The Armada Parallel I/O framework for Computational Grids

Ron Oldfield and David Kotz

{raoldfi,dfk}@cs.dartmouth.edu.

Department of Computer Science, Dartmouth College

http://www.cs.dartmouth.edu/∼dfk/armada/

September 20, 2002

Computational Grids

Networks of geographically distributed heterogeneous systems and devices.

Properties of computational grids

- Dynamic resources
- Heterogeneous components
- Multiple administrative domains
- High-latency networks

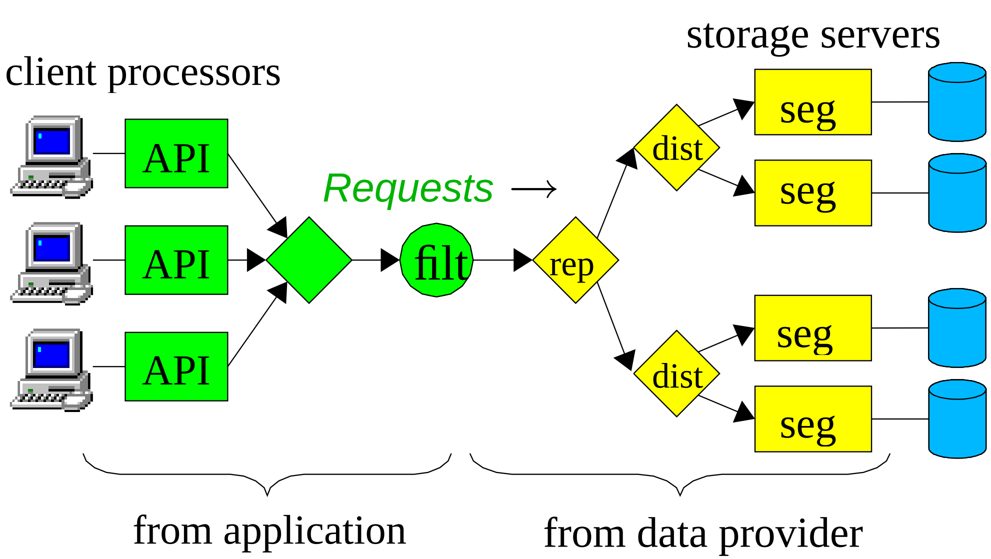
An important challenge facing grid computing is efficient I/O for data-intensive applications.

Grid Applications

- *Computationally intensive:* may require supercomputers
- Many are also *data intensive:*
  - Access large remote datasets (terabytes)
  - Datasets often need pre, and/or post-processing
- Examples
  - Seismic processing
  - Climate modeling
  - Astronomy
  - Computational Biology
  - High-energy physics

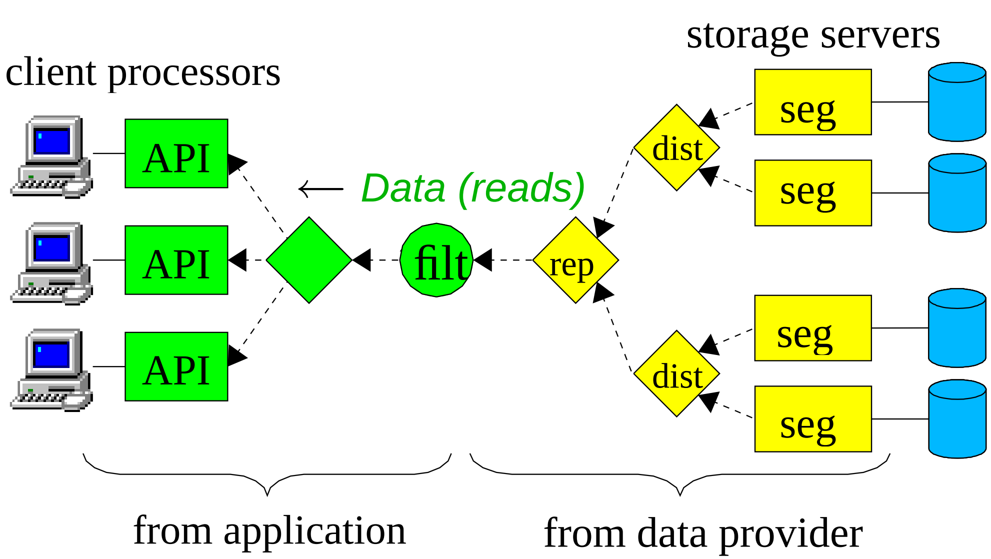
The Armada Framework

- Application deploys a graph of distributed objects (*ships*)
- Data request causes pipelined data flow through graph
- Graph has two distinct portions:
  - from the data provider (describes layout of data set)
  - from the application programmer (pre/post-processing)



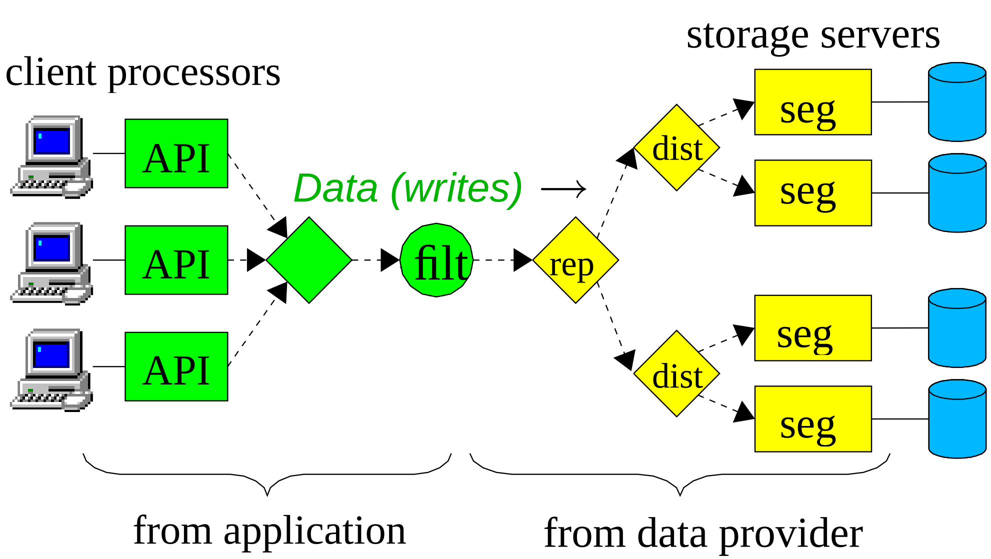
The Armada Framework

- Application deploys a graph of distributed objects (*ships*)
- Data request causes pipelined data flow through graph
- Graph has two distinct portions:
  - from the data provider (describes layout of data set)
  - from the application programmer (pre/post-processing)



The Armada Framework

- Application deploys a graph of distributed objects (*ships*)
- Data request causes pipelined data flow through graph
- Graph has two distinct portions:
  - from the data provider (describes layout of data set)
  - from the application programmer (pre/post-processing)



Armada

Armada is not a data storage system.
Armada is not a parallel file system.

The *data segments* that make up a *data set* are stored in conventional data servers as files, databases, or the like.

The Armada graph encodes most functionality provided by the I/O system:

- programmers interface,
- data layout,
- caching and prefetching policies,
- interfaces to heterogeneous data servers.

Armada can...

With Armada, one can

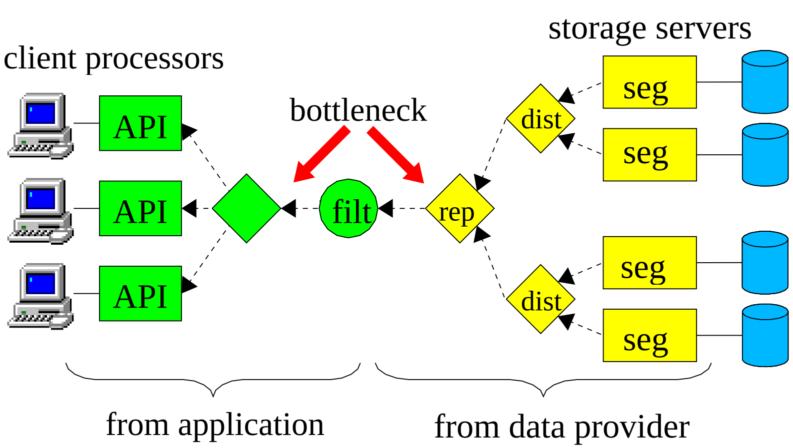
- build a graph for parallel access to a group of legacy files,
- present many similar data sets through a standard interface, and
- provide transparent access to derived “virtual” data—either cached or calculated as needed.

Restructuring

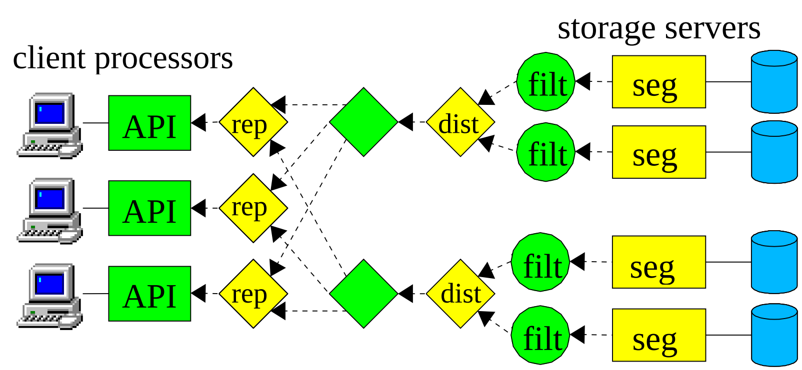
Problems with the example application:

- potential bottlenecks in the composed graph
- original graph restricts placement alternatives for filter

Original graph



Restructured graph

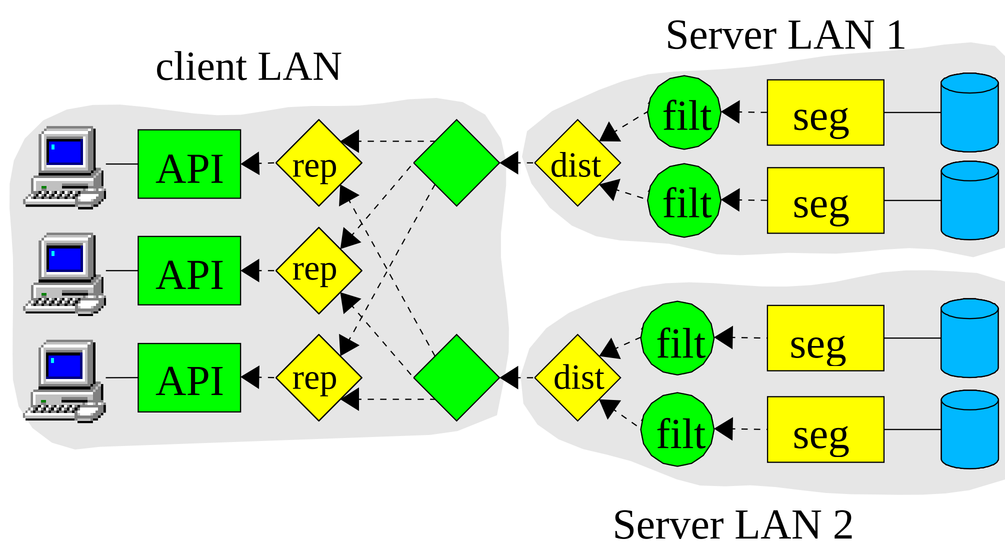


Armada restructures original graph to improve data flow.

Placement

After restructuring:

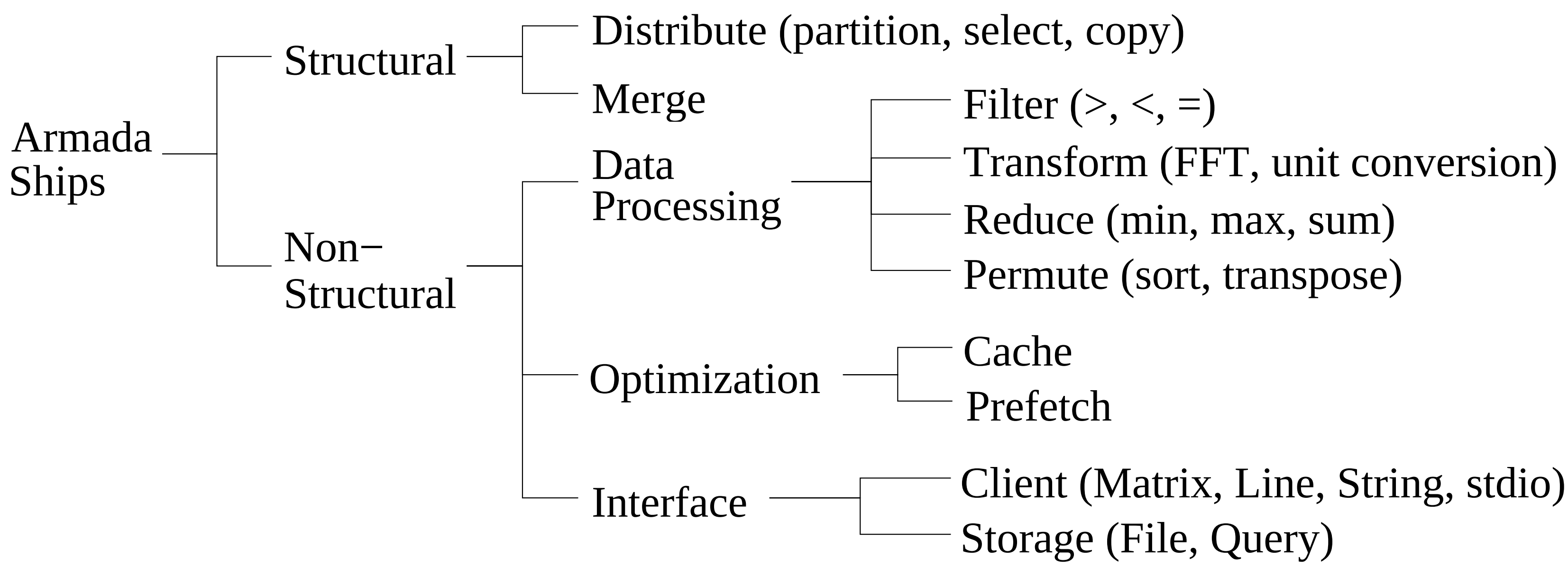
1. Armada deploys ships to appropriate administrative domains to optimize data flow, then
2. domain-level resource managers decide placement of individual ships.



Work in progress...

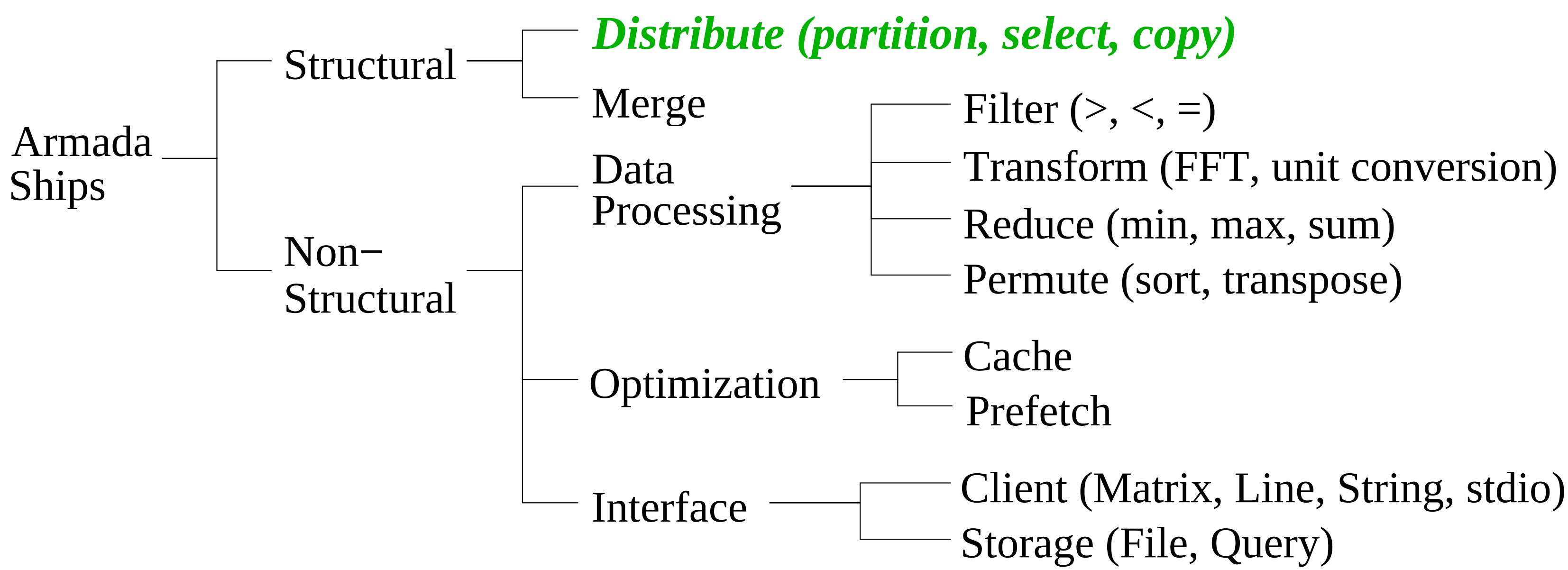
Ships

Armada includes a rich set of extensible ship classes.

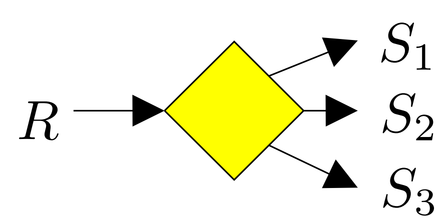


Ships

Armada includes a rich set of extensible ship classes.

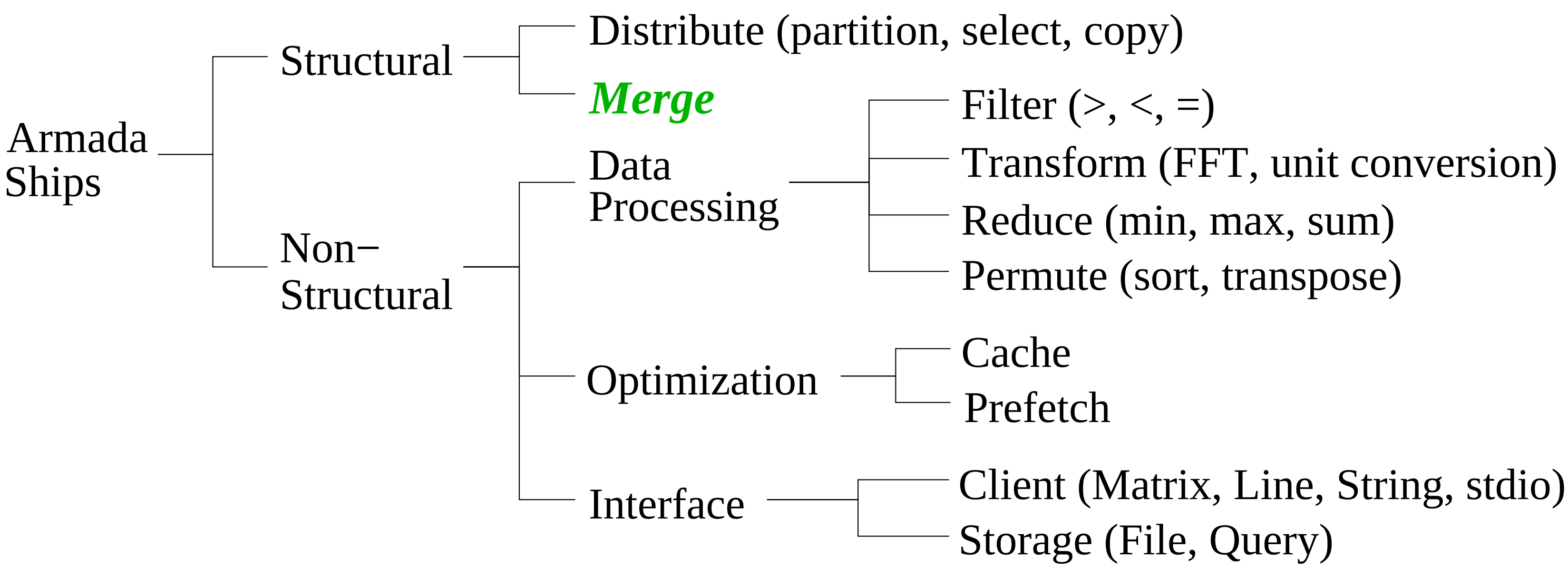


Distribute ships partition requests or data to multiple output streams.

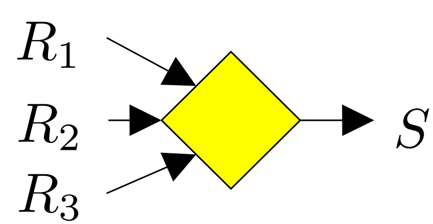


Ships

Armada includes a rich set of extensible ship classes.

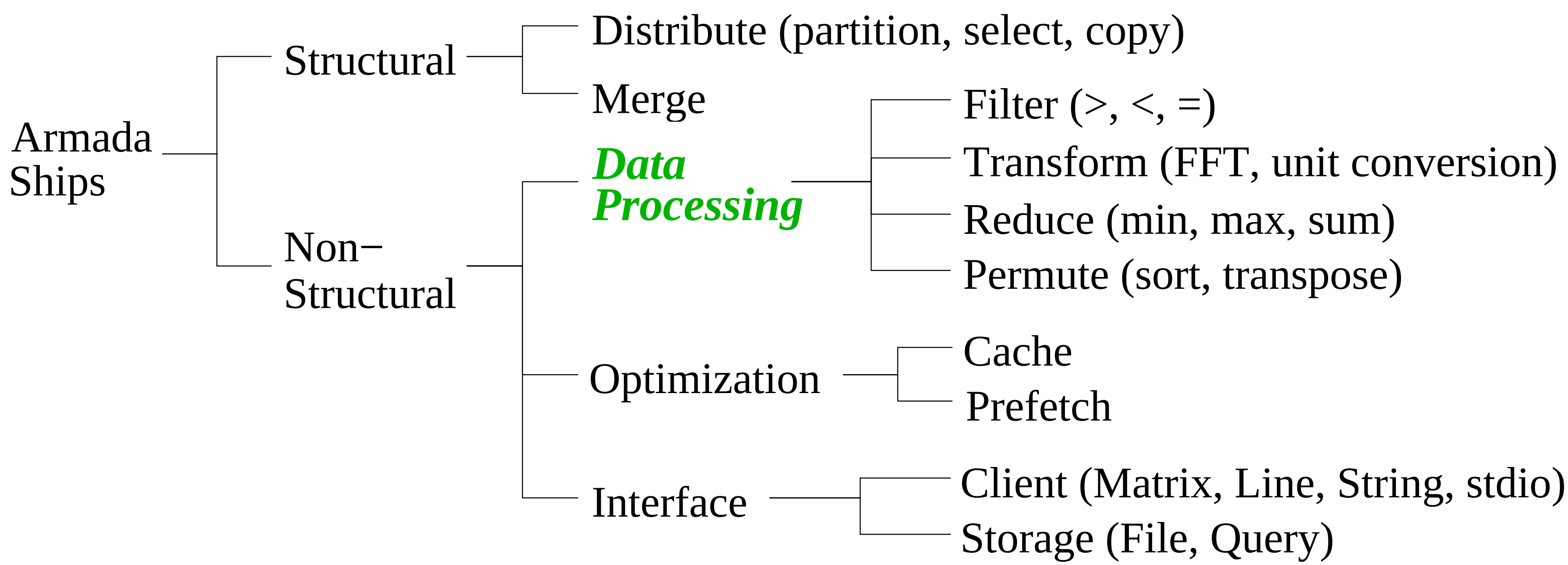


Merge ships interleave requests or data from multiple input streams.

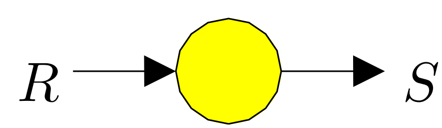


Ships

Armada includes a rich set of extensible ship classes.

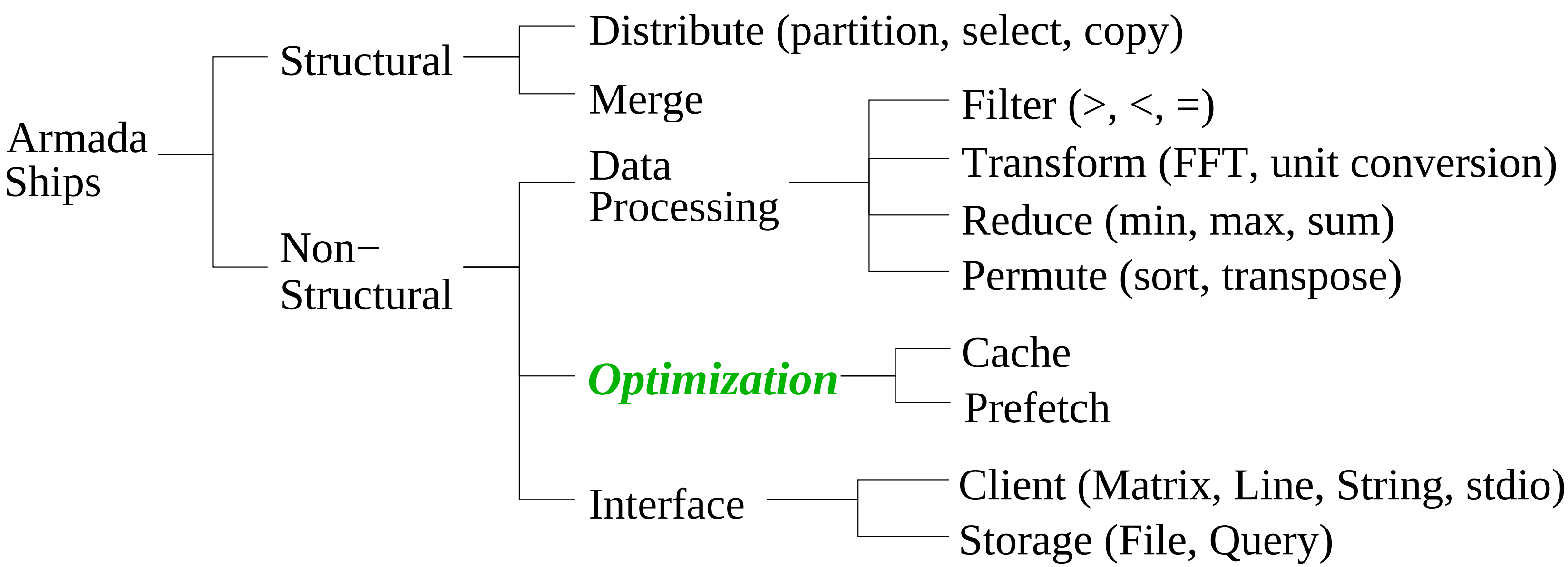


Data-processing ships manipulate data, either individually, or in groups as it passes through the ship.

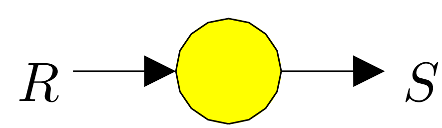


Ships

Armada includes a rich set of extensible ship classes.

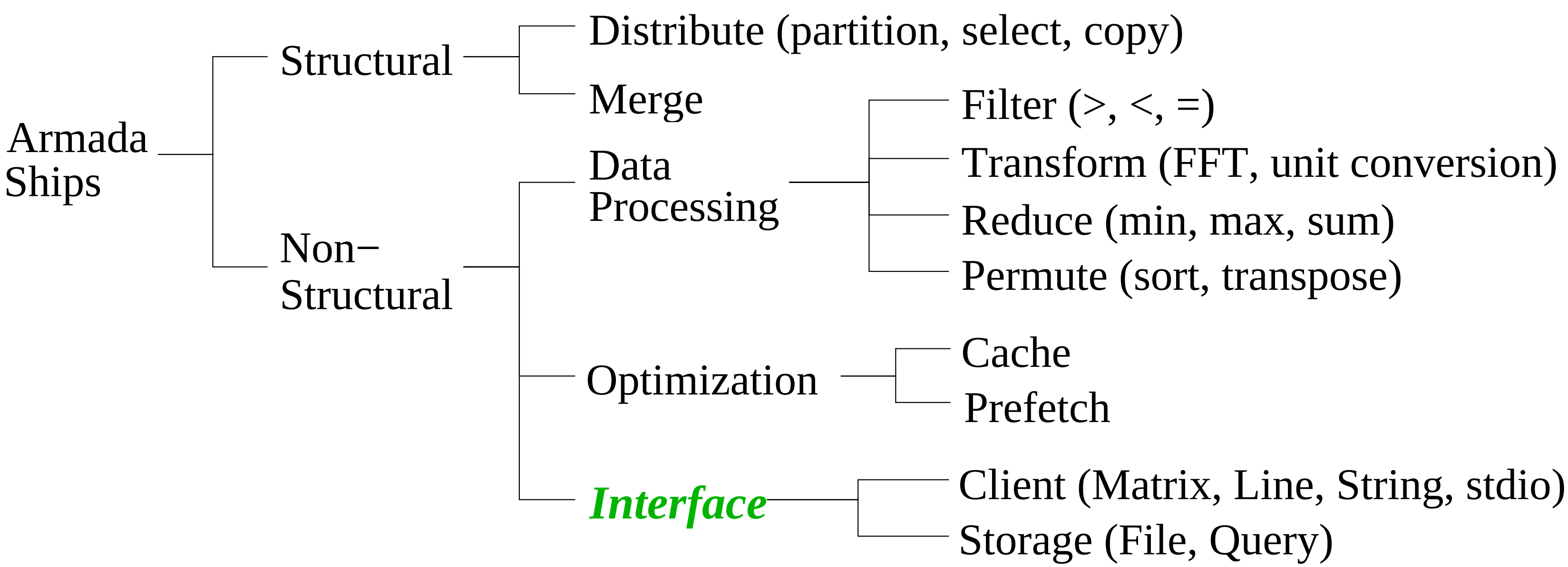


Optimization ships improve I/O performance through latency-reduction techniques like caching and prefetching.

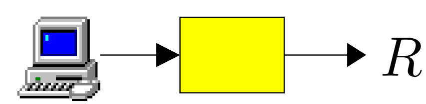


Ships

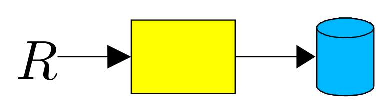
Armada includes a rich set of extensible ship classes.



Client-interface ships
convert method calls to a set of requests for data.



Storage-interface ships
access storage devices to process requests.



Properties of Ships

Properties of ships are

- used by restructuring and placement algorithms
- assigned by the programmer
- encoded in the ship's description

Properties identify whether a ship

- is data- or request-equivalent
- increases or decreases data flow
- is parallelizable

Request- and Data-Equivalent Ships

A sequence A is *equivalent* to sequence B (denoted $A \equiv B$)
if B is a permutation of A, or
if B is a set of subsequences that partition A.

Examples:

$$\{1,2,3,4,5\} \equiv \{2,3,5,1,4\}$$
$$\{1,2,3,4,5\} \equiv \{\{2,3\},\ \{1,4,5\}\}$$
$$\{1,2,3,4,5\} \equiv \{\{2,3\},\ \{1,5,4\}\}$$

In other words, order does not matter.

Request- and Data-Equivalent Ships

A sequence A is *equivalent* to sequence B (denoted $A \equiv B$)
if B is a permutation of A, or
if B is a set of subsequences that partition A.

A *request-equivalent* ship
produces request sequence equivalent to its input.
A *data-equivalent* ship
produces data sequence equivalent to its input.

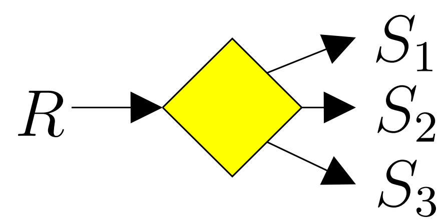
Most structural ships are both request and data-equivalent.

Request- and Data-Equivalent Ships

A sequence A is *equivalent* to sequence B (denoted $A \equiv B$)
if B is a permutation of A, or
if B is a set of subsequences that partition A.

Distribution ships partition requests or data

- S_1, S_2, and S_3 are disjoint subsets of R.
- $R \equiv \{S_1, S_2, S_3\}$

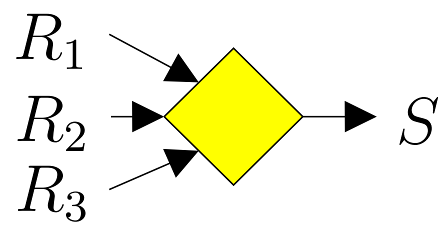


Request- and Data-Equivalent Ships

A sequence A is *equivalent* to sequence B (denoted $A \equiv B$)
if B is a permutation of A, or
if B is a set of subsequences that partition A.

Merge ships interleave requests or data

- R_1, R_2, and R_3 are disjoint subsets of S.
- $\{R_1, R_2, R_3\} \equiv S$



Ships that Change Data Flow

Data-reducer: a ship that decreases the data flow

- filter
- compress
- reduce (min, max, sum)

Data-increaser: a ship that increases the data flow

- cache
- decompress

Parallelizable Ships

Parallelizable: a ship that can transform into multiple ships

- process requests and data in parallel
- parallelized by “swapping” with structural ships
- parallel version produces *equivalent* output

Types of parallelizable ships: *replicatable*, *recursive*

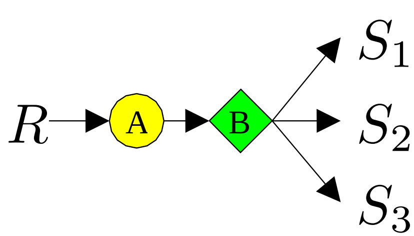
Parallelizable Ships

Parallelizable: a ship that can transform into multiple ships

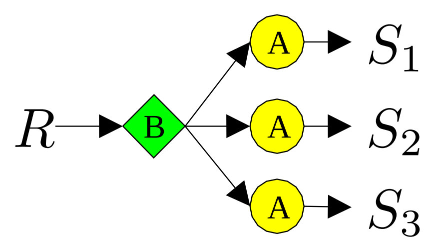
- process requests and data in parallel
- parallelized by “swapping” with structural ships
- parallel version produces *equivalent* output

Types of parallelizable ships: *replicatable*, *recursive*

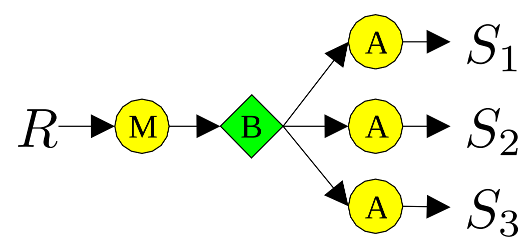
Right-parallelizable



Original



Replicated



Recursed

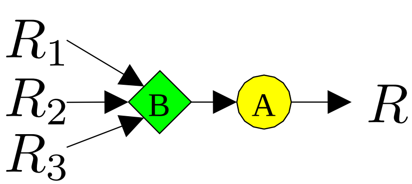
Parallelizable Ships

Parallelizable: a ship that can transform into multiple ships

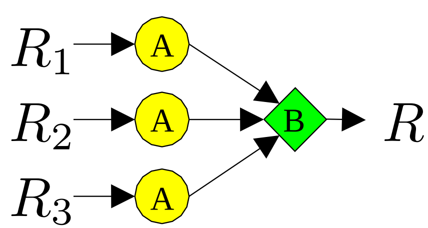
- process requests and data in parallel
- parallelized by “swapping” with structural ships
- parallel version produces *equivalent* output

Types of parallelizable ships: *replicatable*, *recursive*

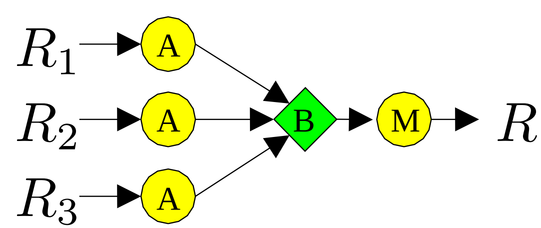
Left-parallelizable



Original



Replicated

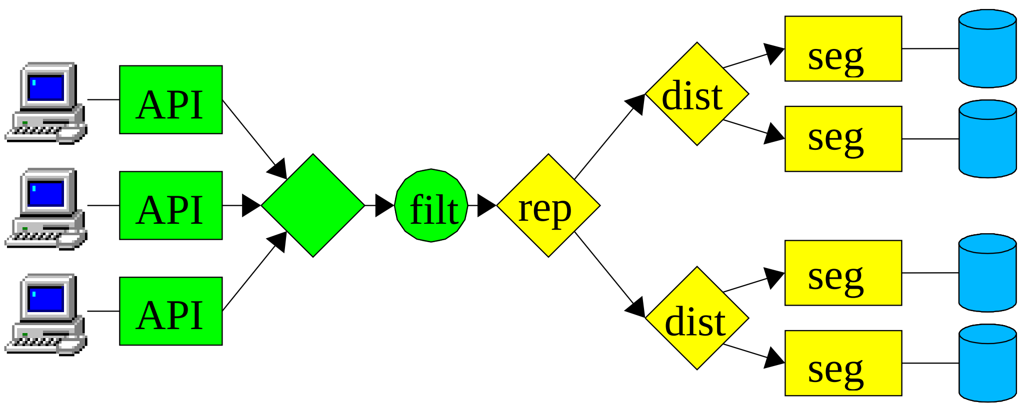


Recursed

Graph Representation

We use a *series-parallel tree* (SP-tree) to describe the composition of an Armada graph.

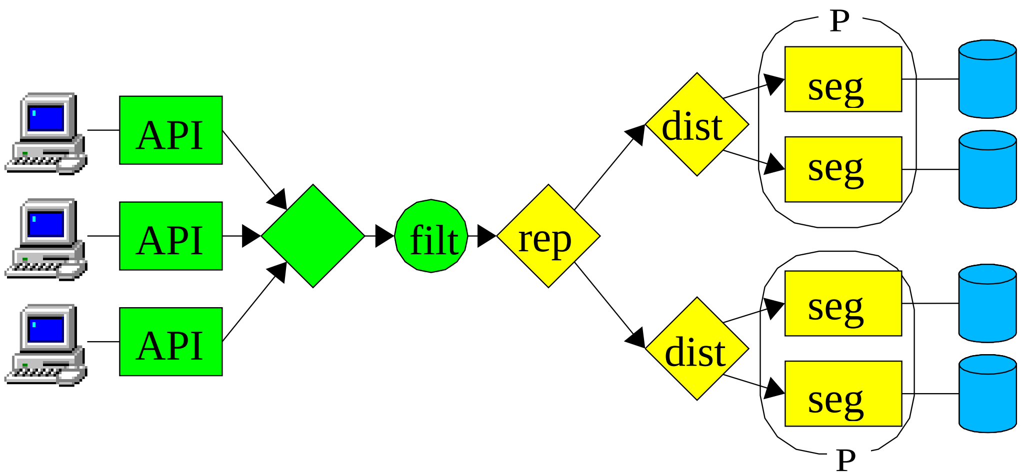
- Syntactically easy to describe (we use XML)
- Easy to manipulate internally
- Constrains the graph to be an SP-DAG (important for restructuring)

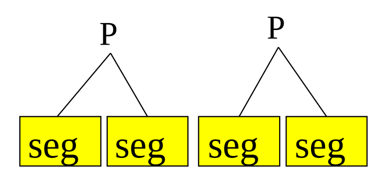


Graph Representation

We use a *series-parallel tree* (SP-tree) to describe the composition of an Armada graph.

- Syntactically easy to describe (we use XML)
- Easy to manipulate internally
- Constrains the graph to be an SP-DAG (important for restructuring)

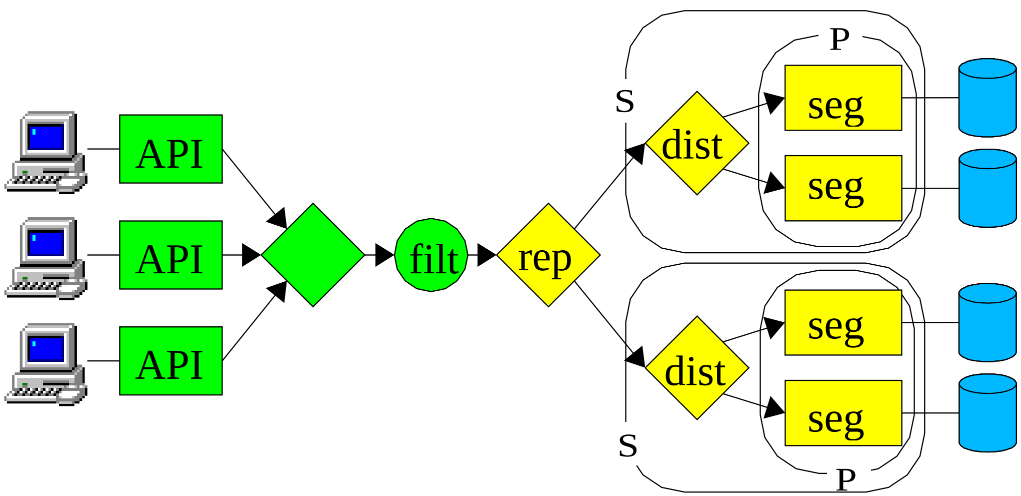


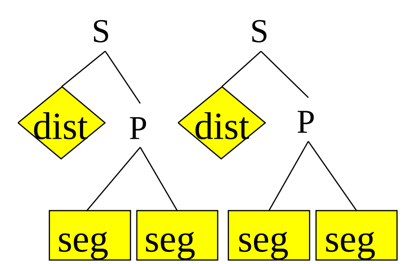


Graph Representation

We use a *series-parallel tree* (SP-tree) to describe the composition of an Armada graph.

- Syntactically easy to describe (we use XML)
- Easy to manipulate internally
- Constrains the graph to be an SP-DAG (important for restructuring)

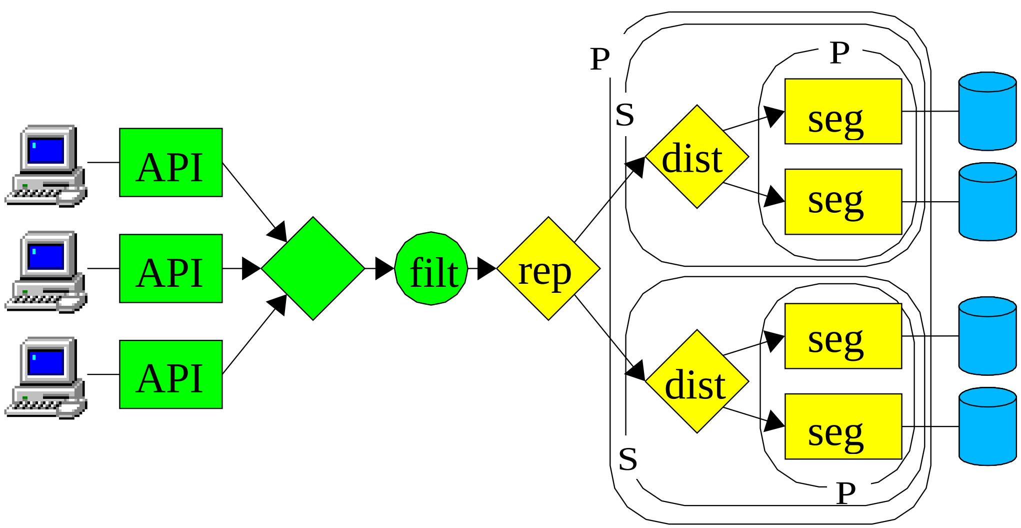


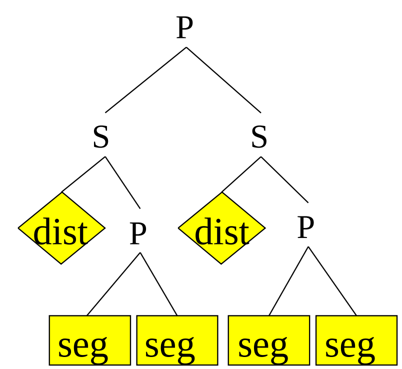


Graph Representation

We use a *series-parallel tree* (SP-tree) to describe the composition of an Armada graph.

- Syntactically easy to describe (we use XML)
- Easy to manipulate internally
- Constrains the graph to be an SP-DAG (important for restructuring)

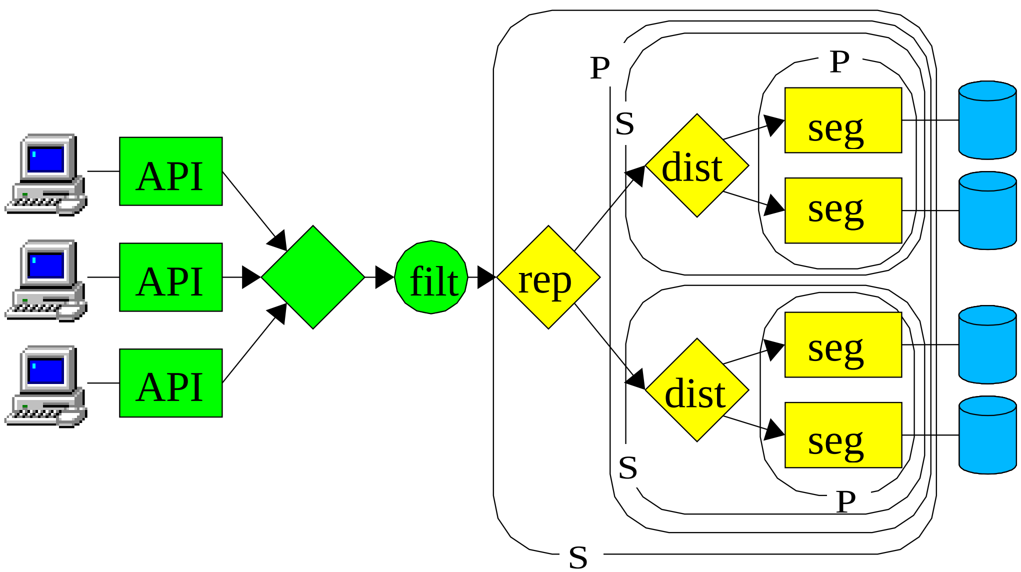


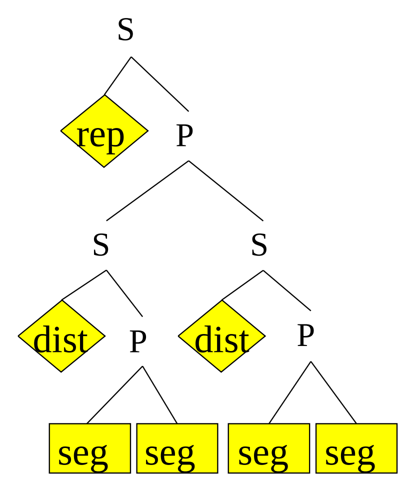


Graph Representation

We use a *series-parallel tree* (SP-tree) to describe the composition of an Armada graph.

- Syntactically easy to describe (we use XML)
- Easy to manipulate internally
- Constrains the graph to be an SP-DAG (important for restructuring)

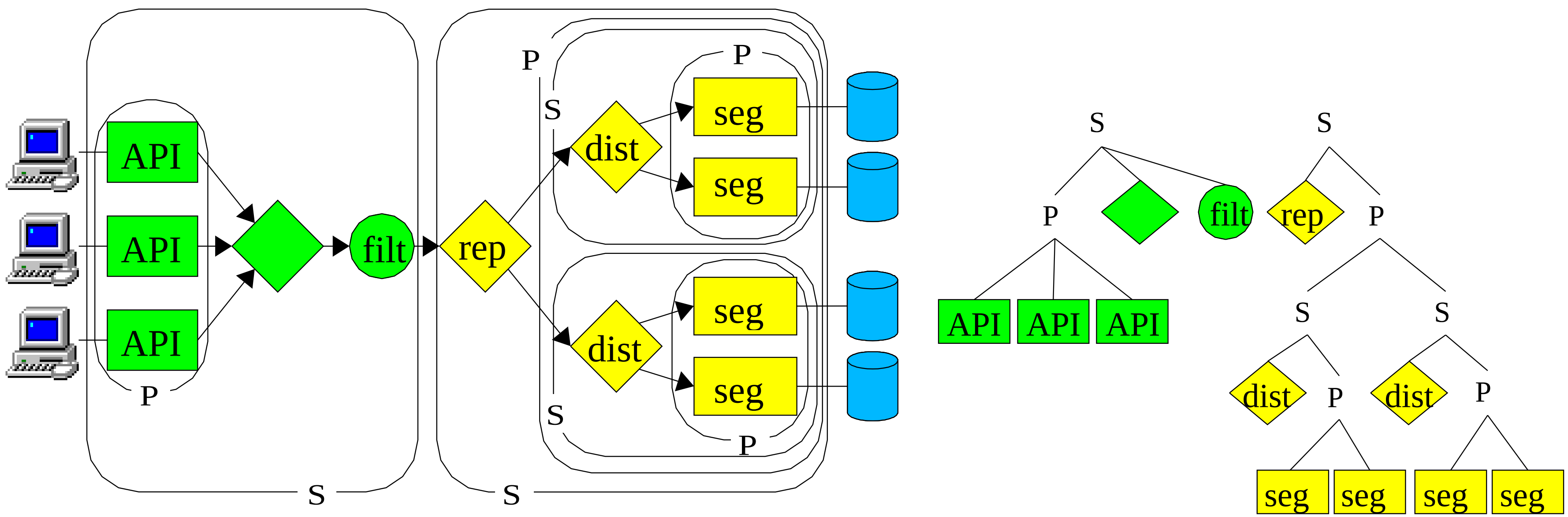




Graph Representation

We use a *series-parallel tree* (SP-tree) to describe the composition of an Armada graph.

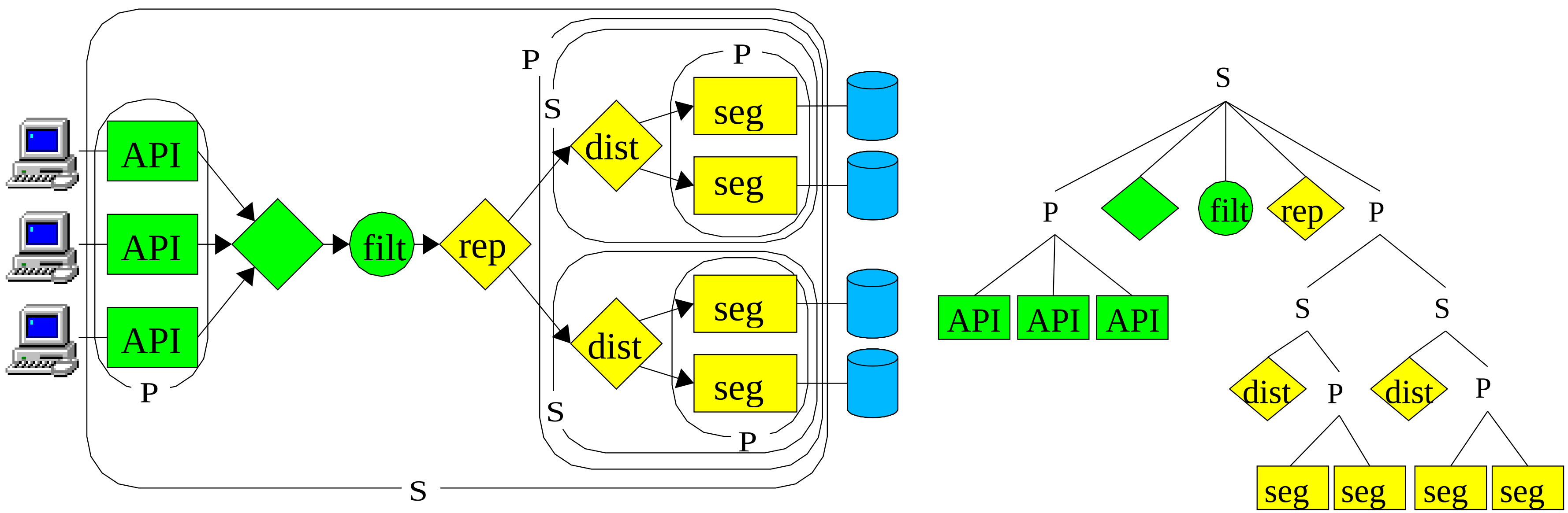
- Syntactically easy to describe (we use XML)
- Easy to manipulate internally
- Constrains the graph to be an SP-DAG (important for restructuring)



Graph Representation

We use a *series-parallel tree* (SP-tree) to describe the composition of an Armada graph.

- Syntactically easy to describe (we use XML)
- Easy to manipulate internally
- Constrains the graph to be an SP-DAG (important for restructuring)



Graph Restructuring

Goals:

- remove bottlenecks (increase parallelism)
- allow better placement to reduce network traffic

We restructure by *swapping* adjacent nodes in the SP-tree

- increase parallelism by swapping *parallelizable* ships with *structural* ships
- reduce network traffic on slow links by
  - moving *data-reducing* ships toward data source,
  - moving *data-increasing* ships toward data destination

The Restruct Algorithm

All series and parallel nodes are initially marked *dirty*.

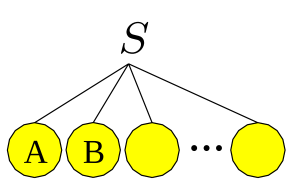
The *Restruct* algorithm traverses the SP-tree (depth-first), revisiting when necessary

1. if node is a leaf or clean (base case)
   (a) do nothing
2. if node is a dirty parallel node
   (a) recursively call *Restruct* on each child
   (b) mark node *clean*
3. if node is a dirty series node
   (a) call the *RestructSeries* algorithm
   (b) mark node *clean*

The RestructSeries Algorithm

1. Partition node into two disjoint series nodes $Head$ and $Tail$
2. Recursively call *Restruct* on both partitions
3. If it is *legal* and *beneficial* to swap last child of $Head$ (A) with first child of $Tail$ (B)
   (a) Swap A and B
   (b) Mark $Head$ and $Tail$ dirty (force restructuring)
4. else
   (a) Append B to $Head$
5. If $Tail$ has children, goto 2

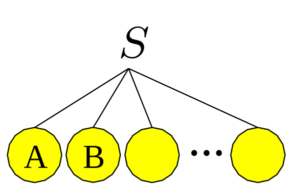
Original



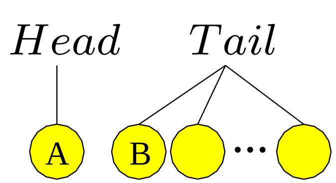
The RestructSeries Algorithm

1. Partition node into two disjoint series nodes $Head$ and $Tail$
2. Recursively call *Restruct* on both partitions
3. If it is *legal* and *beneficial* to swap last child of $Head$ (A) with first child of $Tail$ (B)
   (a) Swap A and B
   (b) Mark $Head$ and $Tail$ dirty (force restructuring)
4. else
   (a) Append B to $Head$
5. If $Tail$ has children, goto 2

Original

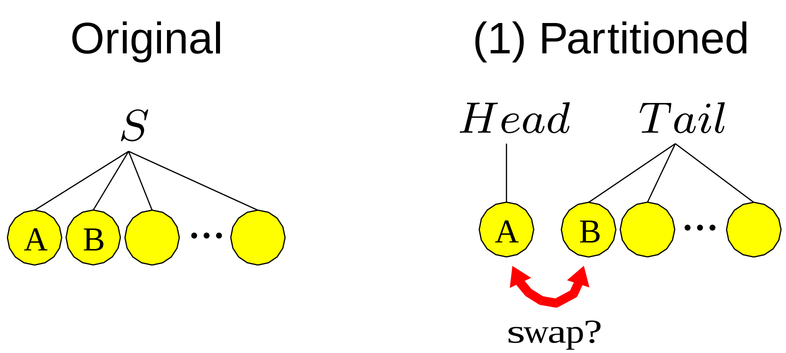


(1) Partitioned



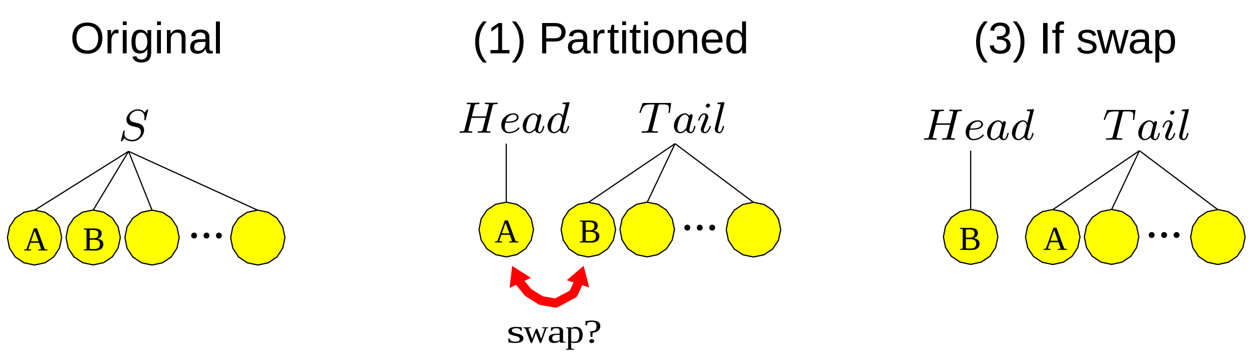
The RestructSeries Algorithm

1. Partition node into two disjoint series nodes $Head$ and $Tail$
2. Recursively call *Restruct* on both partitions
3. If it is *legal* and *beneficial* to swap last child of $Head$ (A) with first child of $Tail$ (B)
   (a) Swap A and B
   (b) Mark $Head$ and $Tail$ dirty (force restructuring)
4. else
   (a) Append B to $Head$
5. If $Tail$ has children, goto 2



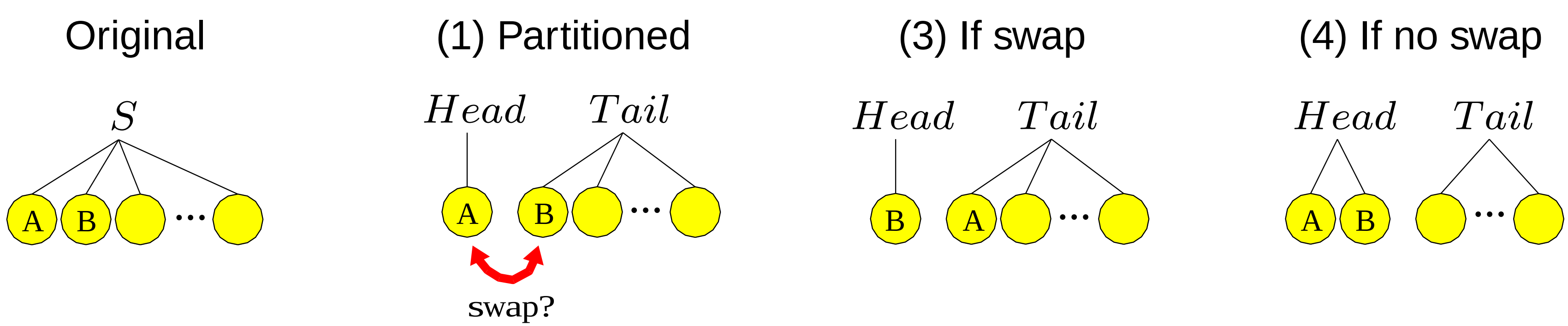
The RestructSeries Algorithm

1. Partition node into two disjoint series nodes $Head$ and $Tail$
2. Recursively call *Restruct* on both partitions
3. If it is *legal* and *beneficial* to swap last child of $Head$ (A) with first child of $Tail$ (B)
   (a) Swap A and B
   (b) Mark $Head$ and $Tail$ dirty (force restructuring)
4. else
   (a) Append B to $Head$
5. If $Tail$ has children, goto 2



The RestructSeries Algorithm

1. Partition node into two disjoint series nodes $Head$ and $Tail$
2. Recursively call *Restruct* on both partitions
3. If it is *legal* and *beneficial* to swap last child of $Head$ (A) with first child of $Tail$ (B)
   (a) Swap A and B
   (b) Mark $Head$ and $Tail$ dirty (force restructuring)
4. else
   (a) Append B to $Head$
5. If $Tail$ has children, goto 2



Legal Swap

It is legal to swap adjacent ships A and B if

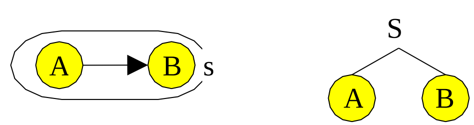
1. the swap must produce an equivalent sequence
   - that is, ship A and B are *commutative*
   - A or B is request-equivalent and A or B is data-equivalent
2. the swap must produce an SP-tree (we allow four configs)

Legal Swap

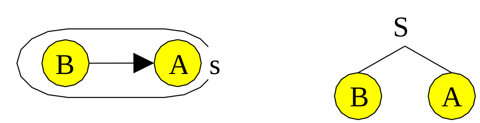
It is legal to swap adjacent ships A and B if

1. the swap must produce an equivalent sequence
   - that is, ship A and B are *commutative*
   - A or B is request-equivalent and A or B is data-equivalent
2. the swap must produce an SP-tree (we allow four configs)

A (non-structural) — B (non-structural)



Original



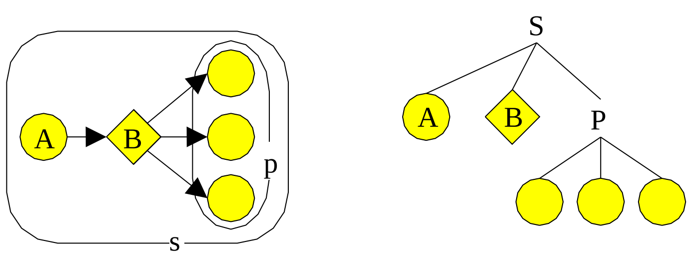
Swapped

Legal Swap

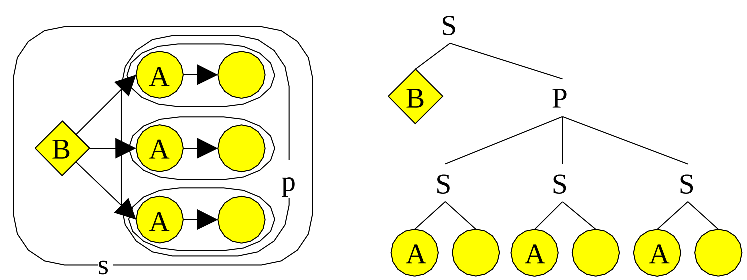
It is legal to swap adjacent ships A and B if

1. the swap must produce an equivalent sequence
   - that is, ship A and B are *commutative*
   - A or B is request-equivalent and A or B is data-equivalent
2. the swap must produce an SP-tree (we allow four configs)

A (non-structural) — B (distribution) — parallel node



Original



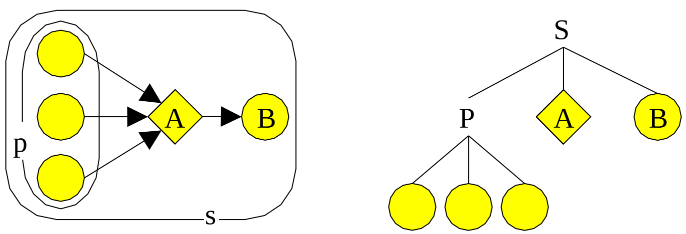
Swapped

Legal Swap

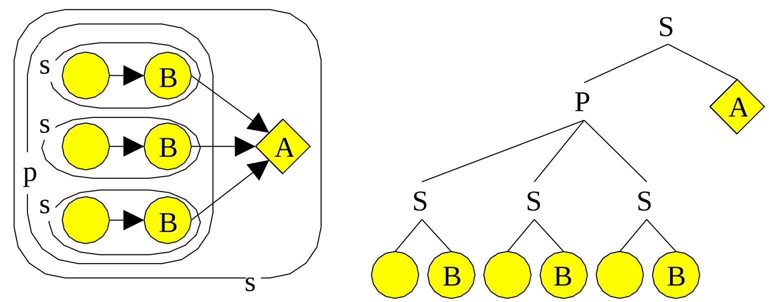
It is legal to swap adjacent ships A and B if

1. the swap must produce an equivalent sequence
   - that is, ship A and B are *commutative*
   - A or B is request-equivalent and A or B is data-equivalent
2. the swap must produce an SP-tree (we allow four configs)

Parallel node — A (merge) — B (non-structural)



Original



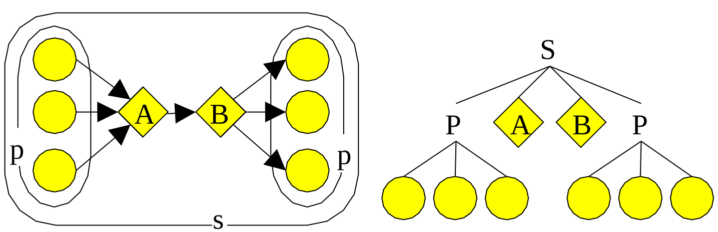
Swapped

Legal Swap

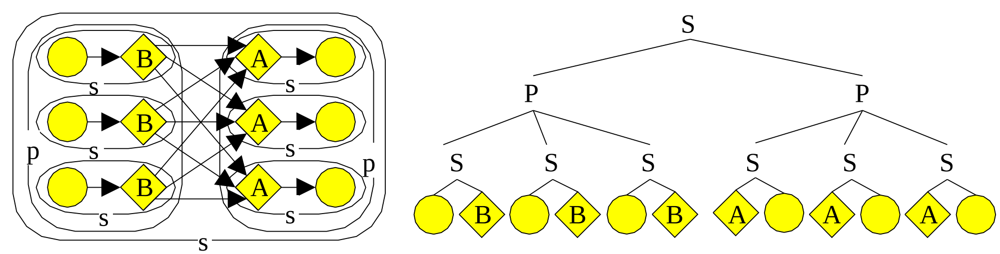
It is legal to swap adjacent ships A and B if

1. the swap must produce an equivalent sequence
   - that is, ship A and B are *commutative*
   - A or B is request-equivalent and A or B is data-equivalent
2. the swap must produce an SP-tree (we allow four configs)

   Parallel node — A (merge) — B (distribution) — parallel node



Original



Swapped

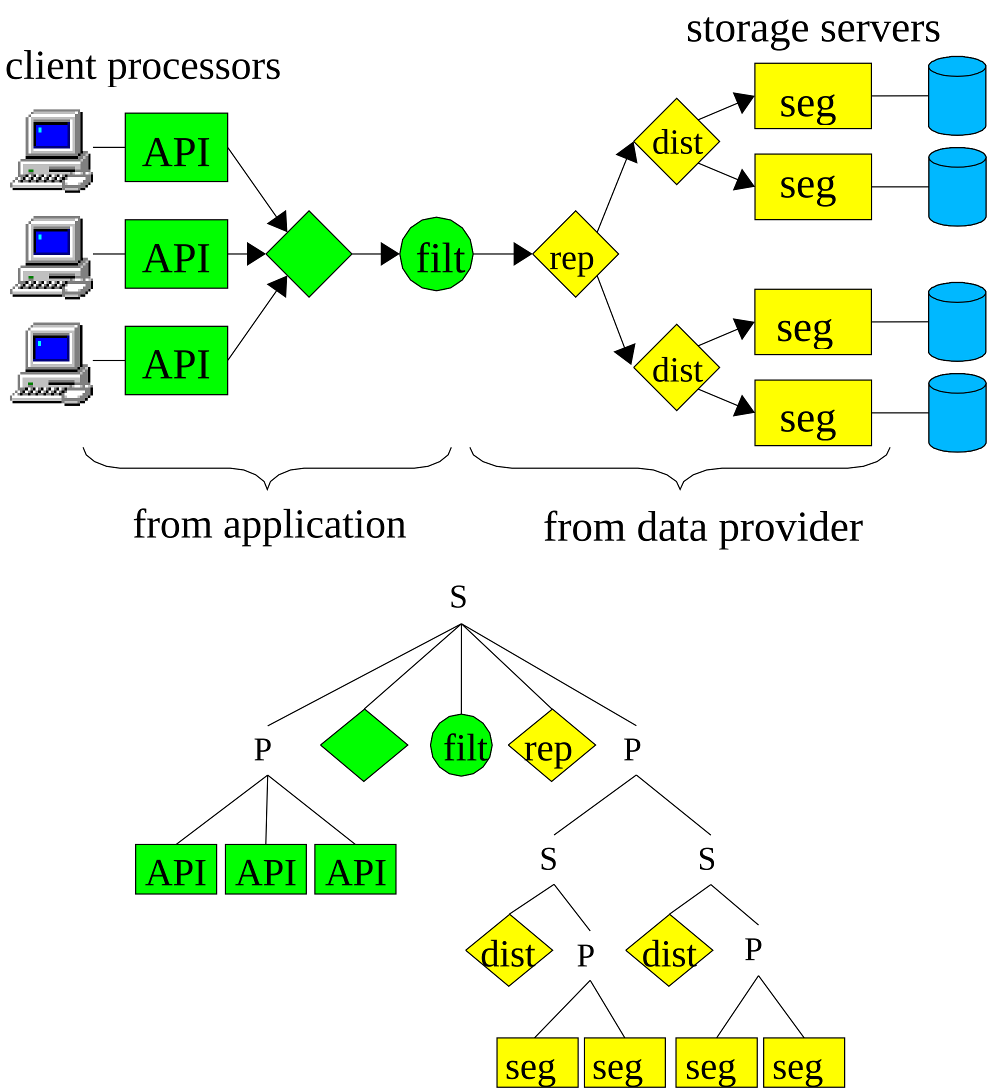
Beneficial Swap

A swap is deemed *beneficial* if it increases parallelism, moves a data-reducing ship closer to the data source, or moves a data-increasing ship closer to data destination.

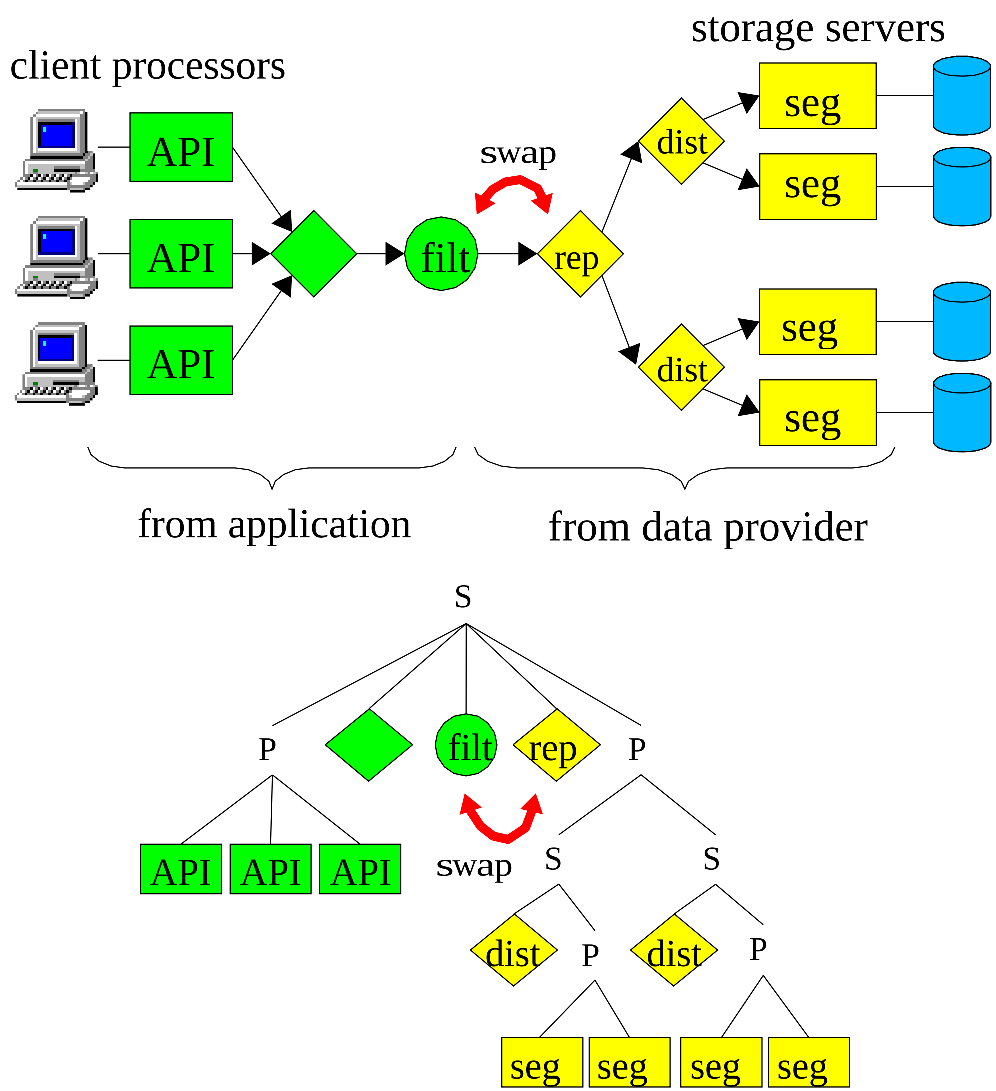
Algorithm to decide a beneficial swap of adjacent ships A and B

1. Assign a preferred direction to each ship (1 for right, -1 for left)
   - Merge ships prefer to go right (increase parallelism)
   - Distribution ships prefer to go left (increase parallelism)
   - Data-reducing ships prefer to swap toward the data destination
   - Data-increasing ships prefer to swap toward the data source
2. return $true$ if preferred direction of A is greater than preferred direction of B
3. else return $false$

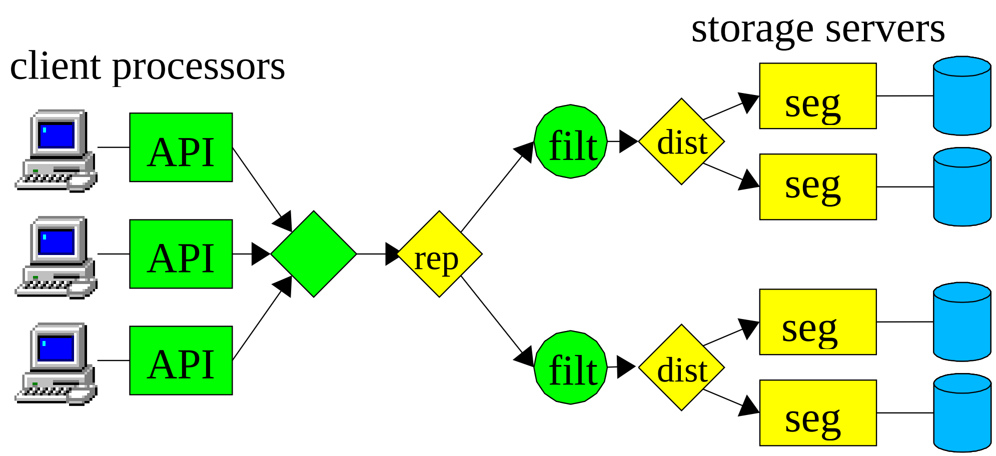
Restructuring the Example Graph

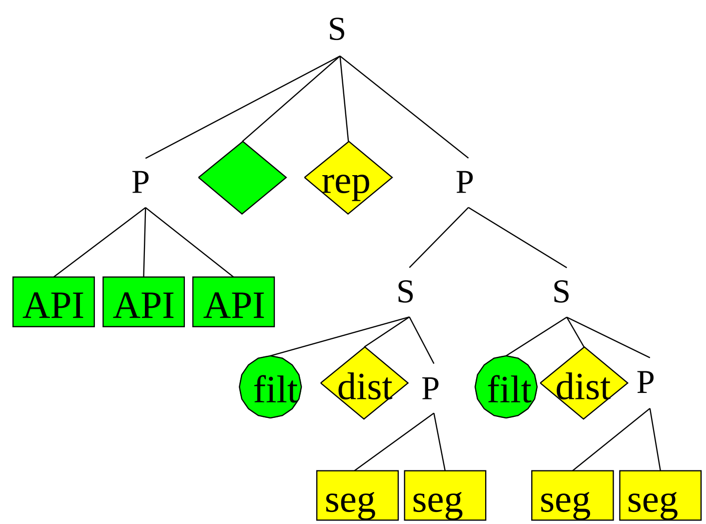


Restructuring the Example Graph

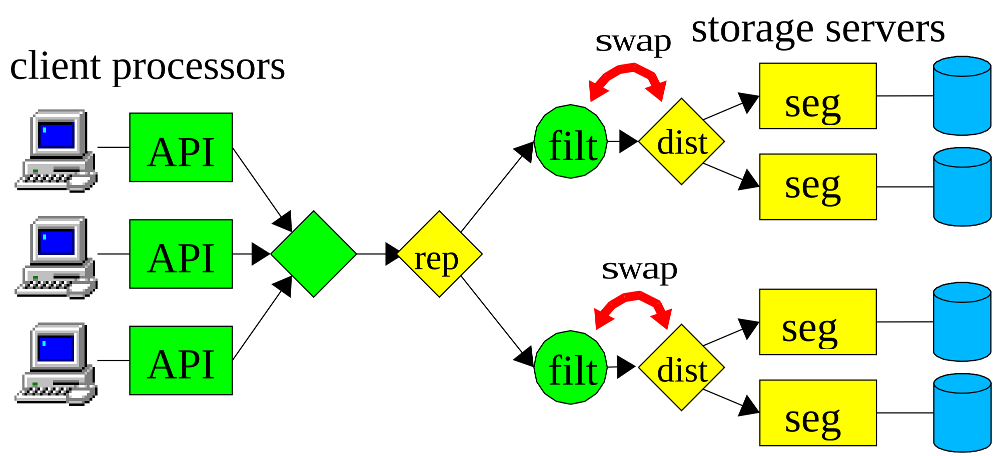


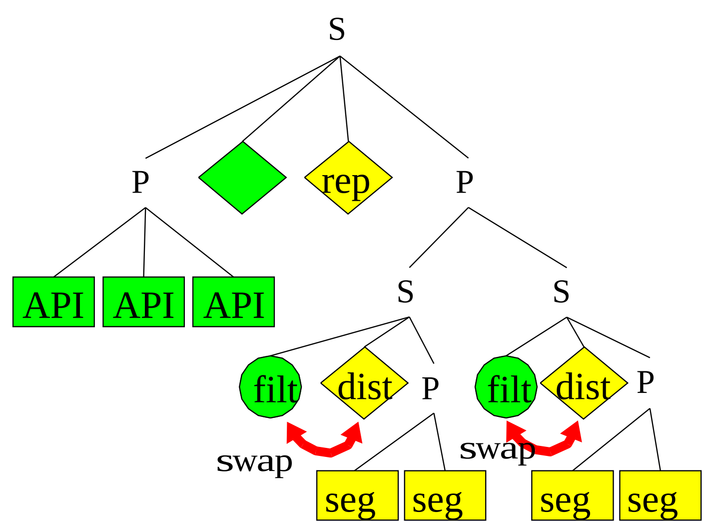
Restructuring the Example Graph



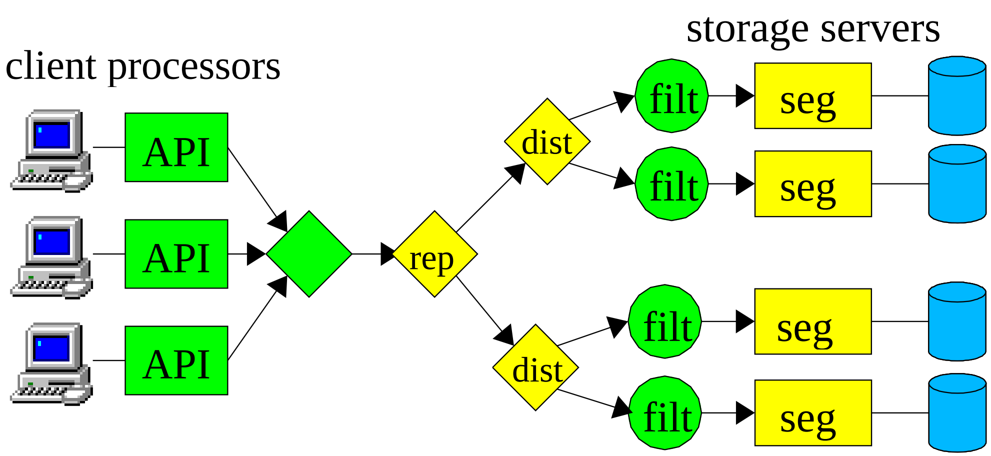


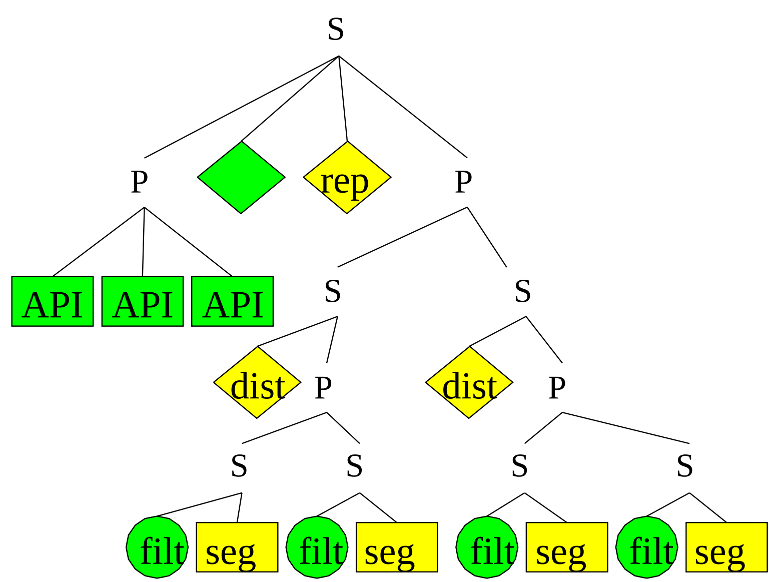
Restructuring the Example Graph



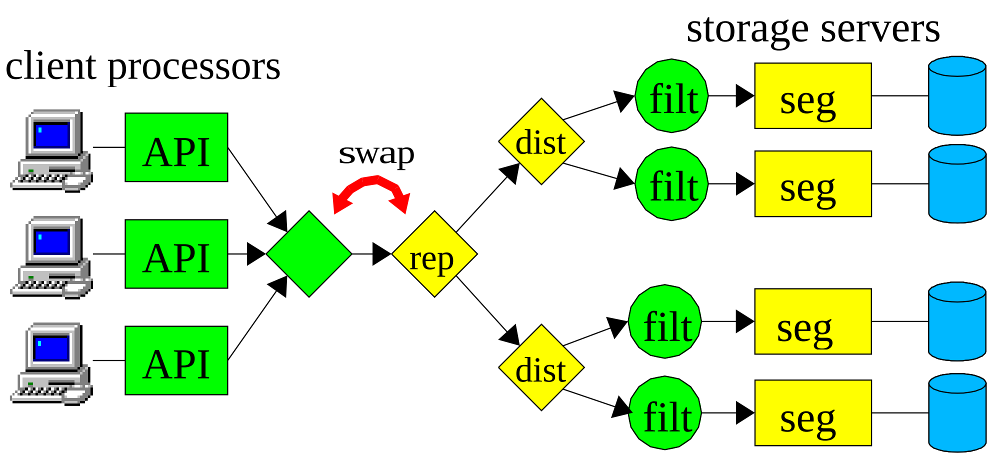


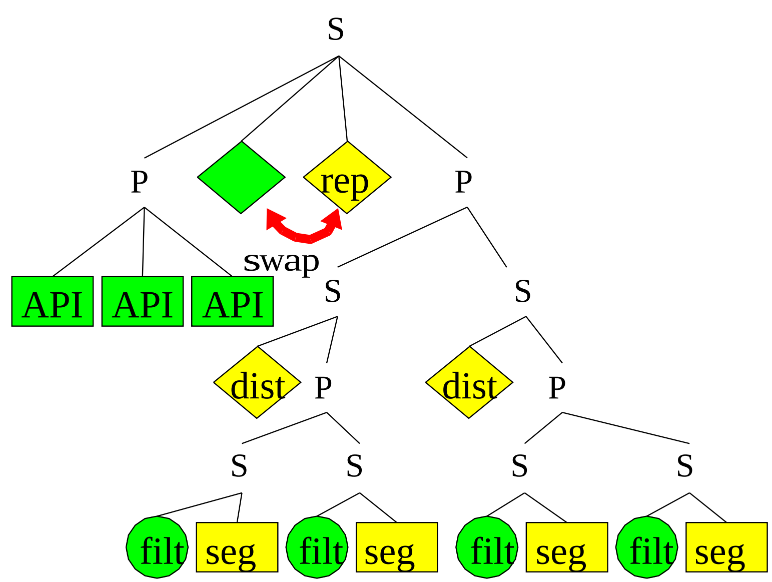
Restructuring the Example Graph



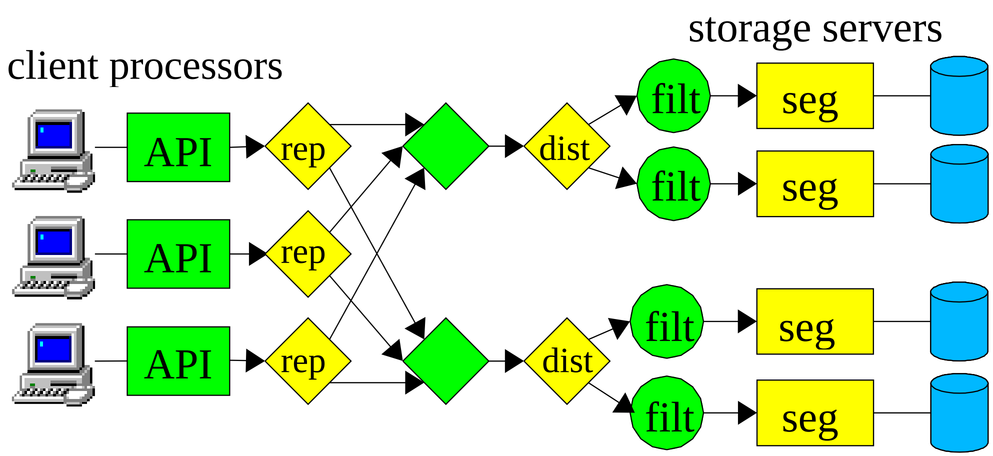


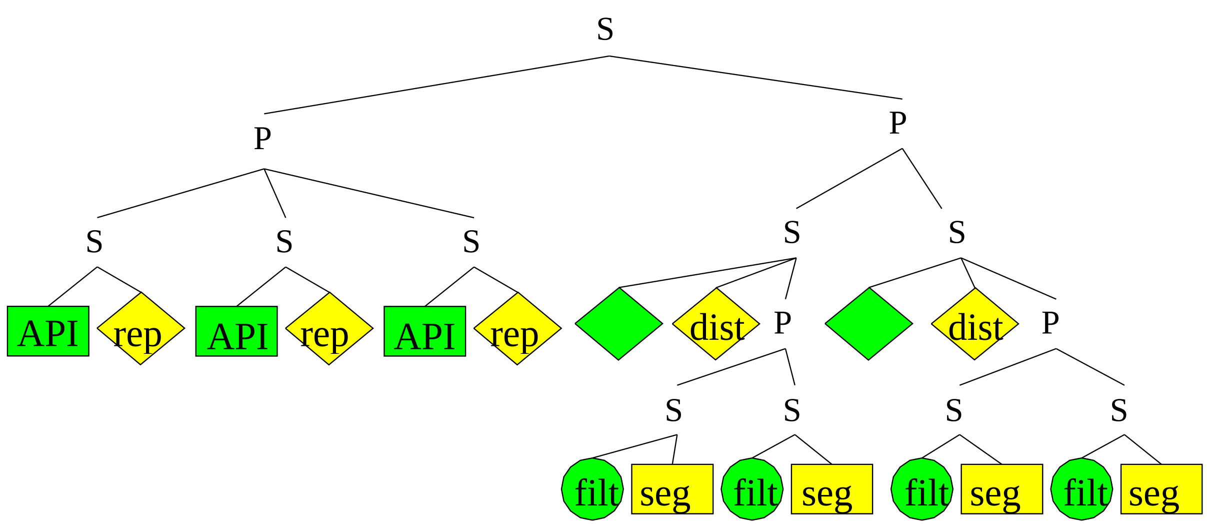
Restructuring the Example Graph





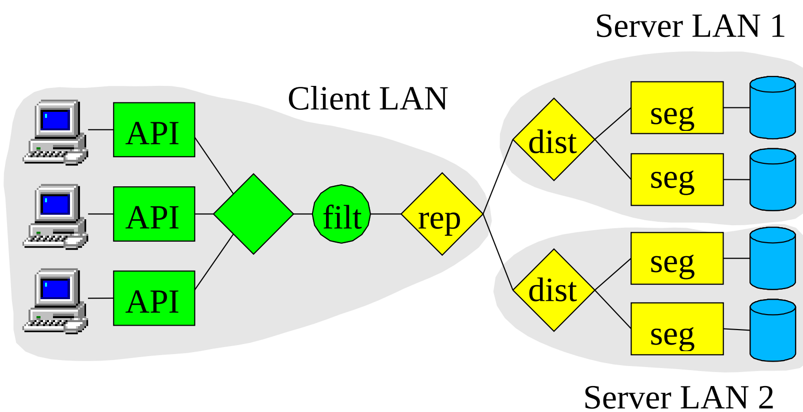
Restructuring the Example Graph



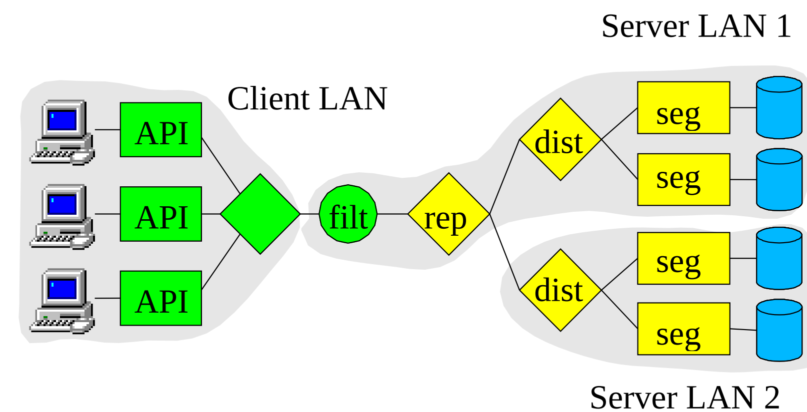


Experiments

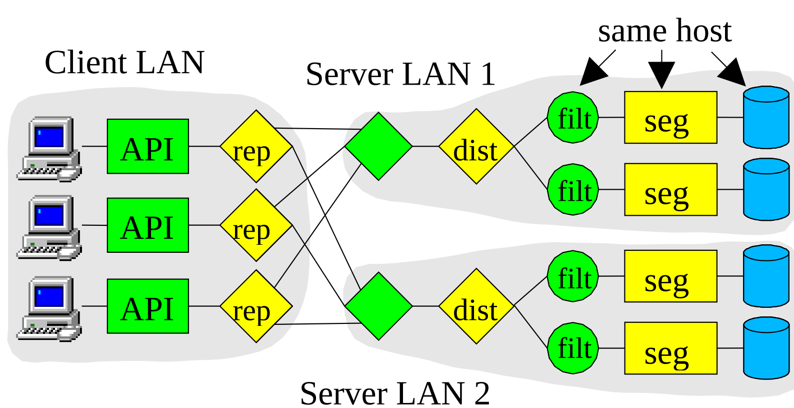
Examined four configurations of the example application with a filter that removed exactly 50% of the data.



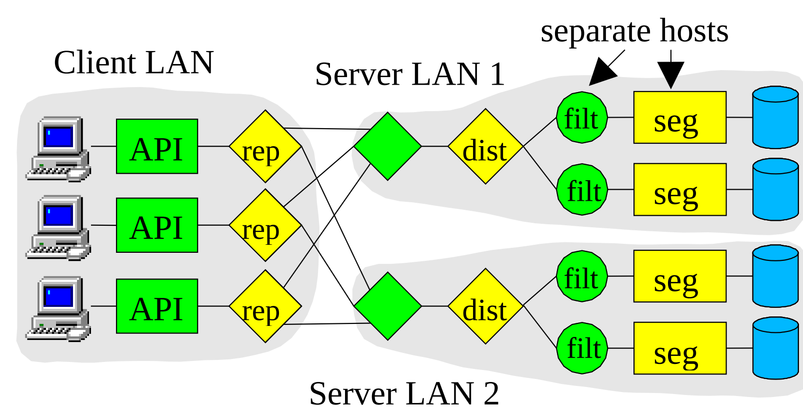
(a) orig1



(b) orig2



(c) restruct1

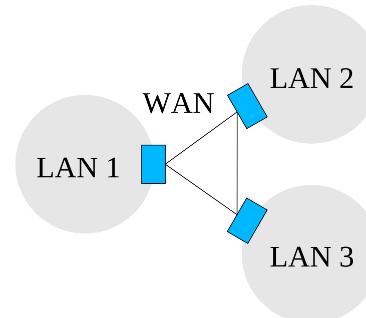


(d) restruct2

Experiment Setup

The area between the blobs represents the WAN

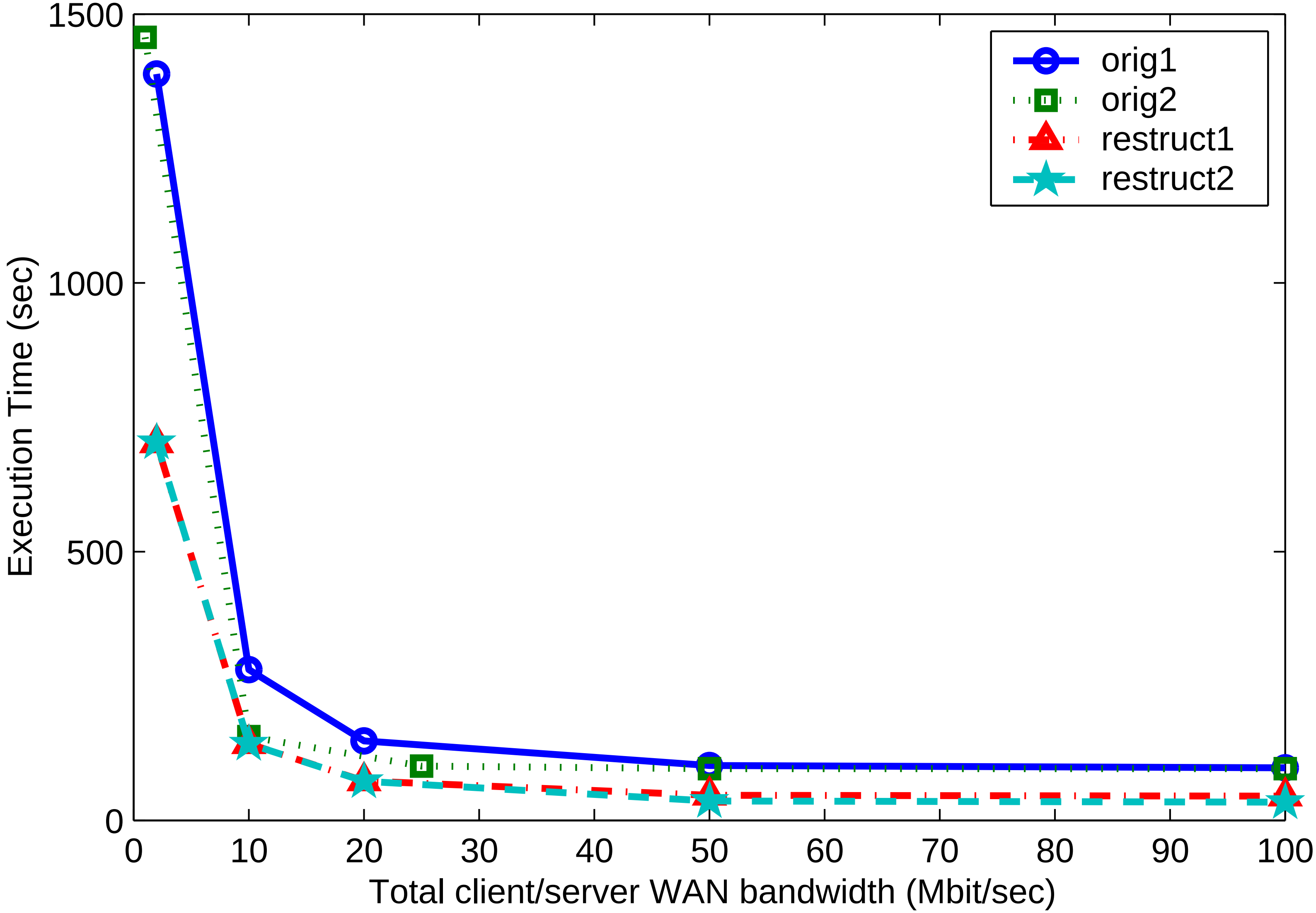
- each LAN connected to the WAN by single router
- each WAN link has limited capacity



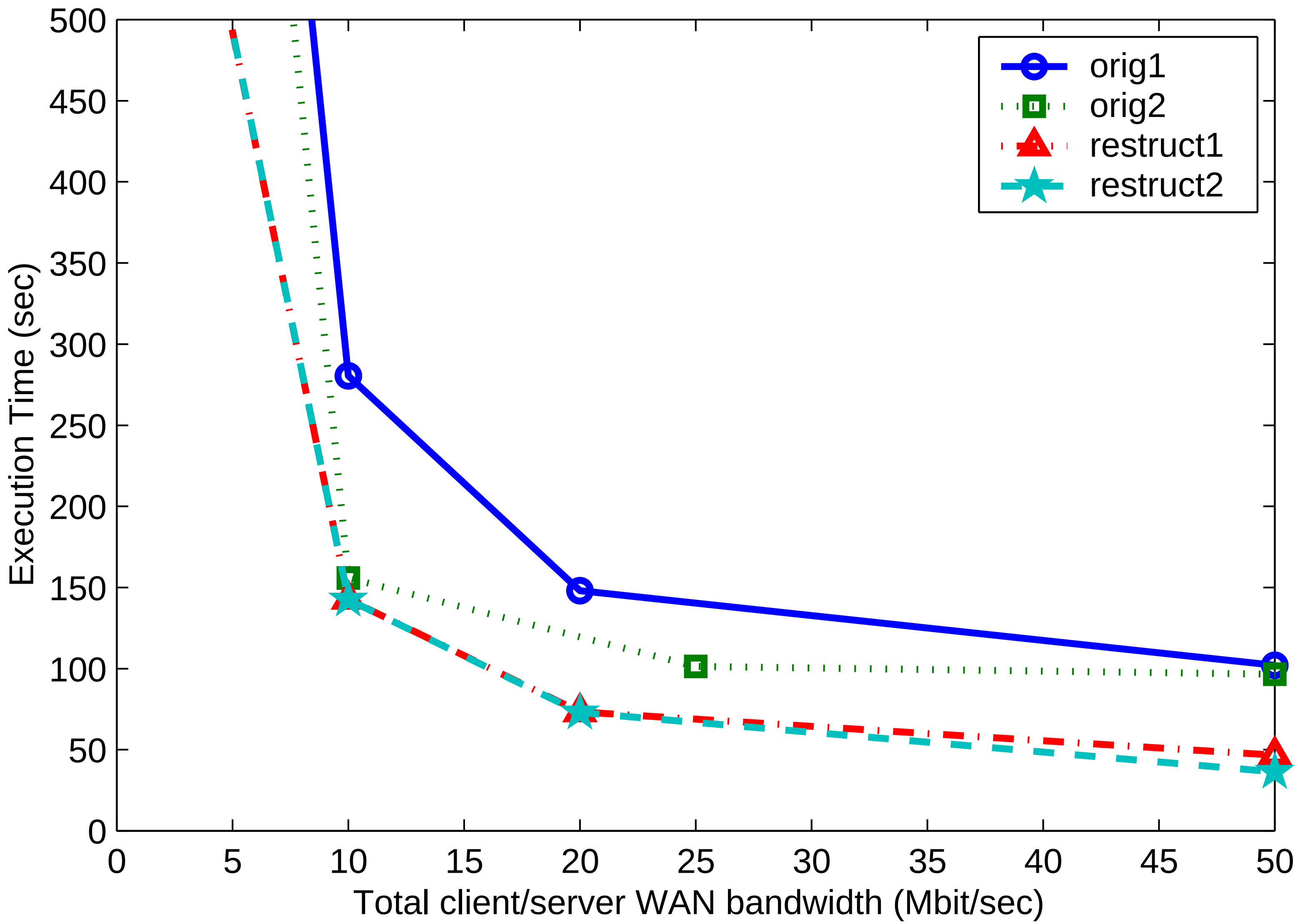
Ran experiments on the Emulab Network Testbed (Univ. Utah)

- Three LANs, each with
  - five 850 MHz Pentium III processors
  - 100 Mbps switched network (0.15 msec latency)
- WAN consisted of
  - three network links with 2.0 msec latency
  - bandwidth ranged from 1 to 50 Mbps (available between client/servers 2-100 Mbps)

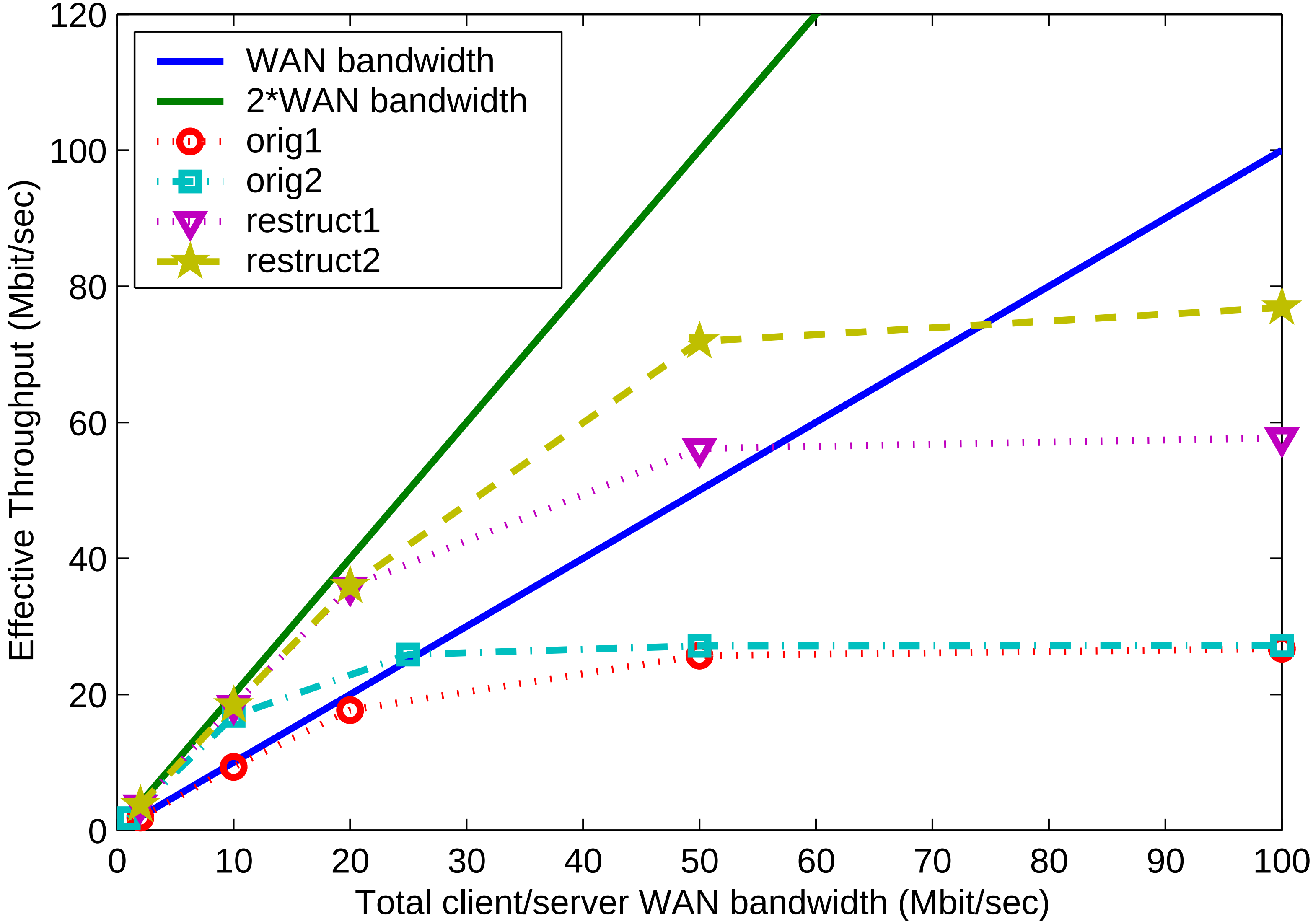
Results: Timings



Results: Timings



Results: Effective Throughput



Discussion

- Below 25 Mbps, all configurations limited by WAN
- Above 25 Mbps, computation associated with Java serialization and the filter code became the bottleneck
- When network bound, placement of filter is critical
  – restruct1 and restruct2 achieve nearly twice the effective throughput
- When compute bound, parallelization of filter is beneficial
  – restruct1 and restruct2 achieve 2-3 times the effective throughput as orig1 and orig2

Related Work

Parallel processing of I/O streams

- PS^2 [Messerli 1999]
  - data-flow model with automatic parallelization
- TPIE [Vengroff et al. 1996 and 2002]
  - data-flow model for I/O-optimal algorithms

Armada does not force whole application into data-flow model
Armada widens data flow for parallel clients and parallel servers

Operation ordering to improve data flow, e.g., in databases

- dQUOB [Plale et al. 2000]
  - optimize query tree to move high-filtering portions close to data
  - exploit well-defined properties associated with query processing

Armada provides a more general approach

Future Work

- Real applications
  - How to push some application function into Armada framework?
  - Can components (ships) be re-used between applications?
  - How much can performance benefit?
- Analytic model of “beneficial”
- Placement algorithm
  - Static: deploy graph at start
  - Dynamic: re-deploy when network conditions change

Conclusion

The Armada framework

- allows data provider to describe complex distributed data sets
- allows the application to describe processing required before computation
- provides a latency-tolerant data-flow approach useful for wide-area computing

Restructuring algorithm

- arranges graph to provide end-to-end parallel I/O
- enables effective placement of data-processing components to reducing network traffic over slow network links

Experiments show that restructuring is beneficial in both low and high-bandwidth environments.

The Armada Parallel I/O Framework for Computational Grids

Ron Oldfield and David Kotz

Department of Computer Science, Dartmouth College

http://www.cs.dartmouth.edu/∼dfk/armada/

Supported by Sandia National Laboratories under contract DOE-AV6184.