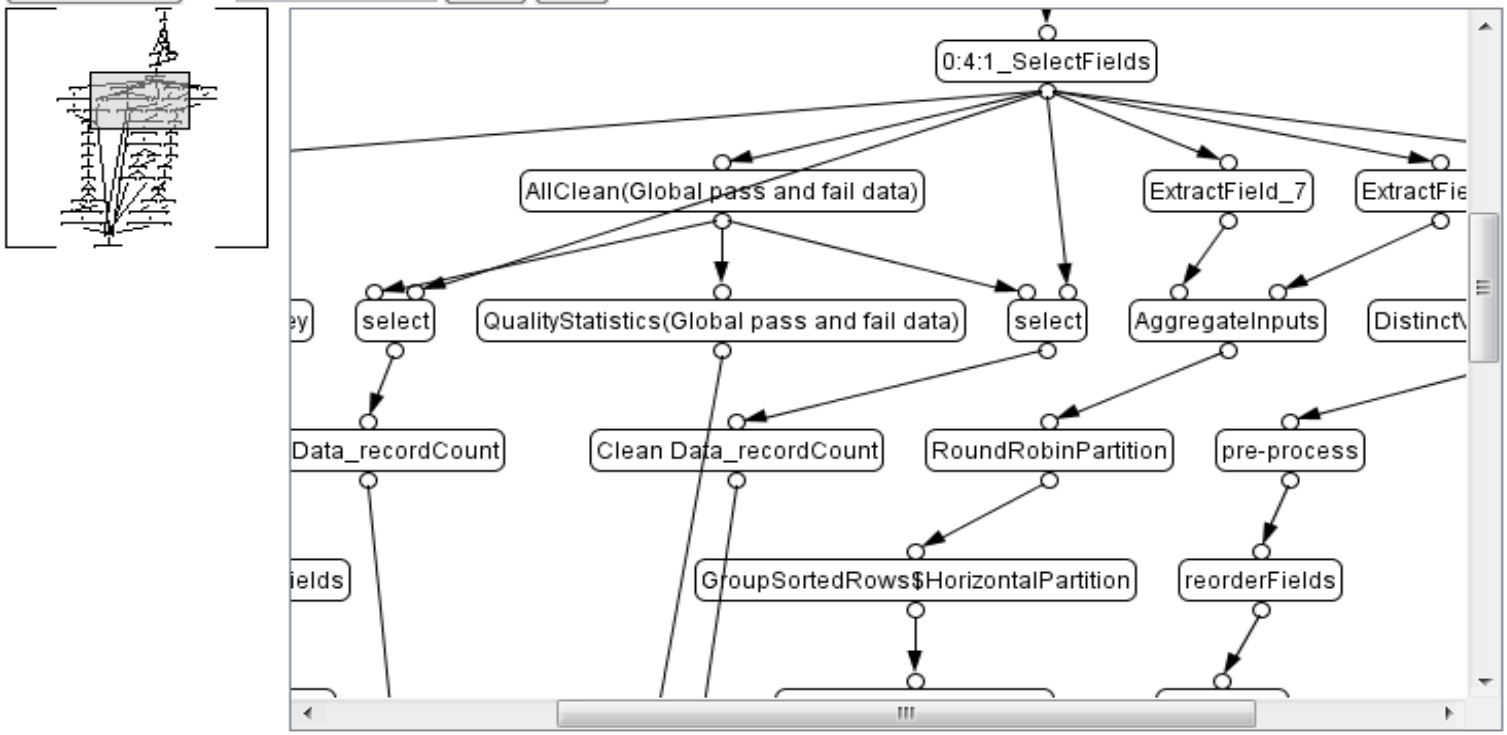
<Unknown Application> (pid 5104)

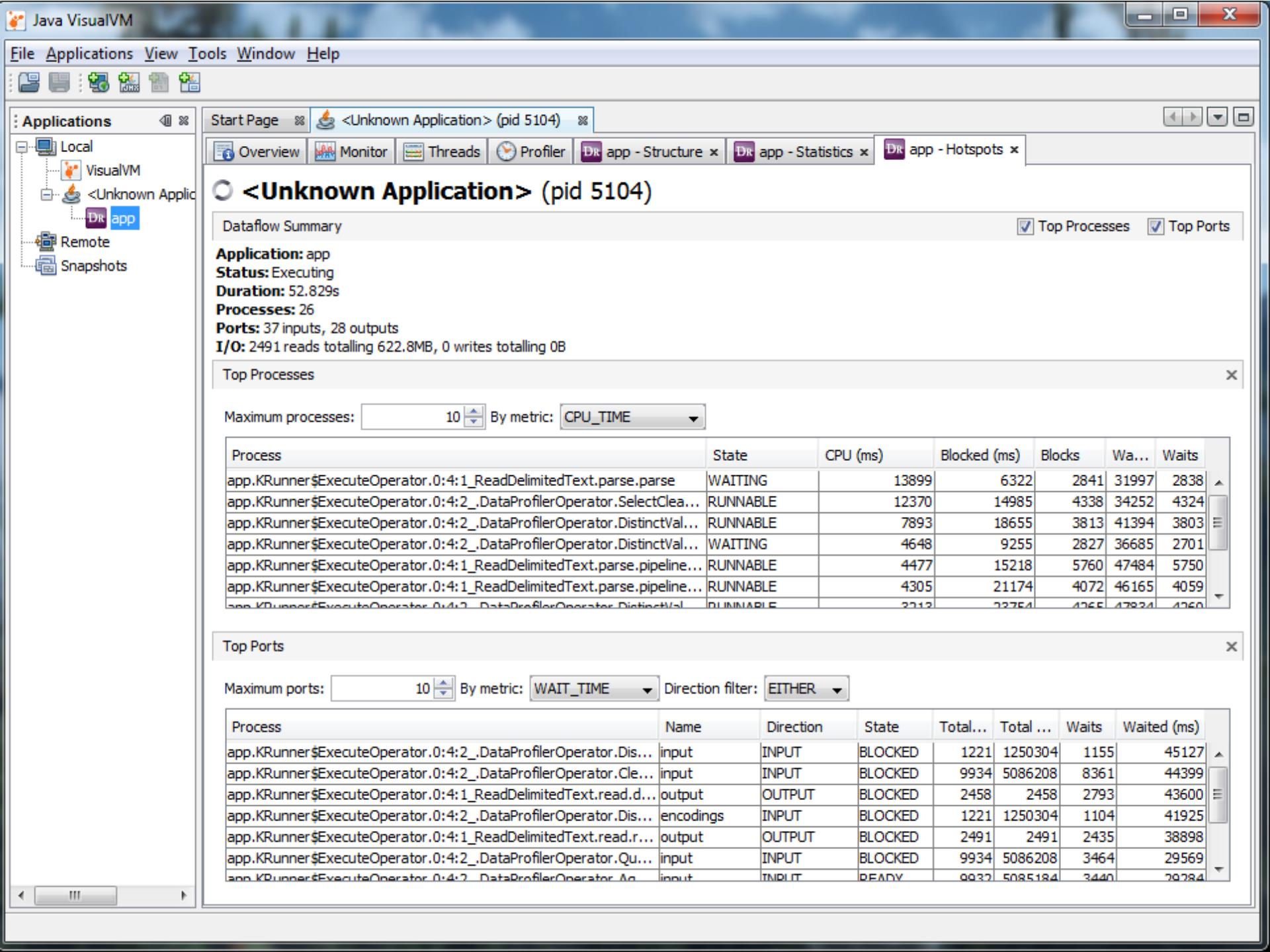
Dataflow Overview — Visualization

Application: app
Status: Executing
Duration: 37.425s
Processes: 26
Ports: 37 inputs, 28 outputs
I/O: 1358 reads totalling 339.5MB, 0 writes totalling 0B

Visualization

Print Graph | 100% | + | -




Java VisualVM
File Applications View Tools Window Help
Applications
Local
VisualVM
<Unknown Applic
app
Remote
Snapshots
Start Page
<Unknown Application> (pid 5104)
Overview
Monitor
Threads
Profiler
app - Structure
app - Statistics
app - Hotspots
<Unknown Application> (pid 5104)
Dataflow Summary
Top Processes
Top Ports
Application: app
Status: Executing
Duration: 52.829s
Processes: 26
Ports: 37 inputs, 28 outputs
I/O: 2491 reads totalling 622.8MB, 0 writes totalling 0B
Top Processes
Maximum processes: 10 By metric: CPU_TIME
Process | State | CPU (ms) | Blocked (ms) | Blocks | Wa... | Waits
app.KRunner$ExecuteOperator.0:4:1_ReadDelimitedText.parse.parse | WAITING | 13899 | 6322 | 2841 | 31997 | 2838
app.KRunner$ExecuteOperator.0:4:2_.DataProfilerOperator.SelectClea... | RUNNABLE | 12370 | 14985 | 4338 | 34252 | 4324
app.KRunner$ExecuteOperator.0:4:2_.DataProfilerOperator.DistinctVal... | RUNNABLE | 7893 | 18655 | 3813 | 41394 | 3803
app.KRunner$ExecuteOperator.0:4:2_.DataProfilerOperator.DistinctVal... | WAITING | 4648 | 9255 | 2827 | 36685 | 2701
app.KRunner$ExecuteOperator.0:4:1_ReadDelimitedText.parse.pipeline... | RUNNABLE | 4477 | 15218 | 5760 | 47484 | 5750
app.KRunner$ExecuteOperator.0:4:1_ReadDelimitedText.parse.pipeline... | RUNNABLE | 4305 | 21174 | 4072 | 46165 | 4059
Top Ports
Maximum ports: 10 By metric: WAIT_TIME Direction filter: EITHER
Process | Name | Direction | State | Total... | Total ... | Waits | Waited (ms)
app.KRunner$ExecuteOperator.0:4:2_.DataProfilerOperator.Dis... | input | INPUT | BLOCKED | 1221 | 1250304 | 1155 | 45127
app.KRunner$ExecuteOperator.0:4:2_.DataProfilerOperator.Cle... | input | INPUT | BLOCKED | 9934 | 5086208 | 8361 | 44399
app.KRunner$ExecuteOperator.0:4:1_ReadDelimitedText.read.d... | output | OUTPUT | BLOCKED | 2458 | 2458 | 2793 | 43600
app.KRunner$ExecuteOperator.0:4:2_.DataProfilerOperator.Dis... | encodings | INPUT | BLOCKED | 1221 | 1250304 | 1104 | 41925
app.KRunner$ExecuteOperator.0:4:1_ReadDelimitedText.read.r... | output | OUTPUT | BLOCKED | 2491 | 2491 | 2435 | 38898
app.KRunner$ExecuteOperator.0:4:2_.DataProfilerOperator.Qu... | input | INPUT | BLOCKED | 9934 | 5086208 | 3464 | 29569

DataRush operator libraries

- Data preparation
  - Core: sort, join, aggregate, transform, …
  - Data profiling
  - Fuzzy matching
  - Cleansing
- Analytics
  - Cluster
  - Classify
  - Collaborative filtering
  - Feature selection
  - Linear regression
  - Association rules
  - PMML support

Malstone* B-10 benchmark

- 10 billions rows of web log data
- Nearly 1 Terabyte of data
- Aggregate site intrusion information

DataRush

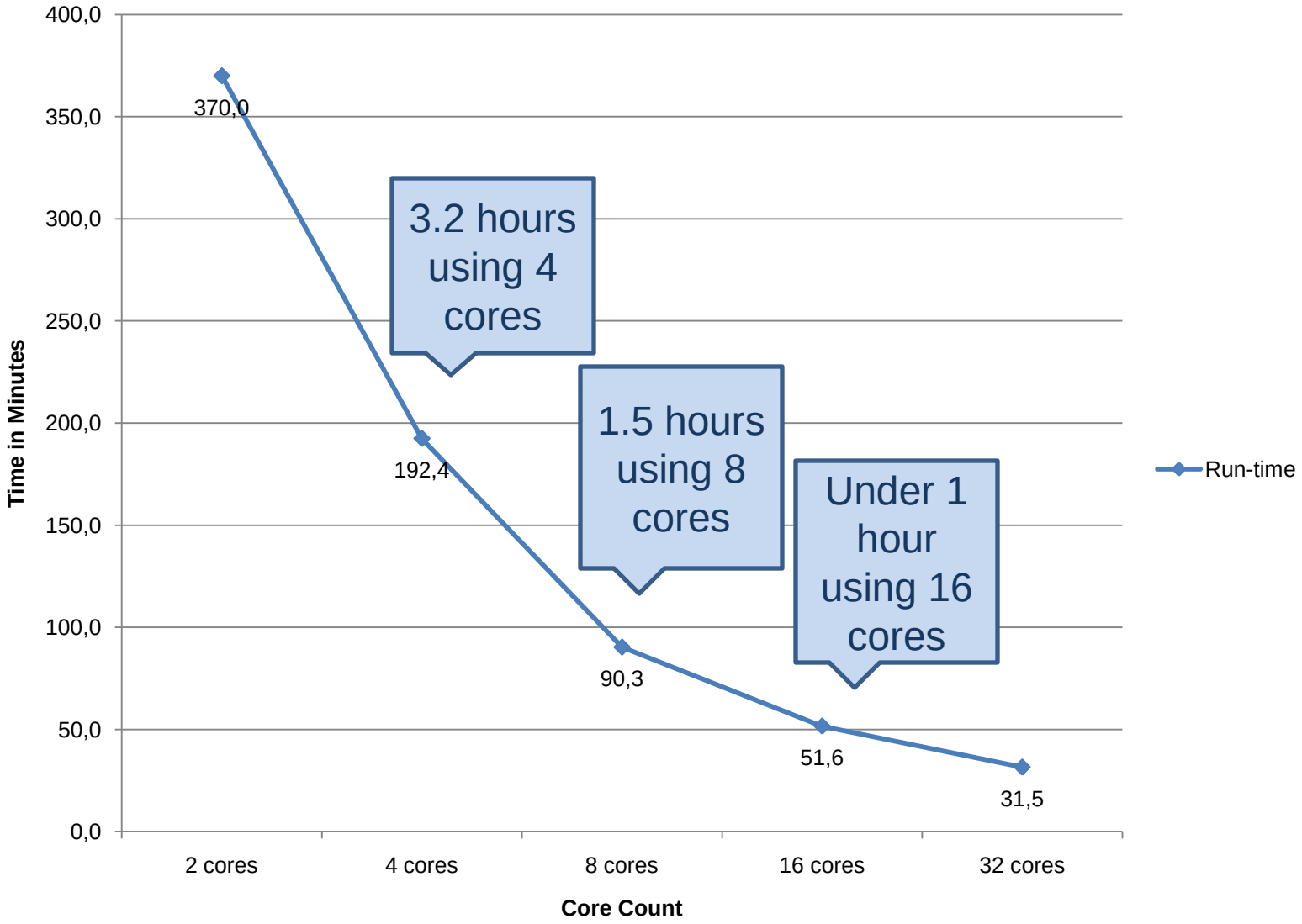
- Configuration
  - Single machine using 4 Intel 7500 processors
  - 32 cores total
  - RAID-0 disk array
  - DataRush + JVM installed
- Results
  - 31.5 minutes
  - Nearly 2 TB/hr throughput

Hadoop (Map-Reduce)

- Configuration
  - 20 node cluster
  - 4-cores per node
  - Hadoop + JVM installed
  - Run by third-party
- Results
  - 14 hours

*www.opencloudconsortium.org/benchmarks

Malstone-B10 Scalability

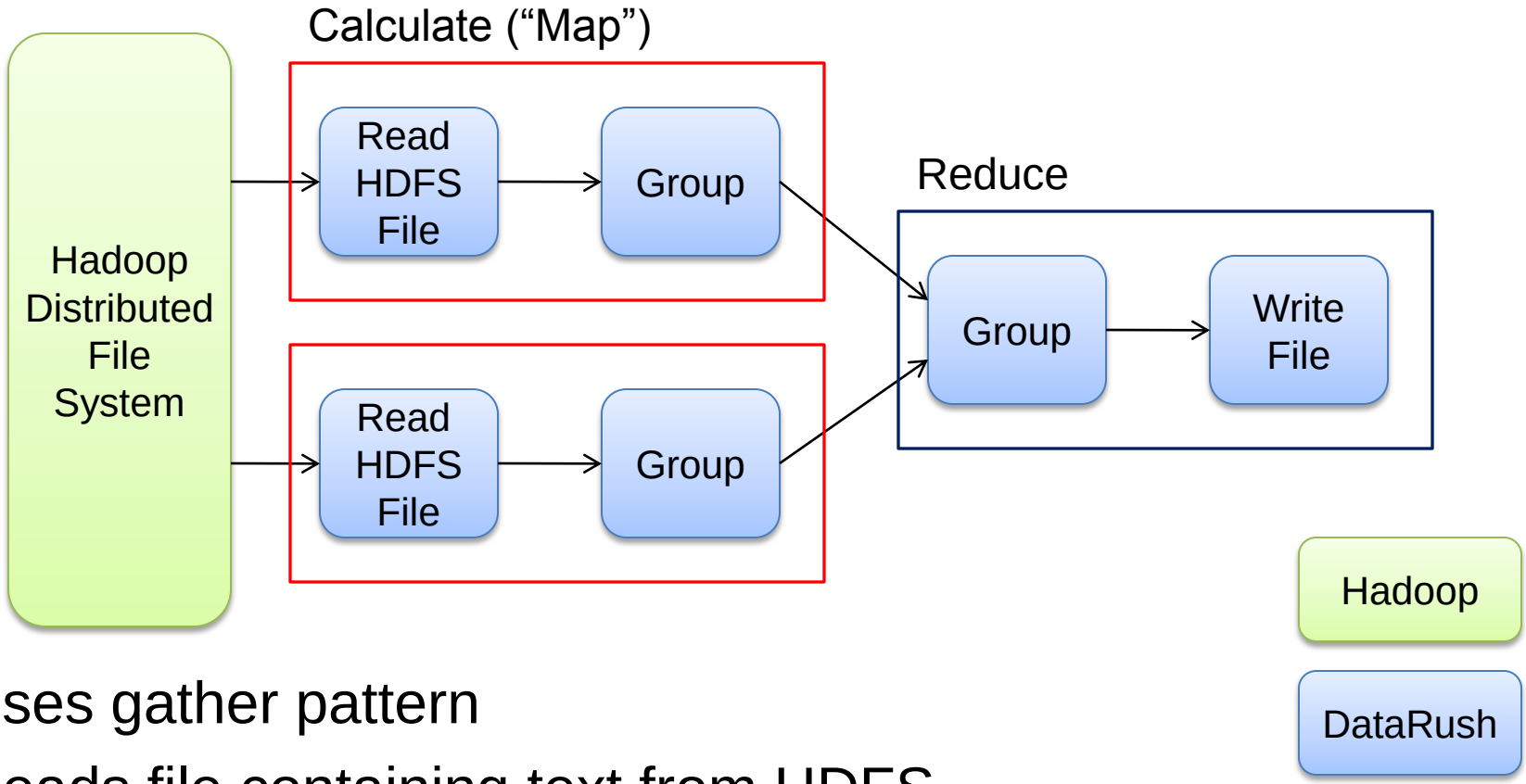


PERVASIVE

Multi-node DataRush

- Extending dataflow to multi-node
  - Execute distributed graph fragments
  - Fragments linked via socket-based queues
  - Used distributed application graph
- Specific patterns supported
  - Scatter
  - Gather
  - Scatter-gather combined
- Available in DataRush 5 (Dec 2010)

Multi-node DataRush example



- Uses gather pattern
- Reads file containing text from HDFS
- Groups by field "state" to count instances
- Groups by "state" to sum counts

Summary

- Dataflow
  - Software architecture based on continuous functions connected via data flows
  - Data focused
  - Easy to grasp and simple to express
  - Simple programming model
  - Utilizes multicore, extendible to multi-node
- DataRush
  - Dataflow based platform
  - Extensive operator library
  - Easy to extend
  - Scales up well with multicore
  - High throughput rates