IEMS5709

Advanced Topics in Information Processing:

Big Data Systems and Information Processing

Spring 2016

DAG-based Dataflow Systems:

Dryad, DryadLINQ, Tez and Beyond

Prof. Wing C. Lau

Department of Information Engineering

wclau@ie.cuhk.edu.hk

Acknowledgements

- The slides are adapted from the following source materials:
  - Michael Isard et al, “Dryad: Distributed and Data-Parallel Programs from Sequential Building Blocks,” Eurosys 2007.
  - Yuan Yu et al, “DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing using a High-level Language,” Usenix OSDI 2008.
  - Michael Isard, “Dryad and Dataflow Systems,” June 2014.
  - The DryadLINQ project page:
    - http://research.microsoft.com/en-us/projects/dryadlinq/default.aspx
  - Andrew Birrell et al, “Distributed Data-Parallel Programming using Dryad,” talk at UC Santa Cruz, Feb 2008.
  - Mihai Budiu, “Cluster Computing with Dryad,” Mar 2008.
  - Tathagata Das, “DryadLINQ,” Talk at UC Berkeley, Oct 2011.
  - Bikas Saha et al, “Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications,” ACM SIGMOD 2015.
  - Bikas Saha, “Apache Tez: Accelerating Hadoop Query Processing,” 2013.
  - Rajesh Balamohan, “Data Processing over YARN,” talk at Hadoop Meetup, 2014.

What is Dryad?

- General-purpose DAG execution engine project launched ca 2005
  - Cited as inspiration for e.g. Hyracks, Tez
- Engine behind Microsoft Cosmos/SCOPE
  - Initially for MSN Search/Bing, later used throughout MSFT
- Core of research batch cluster environment ca 2009
  - DryadLINQ, Quincy scheduler, TidyFS
- In Nov 2011, Microsoft dropped plans to productize/commercialize Dryad to focus on Hadoop instead:
  - http://www.zdnet.com/article/microsoft-drops-dryad-puts-its-big-data-bets-on-hadoop/
- Dryad & DryadLINQ on YARN have been open-sourced under the Apache 2.0 licensed on GitHub:

http://research.microsoft.com/en-us/um/siliconvalley/projects/BigDataDev/

Goals of the Dryad Project

- General-purpose execution environment for distributed, data-parallel applications
  - Concentrates on throughput not latency
  - Assumes private data center (=> security???)
- Automatic management of scheduling, distribution, fault tolerance, etc.

Outline

- Motivation - what's wrong with MapReduce ?
- What is Dryad ?
- What is DryadLINQ ?
- What is Tez ?
- Why is dataflow so useful?
  - An engineering sweet spot
- Beyond Stateless DAG Dataflow

What's wrong with MapReduce?

- Literally Map then Reduce and that's it…
  - Reducers write to replicated storage
- Complex jobs pipeline multiple stages
  - No fault tolerance between stages
    - Map assumes its data is always available: simple!
- Output of Reduce: 2 network copies, write 3 disks
  - In Dryad this collapses inside a single process
  - Big jobs can be more efficient with Dryad

What's wrong with MapReduce?

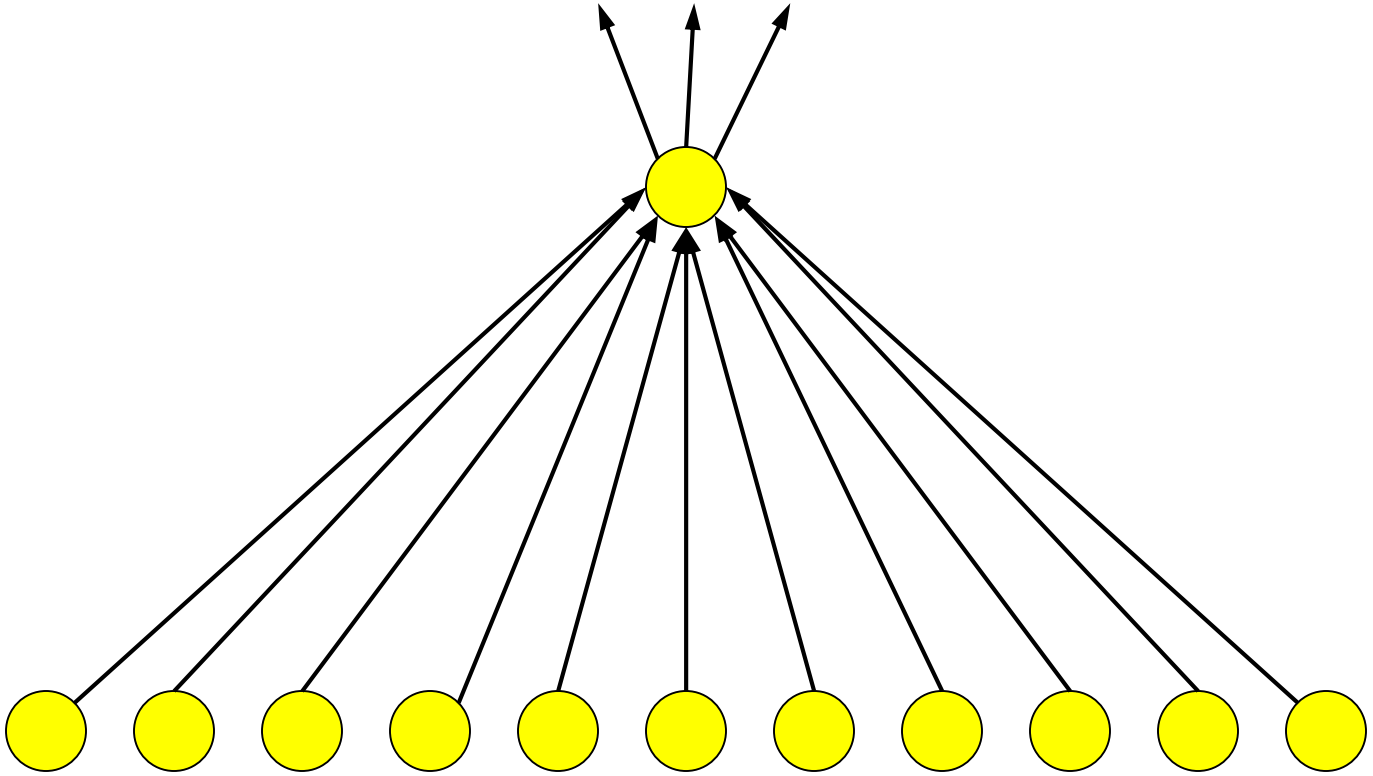
- Literally Map then Reduce and that's it…
  - Reducers write to replicated storage
- Complex jobs pipeline multiple stages
  - No fault tolerance between stages
    - Map assumes its data is always available: simple!
- Output of Reduce: 2 network copies, write 3 disks
  - In Dryad this collapses inside a single process
  - Big jobs can be more efficient with Dryad

What's wrong with Map+Reduce?

- Join combines inputs of different types
- "Split" produces outputs of different types
  - Parse a document, output text and references
- Can be done with Map+Reduce
  - Ugly to program
  - Hard to avoid performance penalty
  - Some merge joins *very* expensive
    - Need to materialize entire cross product to disk

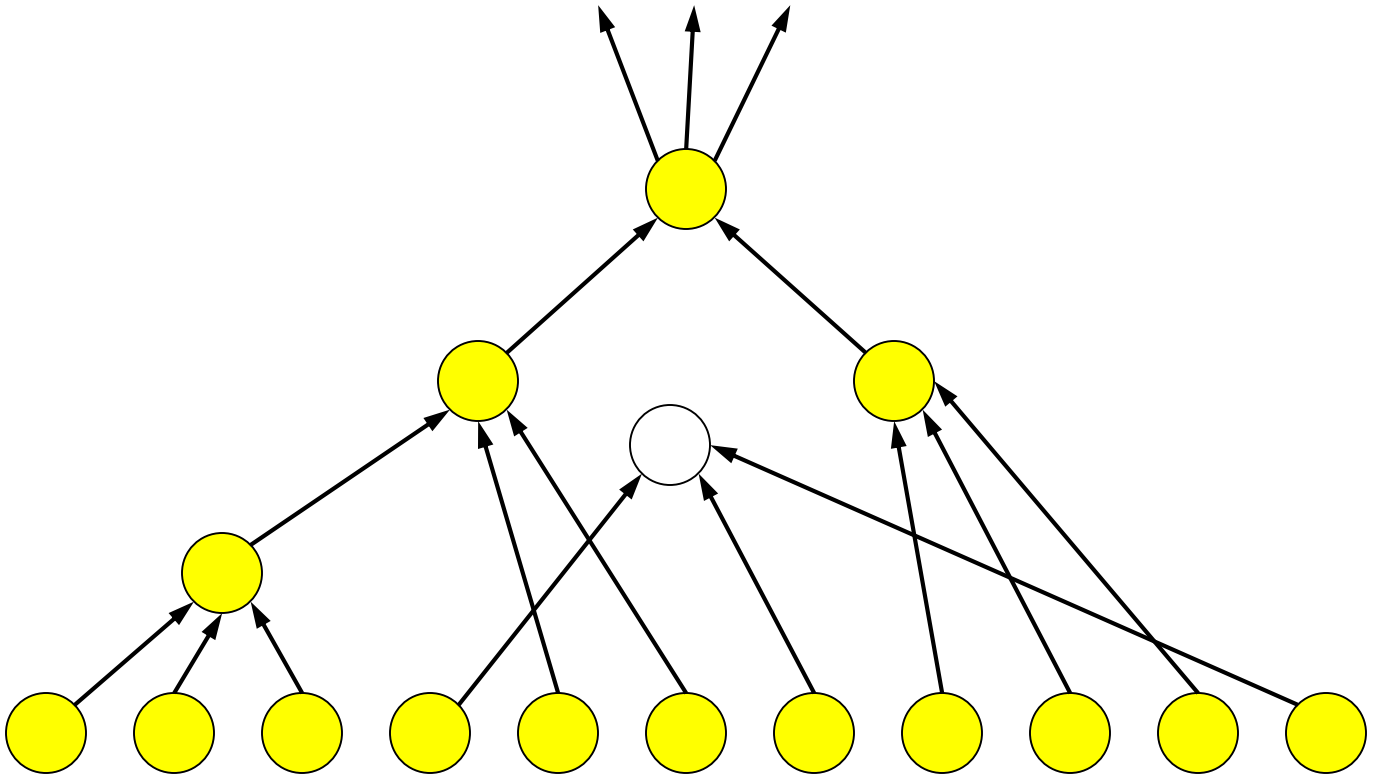
How about Map+Reduce+Join+…?

- “Uniform” stages aren’t really uniform



How about Map+Reduce+Join+…?

- “Uniform” stages aren’t really uniform



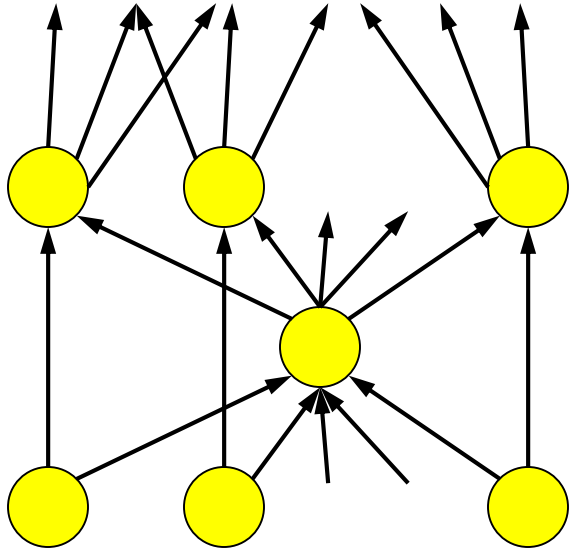
Graph complexity composes

- Non-trees common
- E.g. data-dependent re-partitioning
  - Combine this with merge trees etc.

Distribute to equal-sized ranges

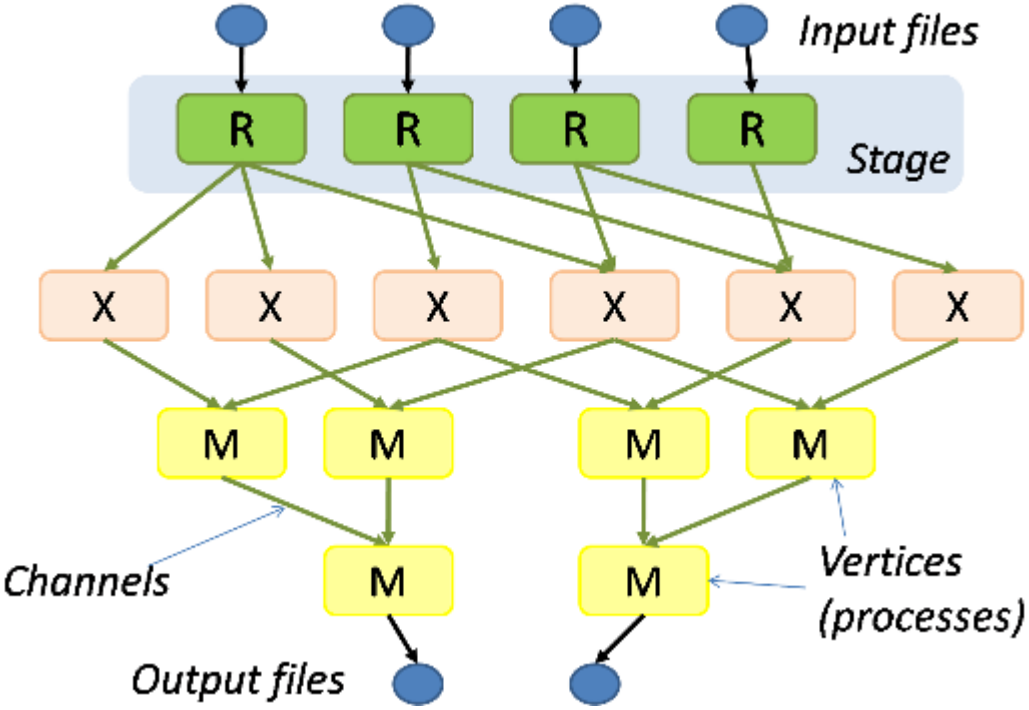
Sample to estimate histogram

Randomly partitioned inputs

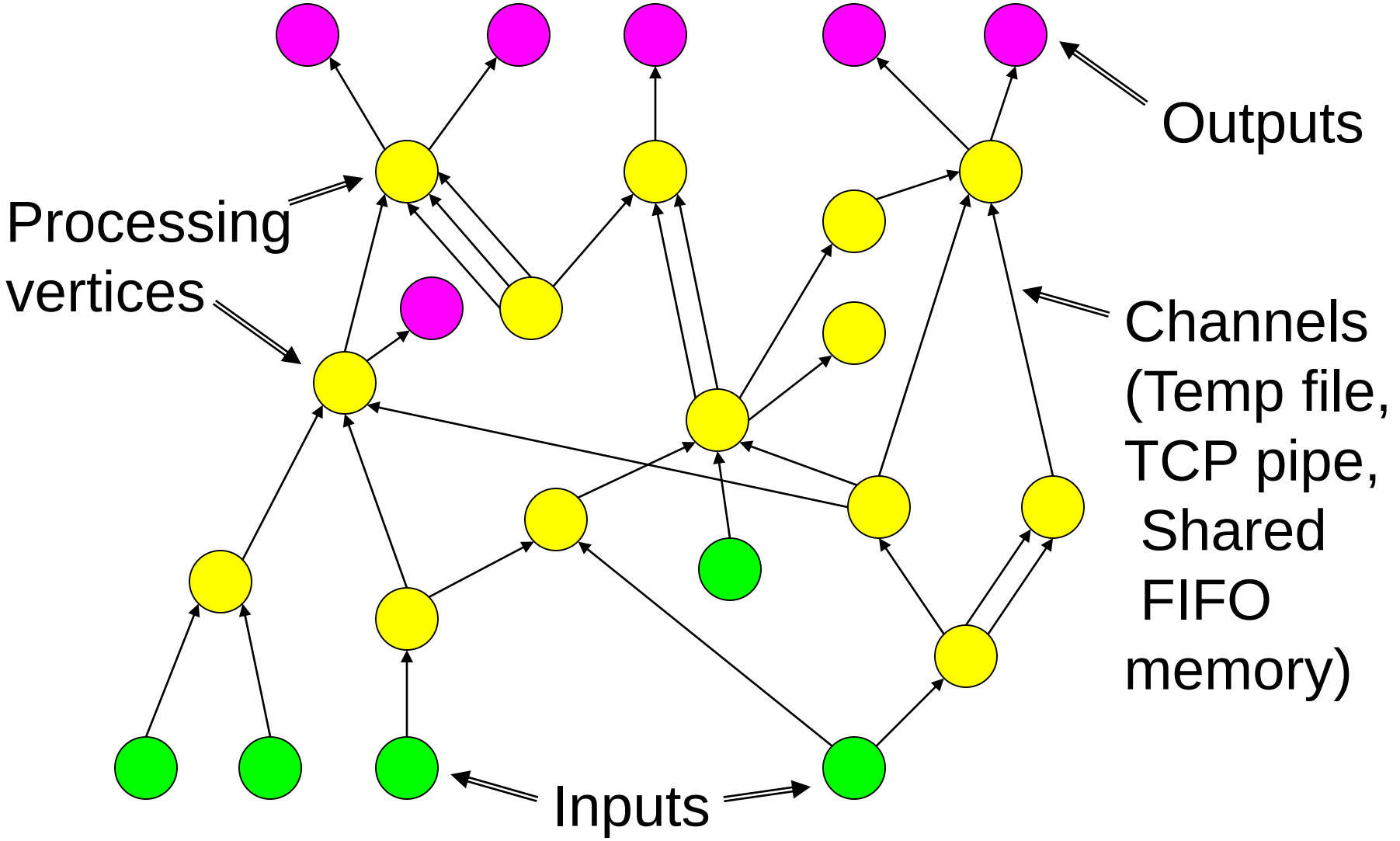


Main Idea of Dryad

- Represent the computation as a Directed Acyclic Graph (DAG) of communicating sequential processes



A Dryad Job = Directed Acyclic Graph



Support Multiple ways of Communications (b/w Vertices of the Computation Graph)

Channel protocol	Discussion
File (the default)	Preserved after vertex execution until the job completes.
TCP pipe	Requires no disk accesses, but both end-point vertices must be scheduled to run at the same time.
Shared-memory FIFO	Extremely low communication cost, but end-point vertices must run within the same process.

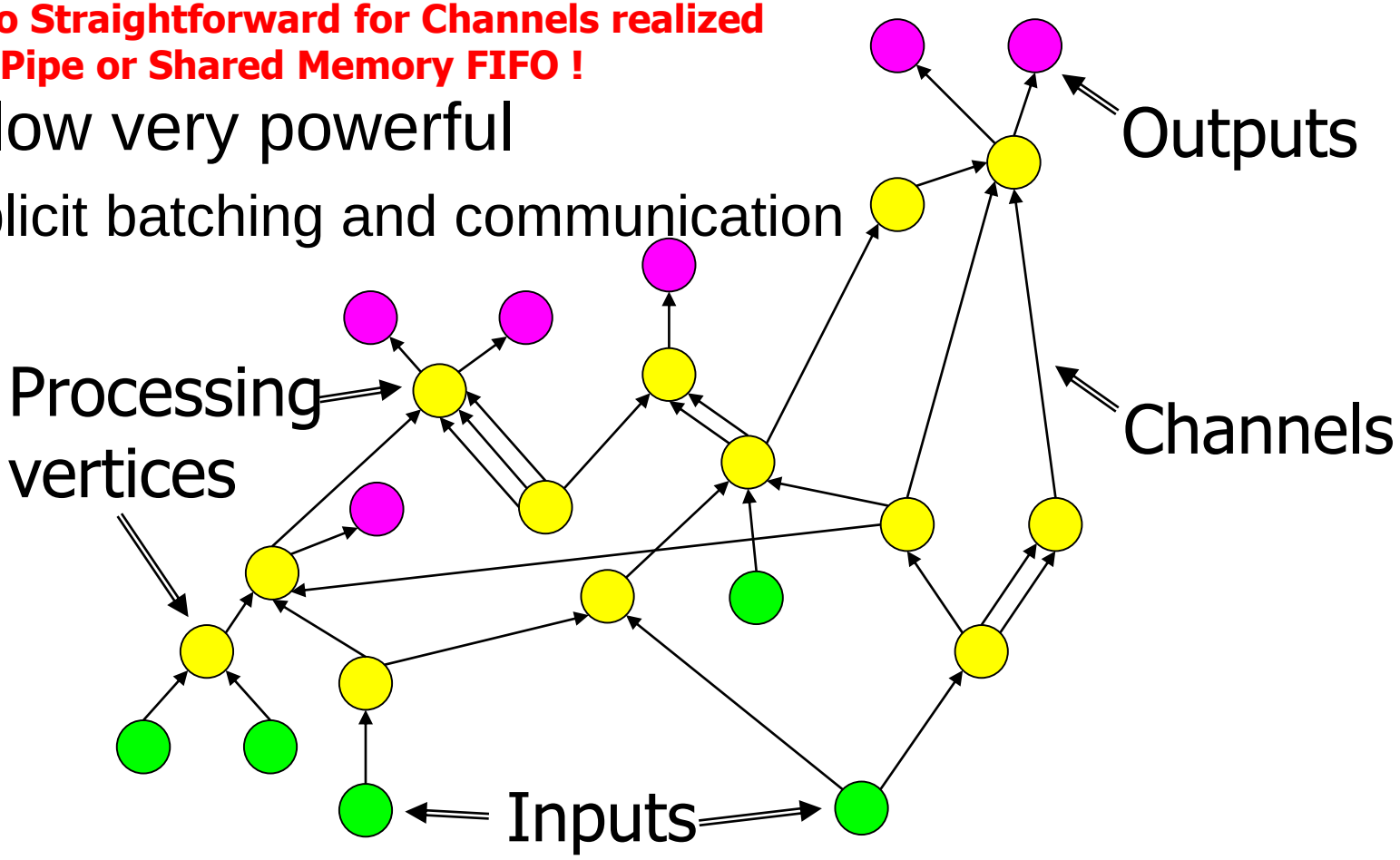
Table 1: Channel types.

The DAG Computational model

- Vertices are independent
  - State and scheduling**

****Not so Straightforward for Channels realized by TCP Pipe or Shared Memory FIFO !**

- Dataflow very powerful
  - Explicit batching and communication



Constructing the Job

- Use graph operators implemented as a C++ library to describe the graph:

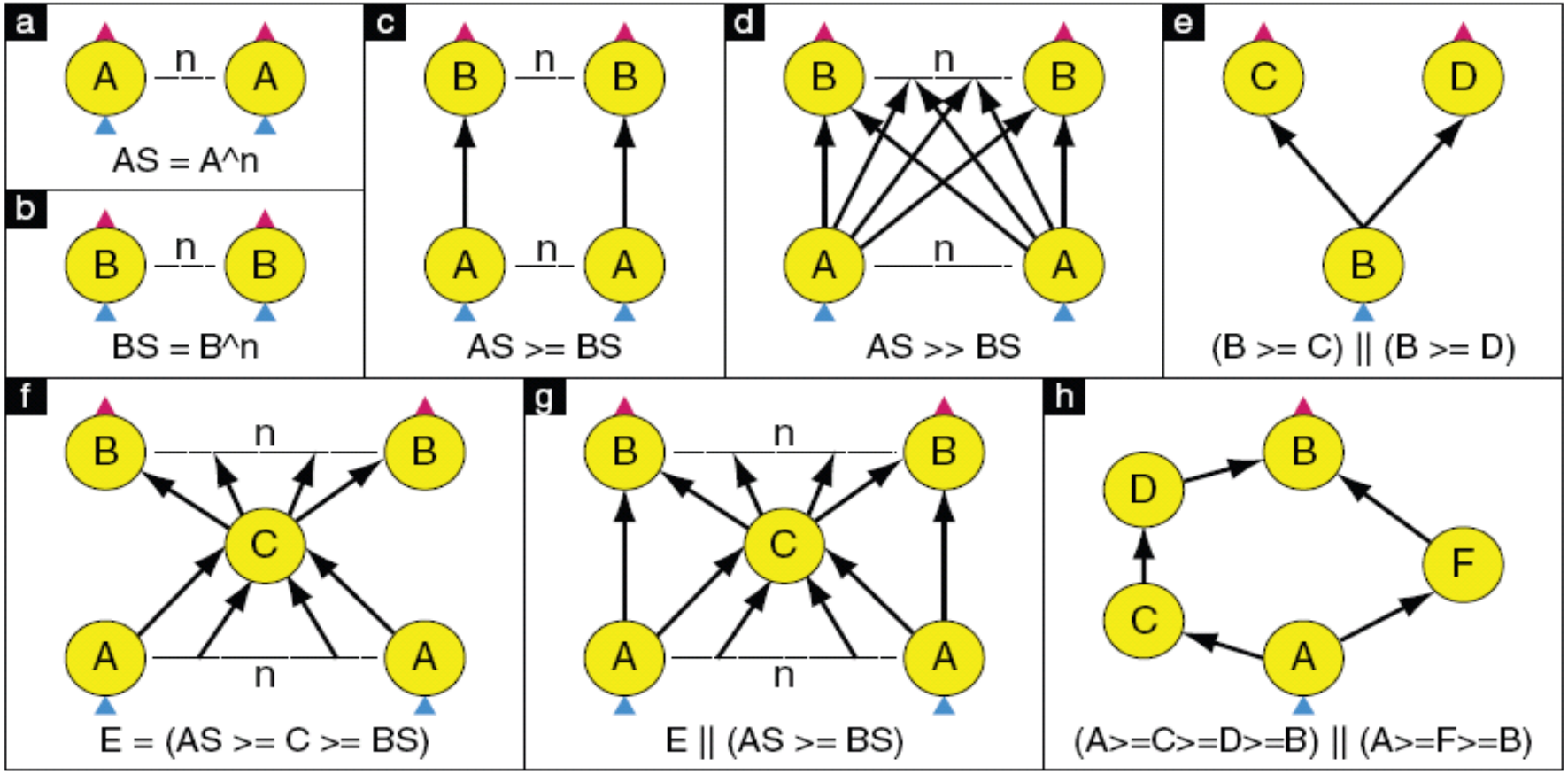


Figure 3: The operators of the graph description language. Circles are vertices and arrows are graph edges. A triangle at the bottom of a vertex indicates an *input* and one at the top indicates an *output*. Boxes (a) and (b) demonstrate cloning individual vertices using the ^ operator. The two standard connection operations are pointwise composition using >= shown in (c) and complete bipartite composition using >> shown in (d). (e) illustrates a merge using | |. The second line of the figure shows more complex patterns. The merge in (g) makes use of a "subroutine" from (f) and demonstrates a bypass operation. For example, each A vertex might output a summary of its input to C which aggregates them and forwards the global statistics to every B. Together the B vertices can then distribute the original dataset (received from A) into balanced partitions. An asymmetric fork/join is shown in (h).

Example: a Dryad Job for Database Query

```
select distinct p.objID
from photoObjAll p
join neighbors n    — call this join "X"
on p.objID = n.objID
  and n.objID < n.neighborObjID
  and p.mode = 1
join photoObjAll l  — call this join "Y"
on l.objid = n.neighborObjID
  and l.mode = 1
  and abs((p.u-p.g)-(l.u-l.g))<0.05
  and abs((p.g-p.r)-(l.g-l.r))<0.05
  and abs((p.r-p.i)-(l.r-l.i))<0.05
  and abs((p.i-p.z)-(l.i-l.z))<0.05
```

```
GraphBuilder XSet = moduleX^N;
GraphBuilder DSet = moduleD^N;
GraphBuilder MSet = moduleM^(N*4);
GraphBuilder SSet = moduleS^(N*4);
GraphBuilder YSet = moduleY^N;
GraphBuilder HSet = moduleH^1;

GraphBuilder XInputs = (ugriz1 >= XSet) || (neighbor >= XSet);
GraphBuilder YInputs = ugriz2 >= YSet;

GraphBuilder XToY = XSet >= DSet >> MSet >= SSet;
for (i = 0; i < N*4; ++i)
{
    XToY = XToY || (SSet.GetVertex(i) >= YSet.GetVertex(i/4));
}

GraphBuilder YToH = YSet >= HSet;
GraphBuilder HOutputs = HSet >= output;

GraphBuilder final = XInputs || YInputs || XToY || YToH || HOutputs;
```

Figure 4: An example graph builder program. The communication graph generated by this program is shown in Figure 2.

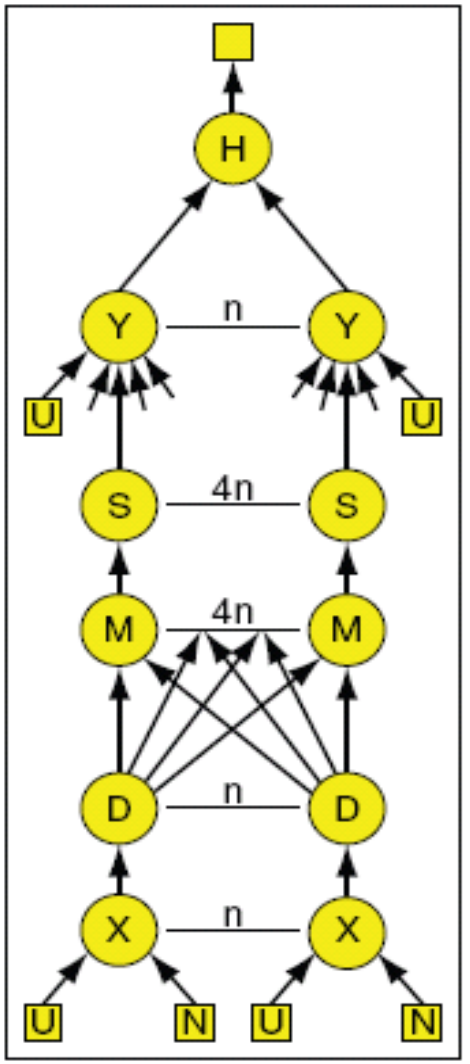


Figure 2: The communication graph for an SQL query. Details are in Section 2.1.

Dryad Key Features

- Dataflow graph is mutable at runtime
  - Repartition to avoid skew
  - Specialize matrices dense/sparse
  - Harden fault-tolerance
- Support general-purpose refinement rules for Optimization
  - Processes formed from subgraphs
  - Re-arrange computations, change I/O type
- *Application code not modified !*
  - System at liberty to make optimization choices
- High-level front ends hide this from user
  - SQL query planner, etc.

Run-time Graph Refinement

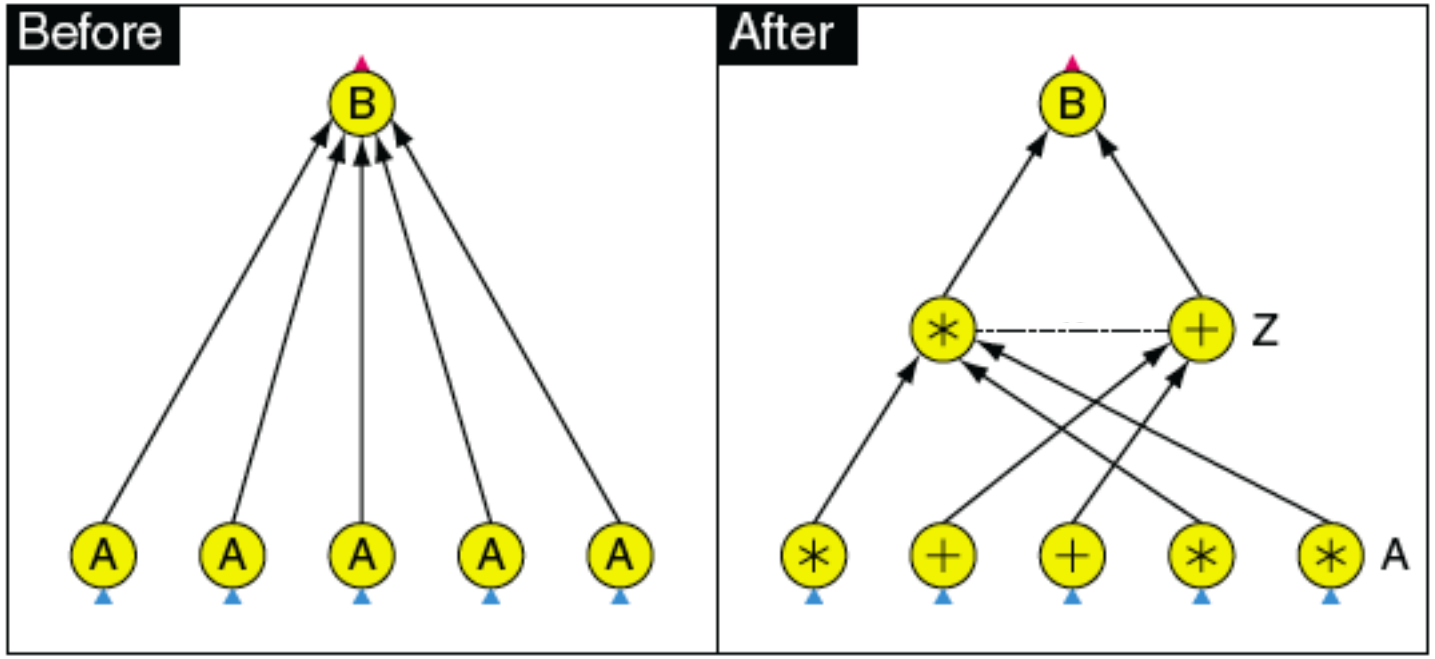


Figure 6: A dynamic refinement for aggregation. The logical graph on the left connects every input to the single output. The locations and sizes of the inputs are not known until run time when it is determined which computer each vertex is scheduled on. At this point the inputs are grouped into subsets that are close in network topology, and an internal vertex is inserted for each subset to do a local aggregation, thus saving network bandwidth. The internal vertices are all of the same user-supplied type, in this case shown as "Z." In the diagram on the right, vertices with the same label ('+' or '$*$') are executed close to each other in network topology.

Run-time Graph Refinement (cont'd)

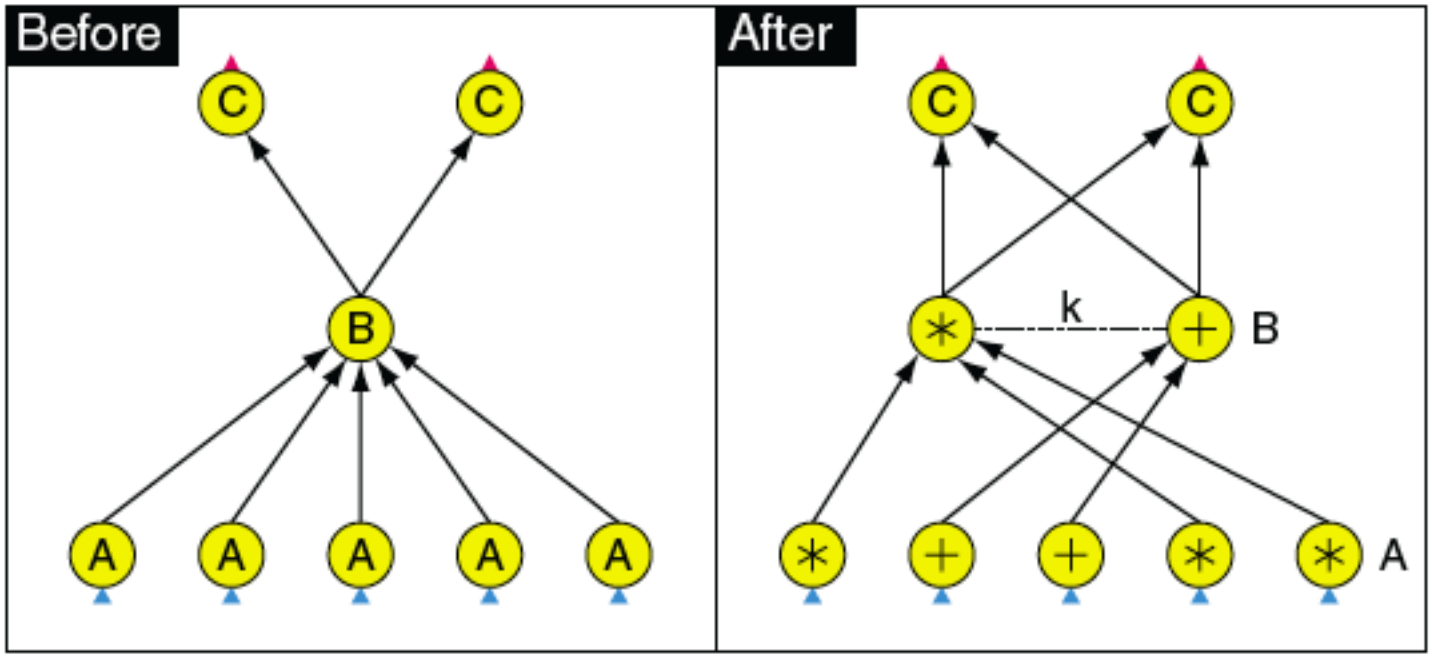
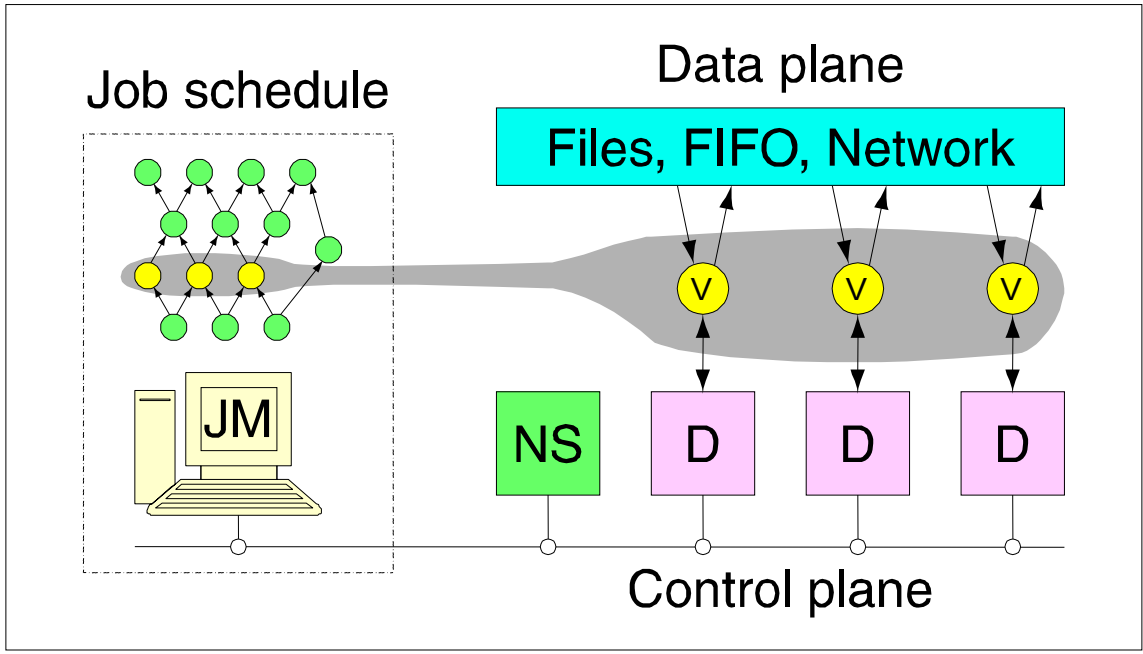


Figure 7: A partial aggregation refinement. Following an input grouping as in Figure 6 into k sets, the successor vertex is replicated k times to process all the sets in parallel.

What Dryad does ?

- Abstracts cluster resources
  - Set of computers, network topology, etc.
- Recovers from transient failures
  - Rerun computations on machine or network fault
  - Speculate duplicates for slow computations
- Schedules a local DAG of work at each vertex

Dryad System Architecture



- Job Manager (JM)
  - Centralized coordinating process
  - User application to construct graph
  - Linked with Dryad libraries for scheduling vertices
- Vertex executable
  - Dryad libraries to communicate with JM
  - User application sees channels in/out
  - Arbitrary application code, can use local Filesystem

Dryad System Architecture (cont'd)

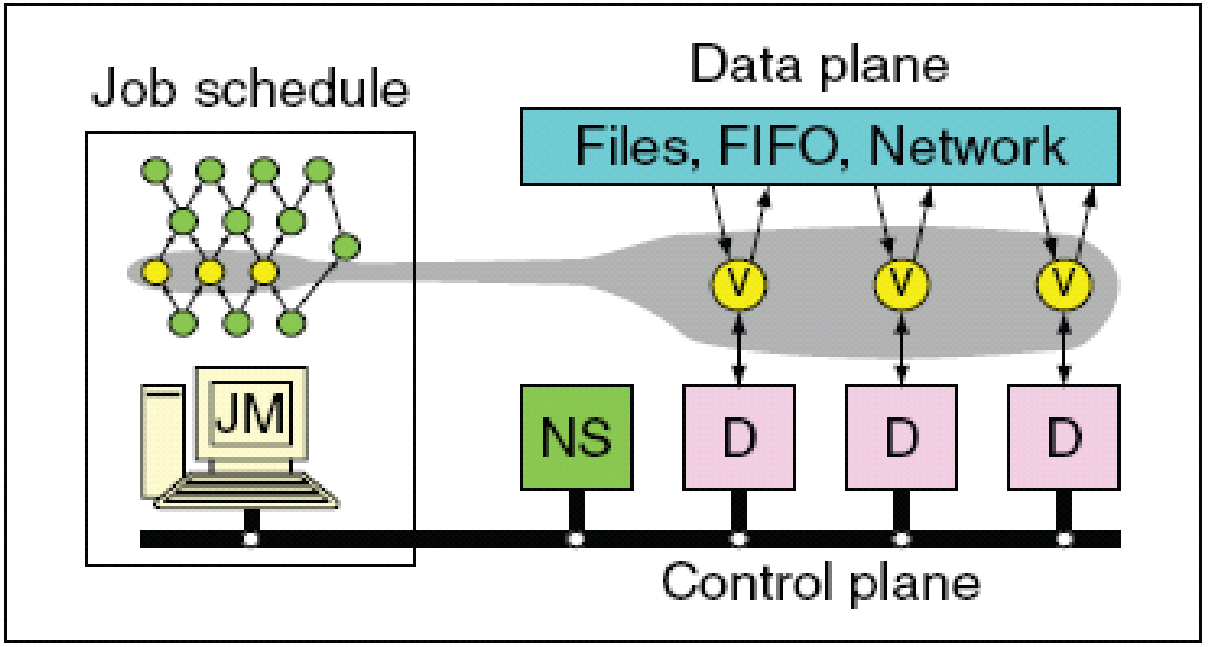


Figure 1: The Dryad system organization. The job manager (JM) consults the name server (NS) to discover the list of available computers. It maintains the job graph and schedules running vertices (V) as computers become available using the daemon (D) as a proxy. Vertices exchange data through files, TCP pipes, or shared-memory channels. The shaded bar indicates the vertices in the job that are currently running.

- Services
  - Name server enumerates all resources
  - Including location relative to other resources
  - Daemon running on each machine for vertex dispatch

Dryad DAG architecture

- Simplicity *depends* on generality
  - Front ends only see graph data-structures
  - Generic scheduler state machine
    - Software engineering: clean abstraction
    - Restricting set of operations which may pollute scheduling logic with execution semantics
- Optimizations all “above the fold”
  - Dryad exports callbacks so applications can react to state machine transitions

Execution

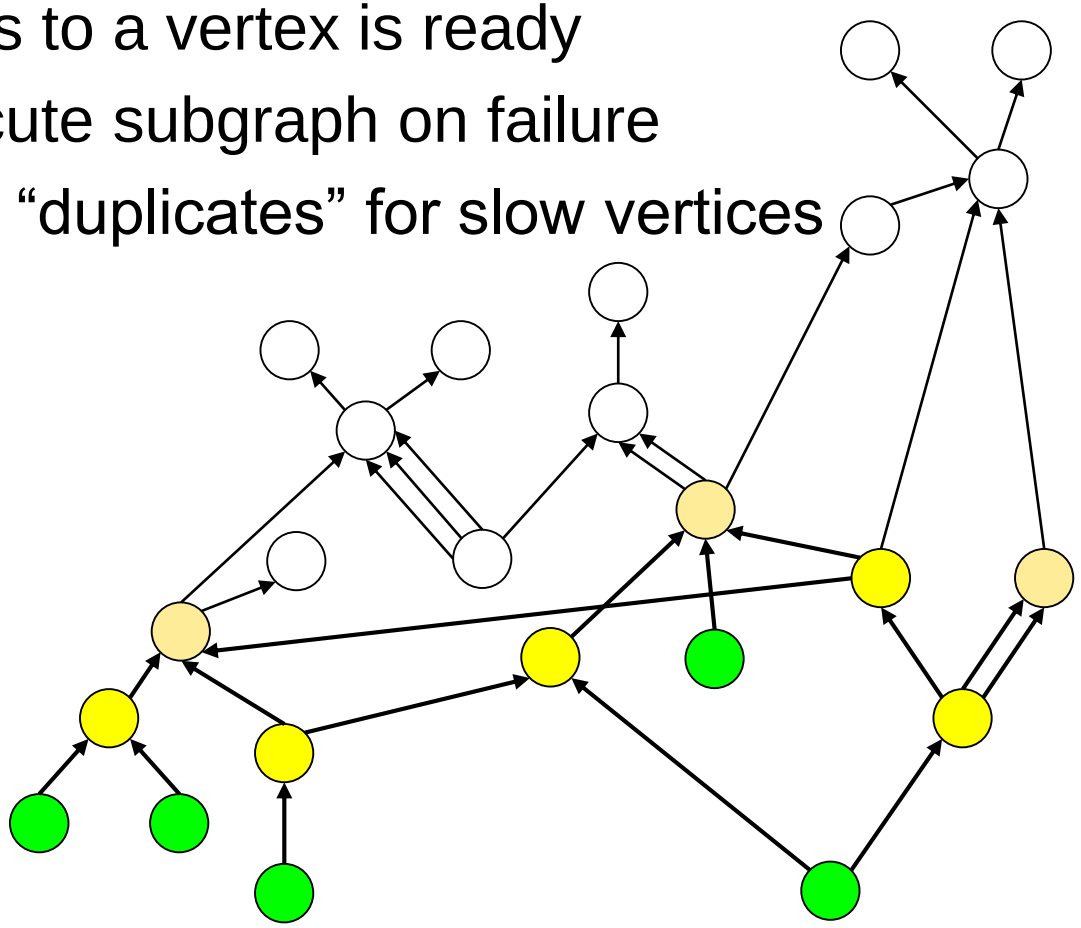
- Job manager not currently fault tolerant
- Vertices may be scheduled multiple times
  - Each execution versioned
  - Execution record kept- including versions of incoming vertices
  - Outputs are uniquely named (versioned)
  - Final outputs selected if job completes
  - Non-file channel communication may cascade failures
- Vertices specify hard constraints or preferences (Hints) for placement
- Scheduling is greedy assuming only one job

Scheduler state machine

- Scheduling is independent of semantics
  - Vertex can run anywhere once all its inputs are ready
  - Fault tolerance
    - If A fails, run it again
    - If A's inputs are gone, run upstream vertices again (recursively)
    - If A is slow, run another copy elsewhere and use output from whichever finishes first

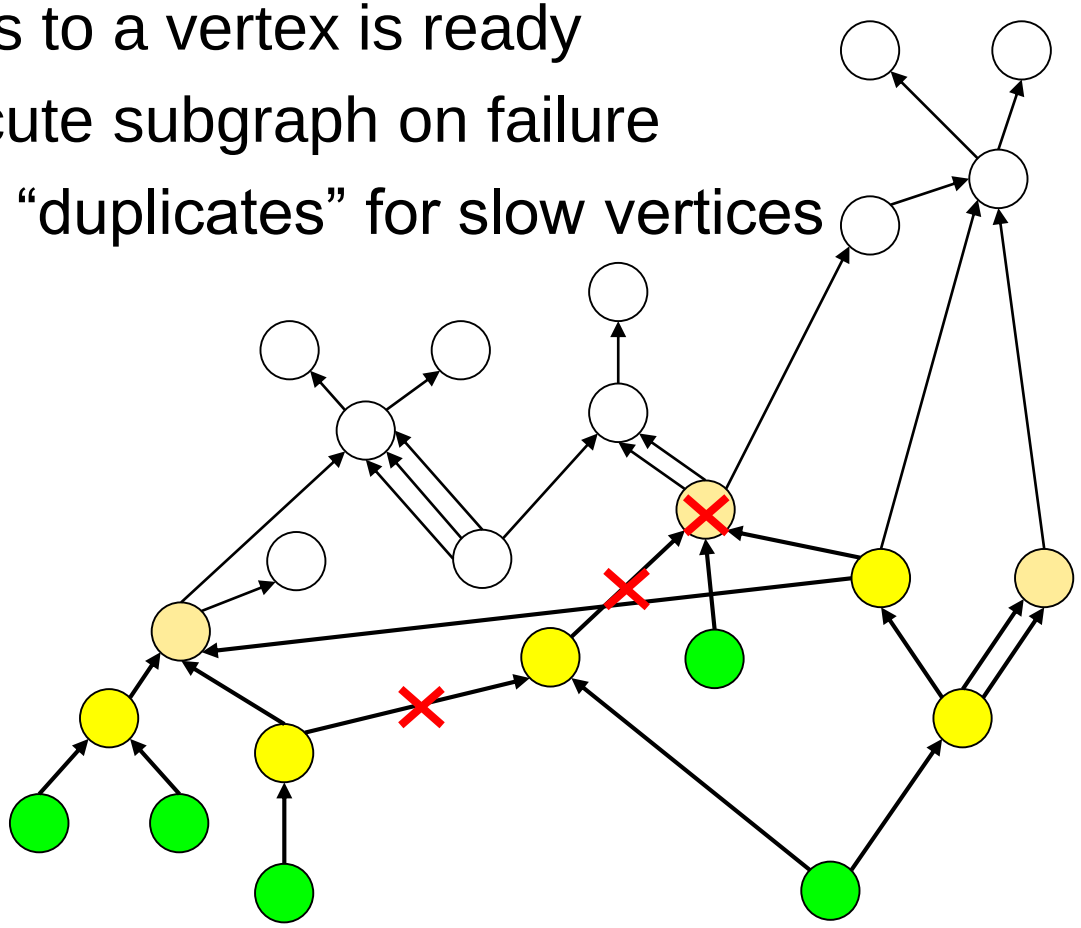
Scheduling and fault tolerance

- DAG makes things easy
  - Schedule from source to sink in any order as long as inputs to a vertex is ready
  - Re-execute subgraph on failure
  - Execute “duplicates” for slow vertices



Scheduling and fault tolerance

- DAG makes things easy
  - Schedule from source to sink in any order as long as inputs to a vertex is ready
  - Re-execute subgraph on failure
  - Execute “duplicates” for slow vertices



Resources are virtualized

- Each graph vertex is a process
  - Write outputs to disk (usually, but not always !)
  - Read inputs from upstream nodes' output files
- Graph generally larger than cluster RAM
  - 1TB partitioned input, 250MB part size, 4000 parts
- Cluster is shared
  - Don't size program for exact cluster
  - Use whatever share of resources are available

Example: SkyServer DB Query

- 3-way join to find gravitational lens effect
- Table U: (objId, color) 11.8GB
- Table N: (objId, neighborId) 41.8GB
- Find neighboring stars with similar colors:
  - Join U+N to find

    T = U.color,N.neighborId where U.objId = N.objId
  - Join U+T to find

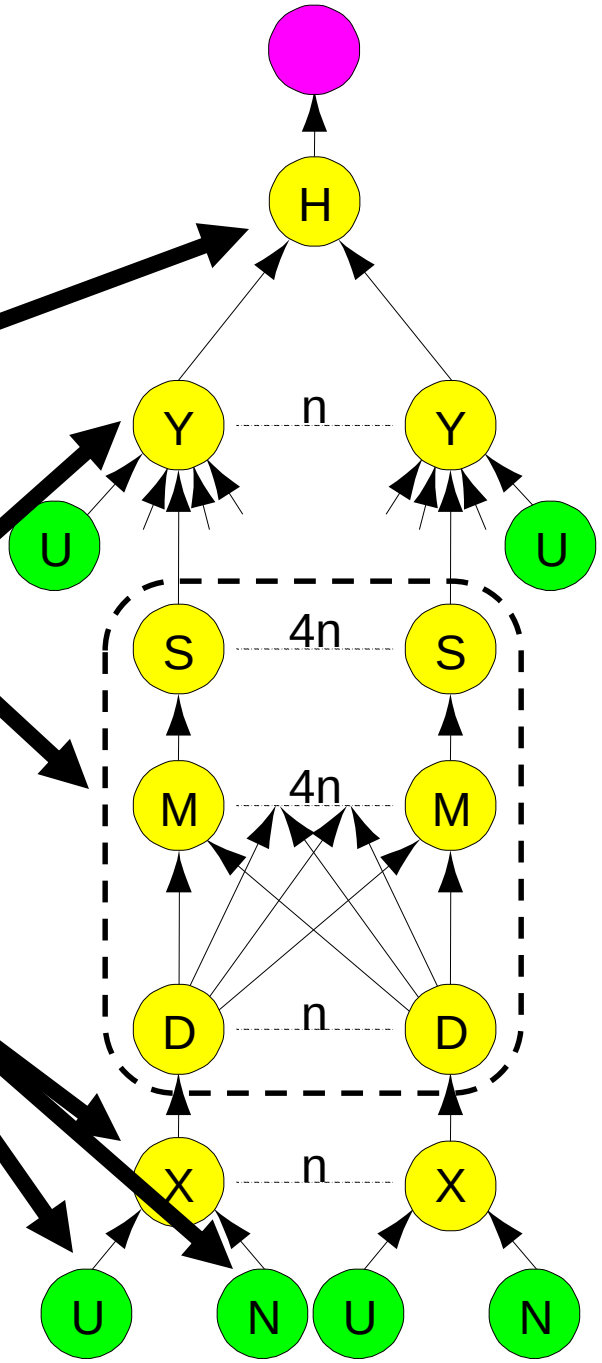
    U.objId where U.objId = T.neighborID

    and U.color ≈ T.color

SkyServer DB query

```
[distinct]
[merge outputs]
```

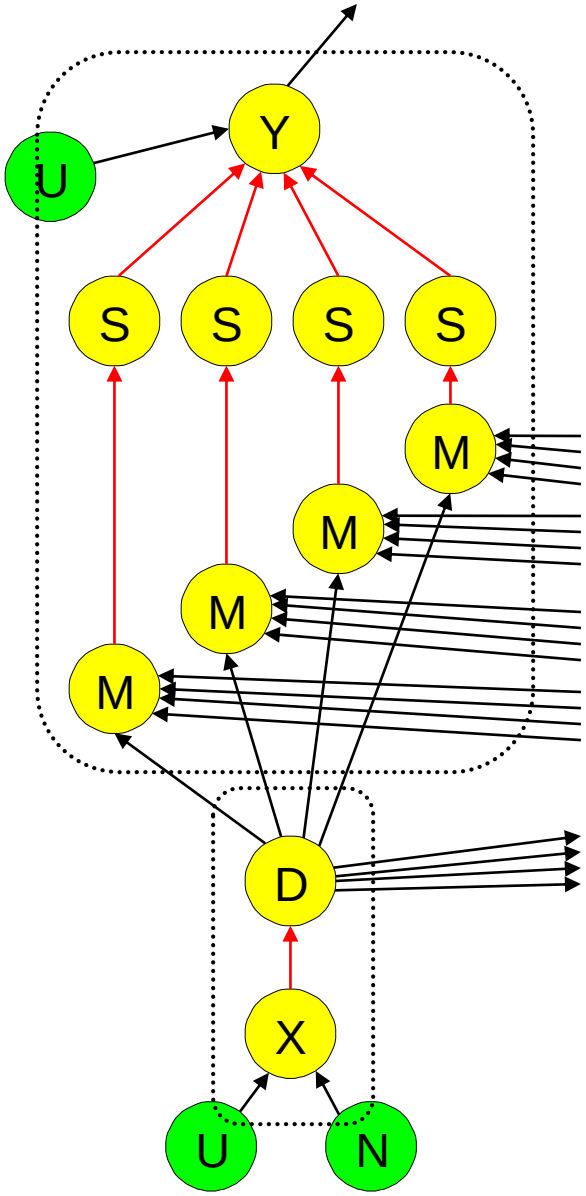
```
select
  u.objid
from u join <temp>
where
  u.objid =
<temp>.neighborobjid and
  |u.color - <temp>.color| < d
```

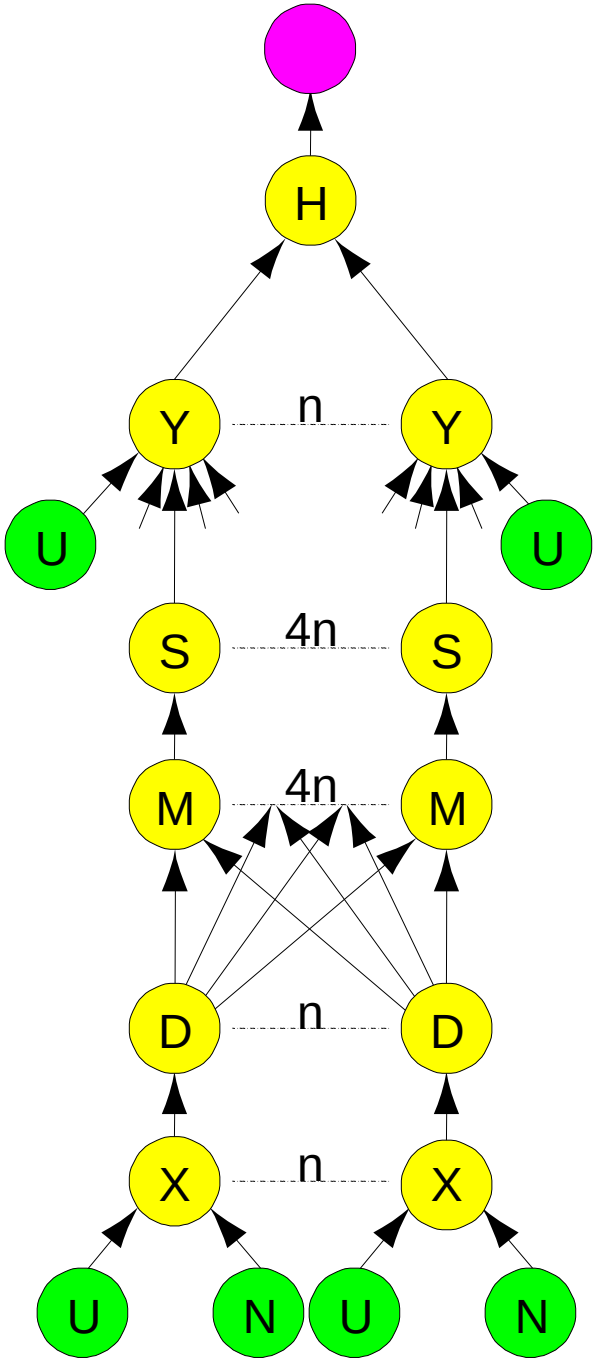


Recap: Automatic Optimizing Dryad applications

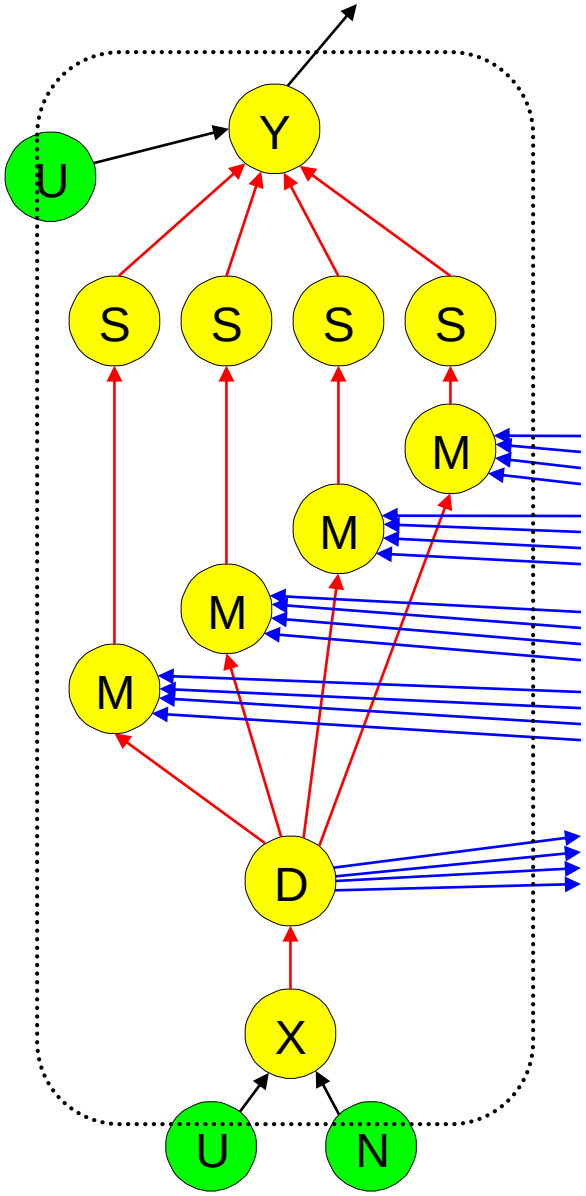
- General-purpose refinement rules
- DAG mutation during run-time !
- Processes formed from subgraphs
  - Re-arrange computations, change I/O type
- ***Application code not modified***
  - System at liberty to make optimization choices
- High-level front ends hide this from user
  - SQL query planner, etc.

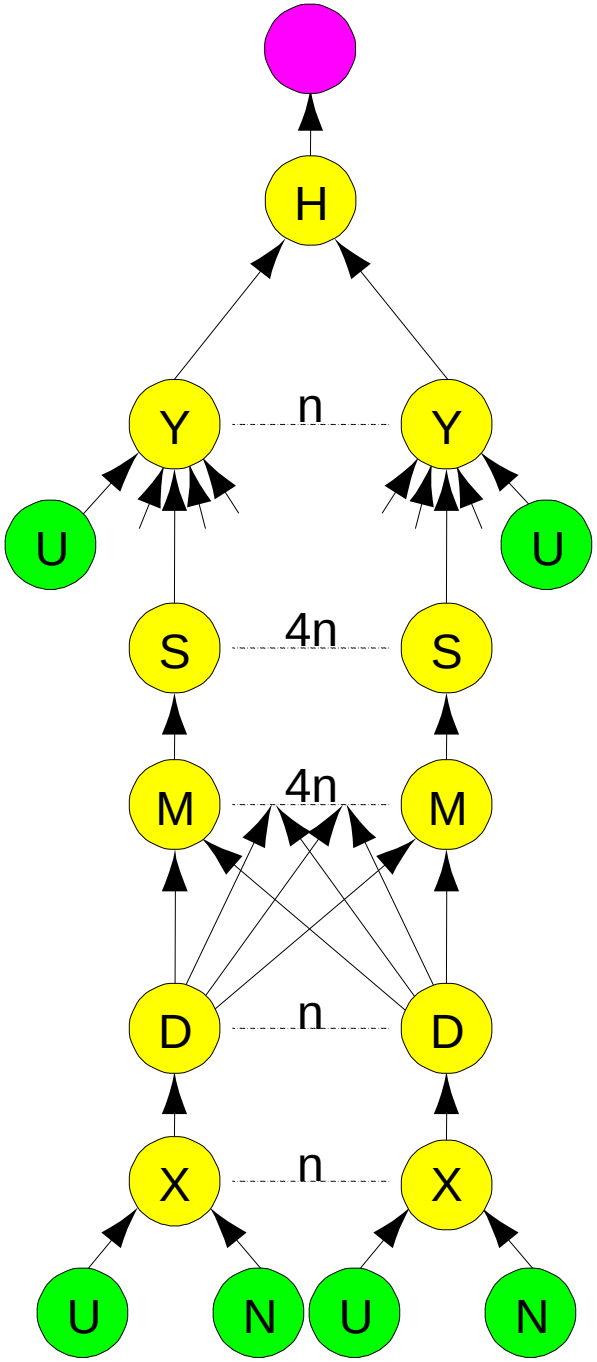
Optimization





Optimization





Performance Results

- SQL Query
- 10 Machines
  - 2 dualcore 2 GHz
  - 8 GB Mem
  - 1 Gb Ethernet
  - 4x400GB disks
  - Winows Server 2003

Computers	1	2	3	4	5	6	7	8	9
SQLServer	3780								
Two-pass	2370	1260	836	662	523	463	423	346	321
In-memory						217	203	183	168

Table 2: Time in seconds to process an SQL query using different numbers of computers. The SQLServer implementation cannot be distributed across multiple computers and the in-memory experiment can only be run for 6 or more computers.

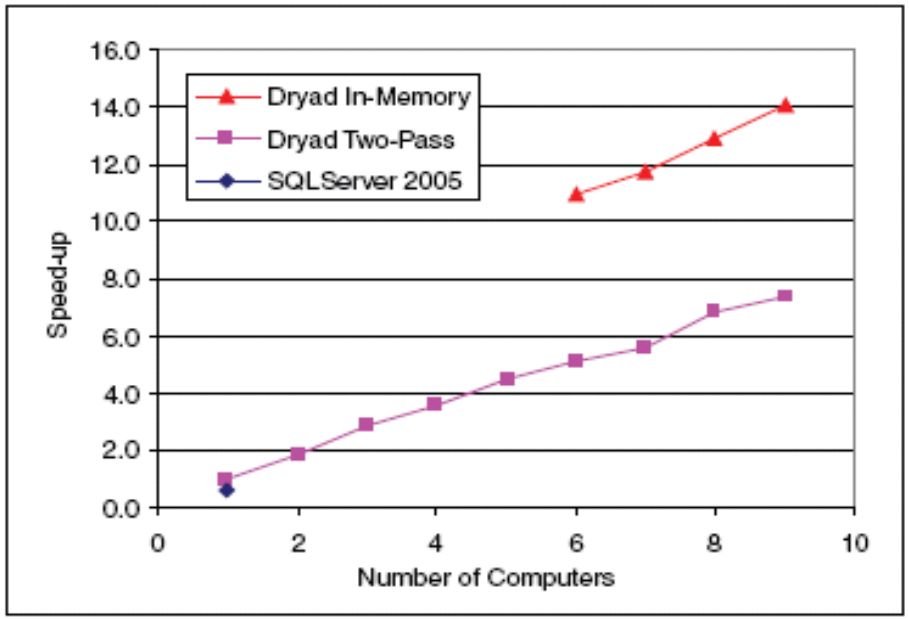


Figure 8: The speedup of the SQL query computation is near-linear in the number of computers used. The baseline is relative to Dryad running on a single computer and times are given in Table 2.

Another Example: Query histogram computation

- Input: log file (n partitions)
- Extract queries from log partitions
- Re-partition by hash of query (k buckets)
- Compute histogram within each bucket

Naïve histogram topology

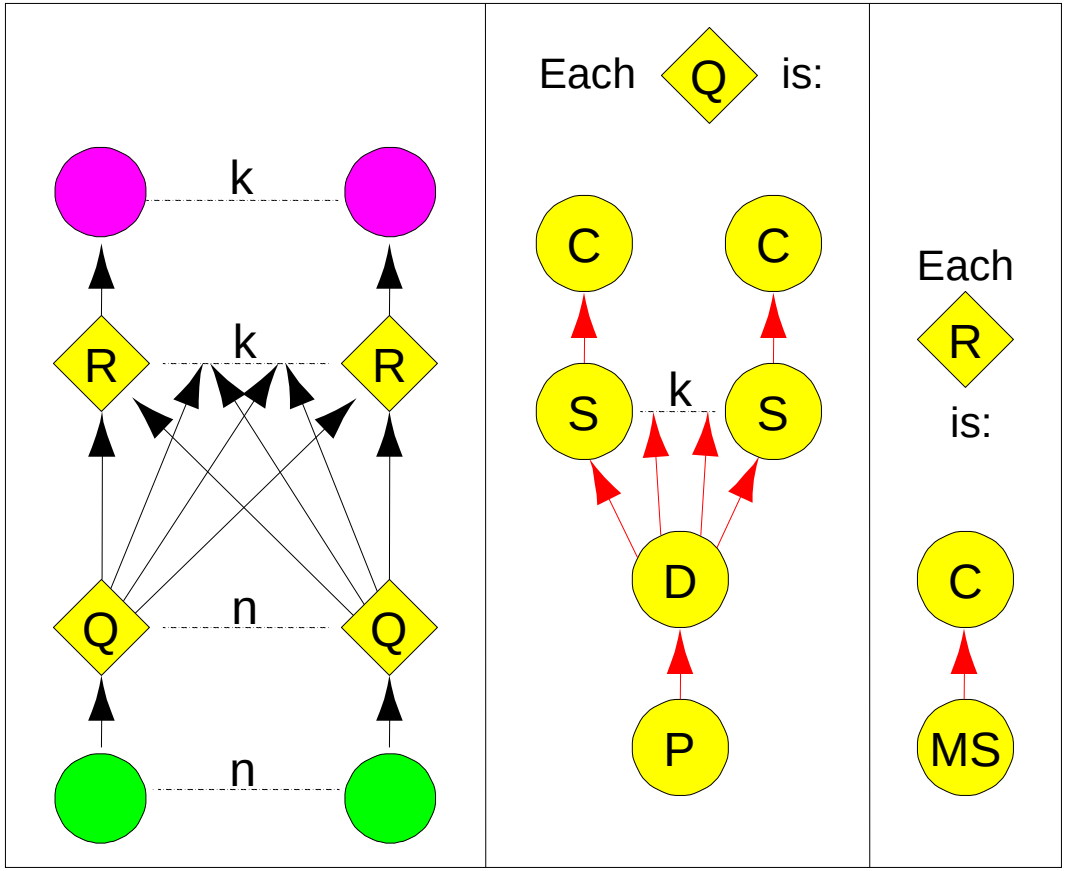
P parse lines

D hash distribute

S quicksort

C count occurrences

MS merge sort



Efficient histogram topology

P parse lines

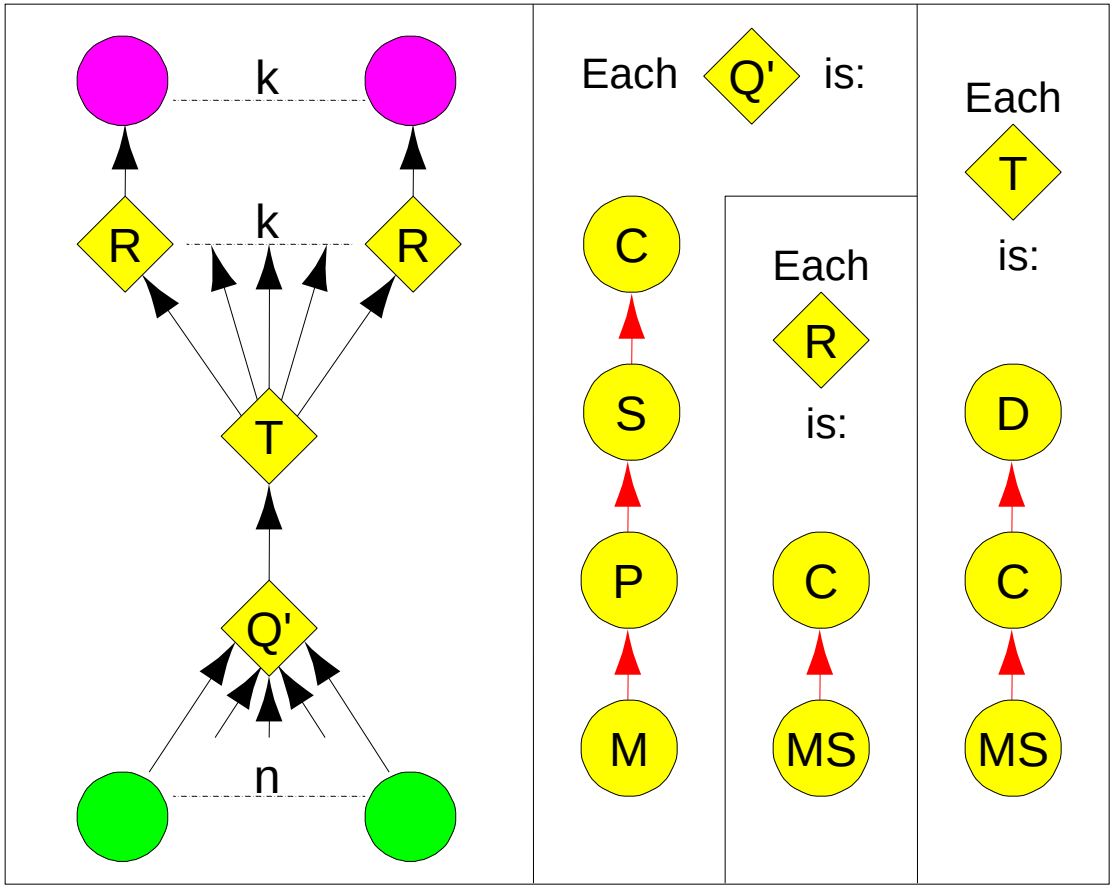
D hash distribute

S quicksort

C count occurrences

MS merge sort

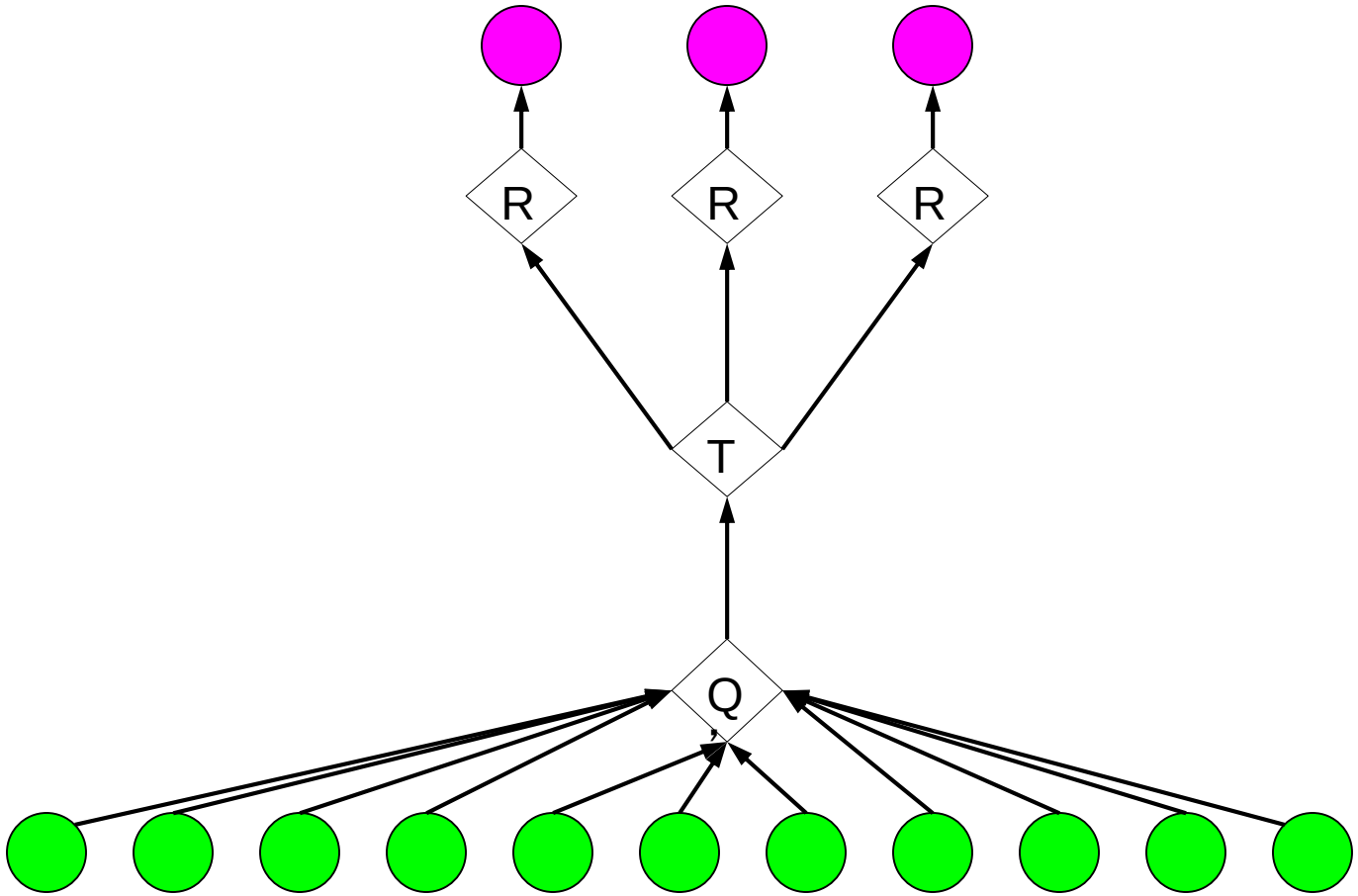
M non-deterministic merge



MS▸C

MS▸C▸D

M▸P▸S▸C



P parse lines

S quicksort

C count occurrences

D hash distribute

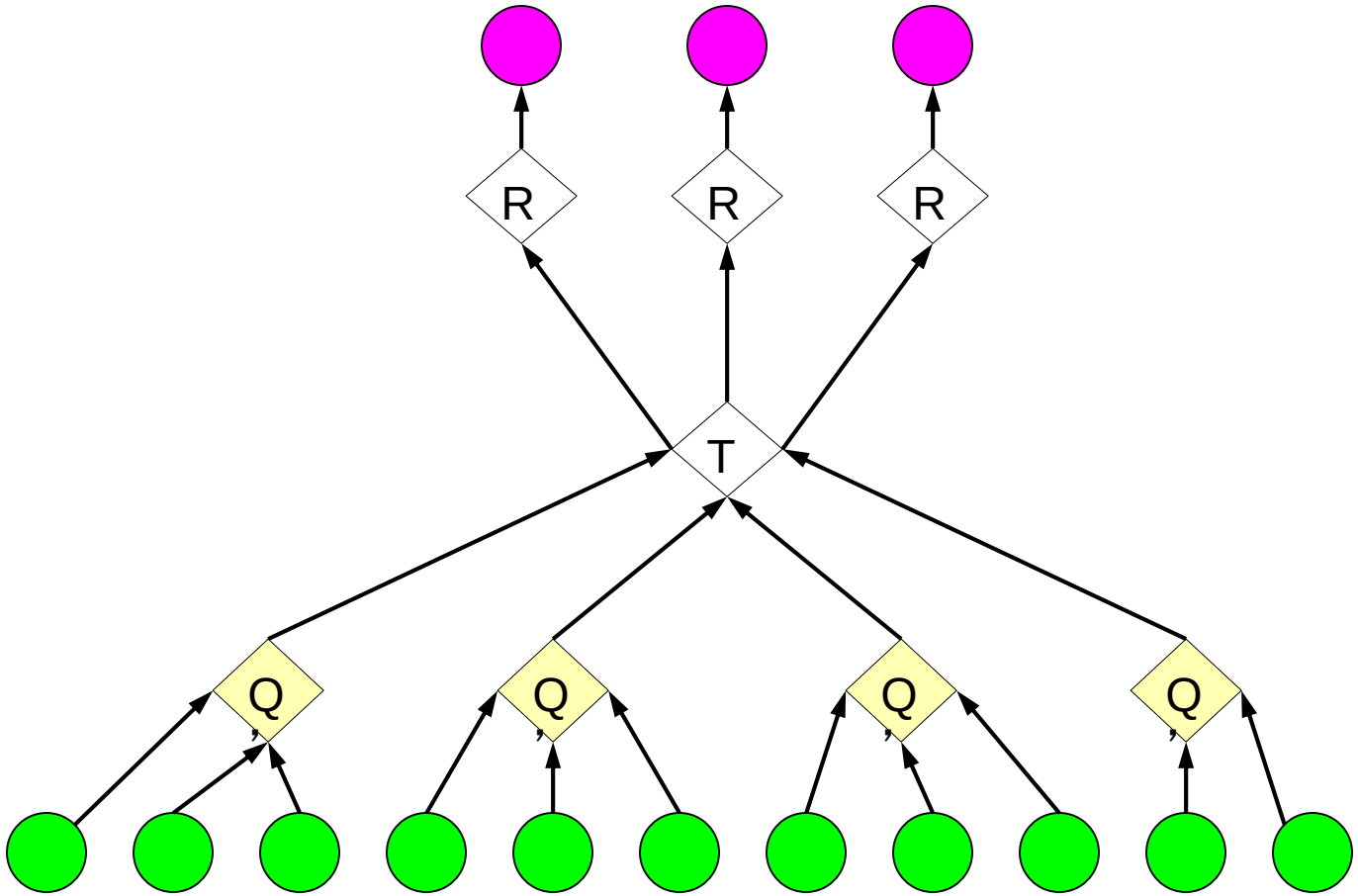
MS merge sort

M non-deterministic merge

MS▸C

MS▸C▸D

M▸P▸S▸C



P parse lines

S quicksort

C count occurrences

D hash distribute

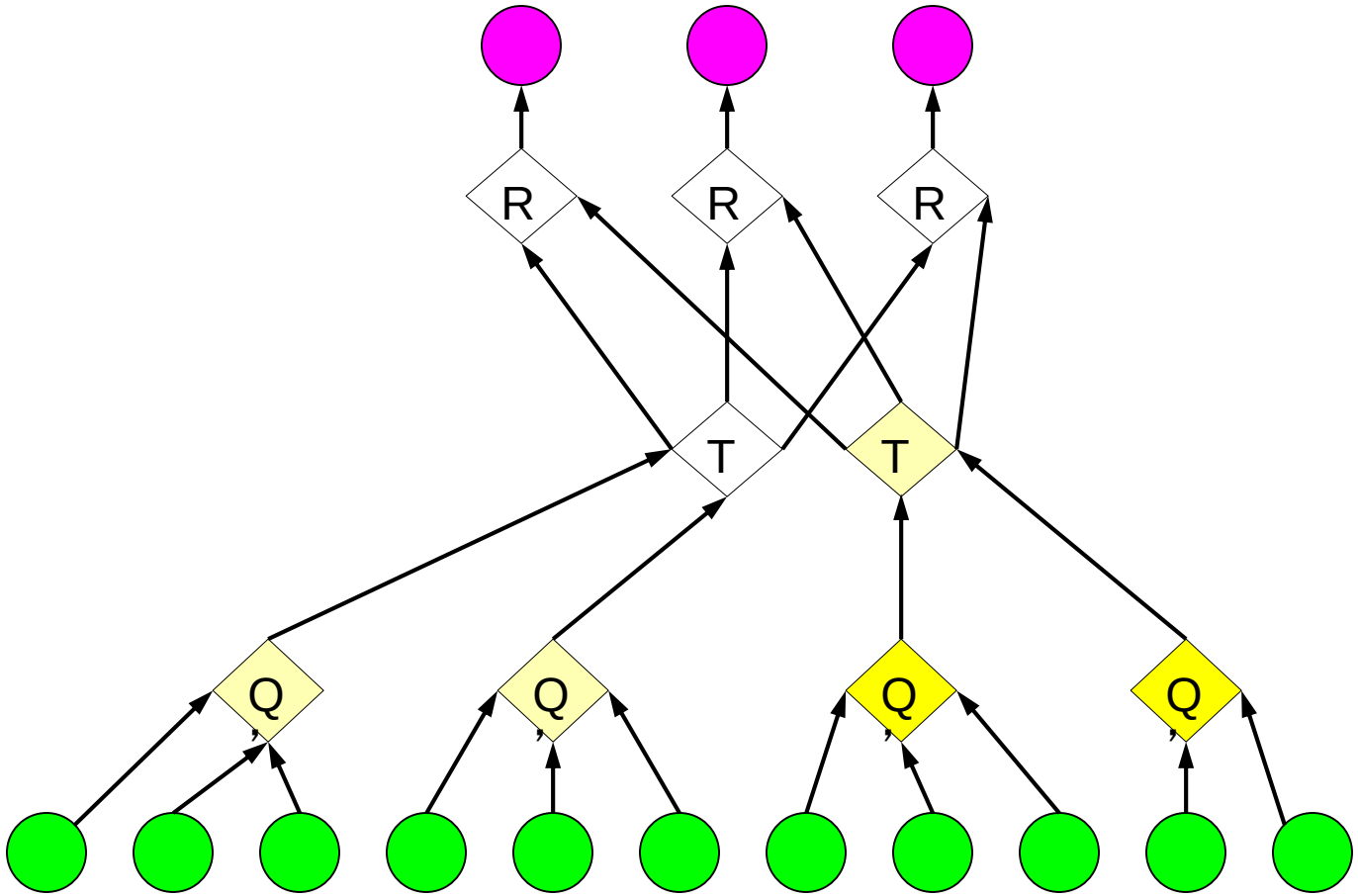
MS merge sort

M non-deterministic merge

MS▸C

MS▸C▸D

M▸P▸S▸C



P parse lines

S quicksort

C count occurrences

D hash distribute

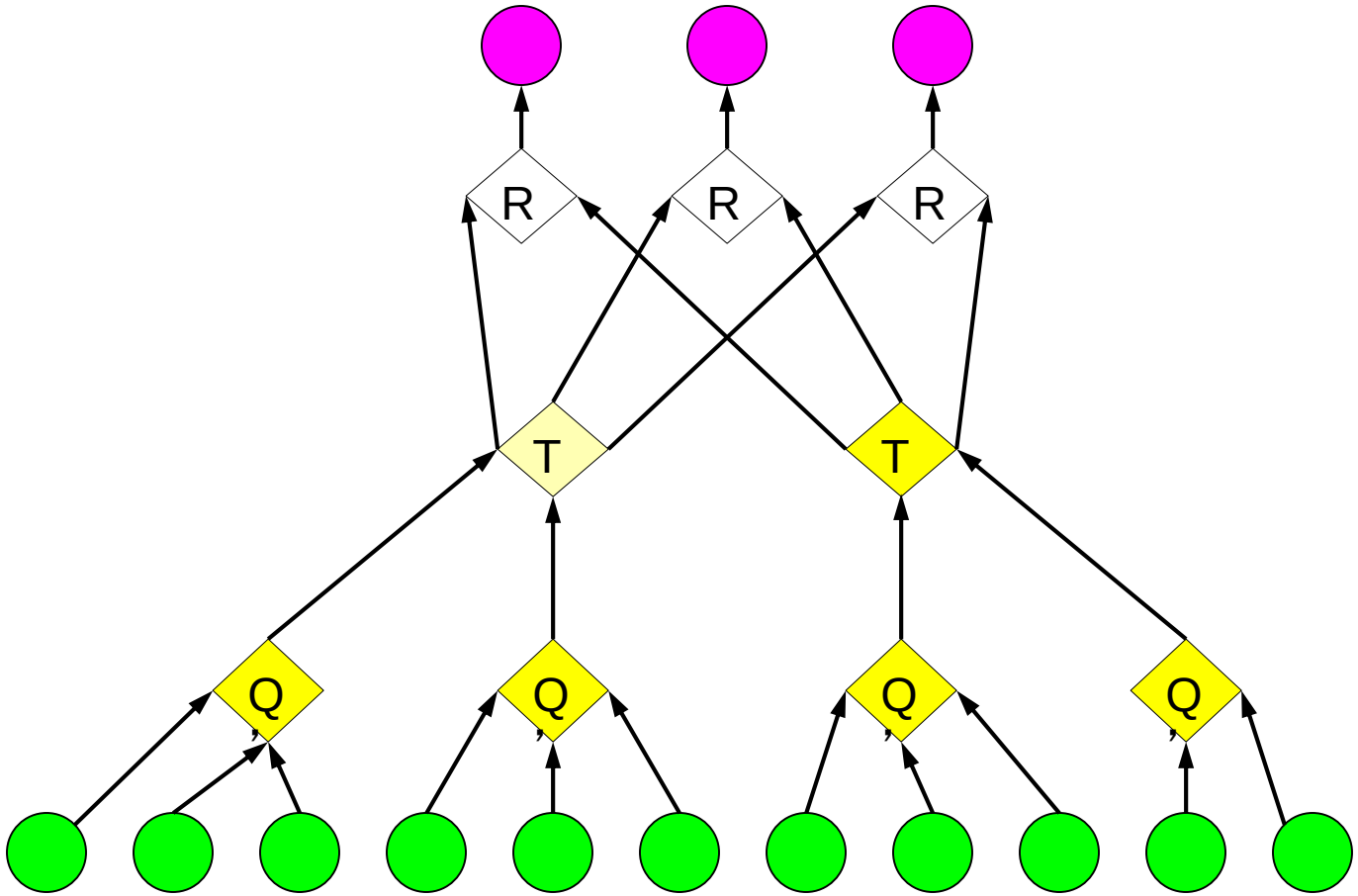
MS merge sort

M non-deterministic merge

MS▸C

MS▸C▸D

M▸P▸S▸C

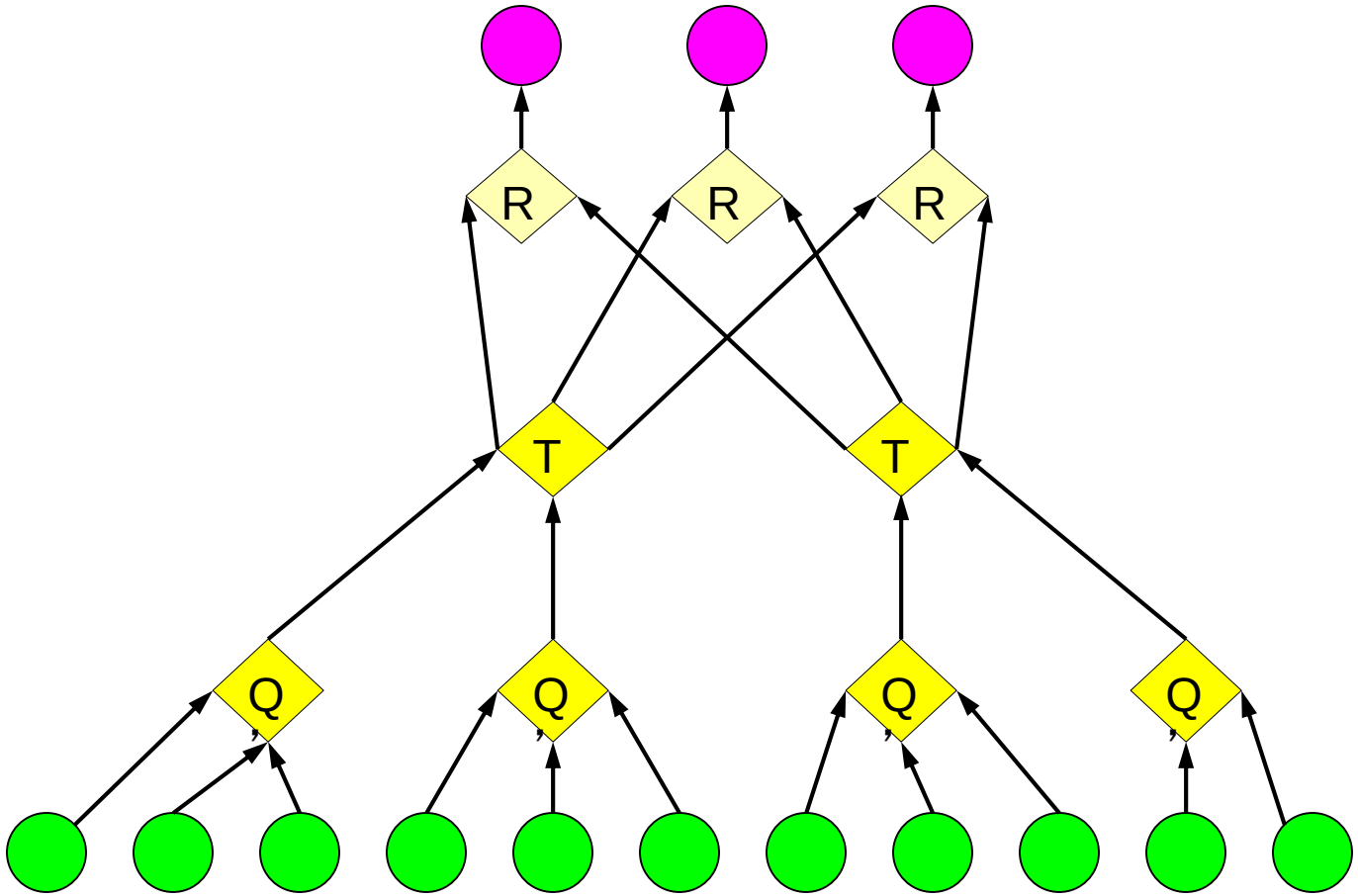


P	parse lines	D	hash distribute
S	quicksort	MS	merge sort
C	count occurrences	M	non-deterministic merge

MS▸C

MS▸C▸D

M▸P▸S▸C

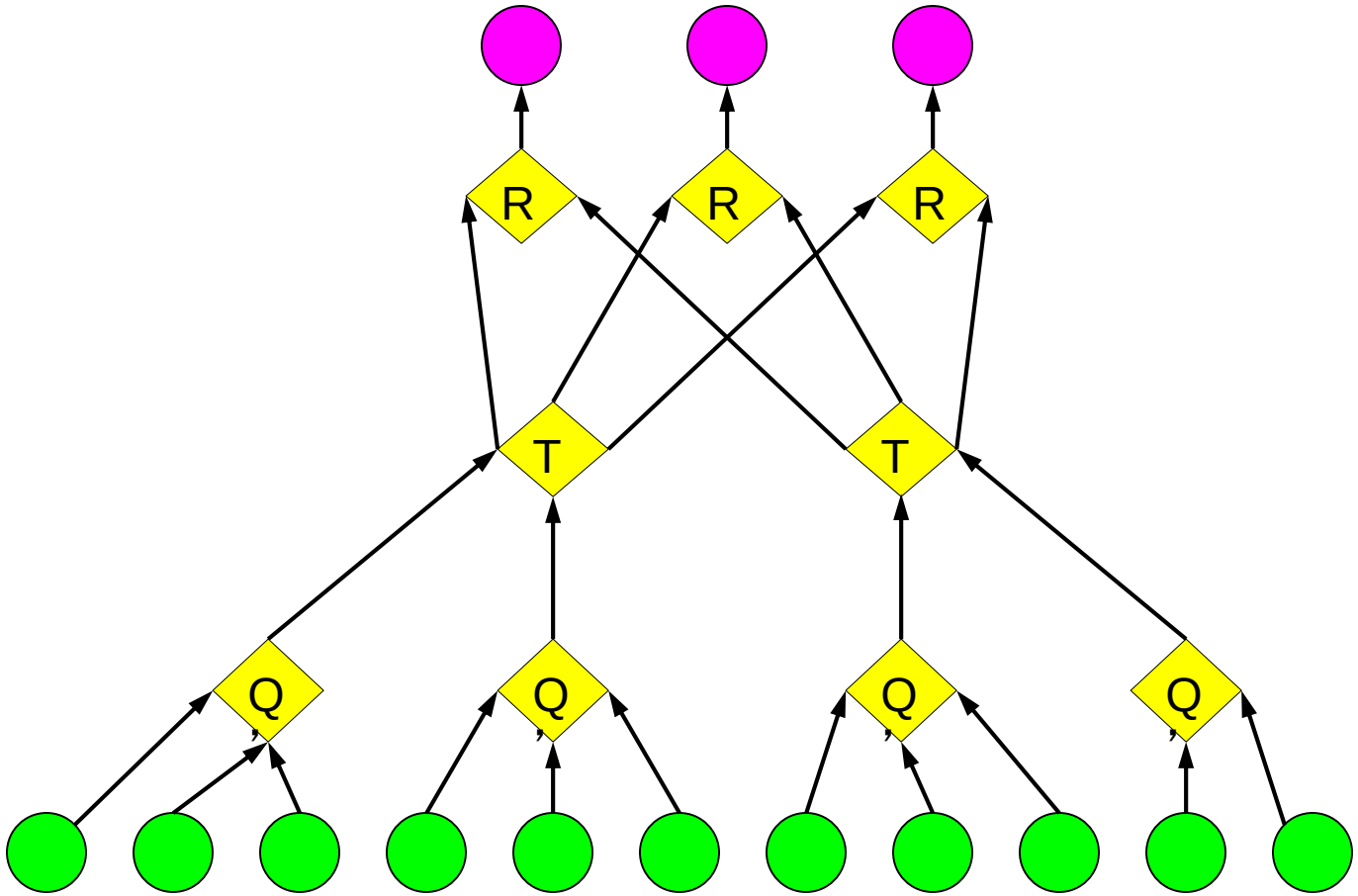


P	parse lines	D	hash distribute
S	quicksort	MS	merge sort
C	count occurrences	M	non-deterministic merge

MS▸C

MS▸C▸D

M▸P▸S▸C



P	parse lines	D	hash distribute
S	quicksort	MS	merge sort
C	count occurrences	M	non-deterministic merge

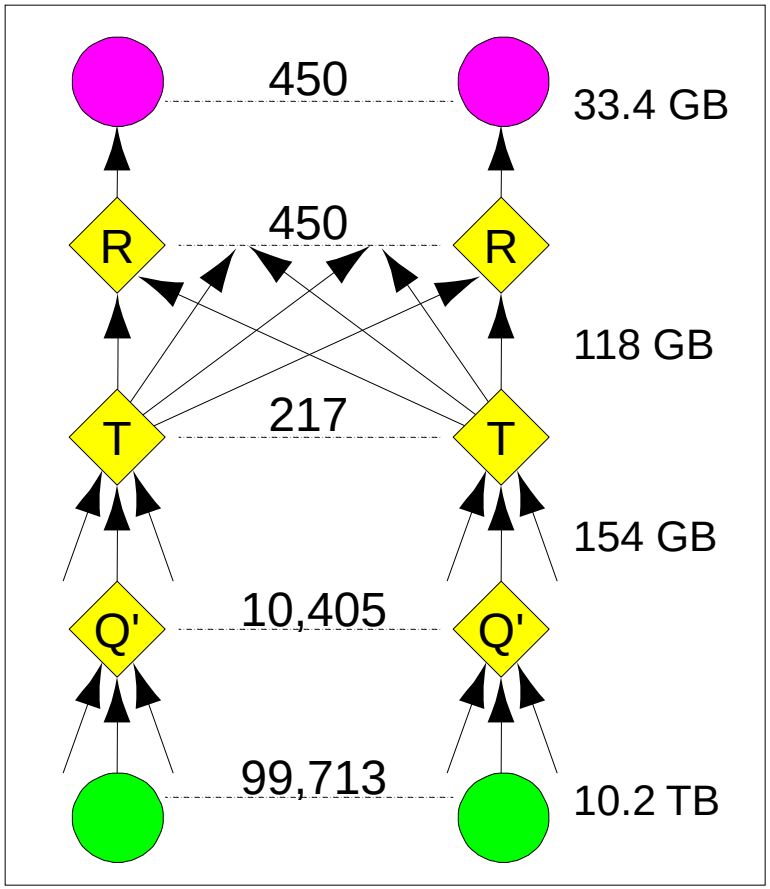
Final histogram refinement

1,800 computers

43,171 vertices

11,072 processes

11.5 minutes

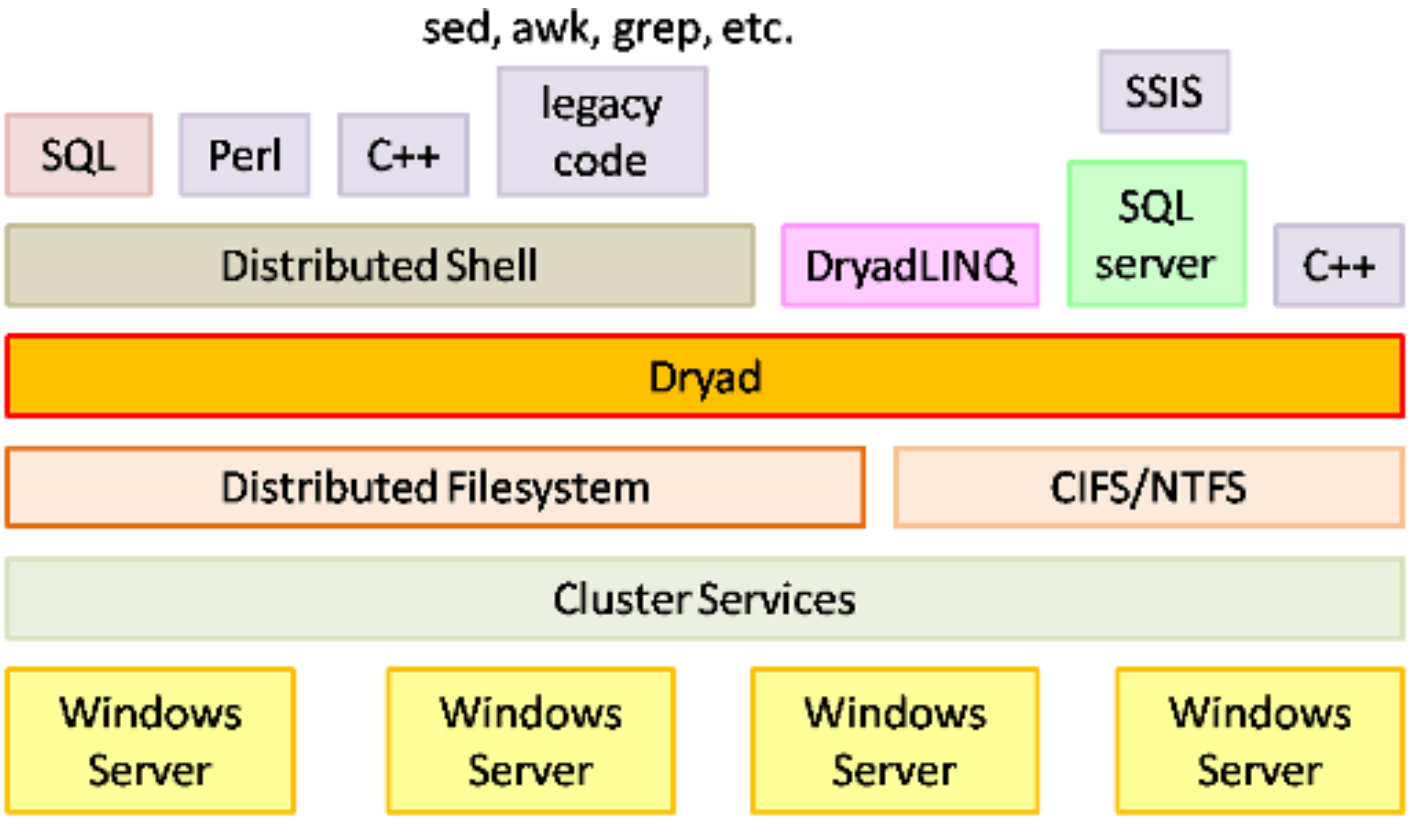


How does Dryad fit in?

- Target Users/Developers of the "Raw" Dryad middleware:
  - Experienced C++ developer
  - Can write good single-threaded code
  - Wants generality, can tune performance

- Higher-level front ends for broader audience:
  - Many programs can be represented as a distributed execution graph
  - The programmer may not have to know this
    - "SQL-like" queries: LINQ
    - High-level queries compiled into DAGs automatically and Dryad will run them for you

The Dryad Ecosystem



- Well-tested at scales up to 15k cluster computers
  - In heavy (Microsoft in-house) production use for 8 years

Integrated system

- Collection-oriented programming model (LINQ)
- Partitioned file system (TidyFS)
  - Manages replication and distribution of large data
- Cluster scheduler (Quincy)
  - Jointly schedule multiple jobs at a time
  - Fine-grain multiplexing between jobs
  - Balance locality and fairness
- Monitoring and debugging (Artemis)
  - Within job and across jobs

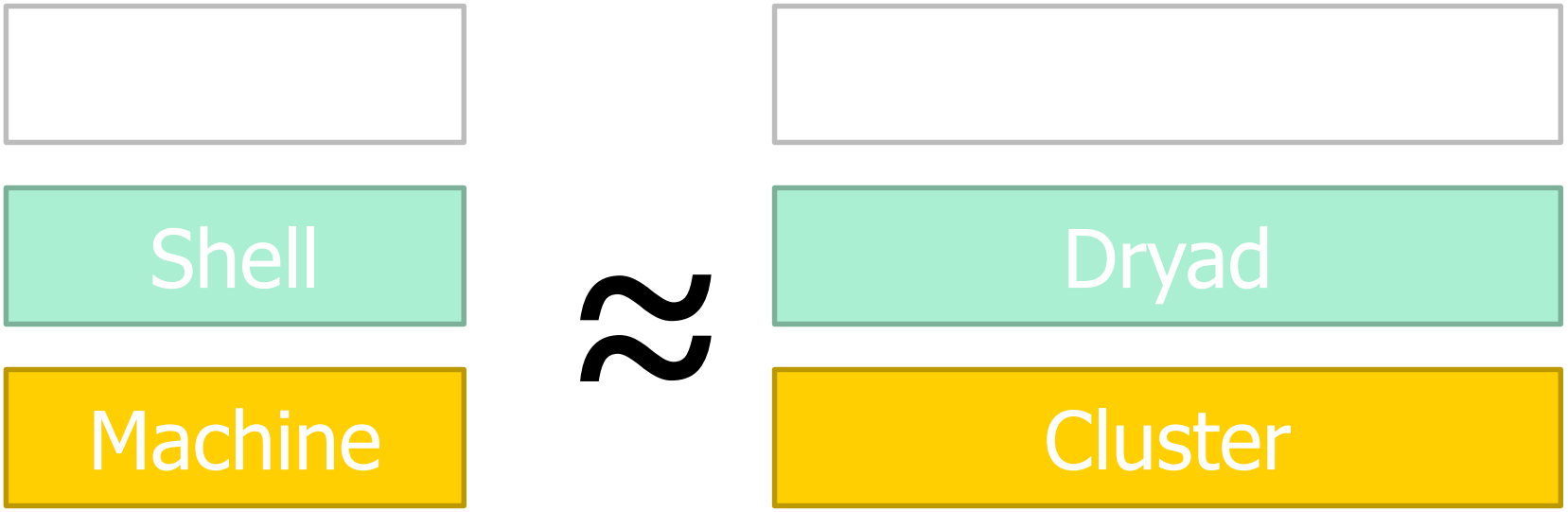
Dryad vs. DryadLINQ

- Dryad provides a low-level parallel data flow processing interface
  - Acyclic data flow graphs
  - Data communication methods include pipes, file-based, message, shared-memory

- DryadLINQ
  - A high level language for app developers
  - It hides the data flow details

LINQ and DryadLINQ

Dryad = Execution Engine



Prior High-level Query Language over Dryad

- Nebula – limited to existing binaries
- Scope – SQL-ish, not general purpose
- Is it possible to do better?
  - Can one get the general purpose-ness of C#/Java and conciseness of SQL?
  - And at the same time, be efficient too?

Language Integrated Query (LINQ)

Language Integrated Query (LINQ)

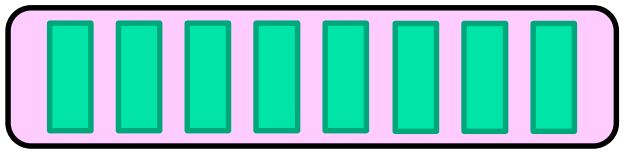
- Microsoft's **L**anguage **IN**tegrated **Q**uery
  - Available in Visual Studio 2008
- The creamy goodness of SQL-like queries within a declarative programming model
- Basic abstraction – collections

"All the world's a *collection*, And all the men and women merely *iterate on collections"* - implied by Shakespeare

- A set of operators to manipulate datasets in .NET
  - Support traditional relational operators
    - Select, Join, GroupBy, Aggregate, etc.
- Data model
  - Data elements are strongly typed .NET objects
  - Much more expressive than SQL tables
- Extensible
  - Add new custom operators
  - Add new execution providers

Collections, Iterators and LINQ

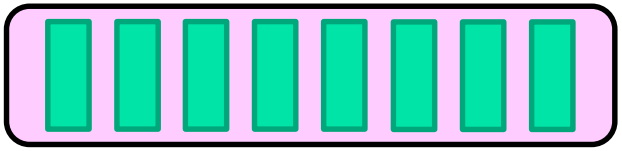
IEnumerable <T>



=>

```
List<int> result = new List<int>();
foreach (int num in numbers) {
    if (num % 2 == 0)
        result.Add(num);
}
result.sort();
```

IEnumerable <T>



\+

LINQ

=>

```
import system.linq;
var result = from num in numbers
                                where num
% 2 == 0
                                orderby
num
                                select
num;
```

Syntactical sweetness of LINQ

Query Style

```
var result = from num in numbers
                                    where num % 2
== 0
                                    orderby num
                                    select num;
```

Method Style

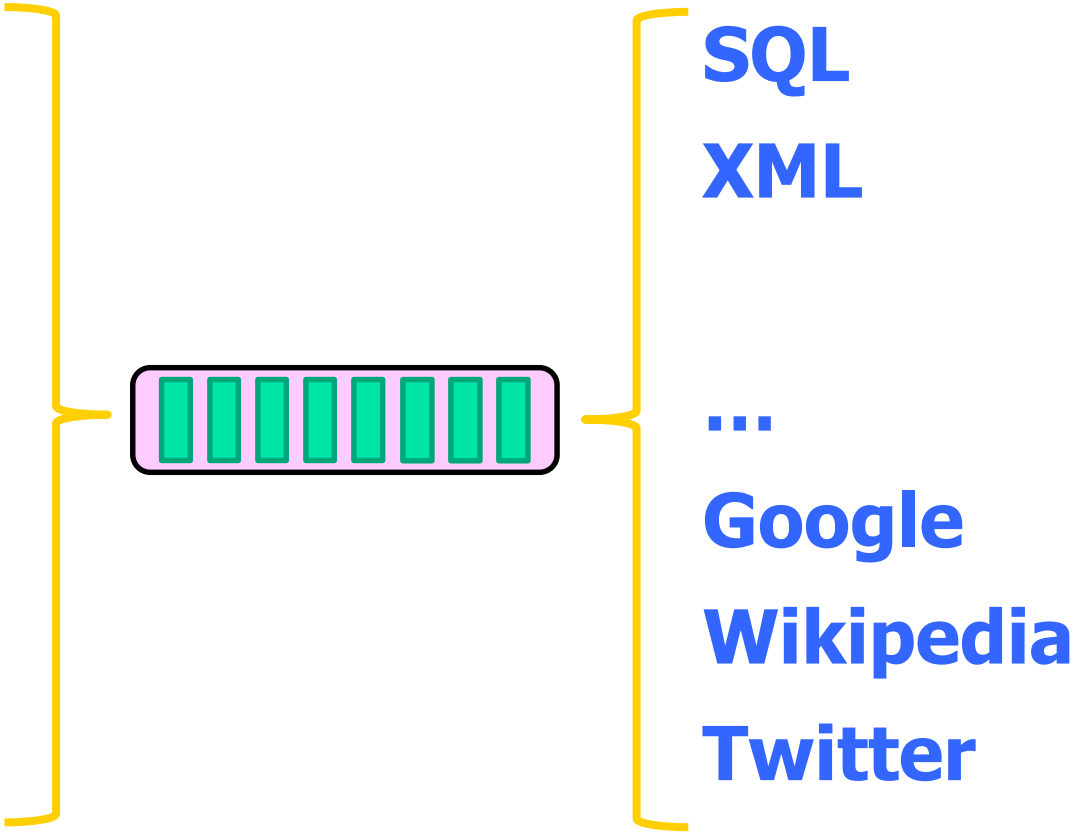
```
var result =
                        numbers.Where(num => num % 2 == 0)
                               .OrderBy(n => n);
```

LINQ Functionality

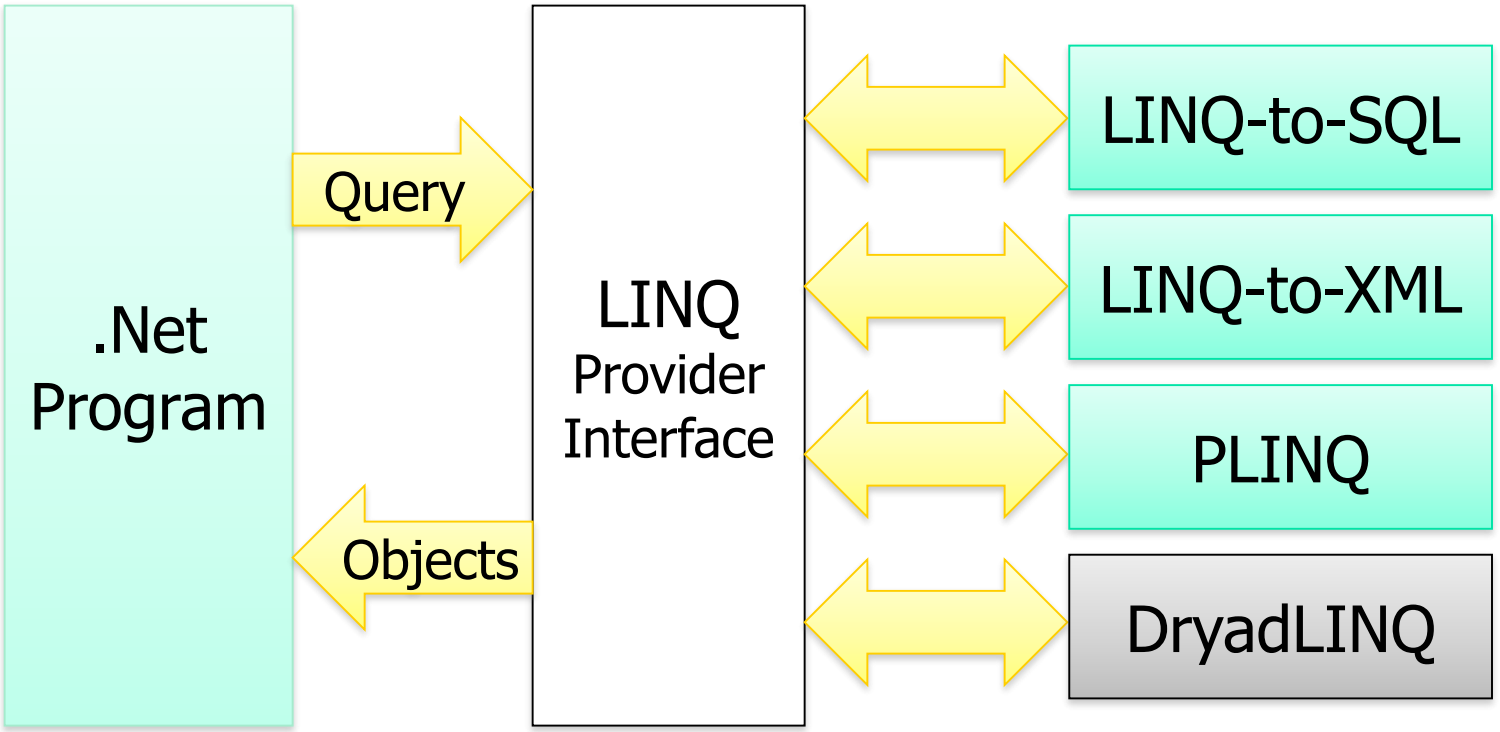
Select / SelectMany	Map (1-to-1 / 1-to-many)
Where	Filter
GroupBy	Reduce
OrderBy	Sort
Join	Join
Union / Intersect / Except	Set operations
…	

LINQ Providers

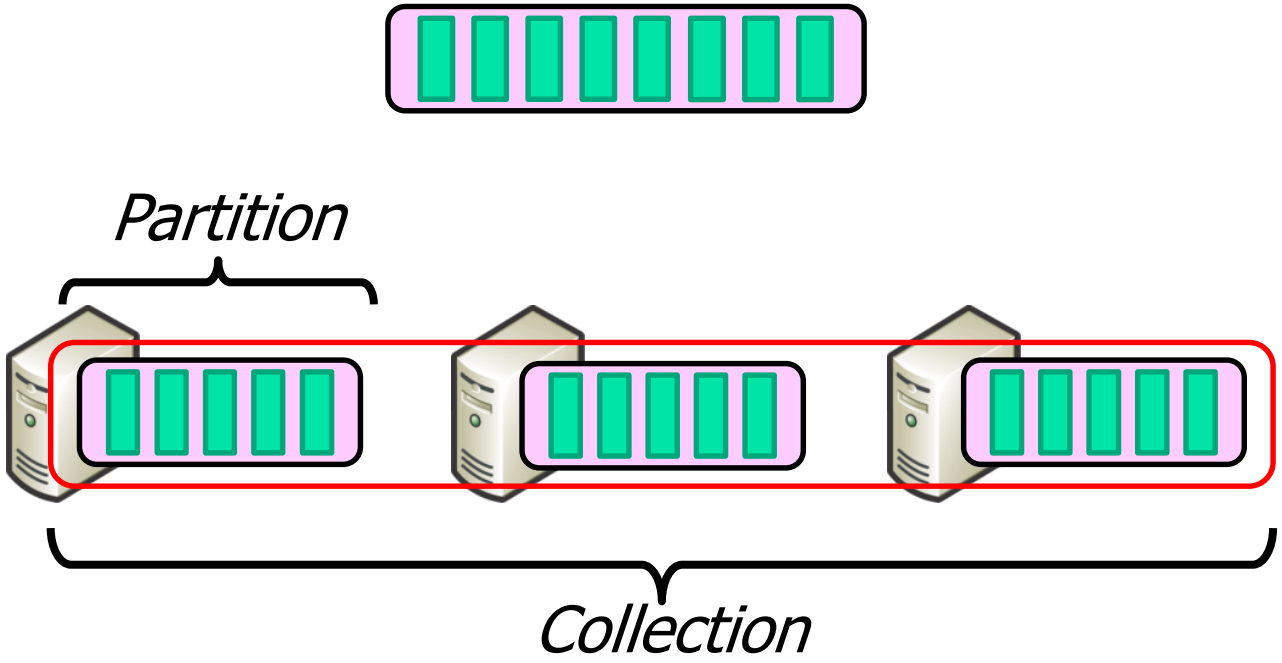
- Select / SelectMany
- Where
- GroupBy
- OrderBy
- Join
- Union / Intersect / Except
- …



LINQ System Architecture

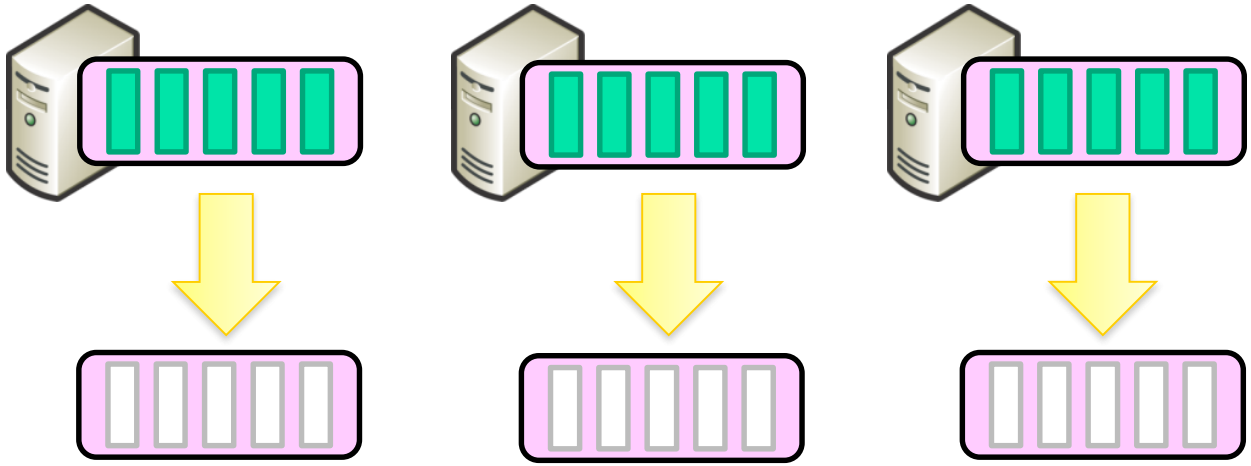


Parallel Collections



Dryad + LINQ = DryadLINQ

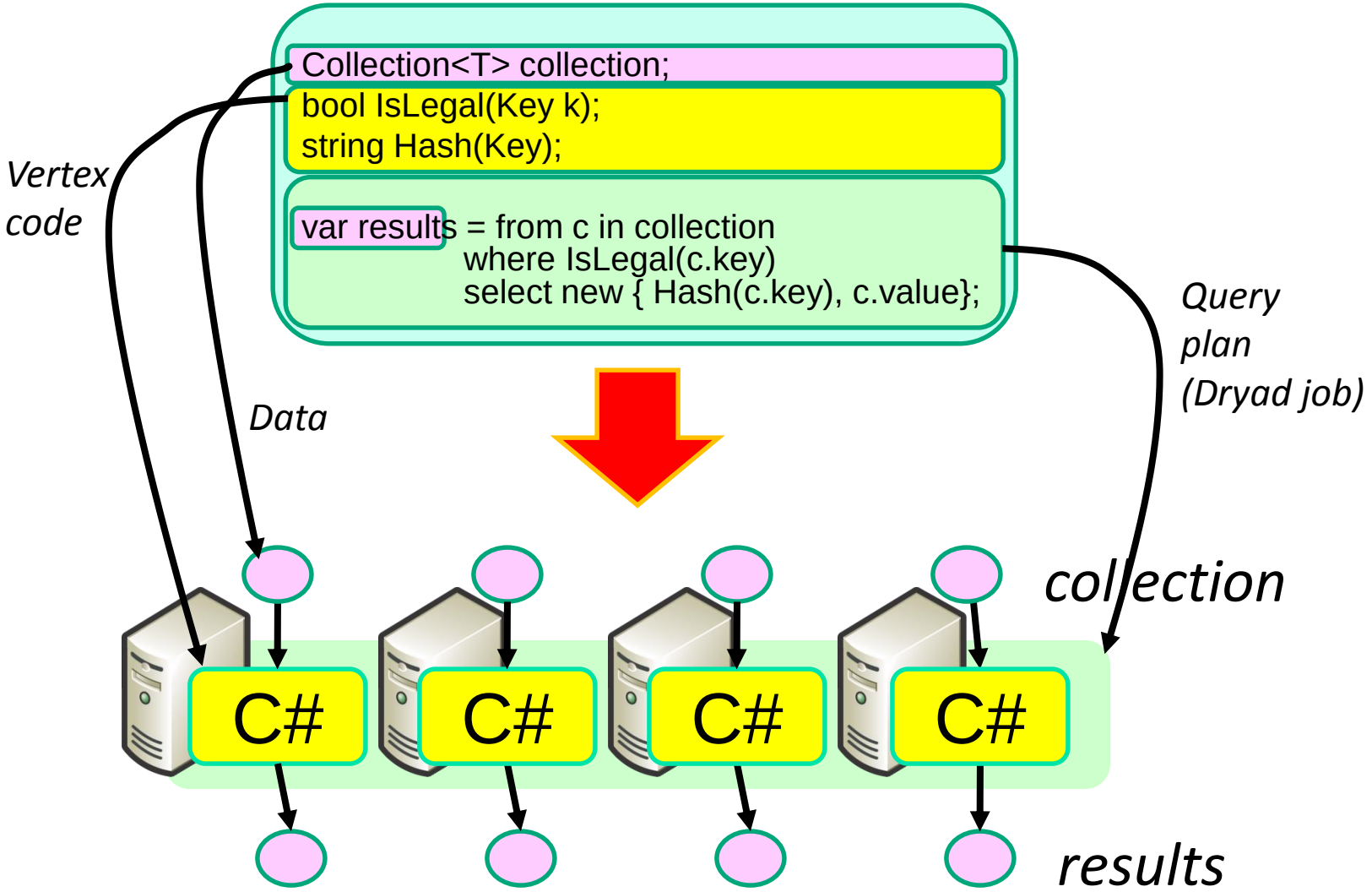
```
string uri = @"file://\\machine\directory\input.pt";
PartitionedTable<LineRecord> input =

        PartitionedTable.Get<LineRecord>(uri);

var lengths = input.Select(line => line.ToString().Length);
```



LINQ

```
Collection<T> collection;
bool IsLegal(Key);
string Hash(Key);
var results = from c in collection
              where IsLegal(c.key)
              select new { Hash(c.key), c.value};
```

DryadLINQ = LINQ + Dryad

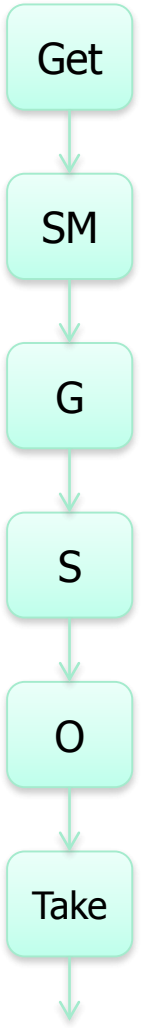


DryadLINQ

- LINQ: Relational queries integrated in C#
- More general than distributed SQL
  - Inherits flexible C# type system and libraries
  - Data-clustering, EM, inference, …
- Uniform data-parallel programming model
  - From SMP to clusters

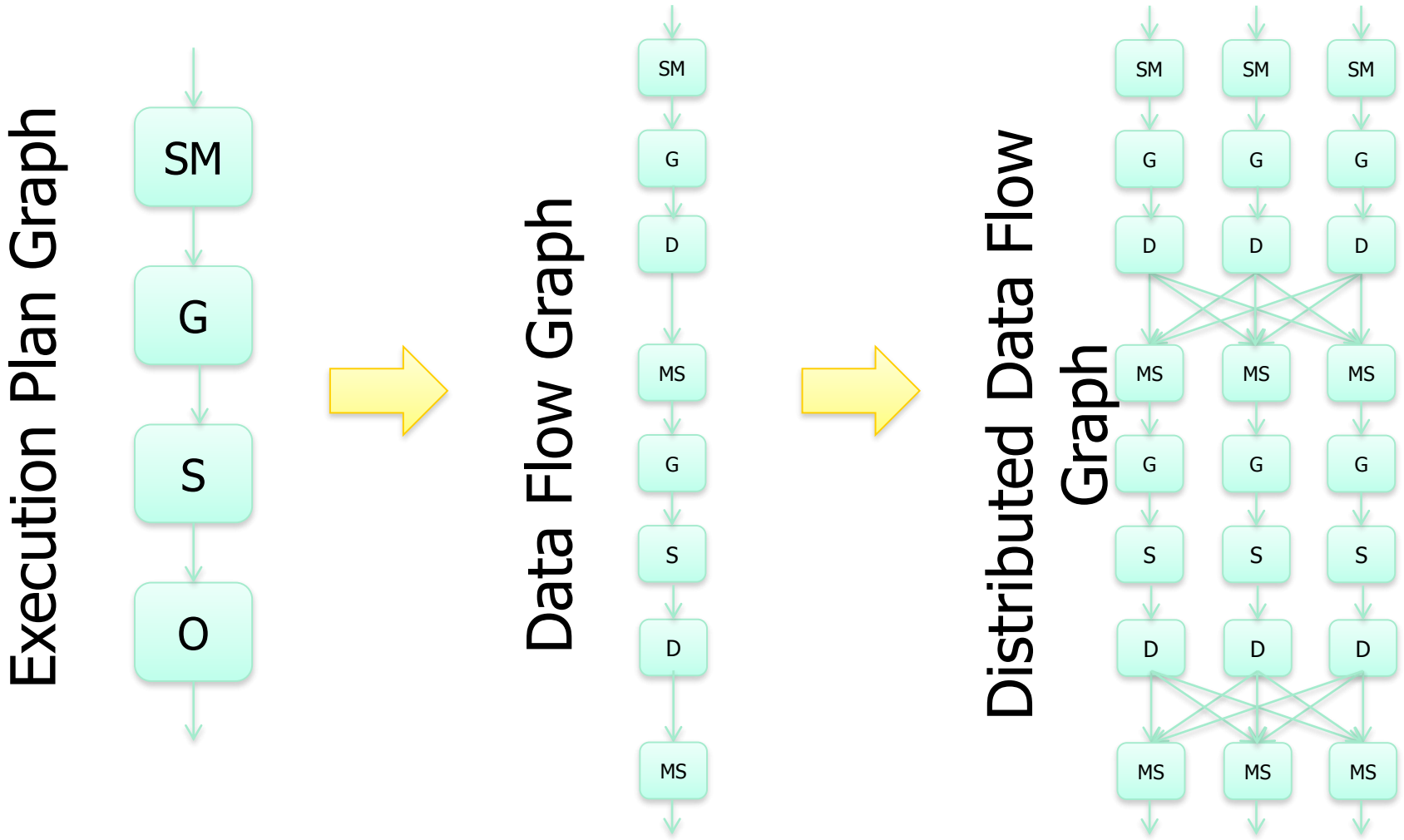
Word Count with DryadLINQ

```
string uri = @"file://\\machine\directory\input.pt";
PartitionedTable<LineRecord> input =

        PartitionedTable.Get<LineRecord>(uri);

string separator = ",";
var words = input.SelectMany(x => SplitLineRecord(separator));

var groups = words.GroupBy(x => x);

var counts = groups.Select(x => new Pair(x.Key, x.Count()));

var ordered = counts.OrderByDescending(x => x[2]);

var top = ordered.Take(k);

top.ToDryadPartitionedTable("matching.pt");
```



Execution Plan Graph

DryadLINQ Word Count →Dryad



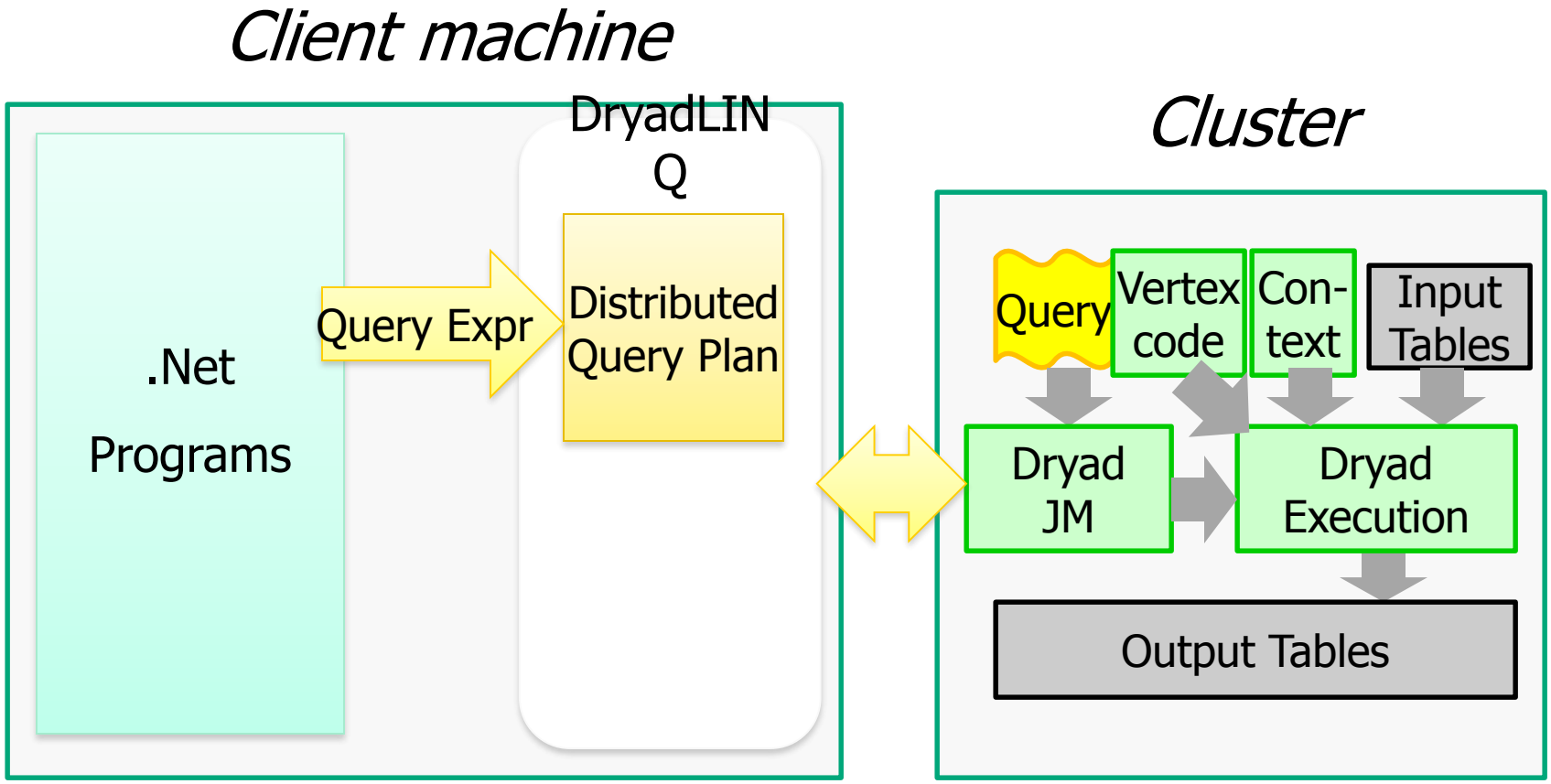
DryadLINQ Code Generation

```
string uri = @"file://\\machine\directory\input.pt";
PartitionedTable<LineRecord> input =
        PartitionedTable.Get<LineRecord>(uri);

string separator = ",";
var words = input.SelectMany(x => SplitLineRecord(separator));

var groups = words.GroupBy(x => x);

var counts = groups.Sel

var ordered = counts.Or

var top = ordered.Take(

top.ToDryadPartitionedT
```

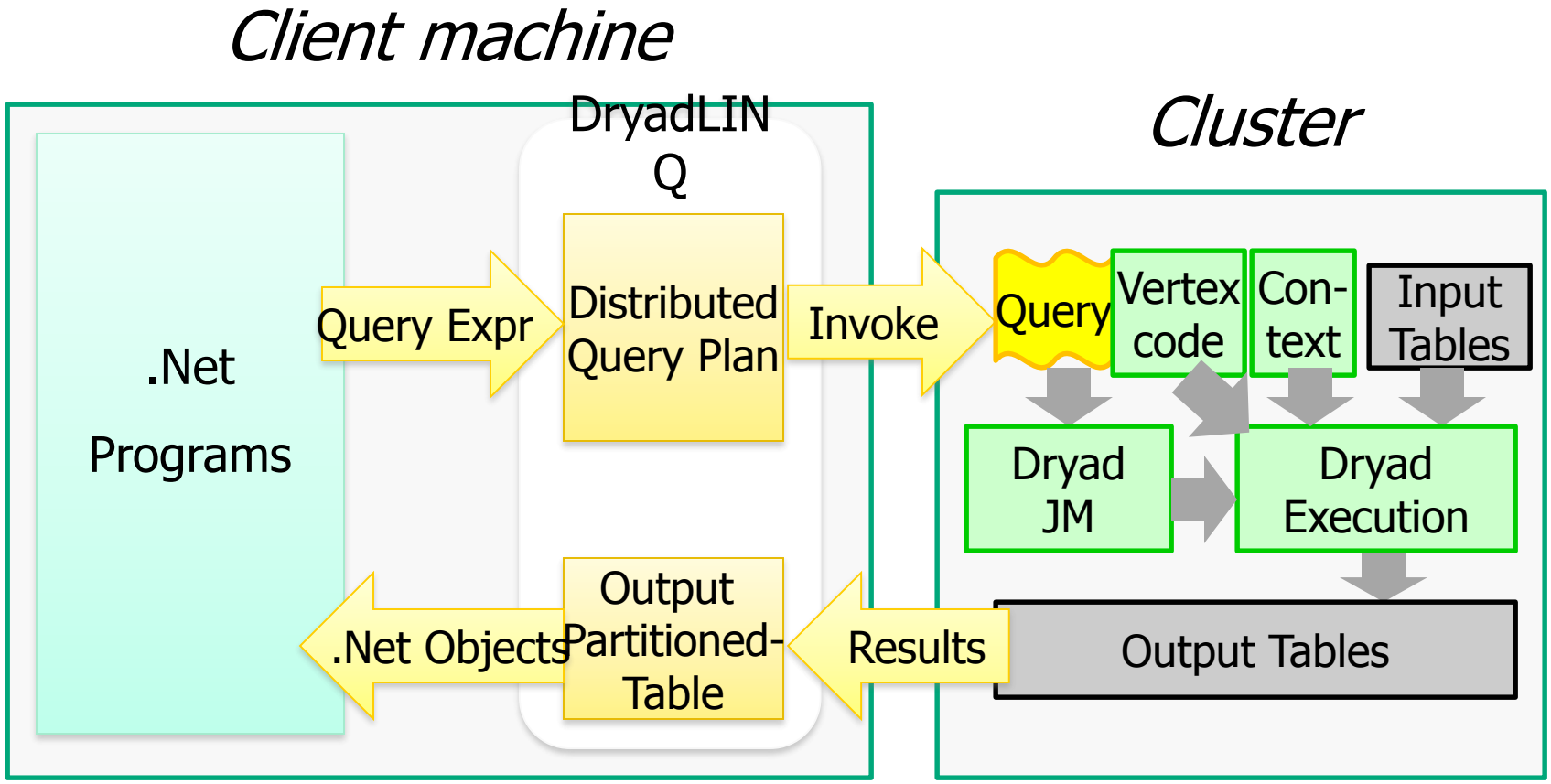
Conversion of subexpressions to code for Dryad vertices…

1. Local variables
2. Local libraries and functions

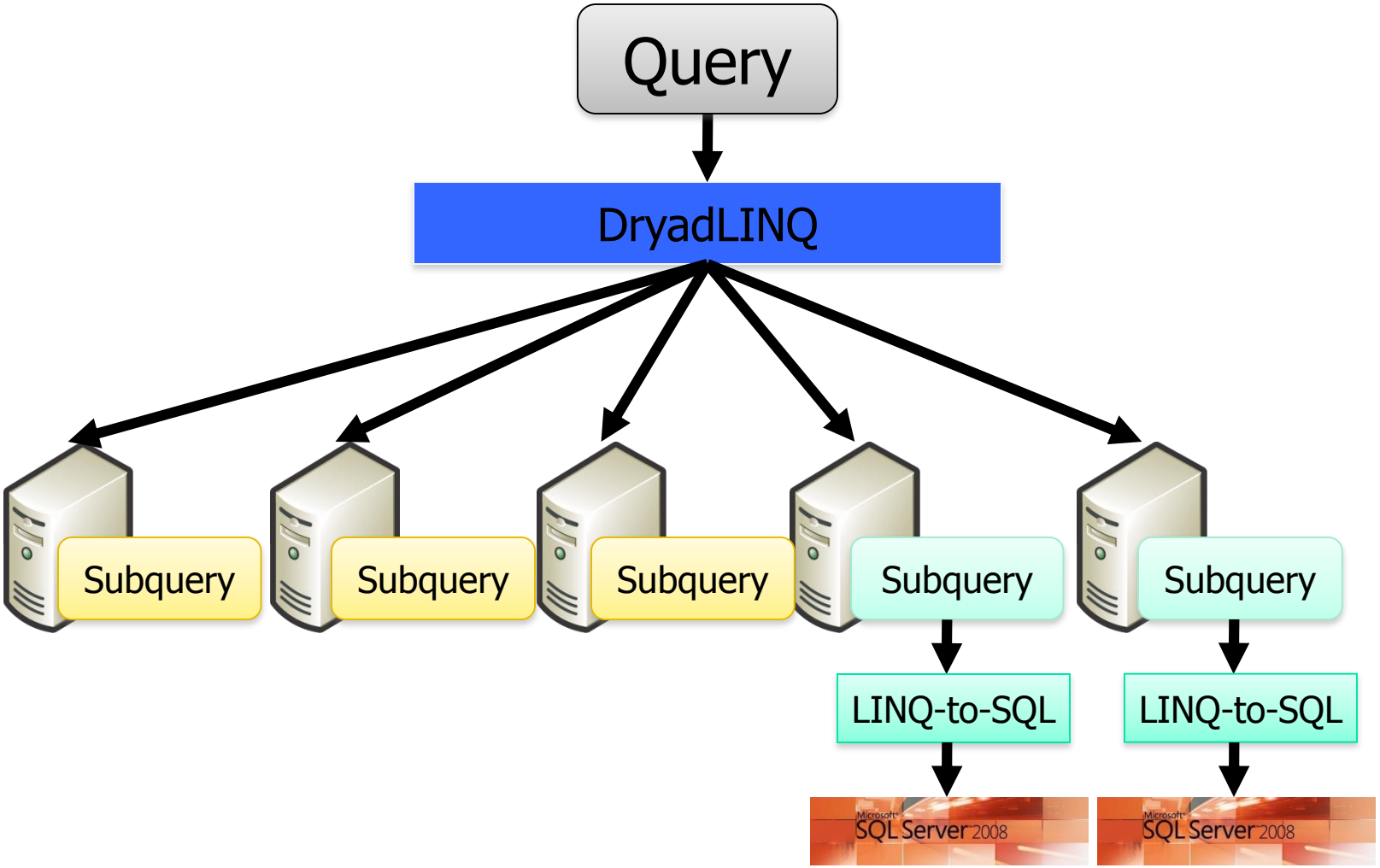
DryadLINQ Architecture



DryadLINQ Architecture (cont'd)



Combining with LINQ-to-SQL



DryadLINQ Optimizations

- Some are similar to existing DB optimizations
  - Eliminate redundant partitioning steps
  - Aggregation steps moved up the graph, before partitioning steps
- Existing Dryad optimizations as well
  - Dynamic reconfiguration of aggregation trees

Summary of Dryad & DryadLINQ

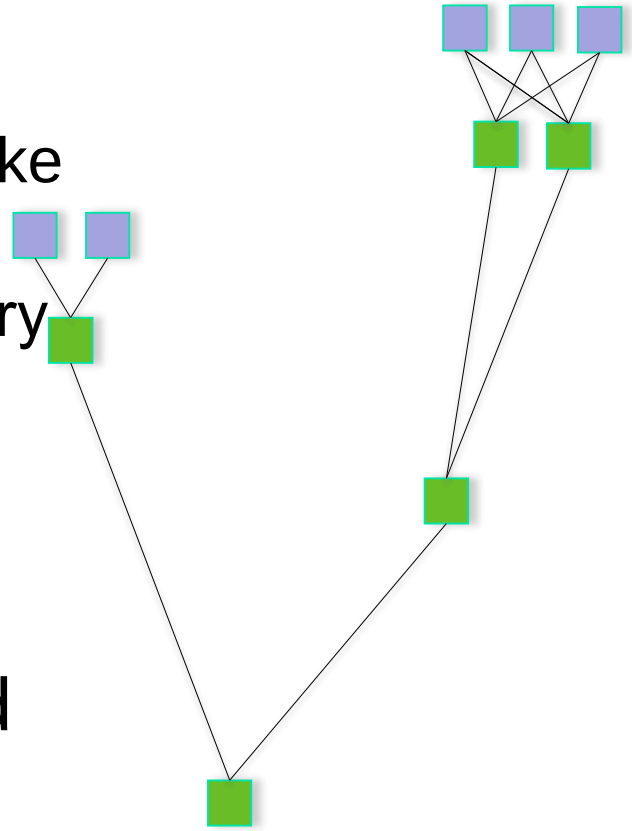
- General-purpose platform for scalable distributed data-processing of all sorts
- Very flexible
  - Optimizations can get more sophisticated
- Designed to be used as middleware
  - Slot different programming models on top, e.g.

e.g. DryadLINQ is a designed as a high-level query language to support the running of LINQ over High-Performance Clusters (HPC)

Apache Tez

Apache Tez – Introduction

- Distributed execution framework targeting data-processing applications.
  - NOT a standlone computation engine like MapReduce or Spark ; Instead, it is intended to be use as a “backend” library
- Based on expressing a computation as a DAG dataflow graph.
  - Claim to be inspired by Dryad
- Highly customizable to meet a broad spectrum of use cases.
- Built on top of YARN – the resource management framework for Hadoop.



Hadoop 1 -> Hadoop 2

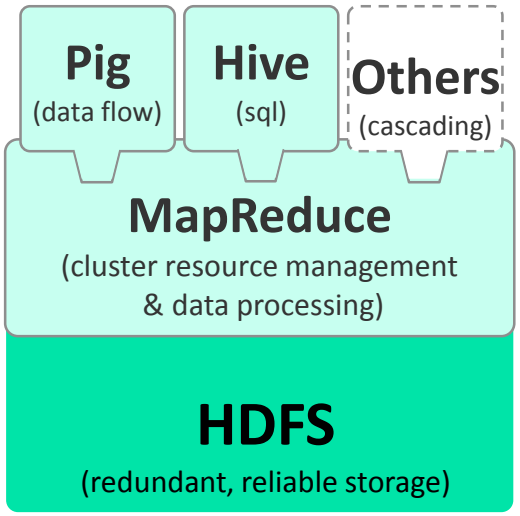
Monolithic

- Resource Management
- Execution Engine
- User API

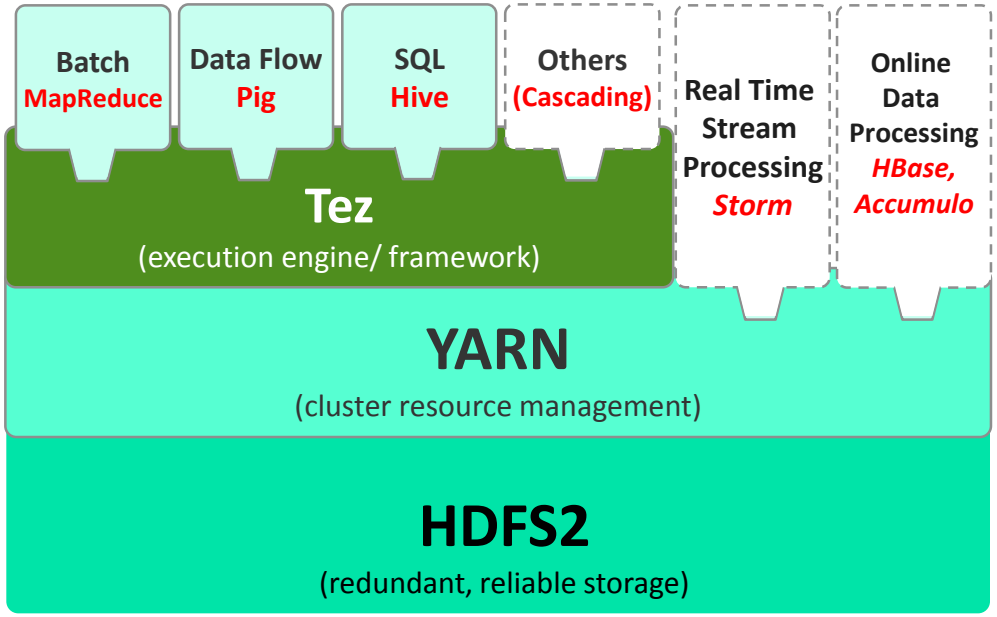
Layered

- Resource Management – YARN
- Execution Engine – Tez
- User API – Hive, Pig, Cascading, Your App, even experimental support for MapReduce and Spark !!

HADOOP 1.0



HADOOP 2.0



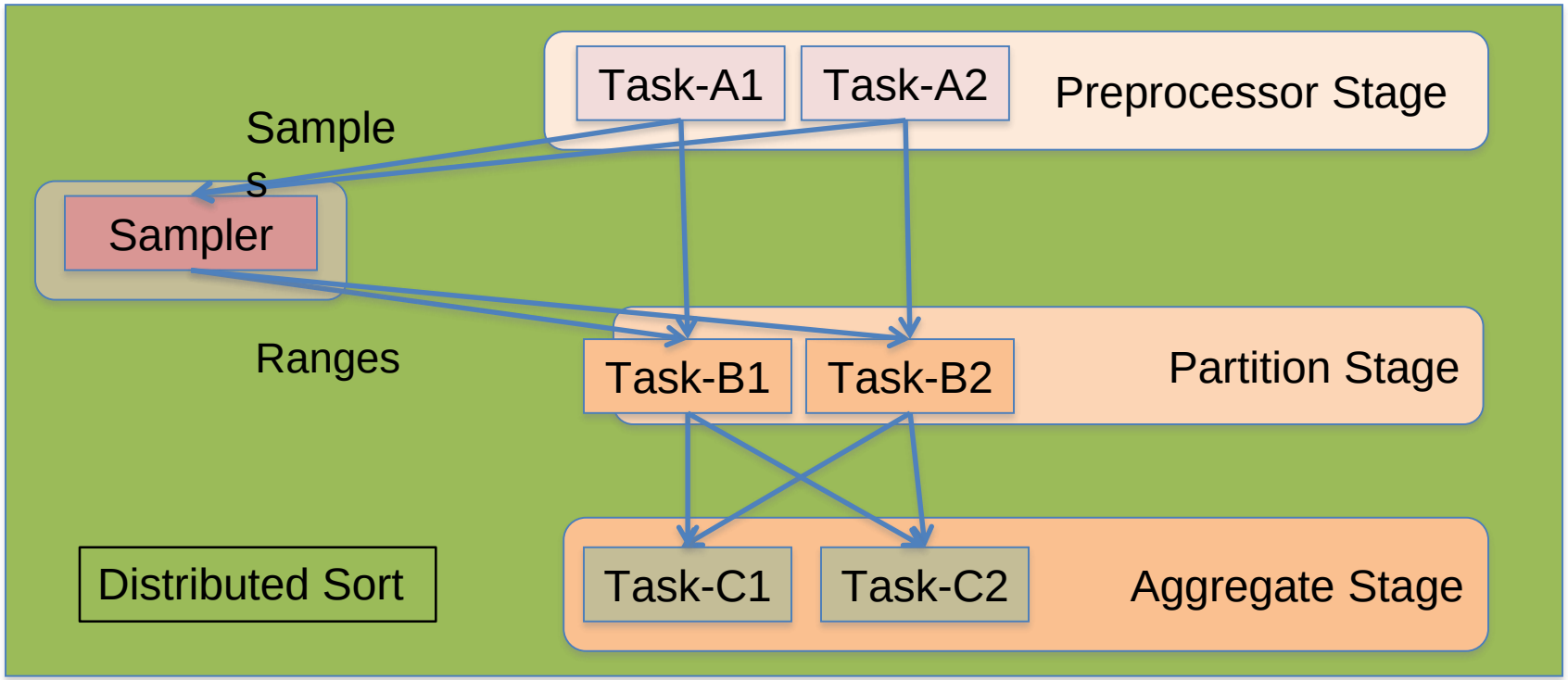
Tez – Design Themes

- Expressing the computation via Dataflow APIs
  - Direct representation of the data processing flow
- Flexible Input-Processor-Output runtime model
- Data Type Agnostic
- Performance Optimizations
  - Late Binding : Make decisions as late as possible using real data from at runtime
  - Leverage the resources of the cluster efficiently
  - Customizable engine to let applications tailor the job to meet their specific requirements
- Simplifying Operations
  - Facilitate installation, operations, testing, experiments and upgrade

Tez – Expressing the computation

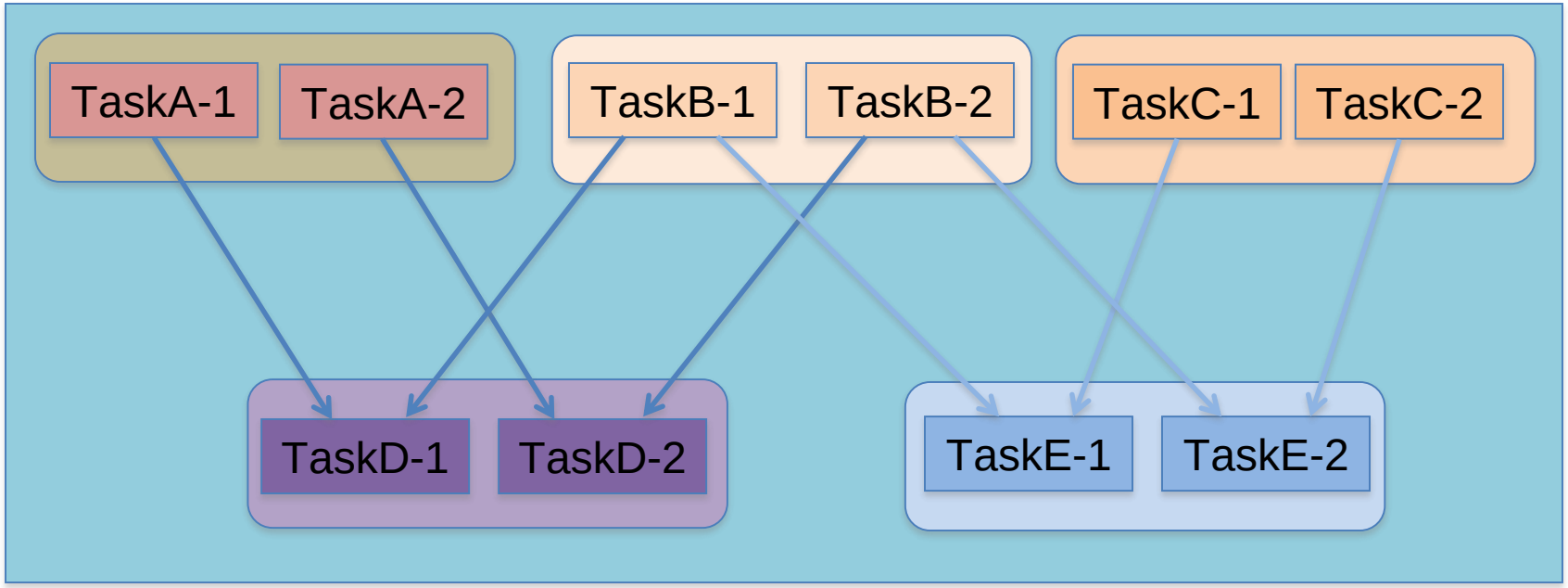
Distributed data processing jobs typically look like DAGs (Directed Acyclic Graph).

- Vertices in the graph represent data transformations
- Edges represent data movement from producers to consumers

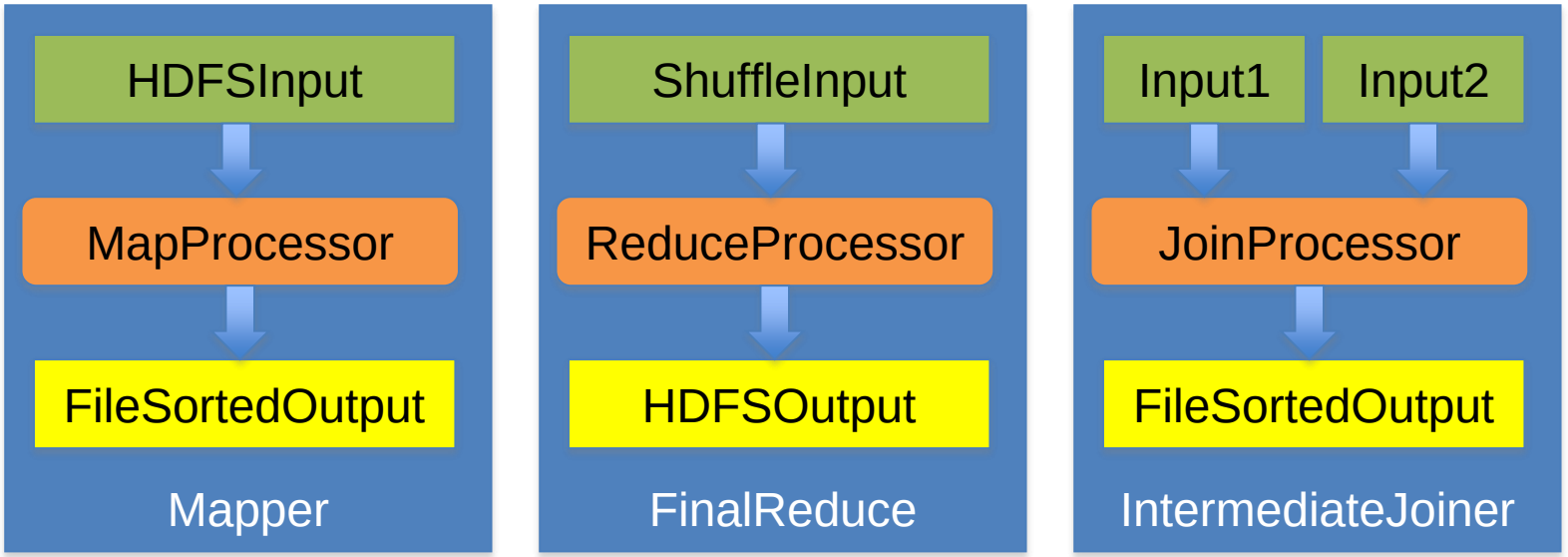


Dataflow definition API's

- Enable definition of complex data flow pipelines using simple graph connection API's. Tez expands the logical plan at runtime.
- Targeted towards data processing applications like Hive/Pig but not limited to it. Hive/Pig query plans naturally map to Tez dataflow graphs with no translation impedance.



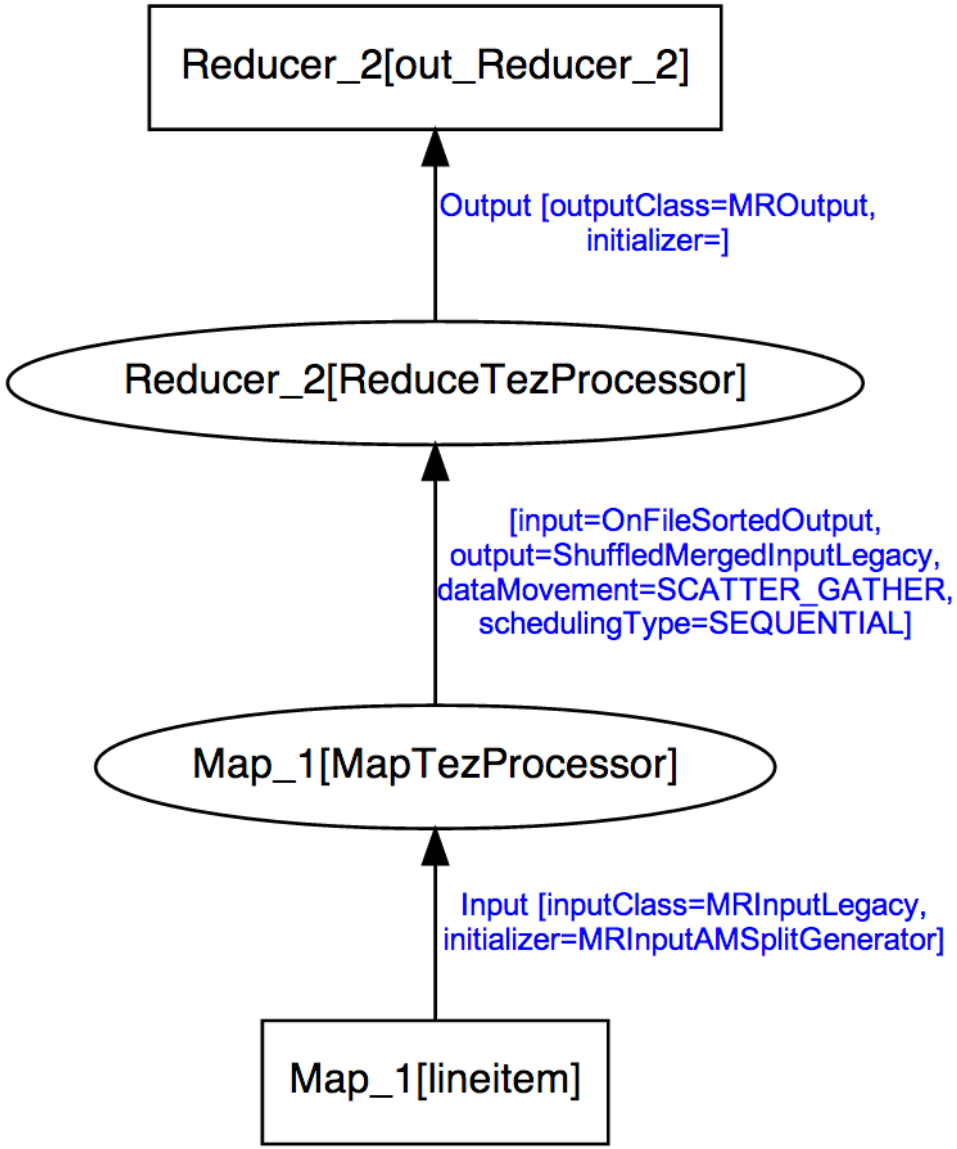
Flexible Input-Processor-Output runtime model in form of Library



- Construct physical runtime executors dynamically by connecting different Inputs, Processors and Outputs.
- End goal is to have a library of Inputs, Outputs and Processors that can be programmatically composed to generate useful tasks.

- **What is built in?**
  - Hadoop InputFormat/OutputFormat
  - OrderedPartitioned Key-Value Input/Output
  - UnorderedPartitioned Key-Value Input/Output
  - Key-Value Input/Output

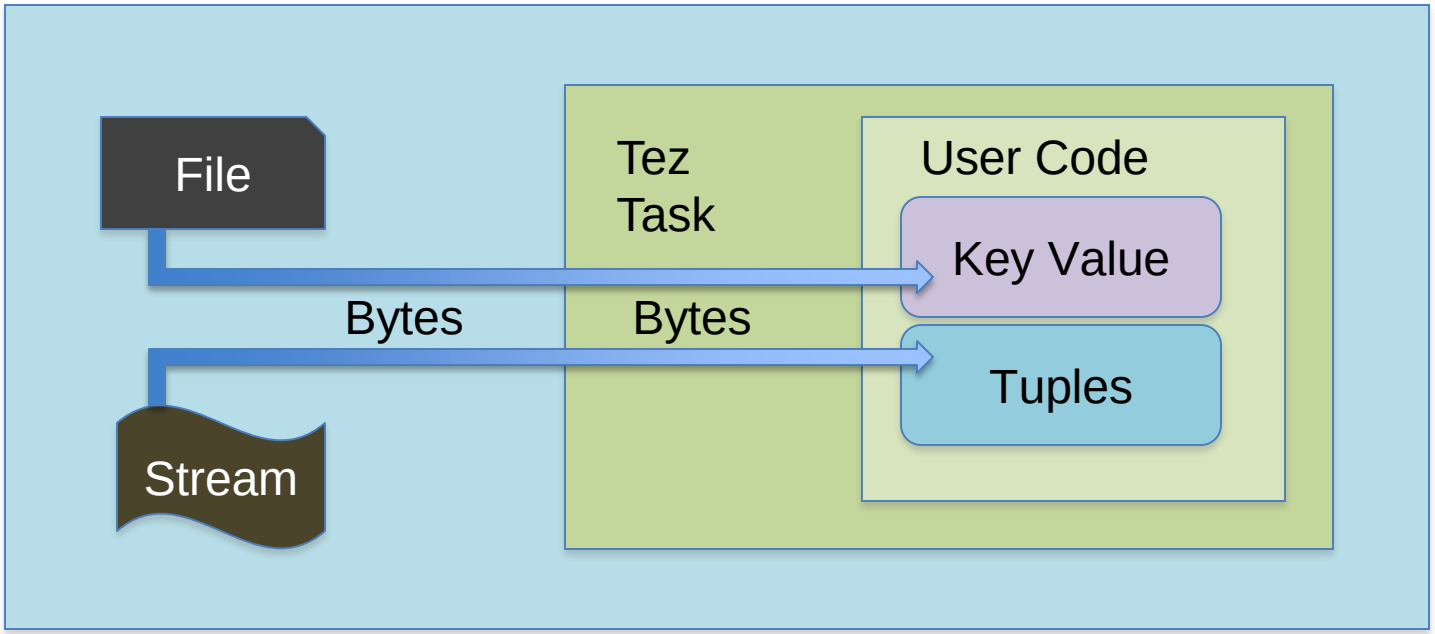
MapReduce as a 2-vertex sub-set of Tez



But Tez can be much more

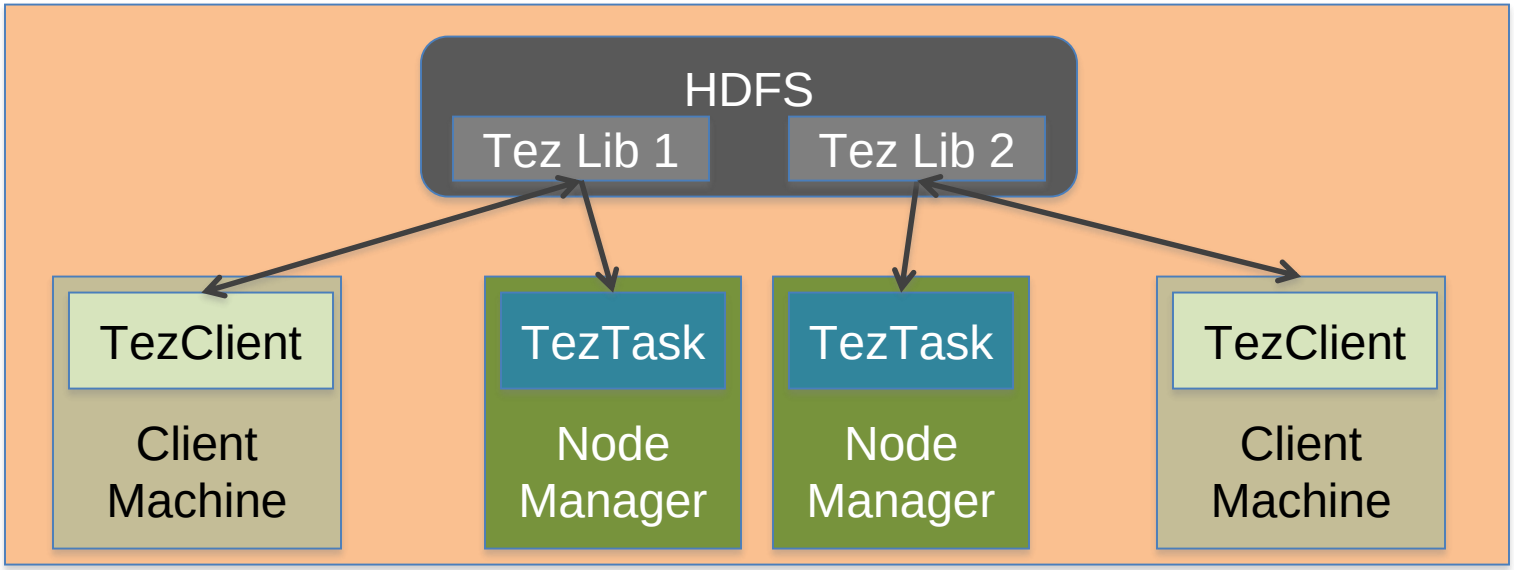
Data Type Agnostic

- Tez is only concerned with the movement of data. Files and streams of bytes.
- Does not impose any data format on the user application. MapReduce application can use Key-Value pairs on top of Tez. Hive and Pig can use tuple oriented formats that are natural and native to them.



Simplifying Deployment

- Tez is a completely-client-side application.
- No servers to be deployed
  - Simply upload to any accessible File System & change local Tez configuration to point to that.
- Enable running different versions concurrently. Facilitate testing of new functionality while keeping stable versions for production.
- Leverages YARN local resources.

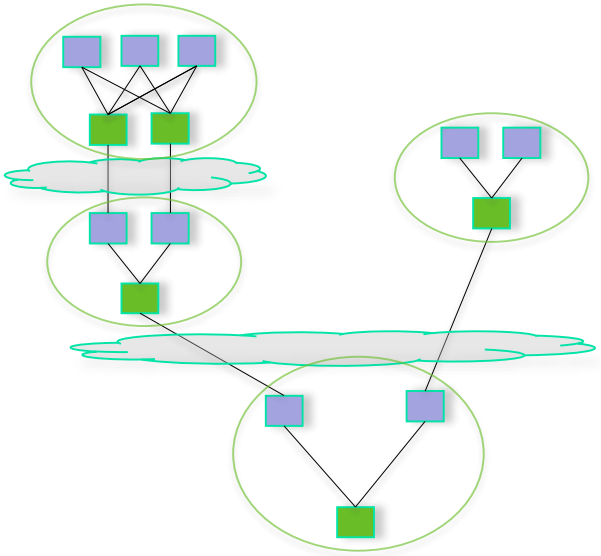


Tez – Execution Performance

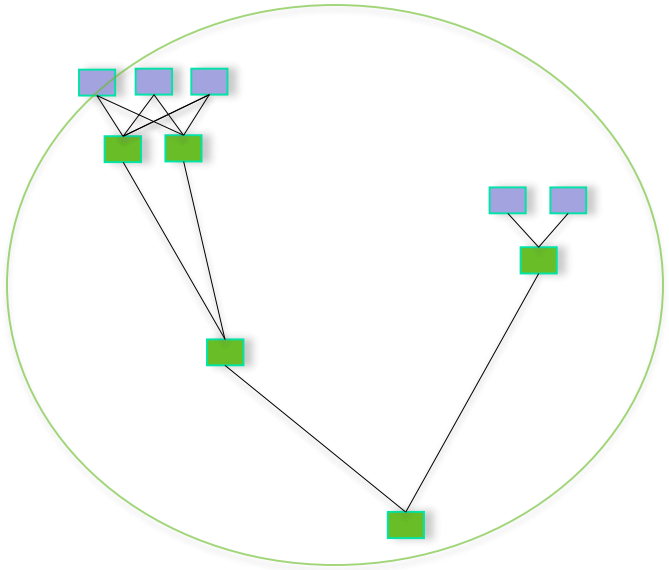
- Performance gains over Map Reduce
- Optimal resource management via YARN Container-Reuse/Sharing
- Plan reconfiguration at runtime
- Dynamic physical data flow decisions

Tez Execution Performance Gain vs. MapReduce

- Eliminate replicated write barrier between successive computations.
- Eliminate job launch overhead of workflow jobs.
- Eliminate extra stage of map reads in every workflow job.
- Eliminate queue and resource contention suffered by workflow jobs that are started after a predecessor job completes.



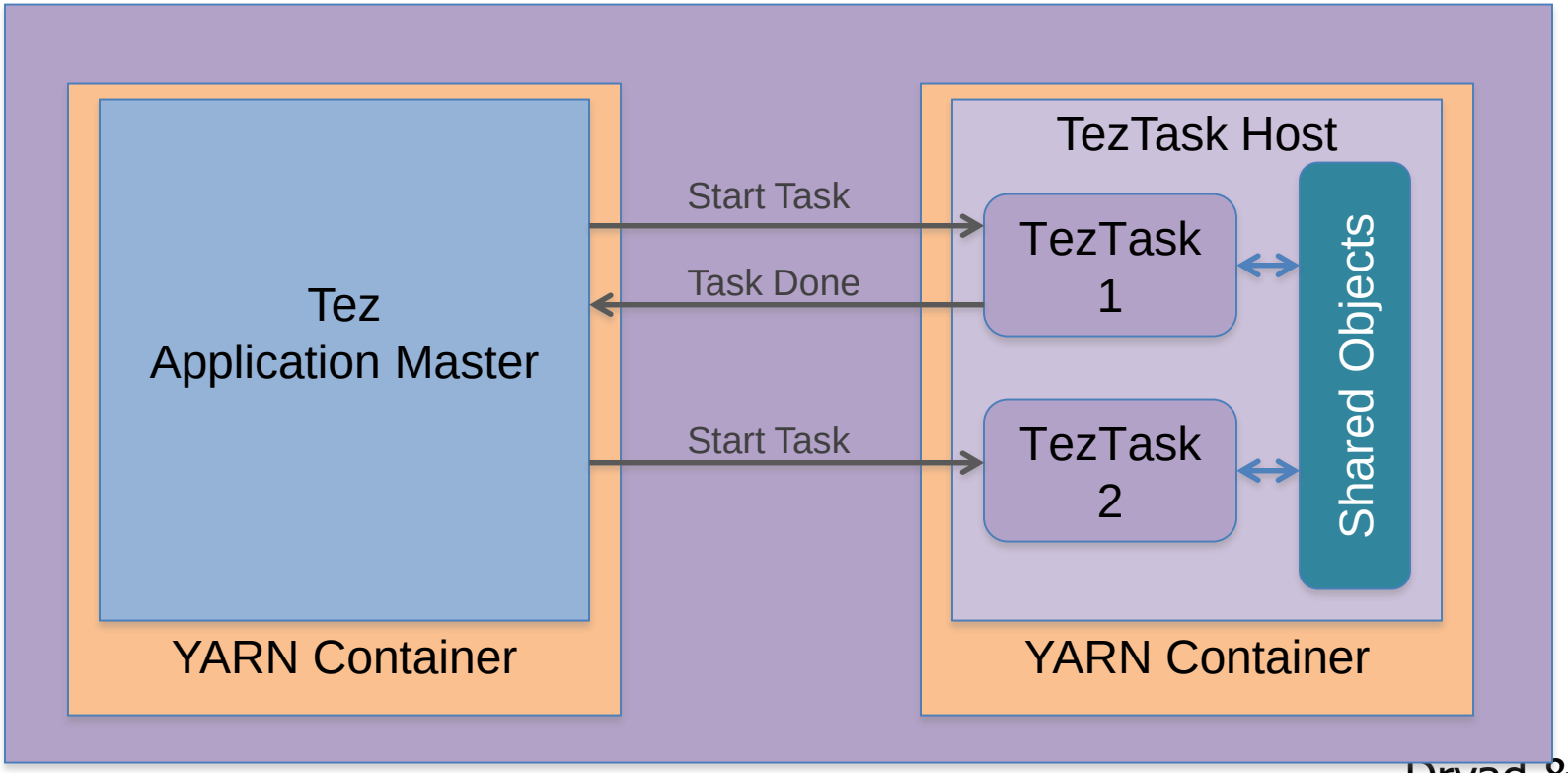
Pig/Hive - MR



Pig/Hive - Tez

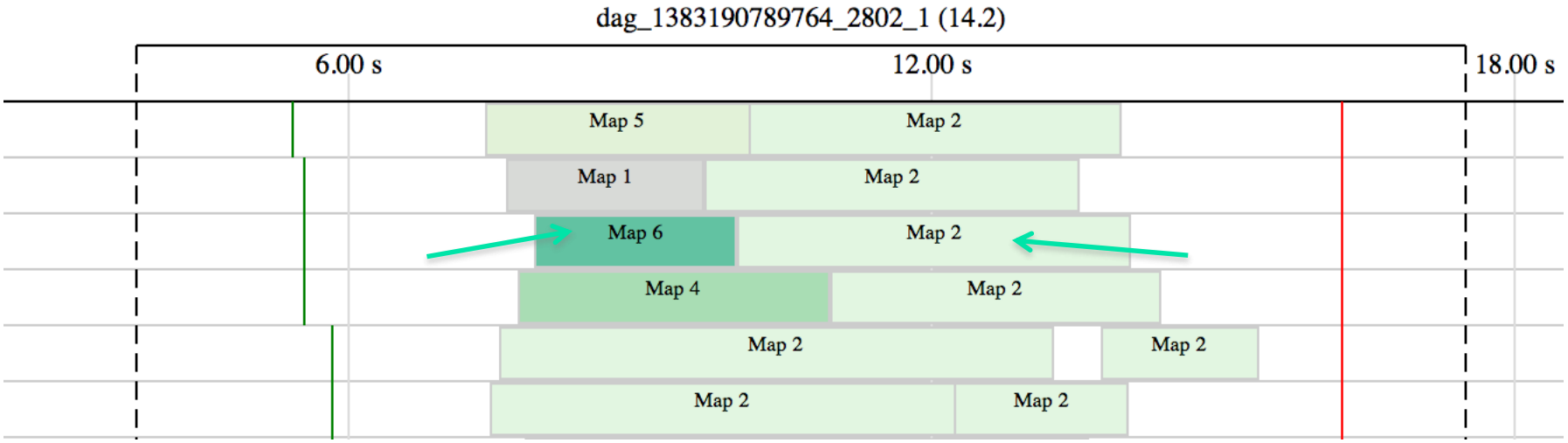
Tez – Execution Performance

- Optimal resource management
  - Reuse YARN containers/JVMs to launch new tasks
  - Reduce scheduling and launching delays
  - Enable Sharing of in-memory Data/Objects across tasks
  - JVM JIT friendly execution



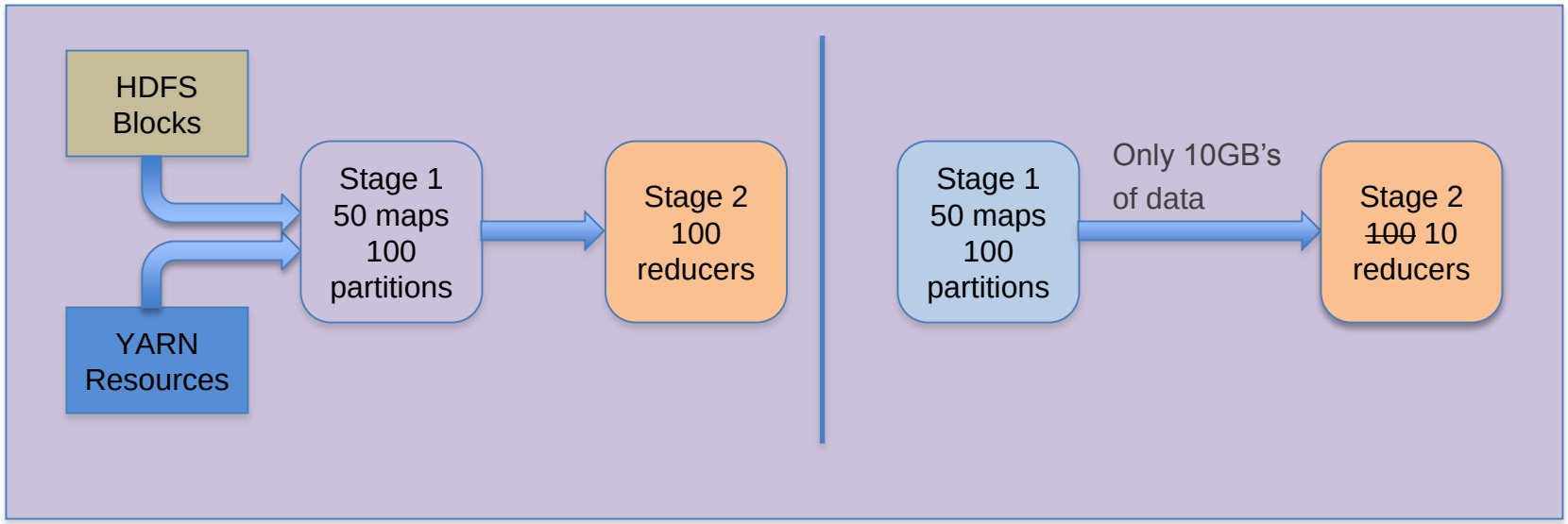
Tez Container reuse

- Tez specific feature
- Run an entire DAG using the same containers
- Different vertices use same container
- Saves time talking to YARN for new containers



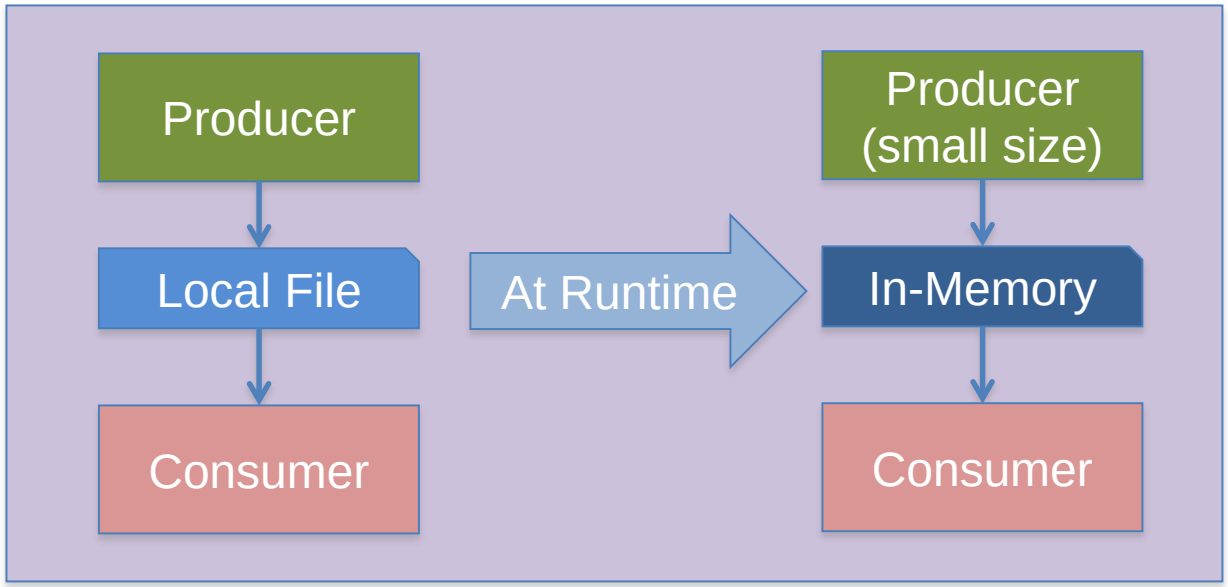
Tez – Execution Performance

- Plan reconfiguration at runtime
  - Dynamic runtime concurrency control based on data size, user operator resources, available cluster resources and locality.
  - Advanced changes in dataflow graph structure.
  - Progressive graph construction in concert with user optimizer.



Tez – Execution Performance

- Dynamic physical data flow decisions
  - Decide the type of physical byte movement and storage on the fly.
  - Store intermediate data on distributed store, local store or in-memory.
  - Transfer bytes via blocking files or streaming and the spectrum in between.



Organization of the Tez Framework

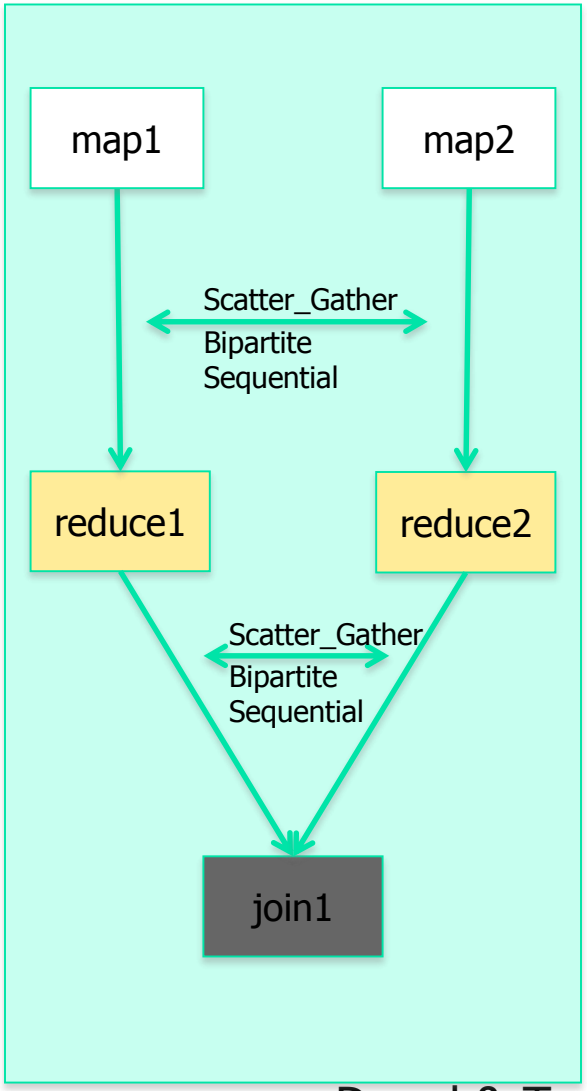
Tez provides the following APIs to define the work

- DAG API
  - Define the structure of the data processing and the relationship between producers and consumers
  - Enable definition of complex data flow pipelines using simple graph connection API's. Tez expands the logical DAG at runtime
  - This is how all the tasks in the job get specified
- Runtime API
  - Define the interface using which the Framework and App code interact with each other
  - App code transforms data and moves it between tasks
  - This is how one can specify what actually executes in each task on the cluster nodes

The DAG API of Tez

Use the DAG API to define Global Processing Flow

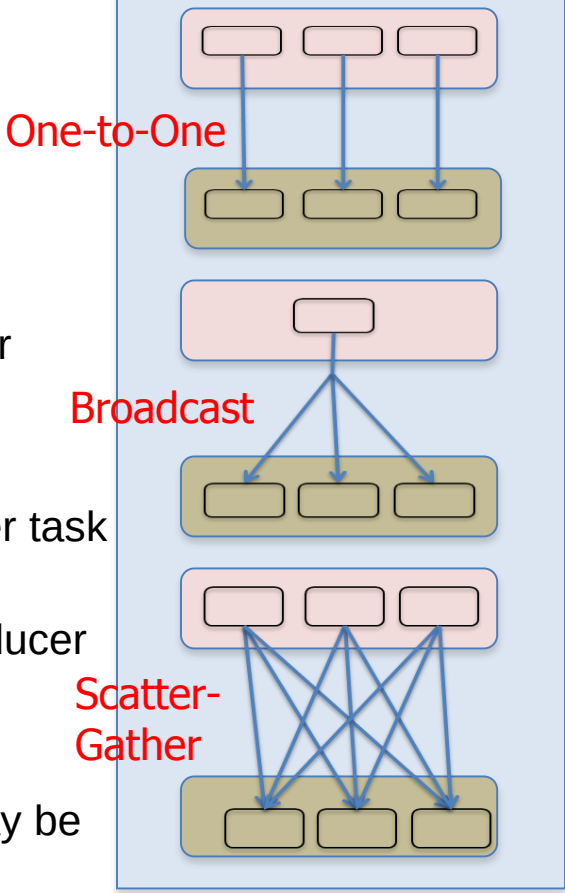
```
DAG dag = new DAG();
 Vertex map1 = new Vertex(MapProcessor.class);
Vertex map2 = new Vertex(MapProcessor.class);
Vertex reduce1 = new Vertex(ReduceProcessor.class);
Vertex reduce2 = new Vertex(ReduceProcessor.class);
Vertex join1 = new Vertex(JoinProcessor.class);
…….
Edge edge1 = Edge(map1, reduce1,
SCATTER_GATHER, PERSISTED, SEQUENTIAL,
MOutput.class, RInput.class);
Edge edge2 = Edge(map2, reduce2, SCATTER_GATHER, PERSISTED,
SEQUENTIAL, MOutput.class, RInput.class);
Edge edge3 = Edge(reduce1, join1, SCATTER_GATHER, PERSISTED,
SEQUENTIAL, MOutput.class, RInput.class);
Edge edge4 = Edge(reduce2, join1, SCATTER_GATHER, PERSISTED,
SEQUENTIAL, MOutput.class, RInput.class);
…….
dag.addVertex(map1).addVertex(map2)
        .addVertex(reduce1).addVertex(reduce2)
        .addVertex(join1)
        .addEdge(edge1).addEdge(edge2)
        .addEdge(edge3).addEdge(edge4);
```



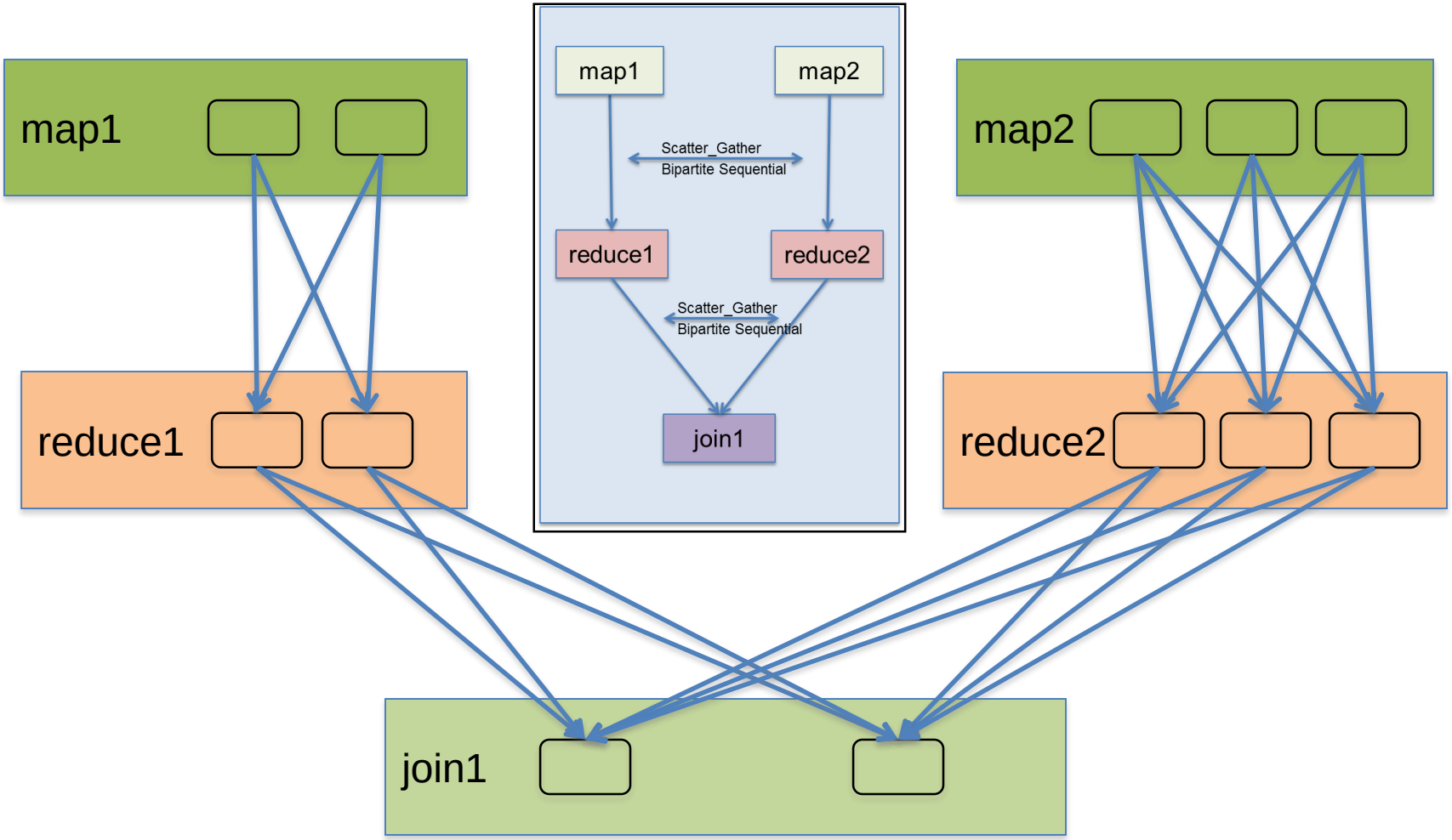
DAG API of Tez: Edge Properties

Edge properties define the connection between producer and consumer vertices in the DAG

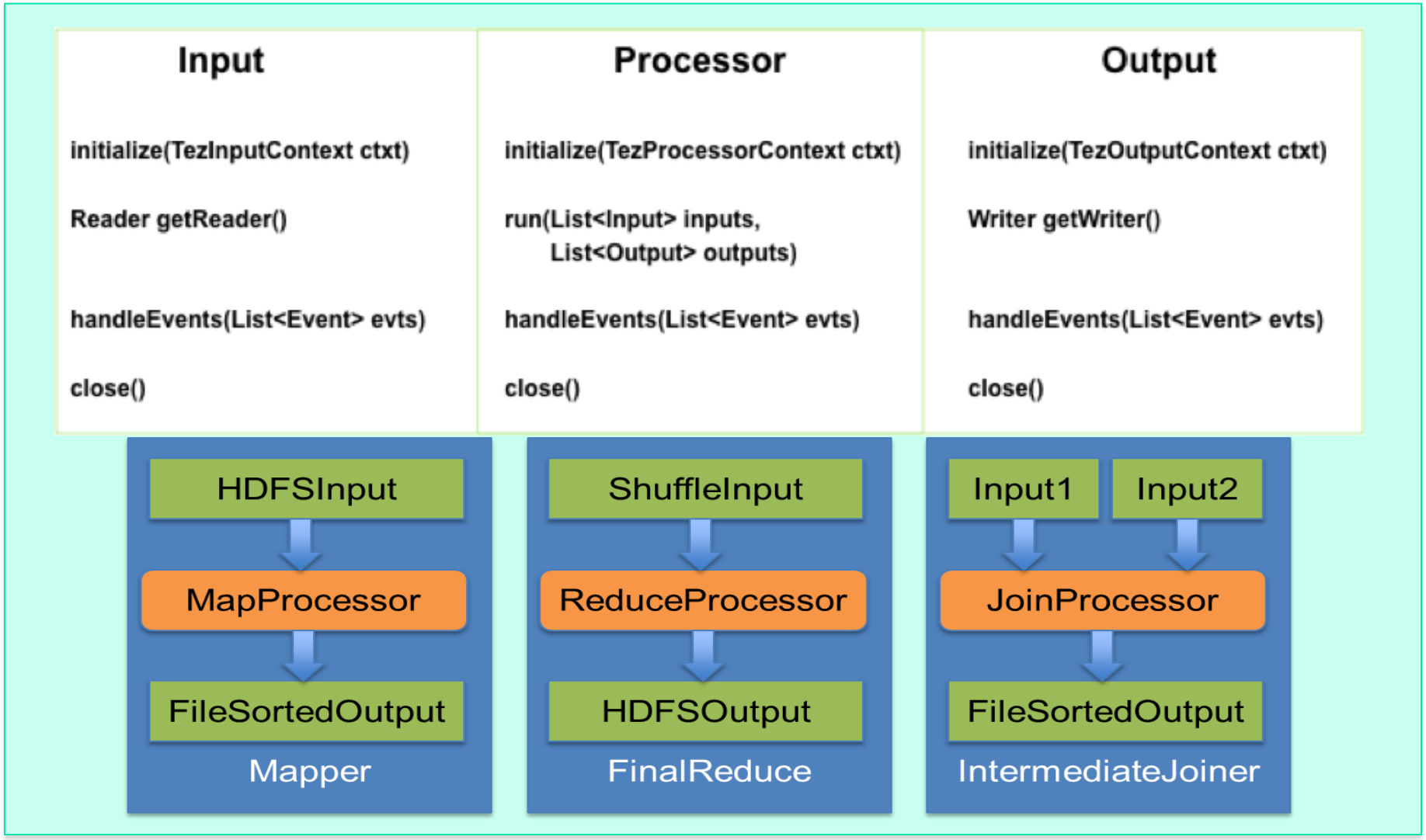
- Data movement – Define routing of data between tasks
  - **One-To-One** : Data from the i^{th} producer task routes to the i^{th} consumer task.
  - **Broadcast** : Data from a producer task routes to all consumer tasks.
  - **Scatter-Gather** : Producer tasks scatter data into shards and consumer tasks gather the data. The i^{th} shard from all producer tasks routes to the i^{th} consumer task.
- Scheduling – Define when a consumer task is scheduled
  - Sequential : Consumer task may be scheduled after a producer task completes.
  - Concurrent : Consumer task must be co-scheduled with a producer task.
- Data source – Define the lifetime/reliability of a task output
  - Persisted : O/P will be available after the task exits. Output may be lost later on.
  - Persisted-Reliable : O/P is reliably stored and will always be available
  - Ephemeral : O/P is available only while the producer task is running



Logical DAG expansion at Runtime



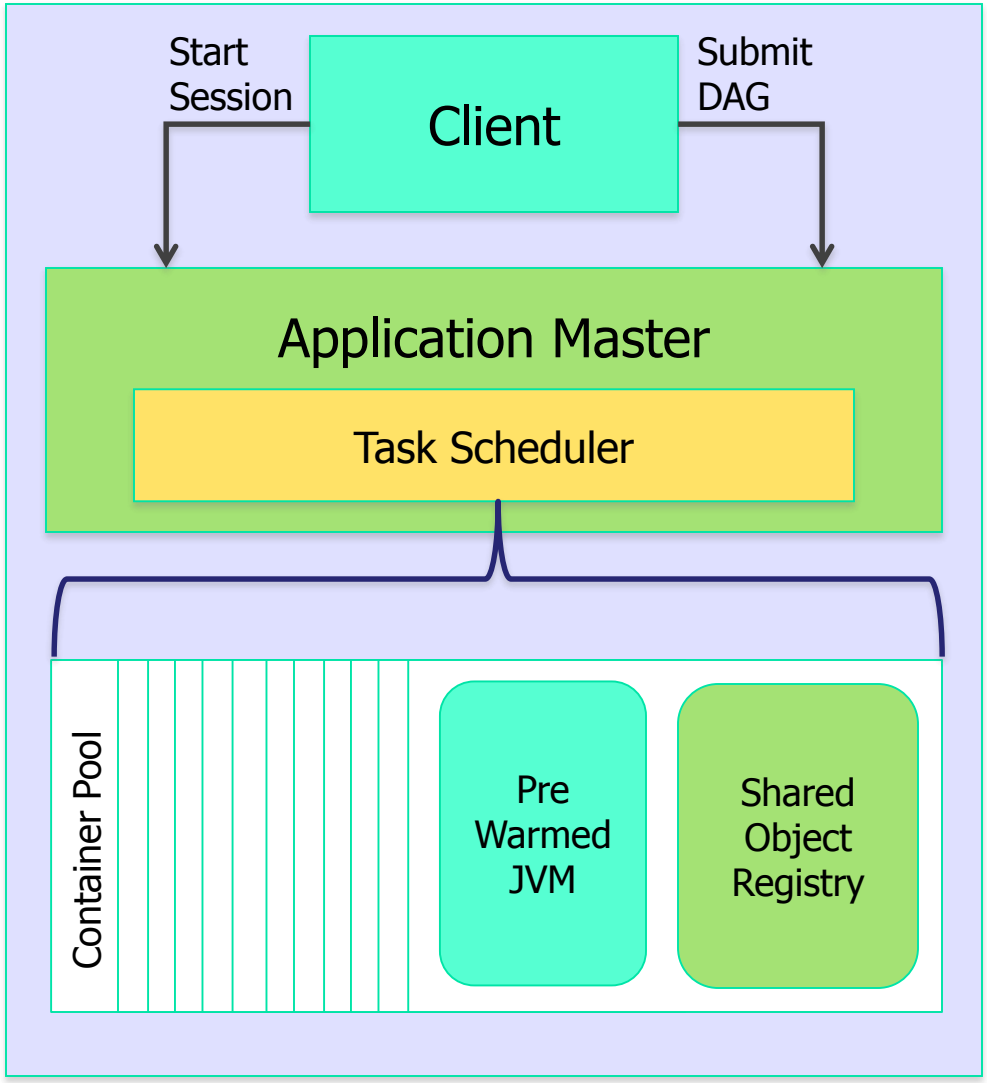
Runtime API building blocks of Tez



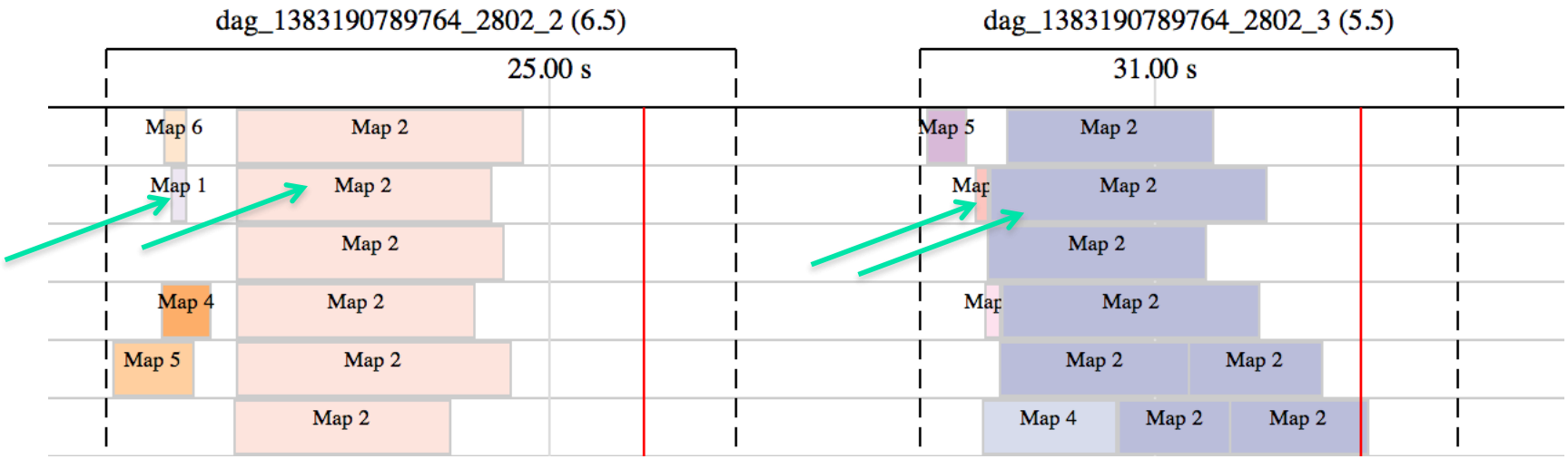
Tez – Sessions

Sessions

- Standard concepts of pre-launch and pre-warm applied
- Key for Interactive queries
- Analogous to database sessions and represent a connection between the user and the cluster
- A session can run Multiple DAGs/Queries executed in the same AM
- Maintains a pool of reusable containers for low latency execution of tasks within and across queries
- Take care of data locality and releasing resources when idle
- Session cache in the Application Master and in the container pool reduce re-computation and re-initialization



Tez – Re-Use in Action (In Session)



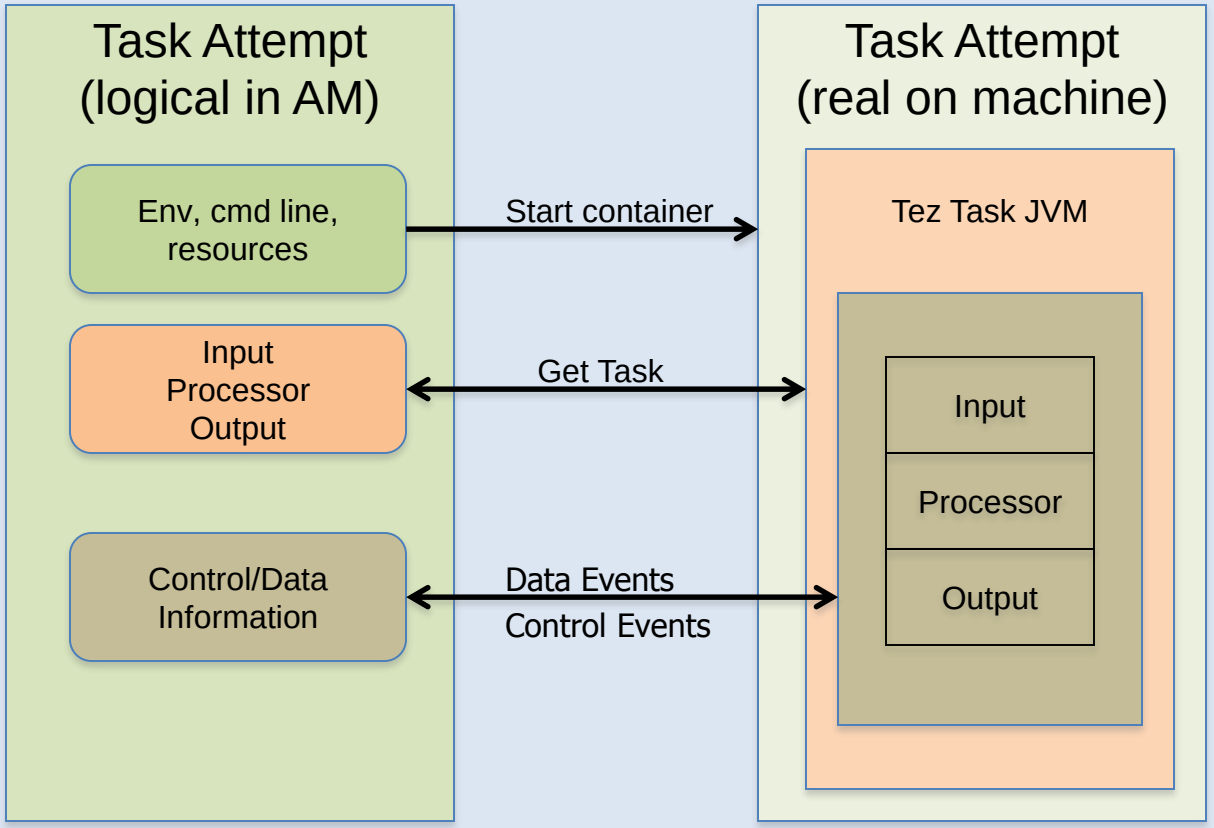
Task Execution Timeline

Tez – Deep Dive

- DAG API
- Runtime API and Event Model
- Dynamic Graph Reconfiguration

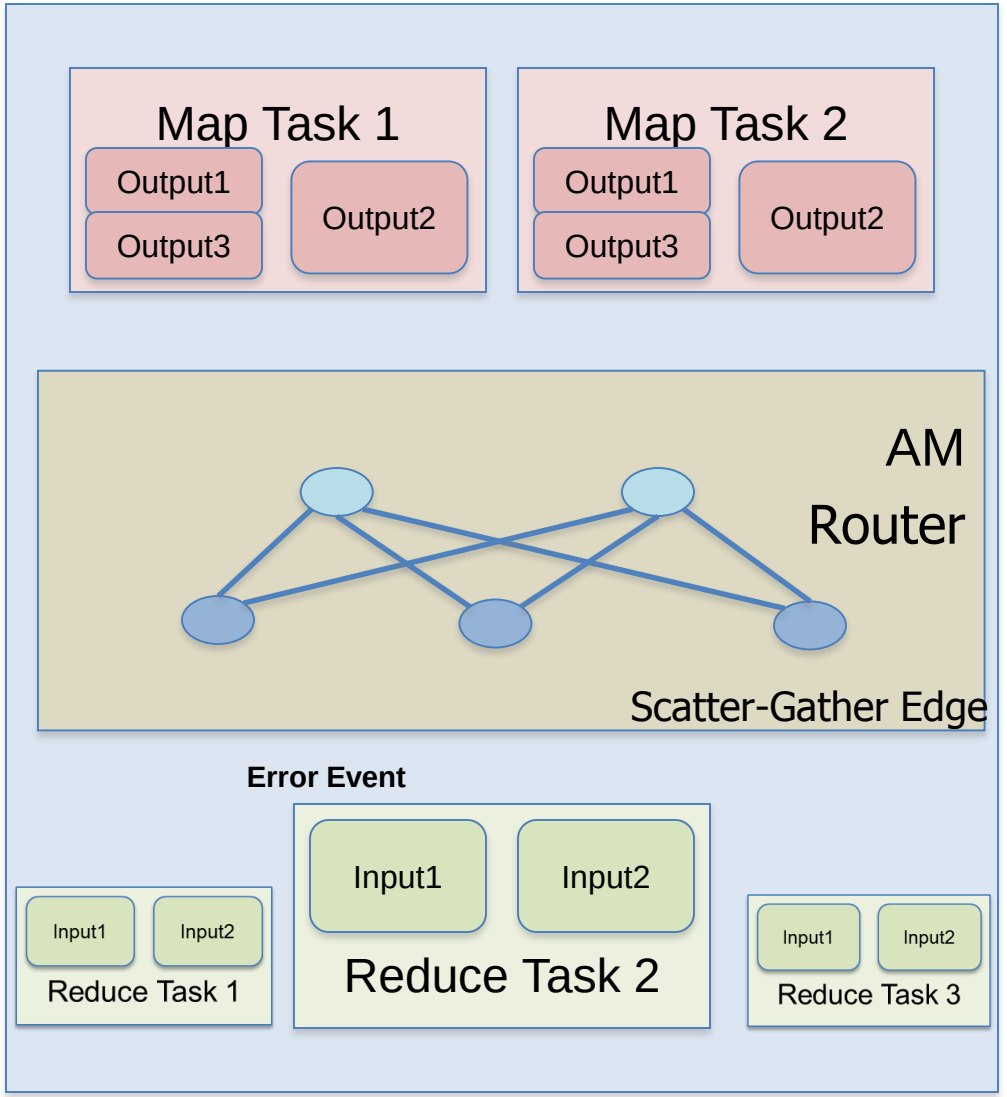
Tez – Deep Dive – Task Execution

- Start task shell with user specified env, resources etc.
- Fetch and instantiate Input, Processor, Output objects
- Receive (incremental) input information and process the input
- Provide output information
- Provide control/error events



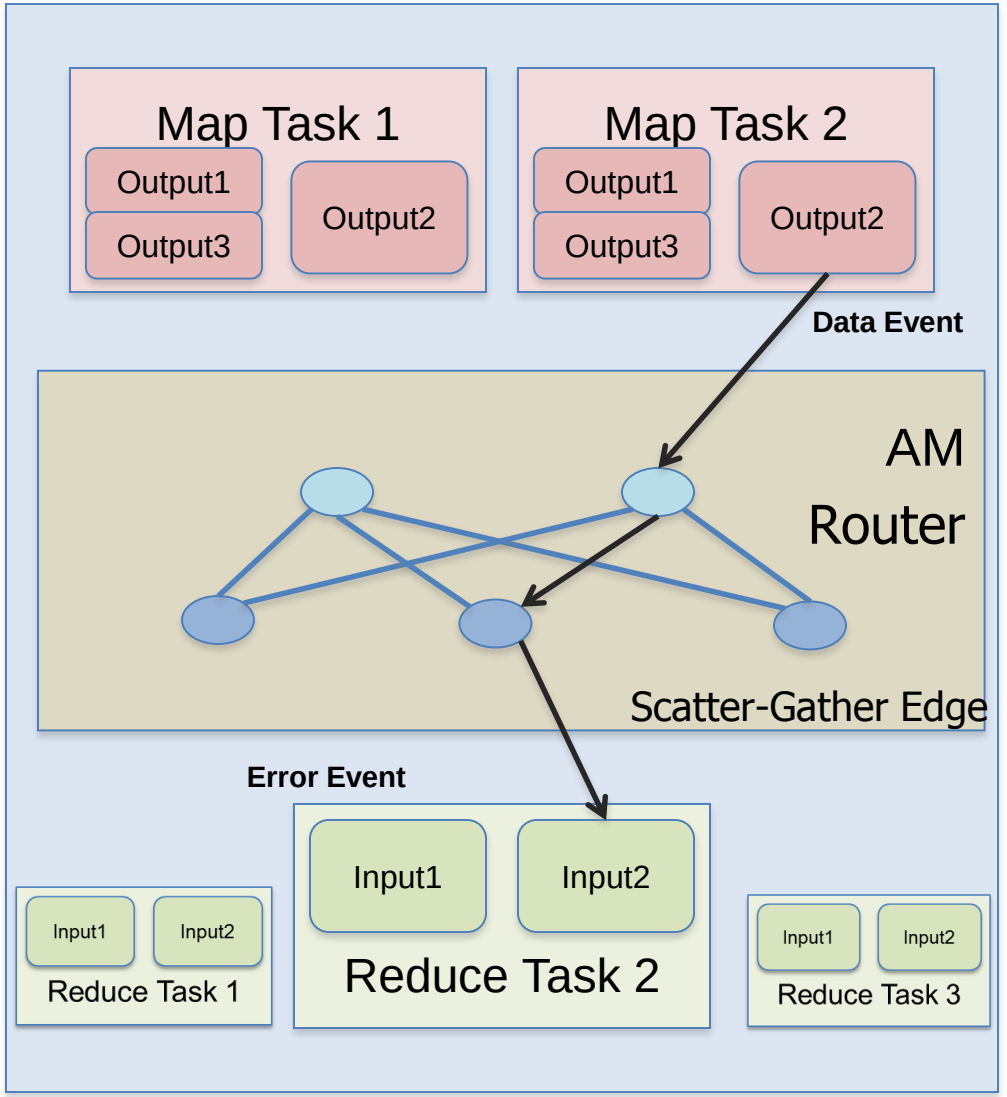
Tez Deep Dive – Runtime Events

- Events used to communicate between the tasks and between task and ApplicationMaster (AM)
- Data Movement Event used by producer task to inform the consumer task about data location, size etc.
- Input Error event sent by task to AM to inform about errors in reading input. AM then takes action by re-generating the input
- Other events to send task completion notification, data statistics and other control plane information



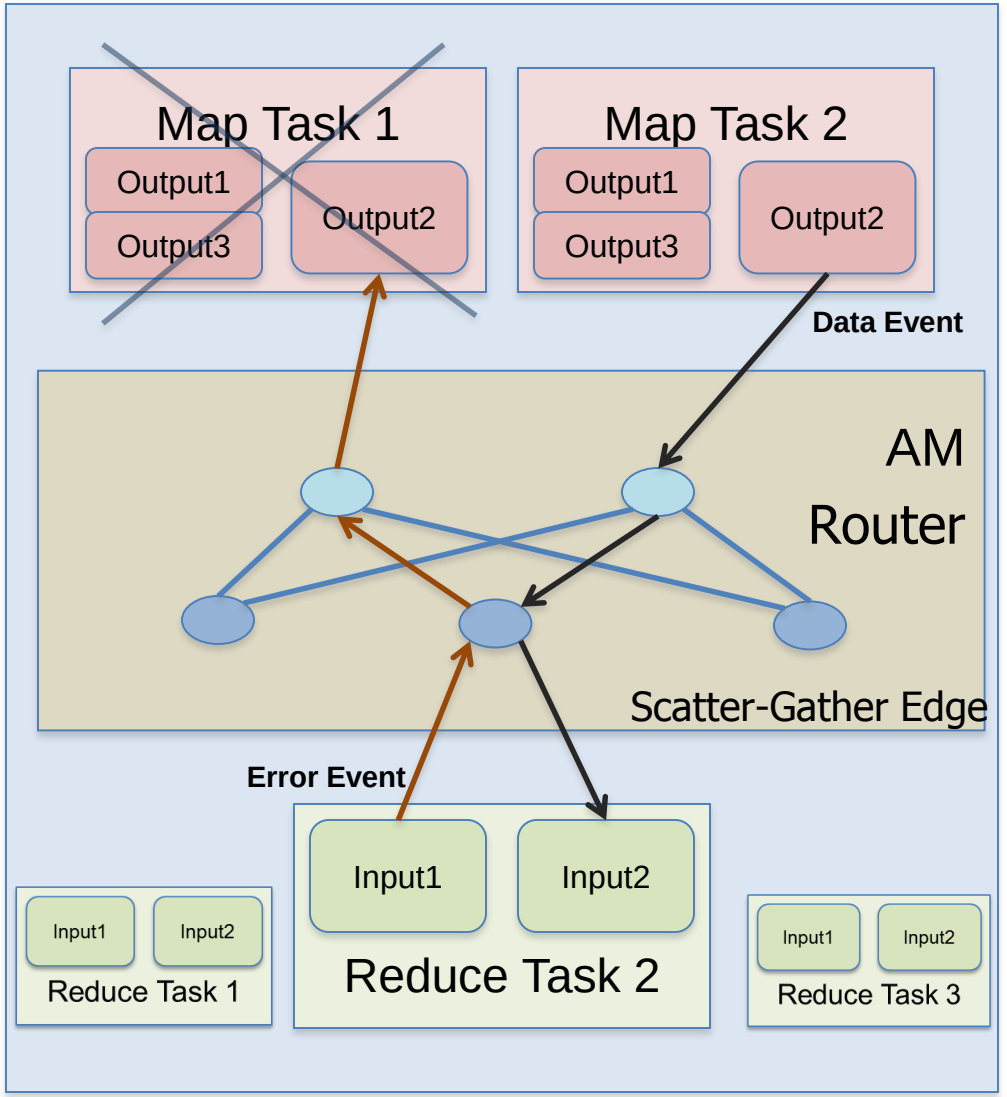
Tez Deep Dive – Runtime Events

- Events used to communicate between the tasks and between task and ApplicationMaster (AM)
- Data Movement Event used by producer task to inform the consumer task about data location, size etc.
- Input Error event sent by task to AM to inform about errors in reading input. AM then takes action by re-generating the input
- Other events to send task completion notification, data statistics and other control plane information



Tez Deep Dive – Runtime Events

- Events used to communicate between the tasks and between task and ApplicationMaster (AM)
- Data Movement Event used by producer task to inform the consumer task about data location, size etc.
- Input Error event sent by task to AM to inform about errors in reading input. AM then takes action by re-generating the input
- Other events to send task completion notification, data statistics and other control plane information



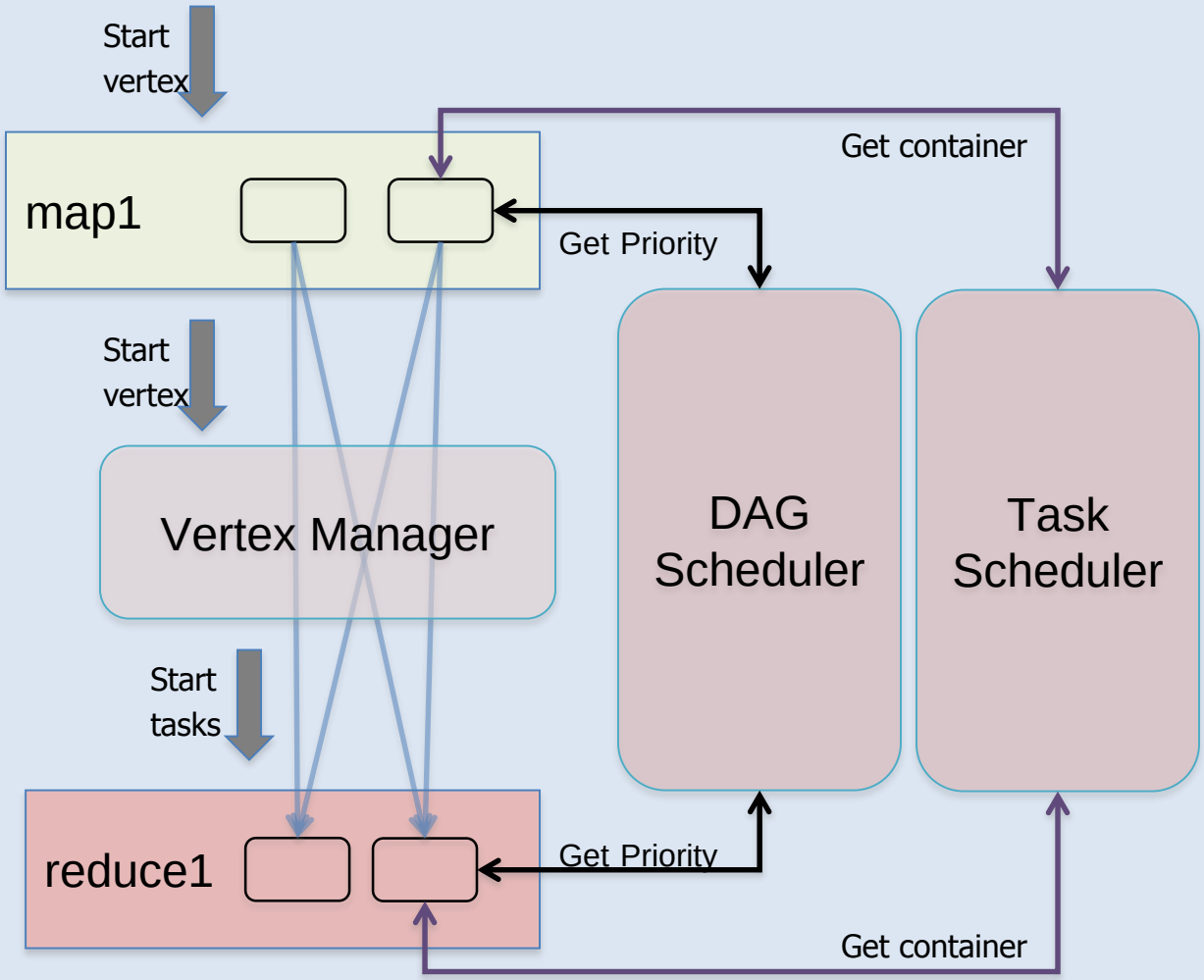
Tez – Deep Dive – Core Engine

- Vertex Manager
  - Determine task parallelism
  - Determine when tasks in a vertex can start.
- DAG Scheduler

Determine priority of task

- Task Scheduler

Allocate containers from YARN and assigns them to tasks



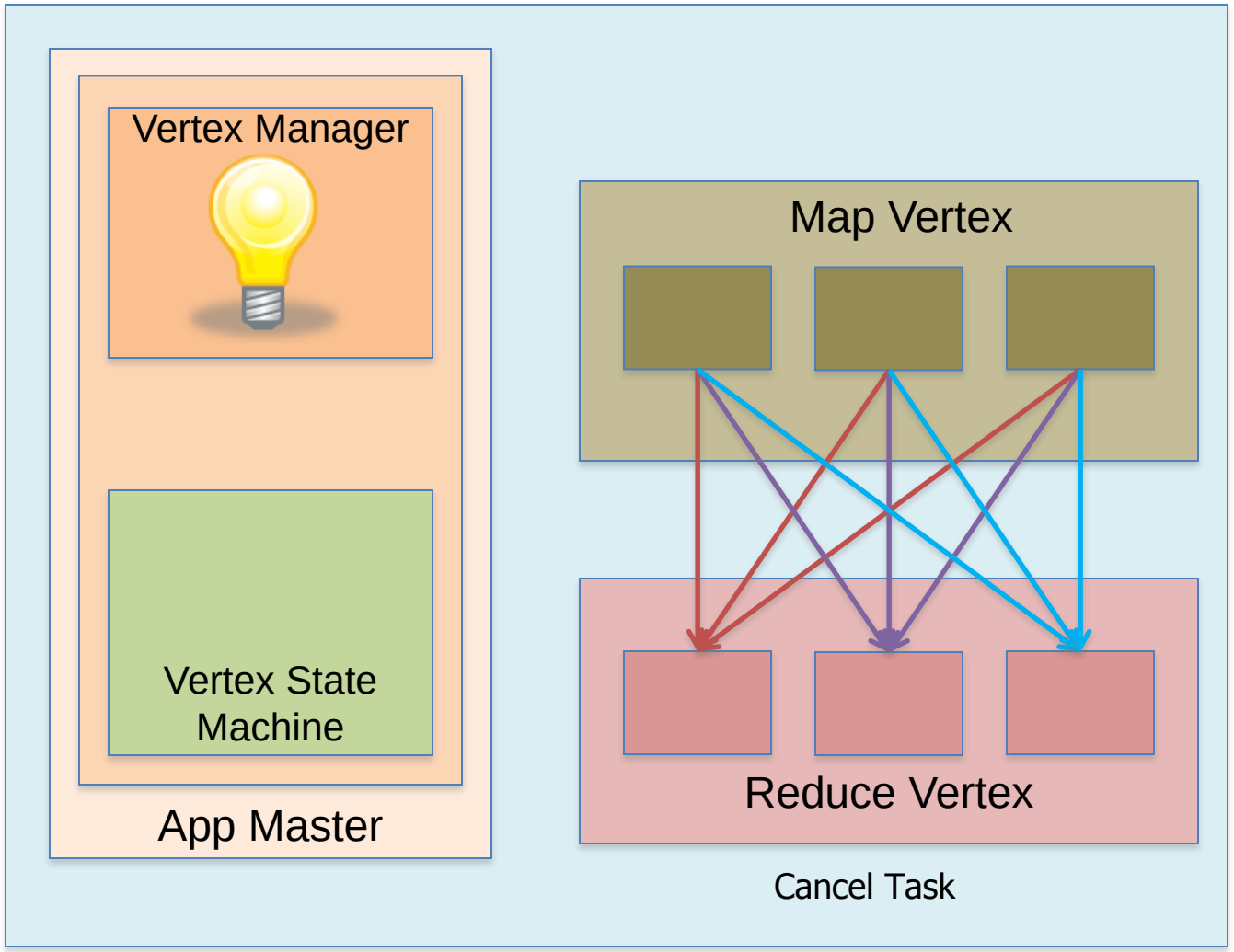
Tez – Automatic Reduce Parallelism

Event Model

Map tasks send data statistics events to the Reduce Vertex Manager.

Vertex Manager

Pluggable user logic that understands the data statistics and can formulate the correct parallelism. Advise vertex controller on parallelism



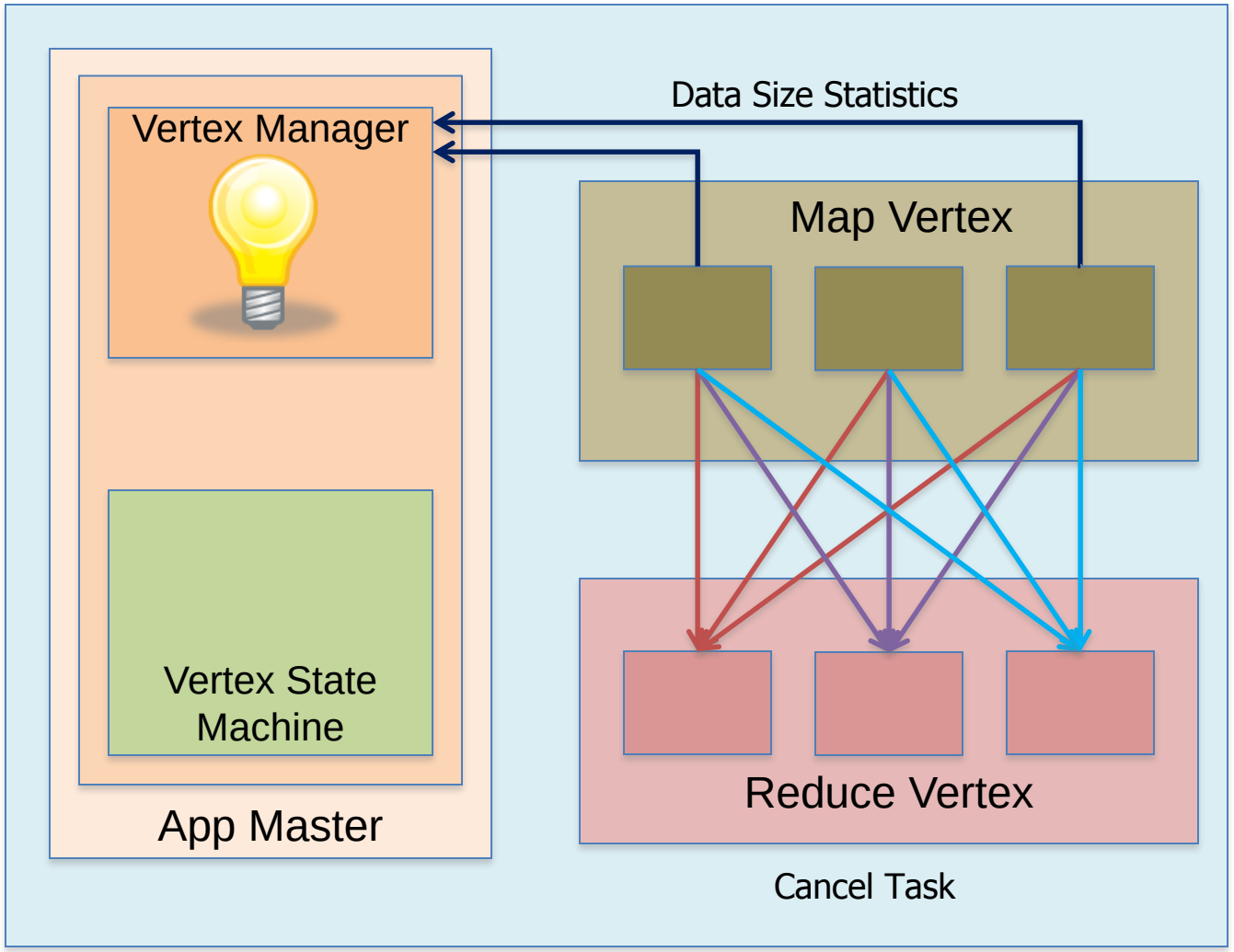
Tez – Automatic Reduce Parallelism

Event Model

Map tasks send data statistics events to the Reduce Vertex Manager.

Vertex Manager

Pluggable user logic that understands the data statistics and can formulate the correct parallelism. Advise vertex controller on parallelism



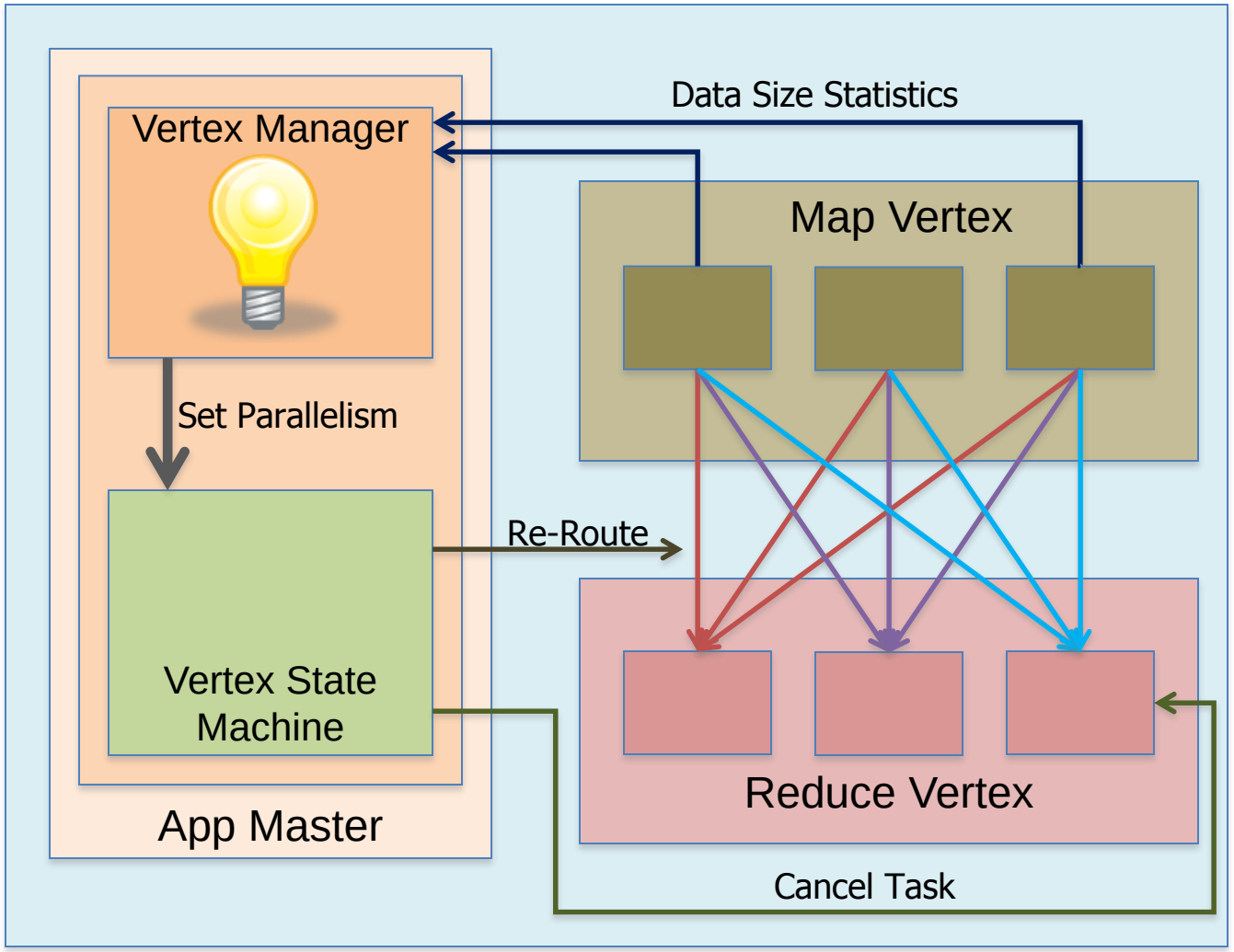
Tez – Automatic Reduce Parallelism

Event Model

Map tasks send data statistics events to the Reduce Vertex Manager.

Vertex Manager

Pluggable user logic that understands the data statistics and can formulate the correct parallelism. Advise vertex controller on parallelism



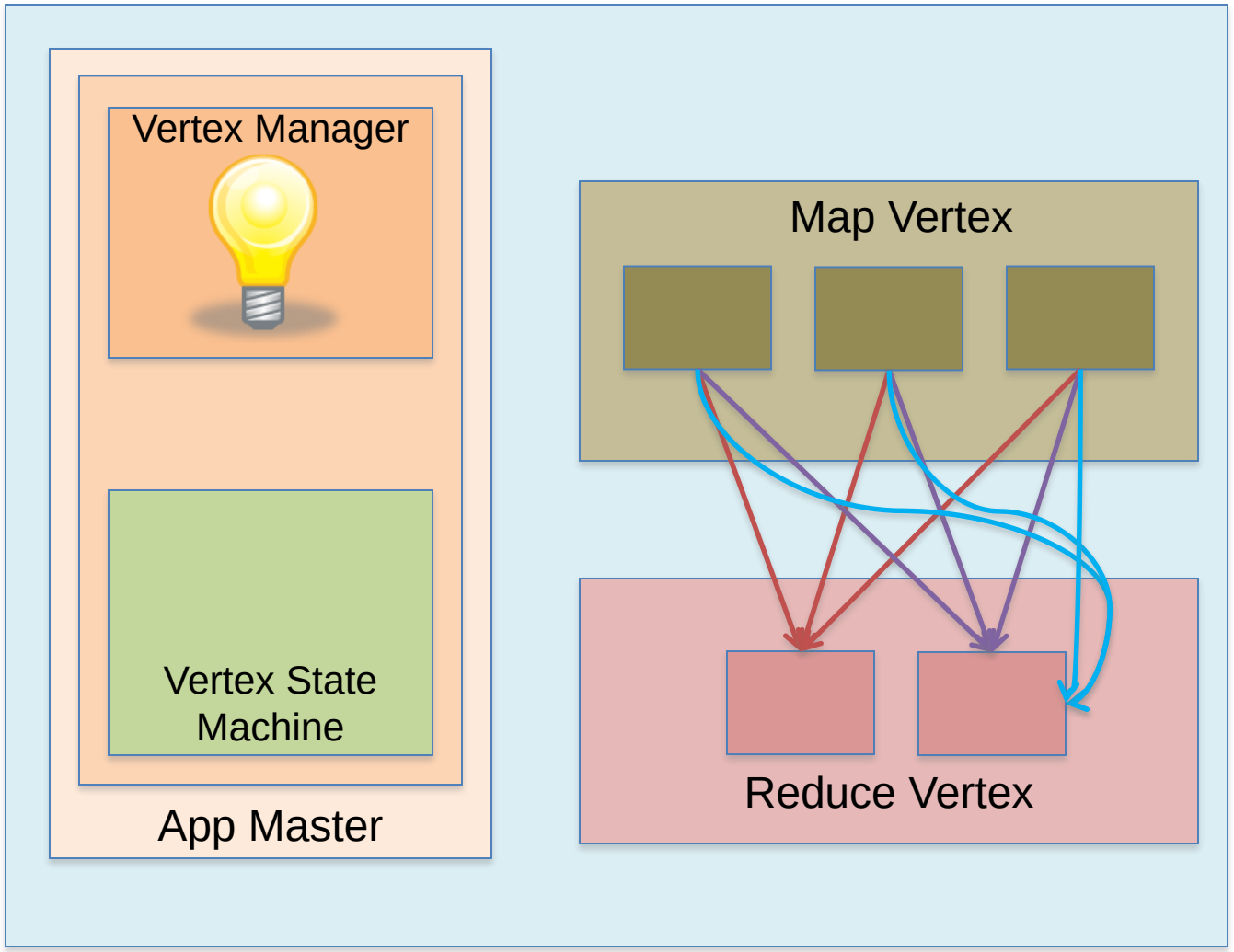
Tez – Reduce Slow Start/Pre-launch

Event Model

Map completion events sent to the Reduce Vertex Manager.

Vertex Manager

Pluggable user logic that understands the data size. Advise the vertex controller to launch the reducers before all maps have completed so that shuffle can start.



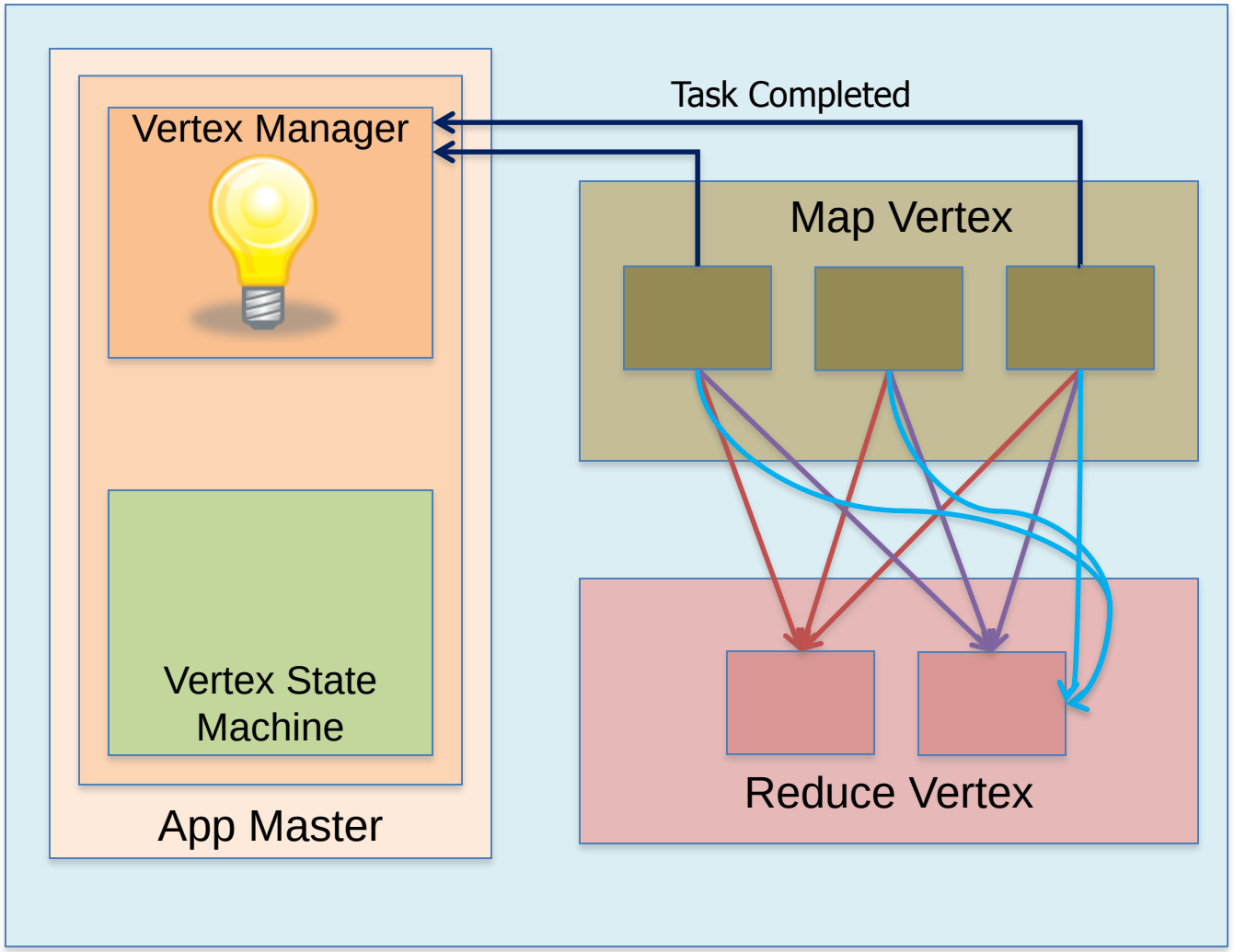
Tez – Reduce Slow Start/Pre-launch

Event Model

Map completion events sent to the Reduce Vertex Manager.

Vertex Manager

Pluggable user logic that understands the data size. Advise the vertex controller to launch the reducers before all maps have completed so that shuffle can start.



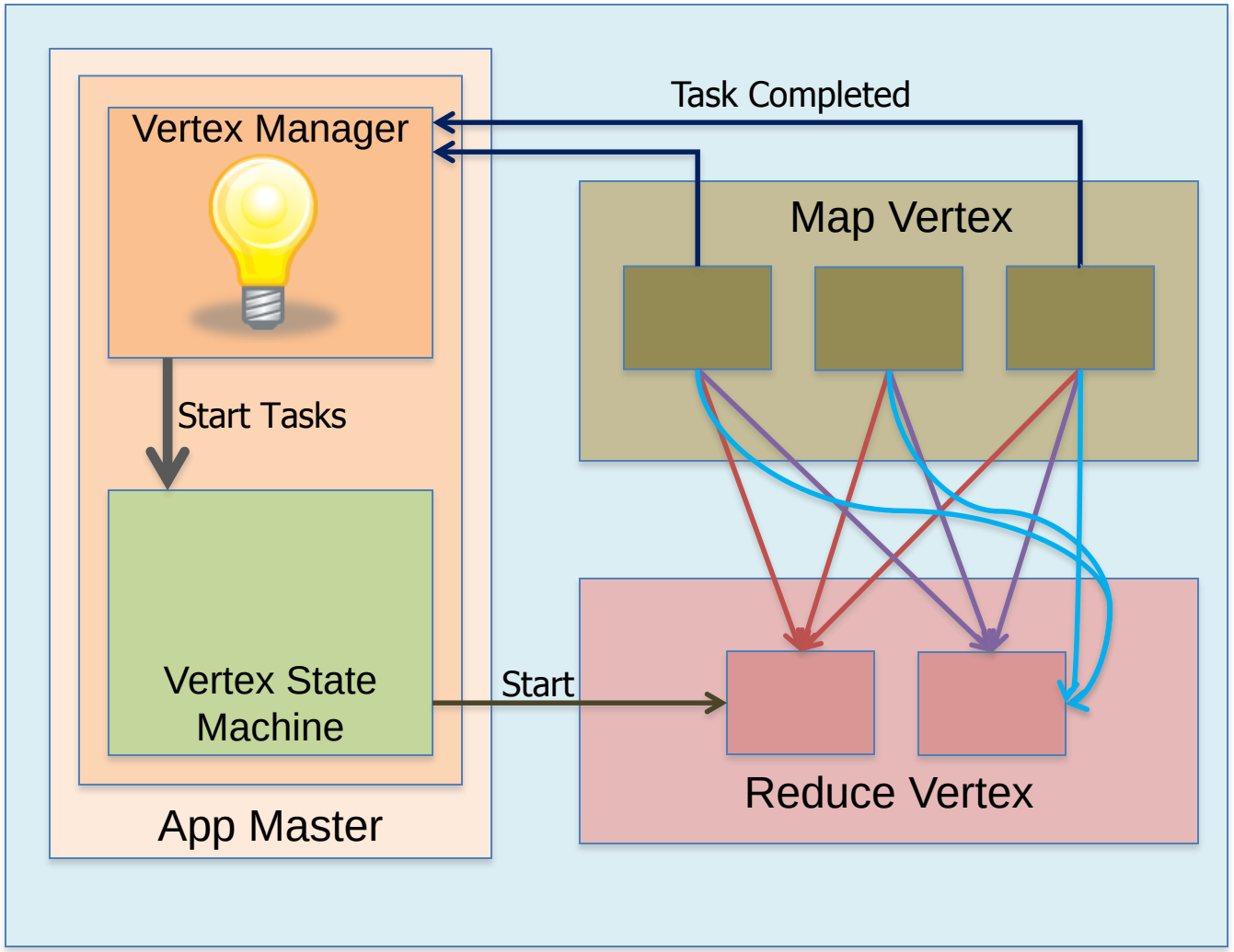
Tez – Reduce Slow Start/Pre-launch

Event Model

Map completion events sent to the Reduce Vertex Manager.

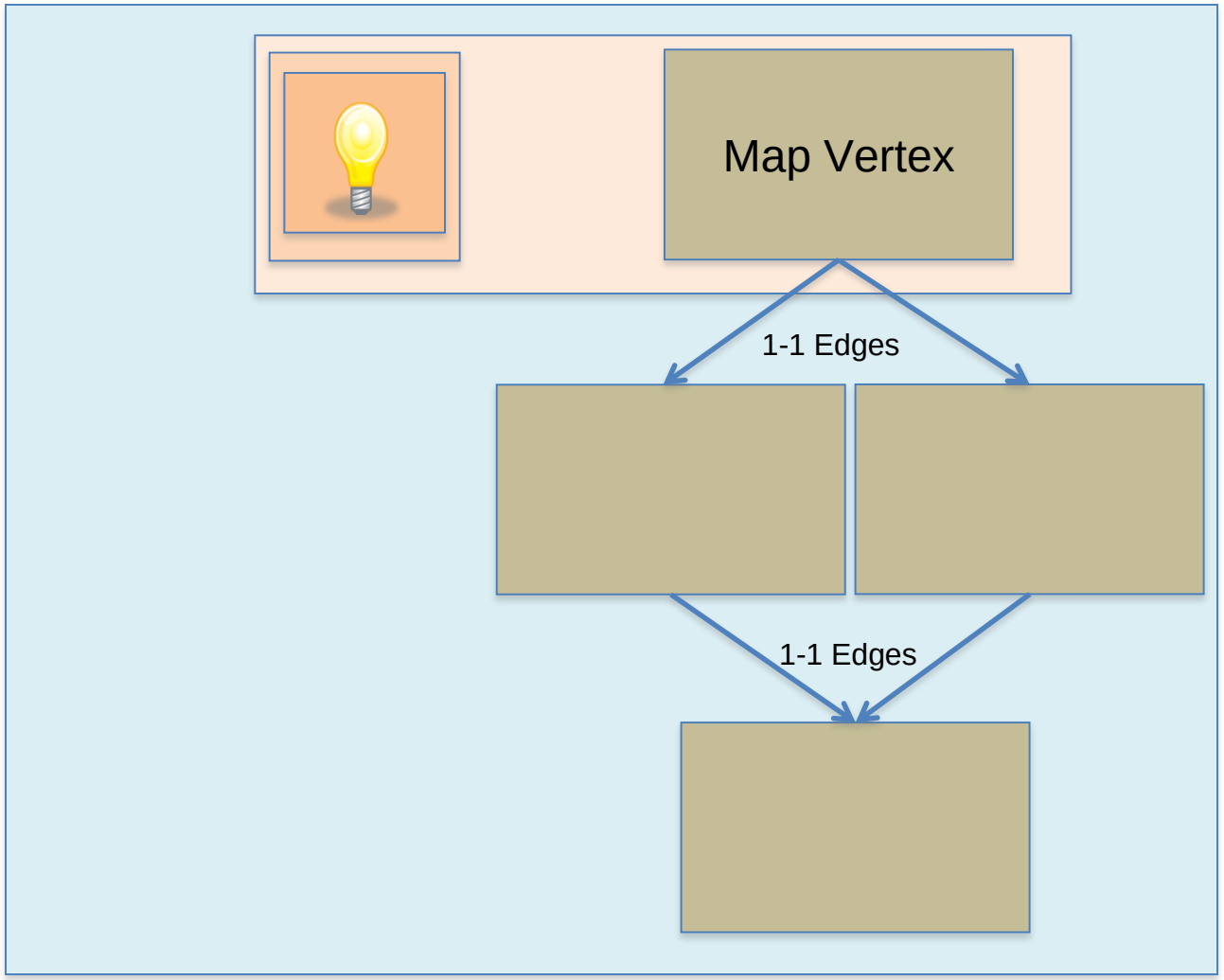
Vertex Manager

Pluggable user logic that understands the data size. Advise the vertex controller to launch the reducers before all maps have completed so that shuffle can start.



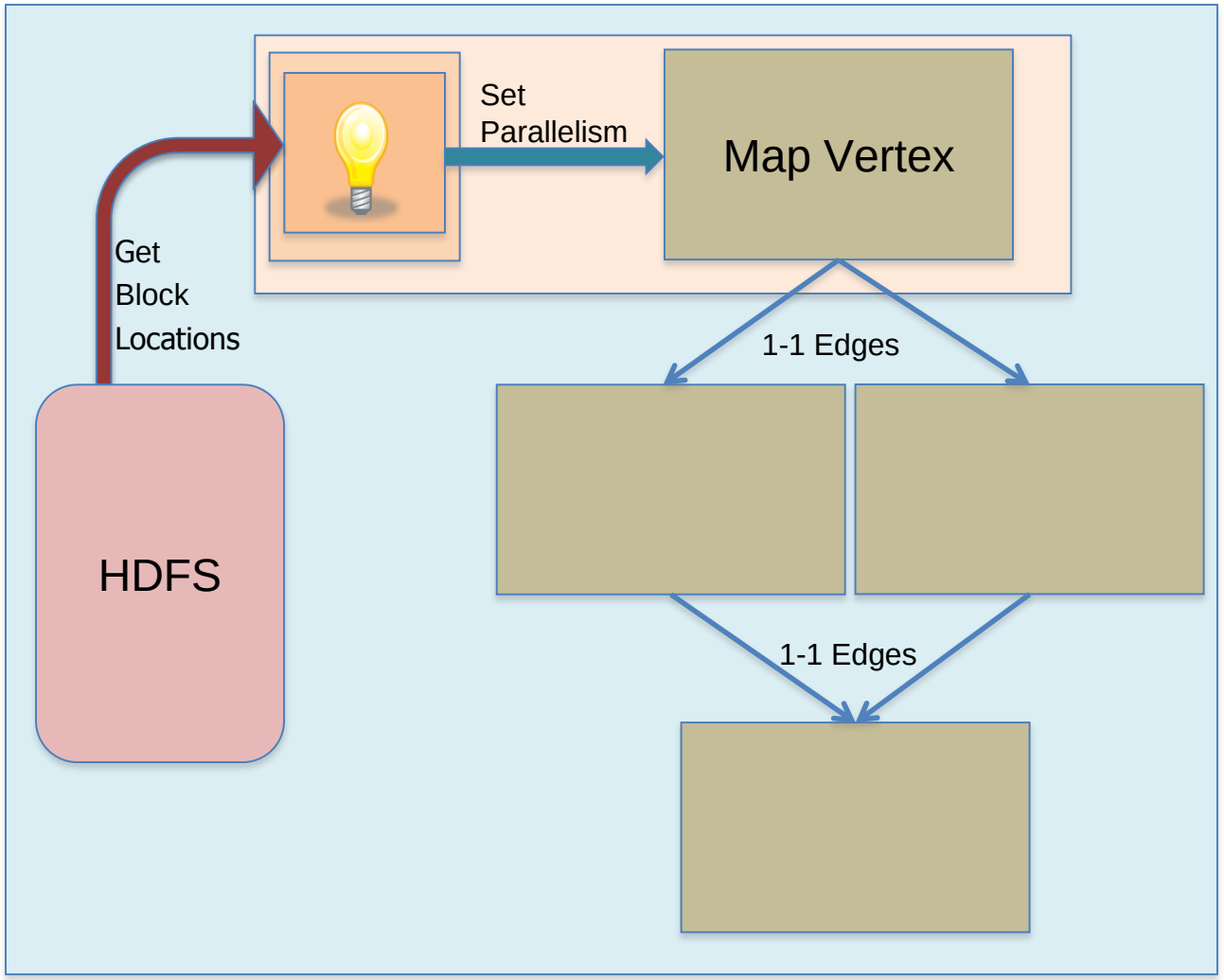
Tez – Automatic Map Parallelism

- Input vertex manager gets block locations and estimates the number of mappers based on data size, cluster capacity and map data limits. Groups block by locality
- Consumer vertex parallelism gets recursively determined through the chain of consumer vertices



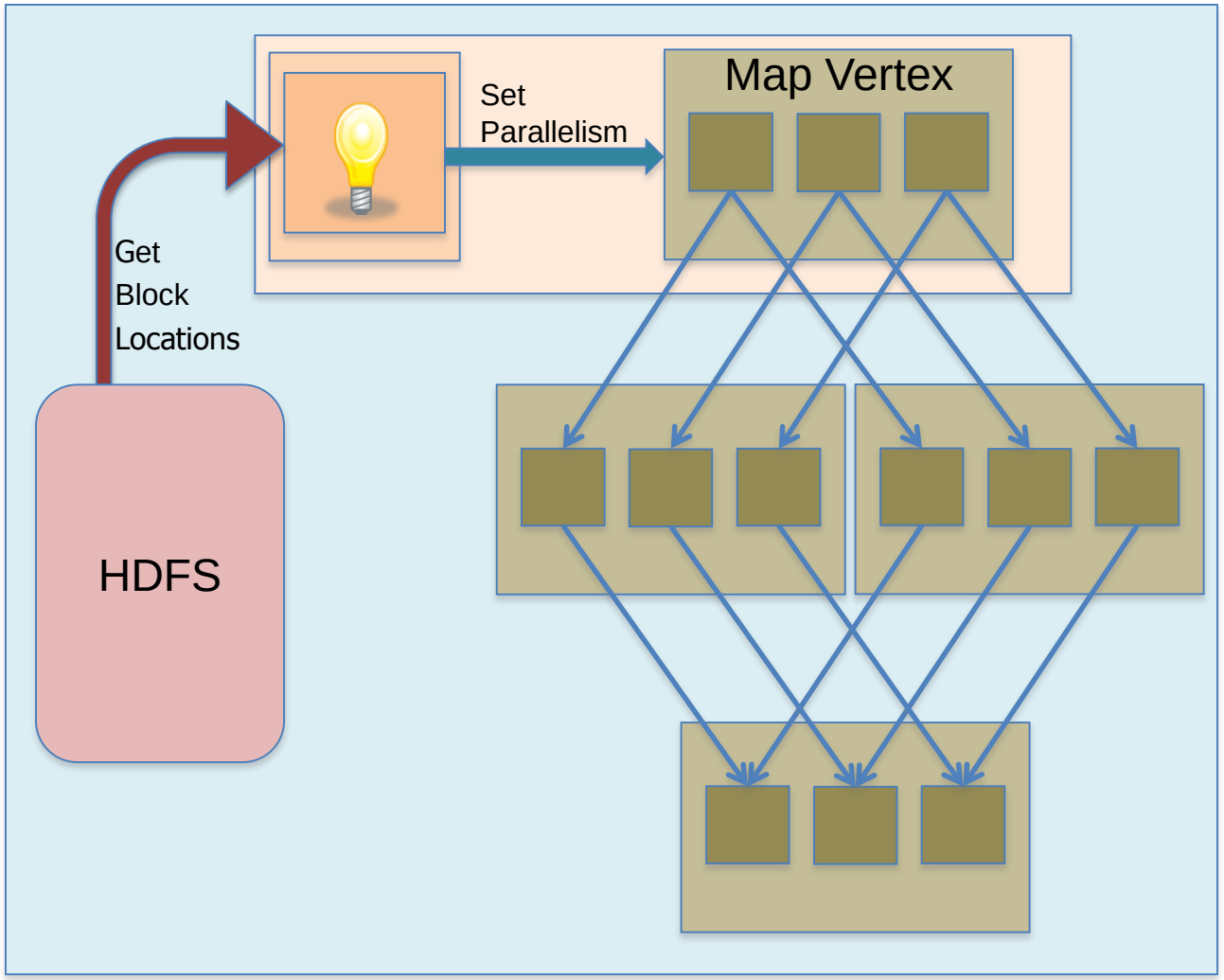
Tez – Automatic Map Parallelism

- Input vertex manager gets block locations and estimates the number of mappers based on data size, cluster capacity and map data limits. Groups block by locality
- Consumer vertex parallelism gets recursively determined through the chain of consumer vertices



Tez – Automatic Map Parallelism

- Input vertex manager gets block locations and estimates the number of mappers based on data size, cluster capacity and map data limits. Groups block by locality
- Consumer vertex parallelism gets recursively determined through the chain of consumer vertices



End User Benefits from Tez

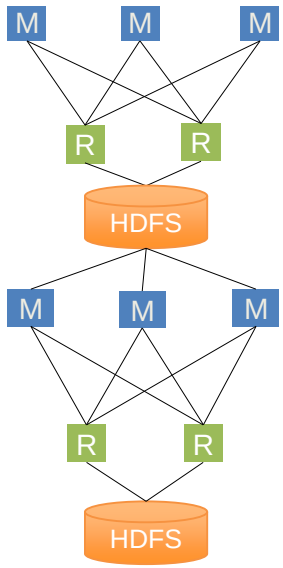
- Better Performance
  - Framework performance + application performance
- Better utilization of cluster resources
  - Efficient use of allocated resources
- Better predictability of results
  - Minimized queuing delays
- Reduced load on HDFS
  - Removes unnecessary HDFS writes
- Reduced network usage
  - Efficient data transfer using new data patterns
- Increased developer productivity
  - Lets the user concentrate on application logic instead of Hadoop internals

Tez – Broadcast Edge

```
SELECT ss.ss_item_sk, avg_price, inv.inv_quantity_on_hand
FROM (select avg(ss_sales_price) as avg_price, ss_item_sk from
   store_sales
group by ss_item_sk) ss
JOIN inventory inv
ON (inv.inv_item_sk = ss.ss_item_sk);
```

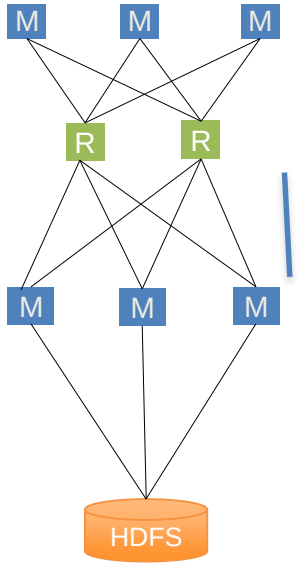
Store Sales scan. Group by and aggregation.

Inventory and Store Sales (aggr.) output scan and shuffle join.



Store Sales scan. Group by and aggregation.

Inventory and Store Sales (aggr.) output scan and shuffle join.



broadcast

Tez – Multiple Outputs

```
FROM (SELECT * FROM store_sales, date_dim WHERE ss_sold_date_sk =
   d_date_sk and d_year = 2000)
INSERT INTO TABLE t1 SELECT distinct ss_item_sk
INSERT INTO TABLE t2 SELECT distinct ss_customer_sk;
```

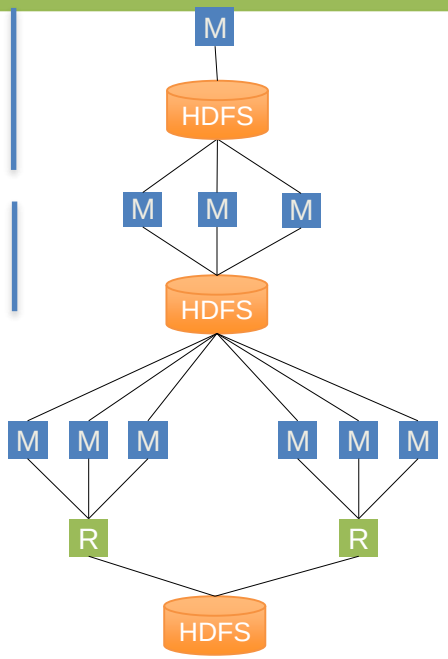
Hive : Multi-insert queries

Hive – MR

Map join date_dim/store sales

Materialize join on HDFS

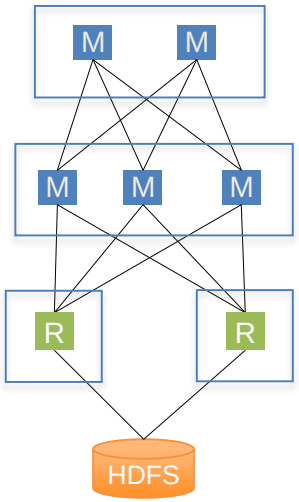
Two MR jobs to do the distinct



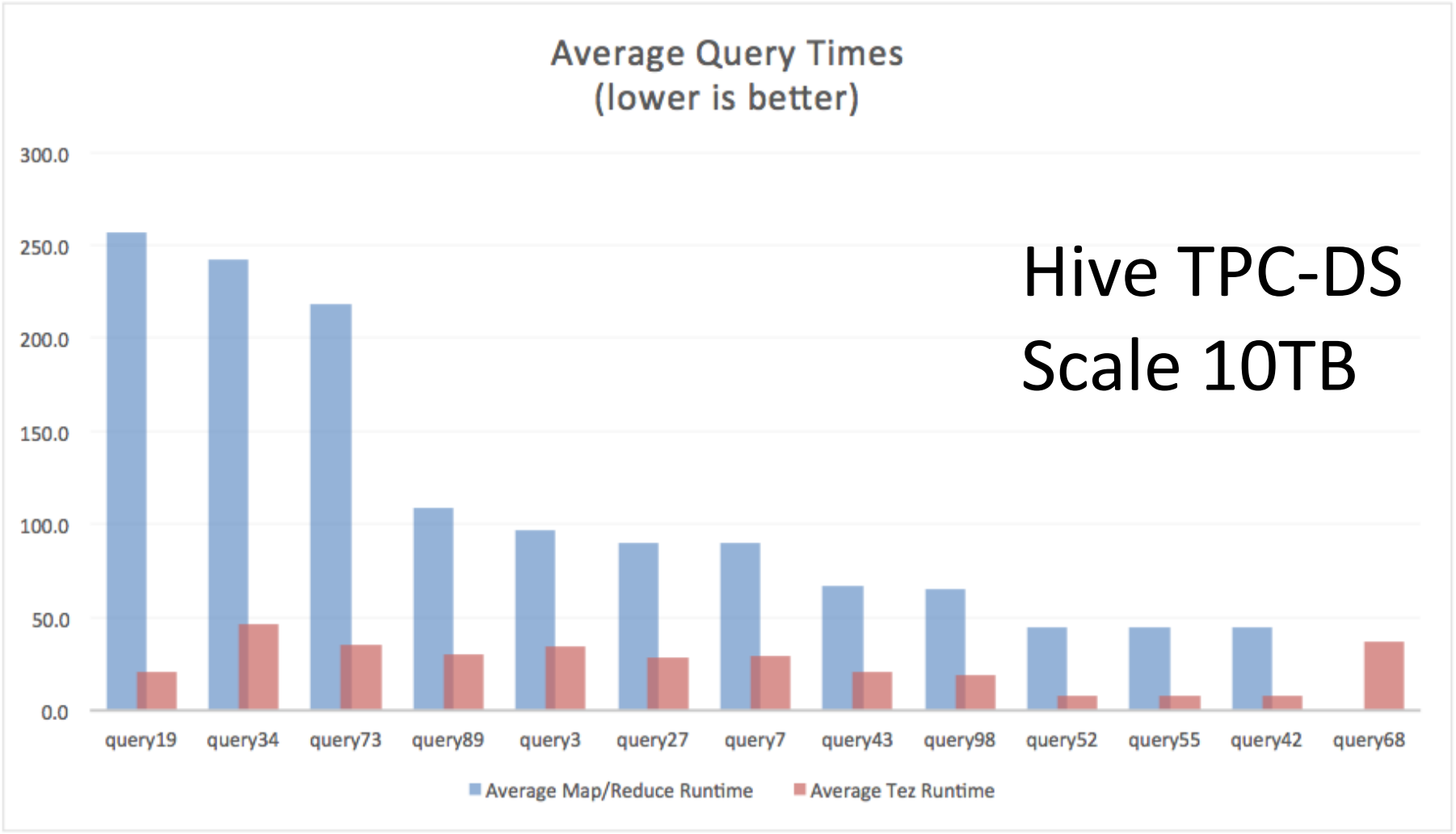
Hive – Tez

Broadcast Join (scan date_dim, join store sales)

Distinct for customer + items

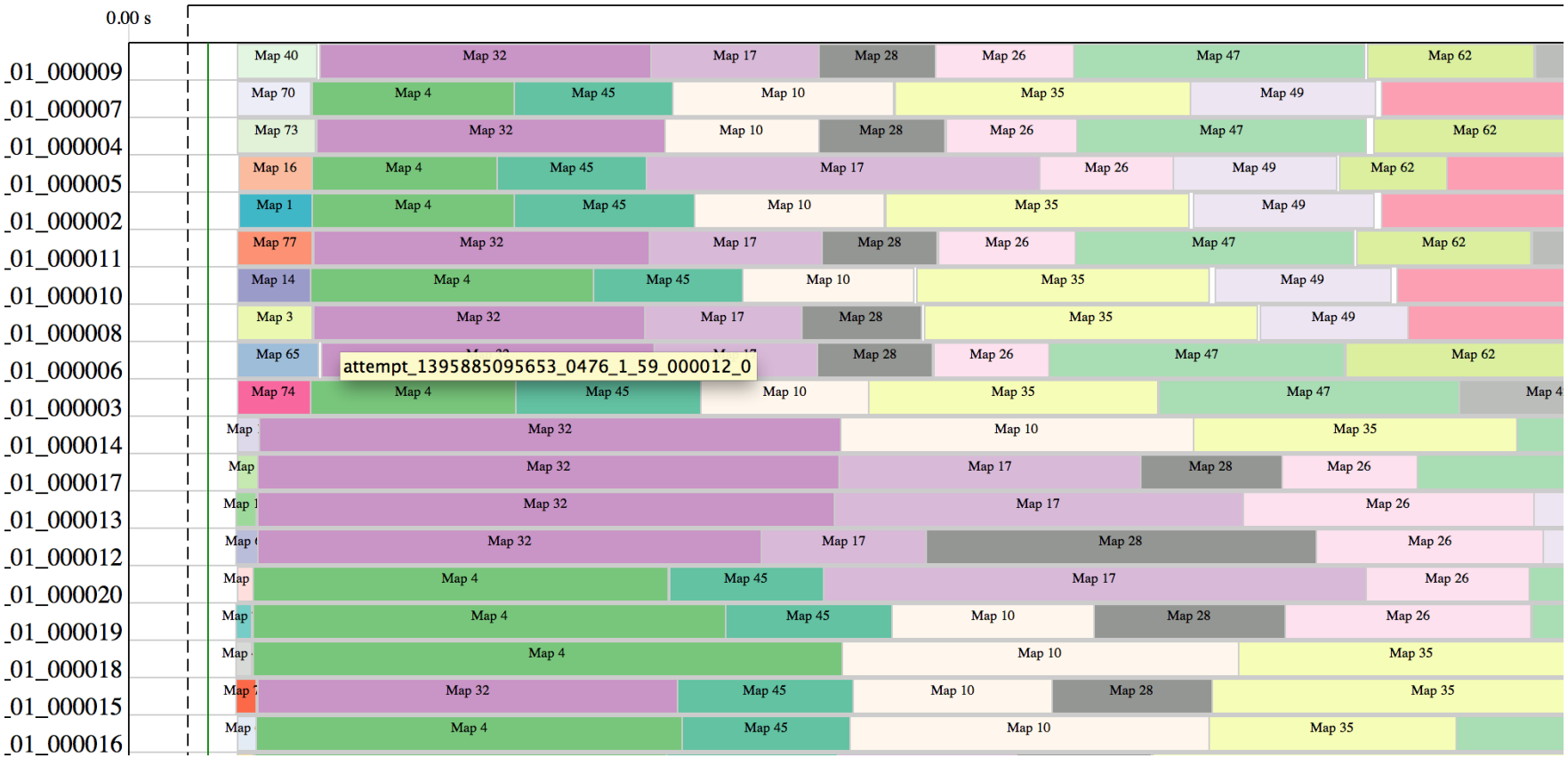


Recap: Tez – Data at scale



Tez – what if you can't get enough containers?

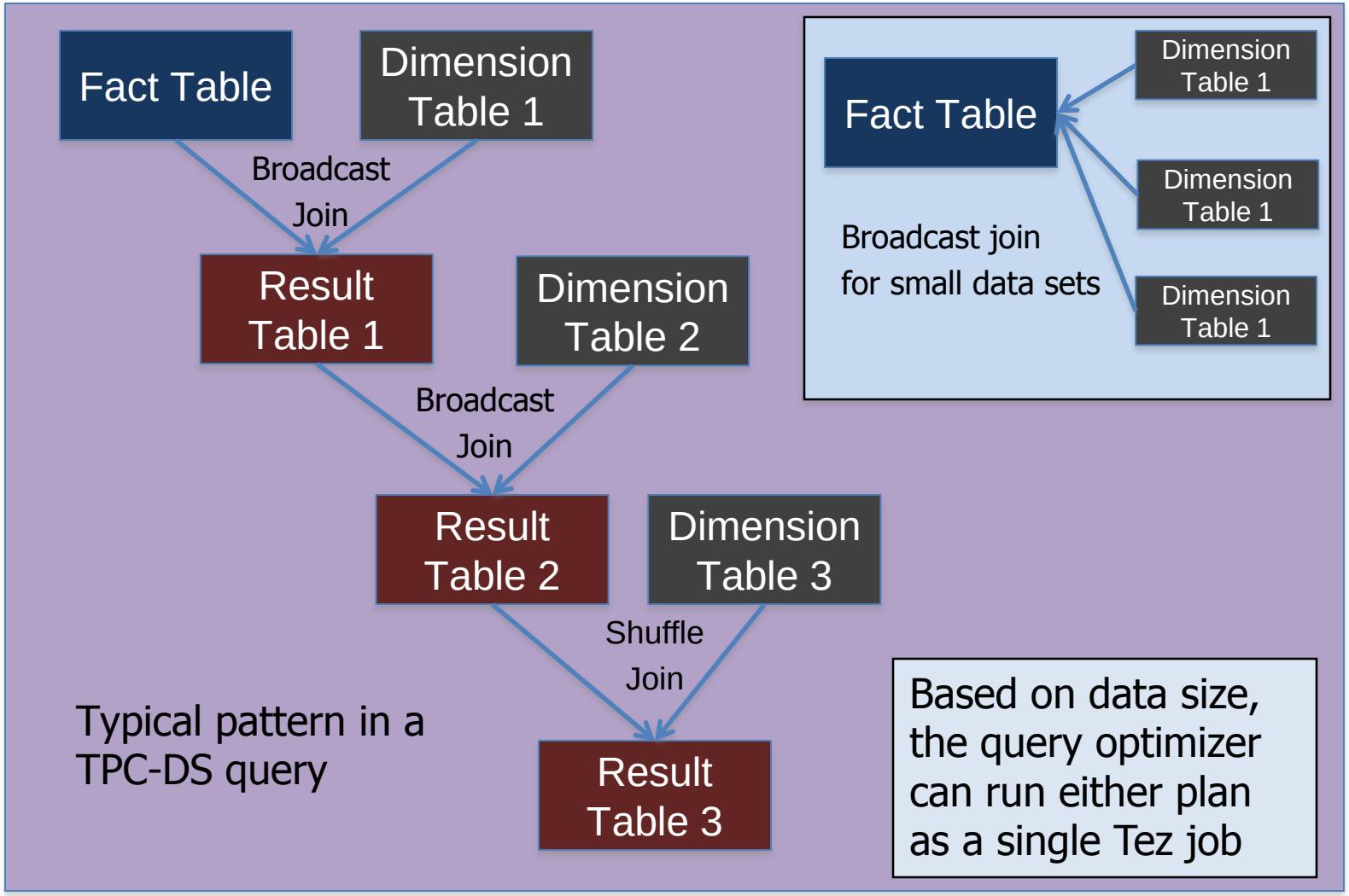
- 78 vertex + 8374 tasks on 50 YARN containers



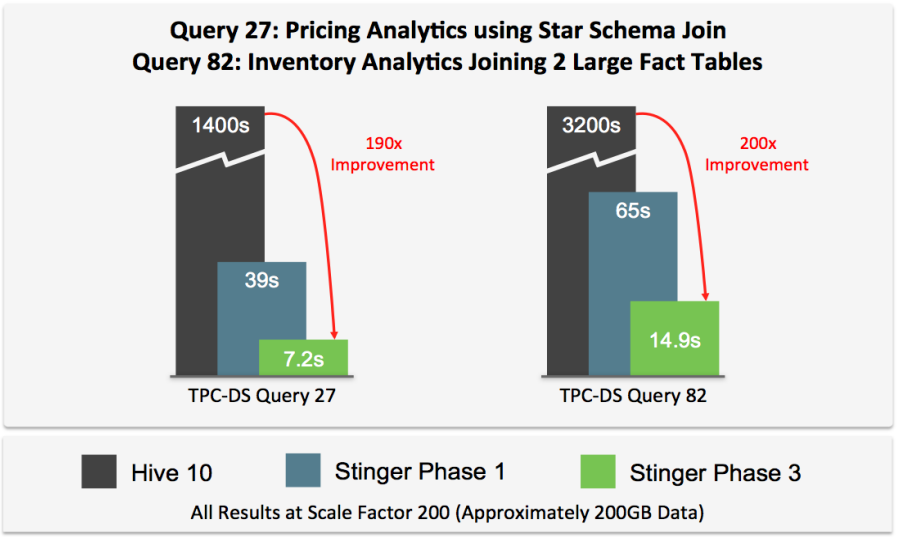
Tez – Designed for big, busy clusters

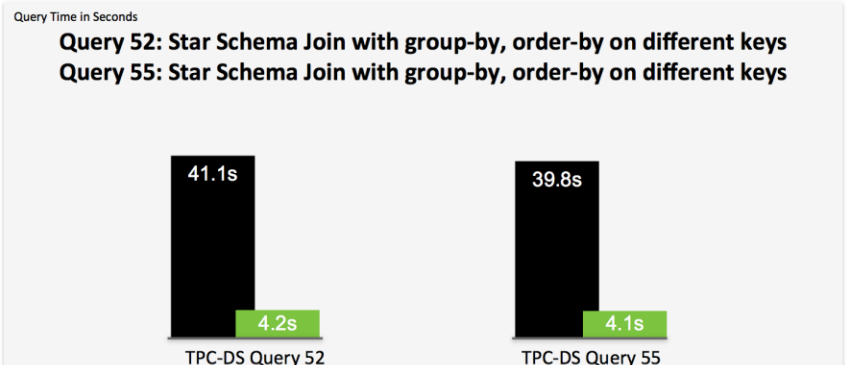
- Number of stages in the DAG
  - Higher the number of stages in the DAG, performance of Tez (over MR) will be better.
- Cluster/queue capacity
  - More congested a queue is, the performance of Tez (over MR) will be better due to container reuse.
- Size of intermediate output
  - Larger the size of intermediate output, the performance of Tez (over MR) will be better due to reduced HDFS usage (cross-rack traffic)
- Size of data in the job
  - For smaller data and more stages, the performance of Tez (over MR) will be better as percentage of launch overhead in the total time is high for smaller jobs.
- Move workloads from gateway boxes to the cluster
  - Move as much work as possible to the cluster by modelling it via the job DAG. Exploit the parallelism and resources of the cluster.

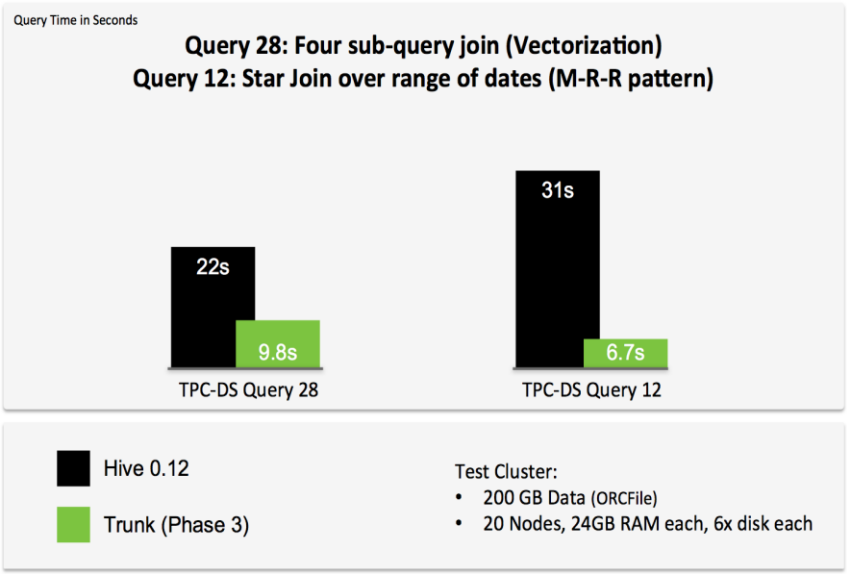
Tez – Bridge the Data Spectrum



Tez – Benchmark Performance



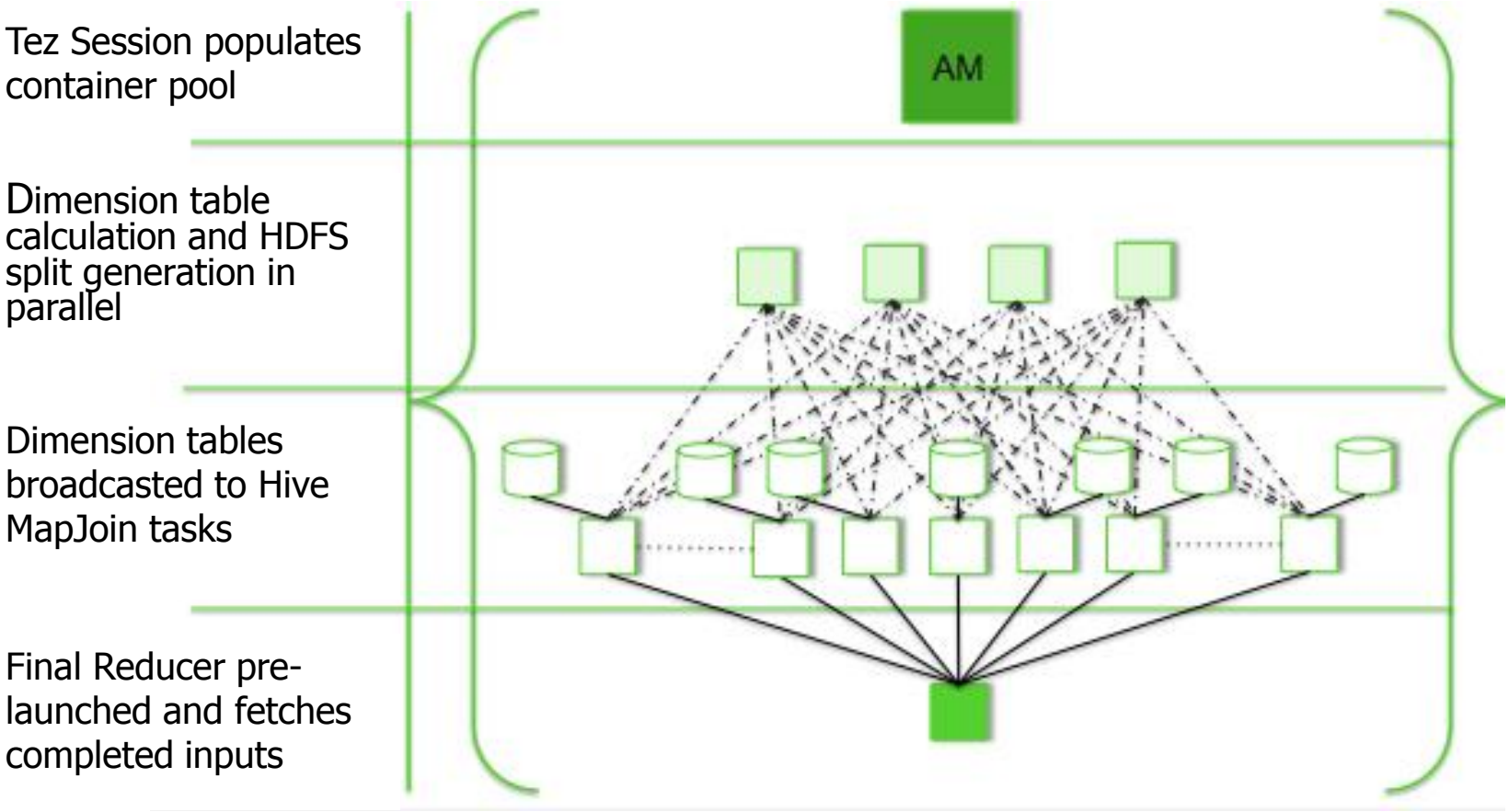




Significant (but not all) speedups due to Tez

- DAG support and runtime graph reconfiguration enable utilizing the parallelism of the cluster
- Tez Session and container reuse enable efficient and low latency execution

Tez – Performance Analysis



TPCDS – Query-27 with Hive on Tez

Architecting the Future of Big Data

Tez – Roadmap

- Richer DAG support
  - Support for co-scheduling and streaming
  - Better fault tolerance with checkpoints
- Performance optimizations
  - More efficiencies in transfer of data
  - Improve session performance
- Usability.
  - Stability and testability
  - Recovery and history
  - Tools for performance analysis and debugging

Tez – Adoption

- Hive
  - Hadoop standard for declarative access via SQL-like interface
- Pig
  - Hadoop standard for procedural scripting and pipeline processing
- Cascading
  - Developer friendly Java API and SDK
  - Scalding (Scala API on Cascading)
- Commercial Vendors
  - ETL : Use Tez instead of MR or custom pipelines
  - Analytics Vendors : Use Tez as a target platform for scaling parallel analytical tools to large data-sets

Tez – Community

- Technical blog series
  - http://hortonworks.com/blog/apache-tez-a-new-chapter-in-hadoop-data-processing
  - Apache Wiki https://cwiki.apache.org/confluence/display/TEZ/Index
- Tez meetup for developers and users
  - http://www.meetup.com/Apache-Tez-User-Group
- Hive and Pig communities are on-board and making great progress -
  - HIVE-4660 and PIG-3446
- Useful links
  - Work tracking: https://issues.apache.org/jira/browse/TEZ
  - Code: https://github.com/apache/tez
  - Developer list: dev@tez.apache.org
    User list: user@tez.apache.org
    Issues list: issues@tez.apache.org

Tez Summary

- Distributed execution framework that works on computations represented as dataflow graphs
- Naturally maps to execution plans produced by query optimizers
- Customizable execution architecture designed to enable dynamic performance optimizations at runtime
- Span the spectrum of interactive latency to batch
- It is already being used by Hive and Pig

Dataflow systems

Why dataflow systems have been so popular upto now?

- Collection-oriented programming model
  - Operations on collections of objects
  - Turn spurious (unordered) `for` into `foreach`
  - Not every `for` is `foreach`
    - Aggregation (sum, count, max, etc.)
    - Grouping
    - Join, Zip
  - Iteration
- LINQ since ca 2008, now Spark via Scala, Java

Well-chosen syntactic sugar

1. Read the lines of text
2. Split each line into its constituent words
3. Count how many times each word appears
4. Find the most commonly occurring words

```
1.  var lines = FS.ReadAsLines(inputFi
2.  var words = lines.SelectMany(x =>
3.  var counts = words.CountInGroups();
4.  var highest =
     counts.OrderByDescending(x => x.count).Take(10);
```

red blue
red,2
blue blue
blue,4
yellow
yellow,3
yellow red

Type inference

Lambda expressions

Generics and extension methods

Collections compile to dataflow

- Each operator specifies a single data-parallel step
- Communication between steps explicit
  - Collections reference collections, not individual objects!
  - Communication under control of the system
    - Partition, pipeline, exchange automatically
- LINQ innovation: embedded user-defined functions

```
var words = lines.SelectMany(x => x.Split(‘ ‘));
```

  - Very expressive
  - Programmer ‘naturally’ writes pure functions

Quiet revolution in parallelism

- Programming model is *more* attractive
  - Simpler, more concise, readable, maintainable
- Program is *easier* to optimize
  - Programmer separates computation and communication
  - System can re-order, distribute, batch, etc. etc.

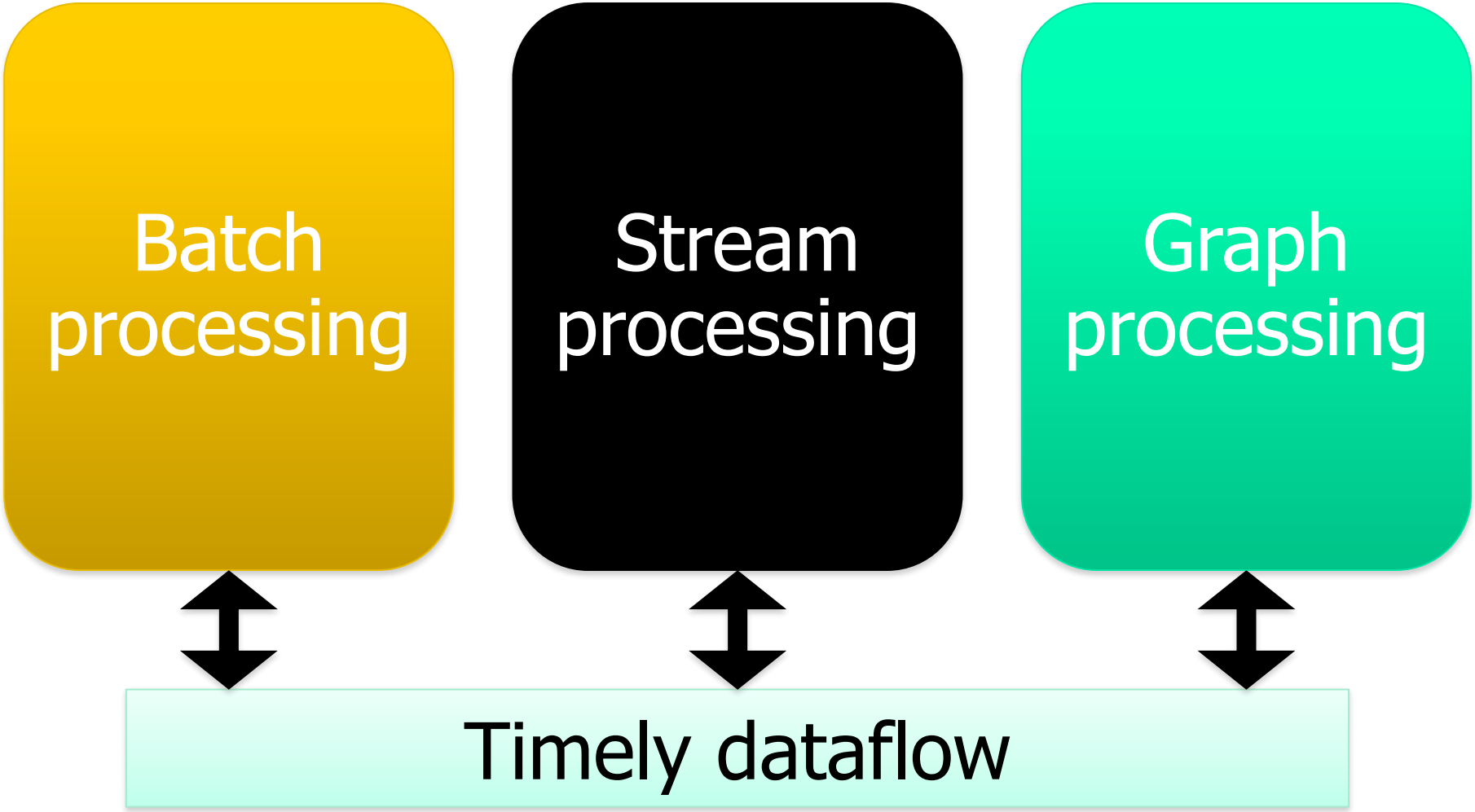
Stateless DAG dataflow

- MapReduce, Dryad, Spark, …
- Stateless vertex constraint hampers performance
  - Iteration and streaming overheads
- Why does this design keep repeating?
  - An Engineering Sweet Spot !

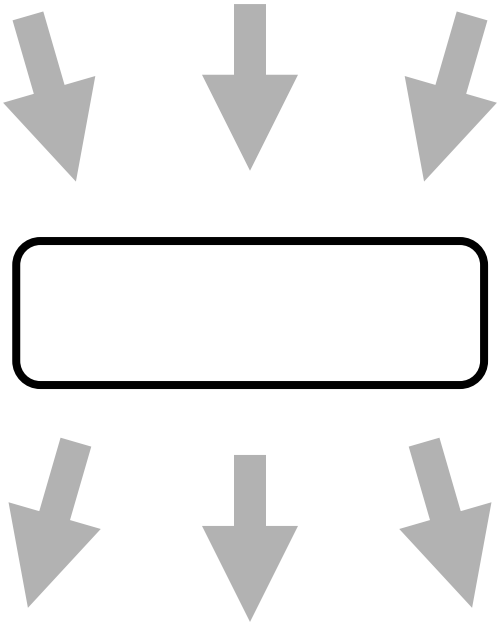
Software engineering

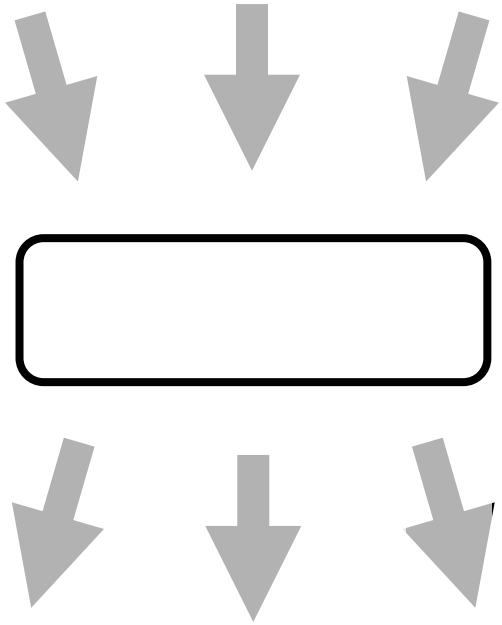
- Fault tolerance well understood
  - E.g., Chandy-Lamport, rollback recovery, etc.
- Basic mechanism: checkpoint plus log
- Stateless DAG: no checkpoint!
- Programming model “tricked” user
  - All communication on typed channels
  - Only channel data needs to be persisted
  - Fault tolerance comes without programmer effort
  - Even with UDFs

Beyond Stateless DAG Dataflow


Batch processing
Stream processing
Graph processing
Timely dataflow

Batching (synchronous) vs. Streaming (asynchronous)





- ✗ Requires coordination
- ✓ Supports aggregation

- ✓ No coordination needed
- ✗ Aggregation is difficult

Mutable state

- In batch DAG systems collections are immutable
  - Functional definition in terms of preceding subgraph
- Adding streaming or iteration introduces mutability
  - Collection varies as function of epoch, loop iteration

What about *stateful* dataflow?

- Microsoft Naiad
  - Add state to vertices
  - Support streaming and iteration
- Opportunities
  - Much lower latency
  - Can model mutable state with dataflow
- Challenges
  - Scheduling
  - Coordination
  - Fault tolerance

What can't dataflow do?

- Programming model for mutable state?
  - Not as intuitive as functional collection manipulation
- Policies for placement still primitive
  - Hash everything and hope
- Great research opportunities
  - Intersection of OS, network, runtime, language

Conclusions

- Dataflow is a great structuring principle
  - We know good programming models
  - We know how to write high-performance systems
- Dataflow is the status quo for batch processing
- Mutable state (i.e. Stateful Dataflow Systems supporting incremental recovery) are the current (circa 2015) research frontier