Relational Stream Processing
Streaming Optimizations

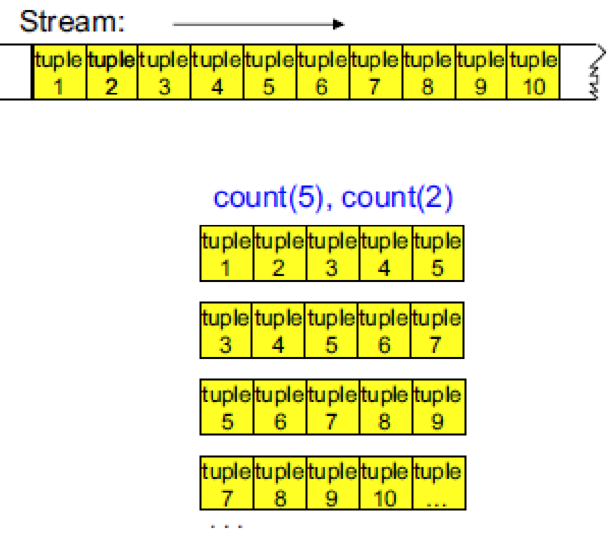
Copyright 2023 – Deepak Turaga

Objectives

- Quick Takeaways from Lecture 3
  - Windowing
  - Apache Beam
- Relational Streaming
  - Operator Concepts
- Streaming Optimizations

Copyright 2023 – Deepak Turaga

Windowing


Stream:

tuple 1 | tuple 2 | tuple 3 | tuple 4 | tuple 5 | tuple 6 | tuple 7 | tuple 8 | tuple 9 | tuple 10

count(5), count(2)

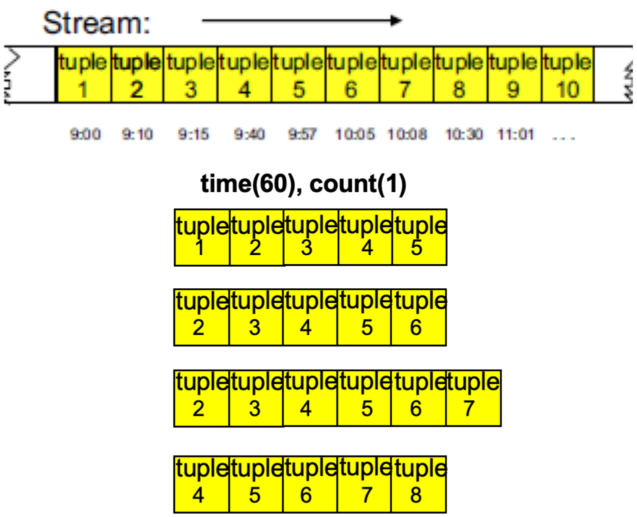
tuple 1 | tuple 2 | tuple 3 | tuple 4 | tuple 5

tuple 3 | tuple 4 | tuple 5 | tuple 6 | tuple 7

tuple 5 | tuple 6 | tuple 7 | tuple 8 | tuple 9

tuple 7 | tuple 8 | tuple 9 | tuple 10 | tuple …

. . .


Stream:

tuple 1 | tuple 2 | tuple 3 | tuple 4 | tuple 5 | tuple 6 | tuple 7 | tuple 8 | tuple 9 | tuple 10

9:00 9:10 9:15 9:40 9:57 10:05 10:08 10:30 11:01 . . .

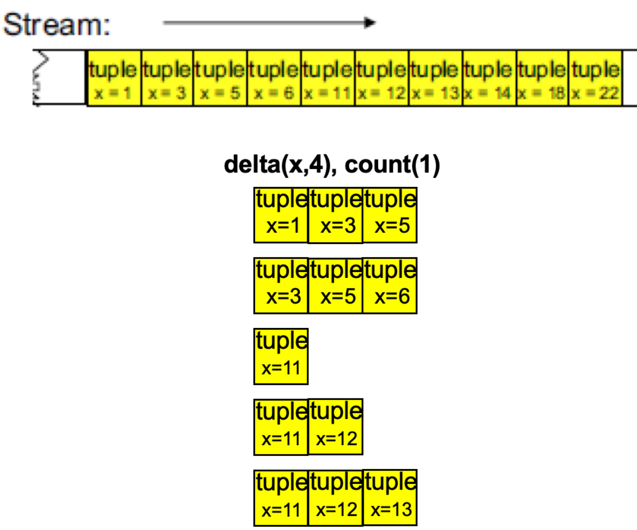
time(60), count(1)

tuple 1 | tuple 2 | tuple 3 | tuple 4 | tuple 5

tuple 2 | tuple 3 | tuple 4 | tuple 5 | tuple 6

tuple 2 | tuple 3 | tuple 4 | tuple 5 | tuple 6 | tuple 7

tuple 4 | tuple 5 | tuple 6 | tuple 7 | tuple 8


Stream:

tuple x = 1 | tuple x = 3 | tuple x = 5 | tuple x = 6 | tuple x = 11 | tuple x = 12 | tuple x = 13 | tuple x = 14 | tuple x = 18 | tuple x = 22

delta(x,4), count(1)

tuple x=1 | tuple x=3 | tuple x=5

tuple x=3 | tuple x=5 | tuple x=6

tuple x=11

tuple x=11 | tuple x=12

tuple x=11 | tuple x=12 | tuple x=13

Window Properties

- Three properties that define a window
  - Eviction policy
    - Defines how large a window can get
    - Determines which older tuples are removed from the window
  - Trigger policy
    - When an operation, such as aggregation, takes place as new tuples arrive into the window
  - Partitioning
    - Maintains separate windows for each group of tuples with the same grouping key value
- Window types
  - Tumbling windows:
    - Non-overlapping sets of consecutive tuples
    - Defined only with an eviction policy
  - Sliding windows:
    - Windows formed by adding new tuples to the end of the window and evicting old tuples from the beginning of the window
    - Defined with both an eviction and a trigger policy

Copyright 2023 – Deepak Turaga

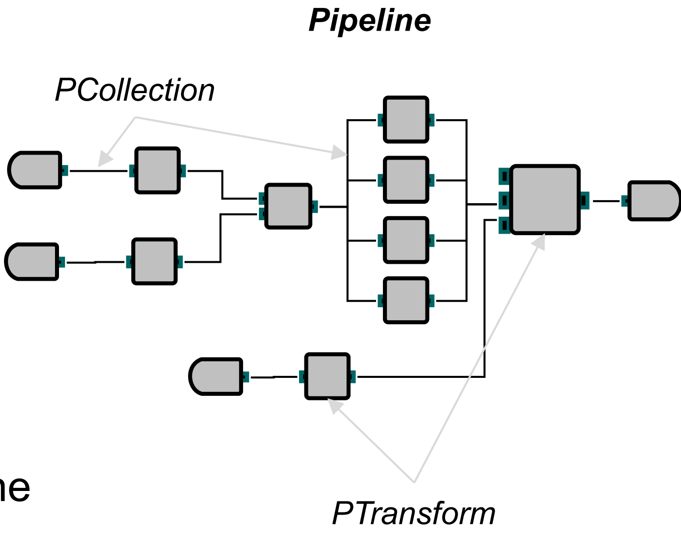
Apache Beam

- Open source unified model
  - Composition language for data pipelines
  - For batch and streaming jobs
- Multiple language SDK
  - Java, Python, Go, Scala
- Multiple execution frameworks (runners)
  - Apache Apex
  - Apache Flink
  - Apache Gearpump
  - Apache Samza
  - Apache Spark
  - Google Dataflow
  - IBM Streams

Copyright 2023 – Deepak Turaga

Apache Beam Programming

- Define driver program using one of Beam SDK languages
  - Java, Python, Go
- Core abstractions
  - *Pipeline* (application flowgraph)
    - Encapsulates entire data processing task
    - Needs to specify where and how to run
  - *PCollection* (Dataset or Stream)
    - Bounded datasets for batch processing
    - Unbounded for stream processing
  - *PTransform* (Operator)
    - Data processing operation or step in the pipeline
    - Multiple special I/O transforms for PCollections

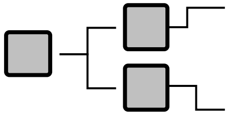


Copyright 2023 – Deepak Turaga

Apache Beam Programming

```
def mypipelinerun():
        p = beam.Pipeline(options=PipelineOptions())

lines = p | 'ReadMyFile' >> beam.io.ReadFromText('gs://some/inputData.txt’)

words = (lines | ‘SplitLines’ >> lines.FlatMap(str.split)



#Compute length for each word (element in PCollection)
class ComputeWordLengthFn(beam.DoFn):
  def process(self, element):
    return [len(element)]

word_lengths = words | beam.ParDo(ComputeWordLengthFn())
```

```
[Output PCollection 1] = [Input PCollection] | [Transform 1]
[Output PCollection 2] = [Input PCollection] | [Transform 2]
```



Copyright 2023 – Deepak Turaga

Core Beam Transformations

- `ParDo`
  - `Map, FlatMap`
- `GroupByKey`
- `CoGroupByKey`
- `Combine`
- `Flatten`
- `Partition`
- Each of these can be extended with user code
- Represent different ways of processing data

Copyright 2023 – Deepak Turaga

Beam Window Triggers

- Determine when results from the window are emitted
- Event time (data time) triggers
- Processing time (wall clock time) triggers
- Data-driven triggers
- Composite triggers

Copyright 2023 – Deepak Turaga

Window Triggers and Composition

- Event Triggers: operate on the event time (data time), e.g. `AfterWaterMark` trigger
  - Processing performed when watermark passes end of window
  - Recall watermarks and Structured Spark Streaming
- Processing Time Triggers: operate on wall clock time, e.g. `AfterProcessingTime` trigger
  - Processing performed interval after the first element in window is received
  - Supports count and time based interval specification
- Data-driven Triggers: element count, e.g. `AfterCount` trigger
  - Processing performed after receiving a certain number of tuples
- Different types of triggers can be composed

```
pcollection | WindowInto( FixedWindows(1 * 60),
trigger=AfterWatermark(late=AfterProcessingTime(10 * 60)),
allowed_lateness=10, accumulation_mode=AccumulationMode.DISCARDING)
```

References

- Apache Beam Documentation: https://beam.apache.org/documentation/programming-guide/

- Apache Beam Community: https://beam.apache.org/community/contact-us/

Copyright 2023 – Deepak Turaga

Operator Concepts

Operator Concepts

- Operators are stream manipulators
- Operators provide a declarative interface
- Some important operator properties
  - Operator state
  - Selectivity and arity
  - Parameters
  - Output assignments and output functions
  - Windowing
  - Punctuations

Copyright 2023 – Deepak Turaga

Operator State

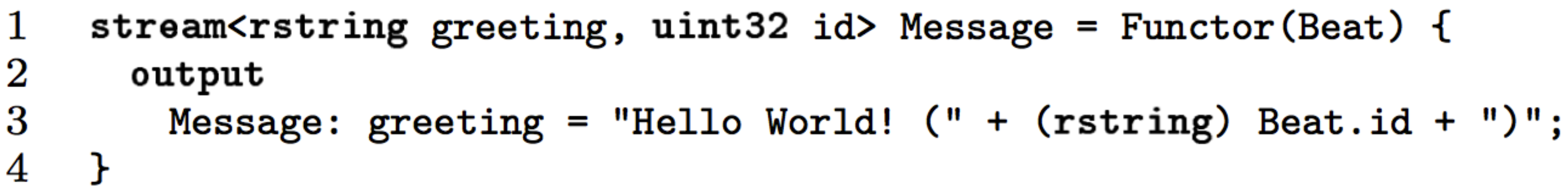
- Three classes of operators categories
  - Stateless
  - Stateful
  - Partitioned stateful

Copyright 2023 – Deepak Turaga

Stateless Operators

- Do not maintain internal state across tuples
- Processes on a tuple-by-tuple basis
- E.g. Projection with `Functor`, `Map`, `ParDo`
- Advantages:
  - Can be parallelized easily
  - No need for synchronization in a multi-threaded context
  - Can be restarted upon failures

Example Stateless Operators



```
stream<rstring greeting, uint32 id> Message = Functor(Beat) {
  output
    Message: greeting = "Hello World! (" + (rstring) Beat.id + ")";
}
```

```
lines = sc.parallelize([1, 4, 3])
//read from file
lines2 = lines.map(lambda x: x*x)
res = lines2.collect()
for num in res:
    print num
```

```
class ComputeWordLengthFn(beam.DoFn):
  def process(self, element):
    return [len(element)]

words = ...
word_lengths = words |
beam.ParDo(ComputeWordLengthFn())
```

Stateful Operators

- Create and maintain state as tuples processed
- Such state, along with internal algorithm, affects the results
- e.g. `DeDuplicate` or `Join` operators or `reduceByKeyAndWindow`
  - *tuple is considered a duplicate if it shares the same key with a previously seen tuple within a pre-defined period of time*
- Runtime support is challenging
  - Require synchronization in a multi-threaded context
  - No direct data parallelization
  - Require some persistence mechanism for fault-tolerance

Copyright 2023 – Deepak Turaga

Partitioned Operators

- Keep state across tuples
- State can be divided into *partitions*
- Each partition depends on independent segments of the data stream
- Another way of looking at it
  - Input stream is divided into non-overlapping *sub-streams*
  - Sub-streams are processed independently
  - Each sub-stream has its own state
- Example: Computing Volume Weighted Average Price (VWAP) on all stock tickers
  - each transaction contains an attribute *ticker* (e.g., “IBM”, “AAPL”, “GOOG”), a *volume*, and a *price*
  - an application wants to compute a VWAP for each ticker over the last 10 transactions on that particular ticker
- `partitionBy` keyword

Copyright 2023 – Deepak Turaga

Example Partitioned Operators

```
//Create a streaming reduce with a 1 second batch, 4 second window
ipDstream = accessLogs.map(lambda x:(x.getIpAddress(),1))
ipCountDstream = ipDstream.reduceByKeyAndWindow(
                    lambda (x,y): x+y,
                    lambda (x,y): x-y,
                    Seconds(4),
                    Seconds(1))
```

Not explicitly partitioned…

Recall partitioning strategy

```
Data = sc.parallelize([(“a”,1),(”b”,2),(“c”,3)])
Res2 = Data.partitionBy(new HashPartitioner(100)).persist()
```

Default mode for Apache Beam is partitioned by key

Copyright 2023 – Deepak Turaga

Operator Selectivity and Arity

- Selectivity: captures the relationship between the number of tuples produced and consumed by an operator
  - Fixed selectivity
  - Variable selectivity
- Fixed: Consumes a fixed number of tuples to generate a fixed number of output tuples
  - E.g. The Map operator with one input tuple to generate one output tuple
- Variable: exhibits a non-fixed relationship between the number of tuples consumed and produced
  - E.g. The Join operator may generate 0, 1, or more output tuples, depending on the match condition, window contents

Copyright 2023 – Deepak Turaga

Operator Arity

- Arity refers to the number of ports an operator has
- An important class is single input / single output operators
- Selectivity of single input / single output operators
  - One-to-one (1:1) – applies a simple map operation
  - One-to-at-most-one (1:[0,1]) – applies a simple filter operation
  - One-to-many (1:*N*) – performs expansion
  - Many-to-one (*M*:1) – performs data reduction
  - Many-to-many (*M*:*N*) – the remaining case

Copyright 2023 – Deepak Turaga

Punctuation

- Special markers inside stream
  - Window punctuations
  - Final punctuations
- Window: marks boundaries so that groups of tuples can be treated together
  - E.g.: Aggregate results from a Join operator
  - Operators that generate output tuples in batches typically insert a window punctuation after each batch
  - Punctuation based windows can be used to operate on these batches

Copyright 2023 – Deepak Turaga

Windows

- Sorting, aggregating, or joining data in a relational table
  - All data in the table can be processed
- For streams, data flows continuously
  - No beginning, no end
  - Requires a different paradigm for sorting, aggregating and joining
  - Can only work with a subset of consecutive tuples
- This finite set of tuples is called a window

Copyright 2023 – Deepak Turaga

Streaming Optimizations

- Stream processing
  - Data flow graphs
- Appears in many areas of computer science
  - Databases
  - Systems (OS/Distributed)
  - Programming Languages
- These are different areas
  - Same underlying optimizations
  - Different terminology (operator vs box, hoisting vs. push-down)
  - Different assumptions (cyclic vs acyclic graphs, shared memory vs distributed)
- Of interest
  - Safety discussions (depends on assumptions, app)
  - Profitability discussions (depends on assumptions, app, workload, resources)

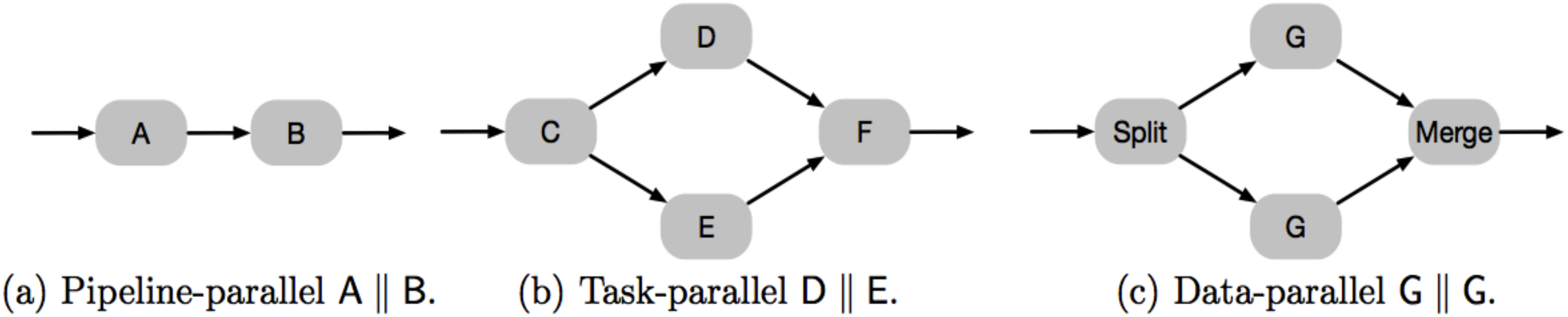
Copyright 2023 – Deepak Turaga

Kinds of Optimization

- Three dimensions
  - Does it change the graph?
    - Optimizations that involve graph transformations change the graph. E.g.: Fission
    - Some optimizations do not require transformation. E.g. Load balancing.
  - Does it impact the application semantics?
    - Ideally, an optimization should not impact the application semantics.
    - There are a few exceptions: load shedding, algorithm selection
  - Is it dynamic?
    - If the optimization can be applied at runtime, potentially based on current resource and workload availablity, then it is dynamic
    - Static optimizations are often done at compile-time

Copyright 2023 – Deepak Turaga

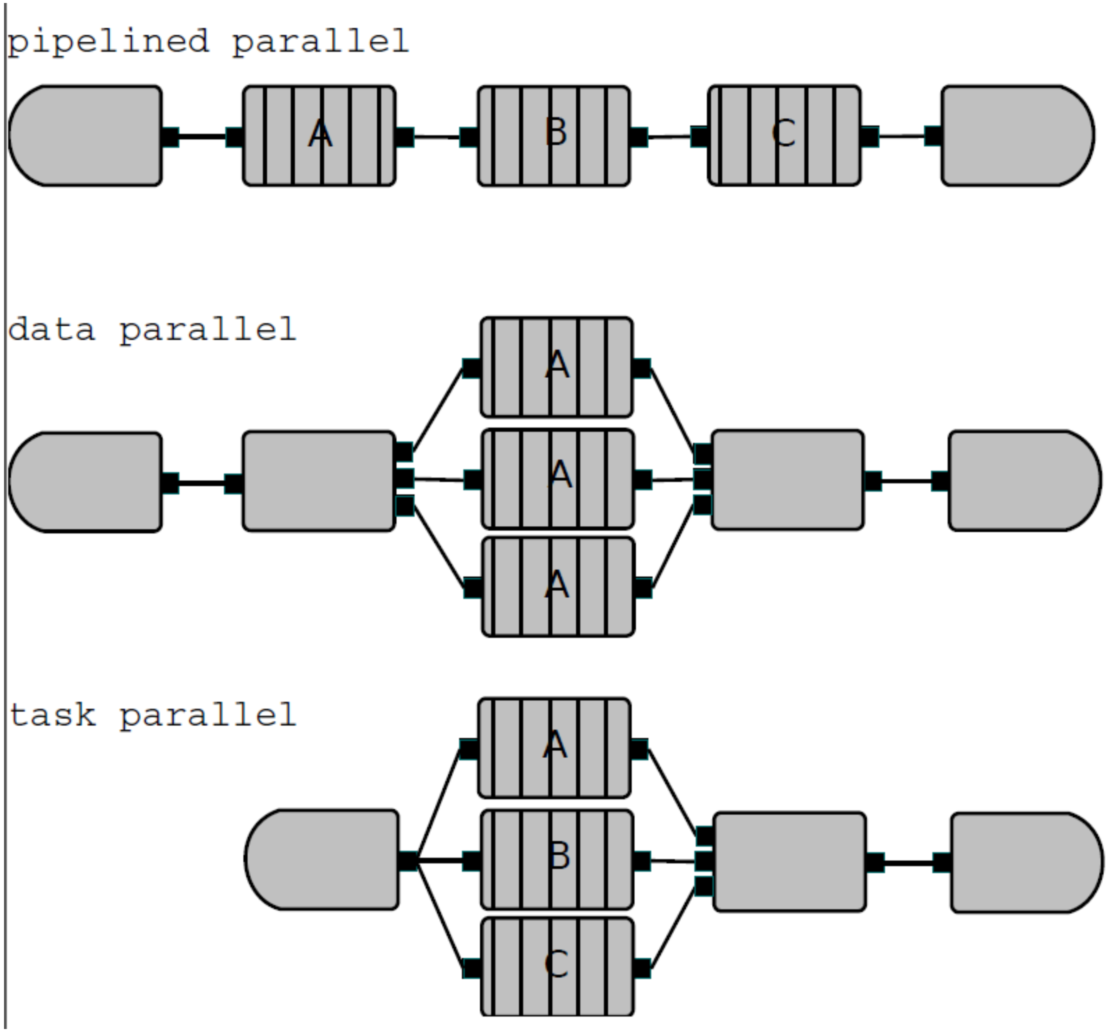
Types of Parallelism



(a) Pipeline-parallel A || B. (b) Task-parallel D || E. (c) Data-parallel G || G.

- (a-pipeline) Successive data items processed by successive tasks at the same time
- (b-task) Same data items processed by multiple tasks at the same time
- (c-data) Successive data items processed by different instances of the same task at the same time

Types of Parallelism



tuple 1	A	B	C					
tuple 2		A	B	C				
tuple 3			A	B	C			
tuple 4				A	B	C		
tuple 5					A	B	C	
tuple 6						A	B	C

time

tuple 1	A	
tuple 2	A	
tuple 3	A	
tuple 4		A
tuple 5		A
tuple 6		A

time

tuple 1_1	A	
tuple 1_2	B	
tuple 1_3	C	
tuple 2_1		A
tuple 2_2		B
tuple 2_3		C

Copyright 2023 – Deepak Turaga

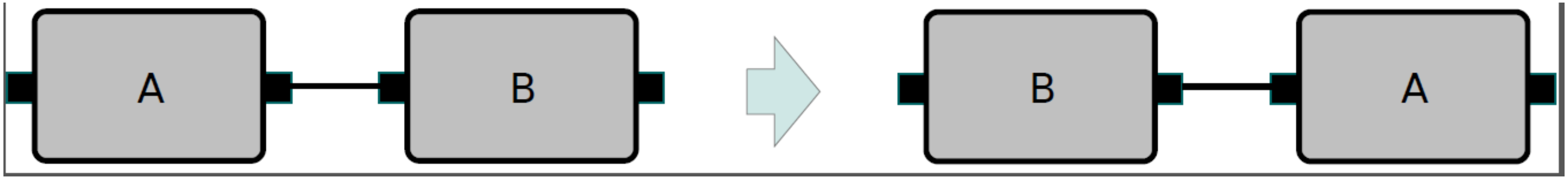
Optimization Catalog

Optimization	Graph	Semantics	Dynamics
Operator re-ordering	changed	unchanged	(depends)
Redundancy elimination	changed	unchanged	(depends)
Operator separation	changed	unchanged	static
Fusion	changed	unchanged	(depends)
Fission	changed	(depends)	(depends)
Placement	unchanged	unchanged	(depends)
Load balancing	unchanged	unchagned	(depends)
State sharing	unchanged	unchanged	static
Batching	unchanged	unchanged	(depends)
Algorithm selection	unchanged	(depends)	(depends)
Load shedding	unchanged	changed	dynamic

Copyright 2023 – Deepak Turaga

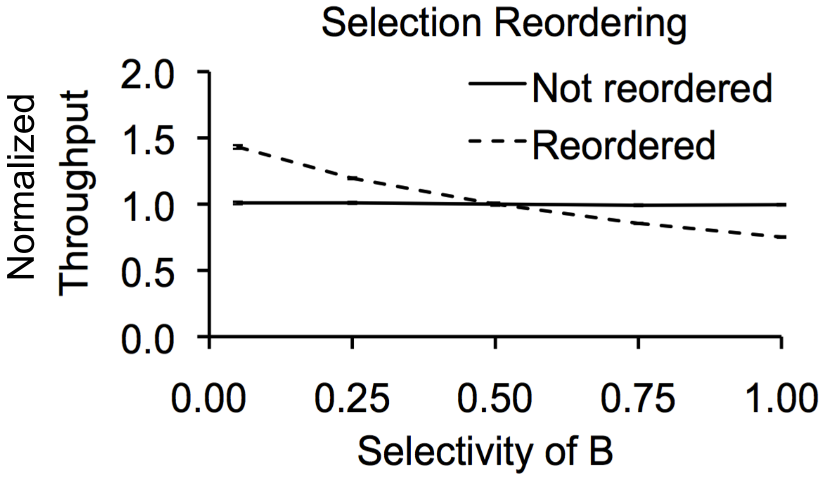
Operator Reordering

- Also known as
  - Hoisting, sinking, rotation, push-down
- *Move more selective operators upstream to filter data early.*



Profitability of Operator Reordering

- $c(A)$: compute cost of A (per unit tuple)
- $s(A)$: selectivity of A
- Compute cost (per input tuple) without reordering
  - $c(A) + s(A)c(B)$
- Compute cost with reordering
  - $c(B) + s(B)c(A)$
- Assume, $c(A)=c(B)=1$ and $s(A)=0.5$ Given a fixed compute budget



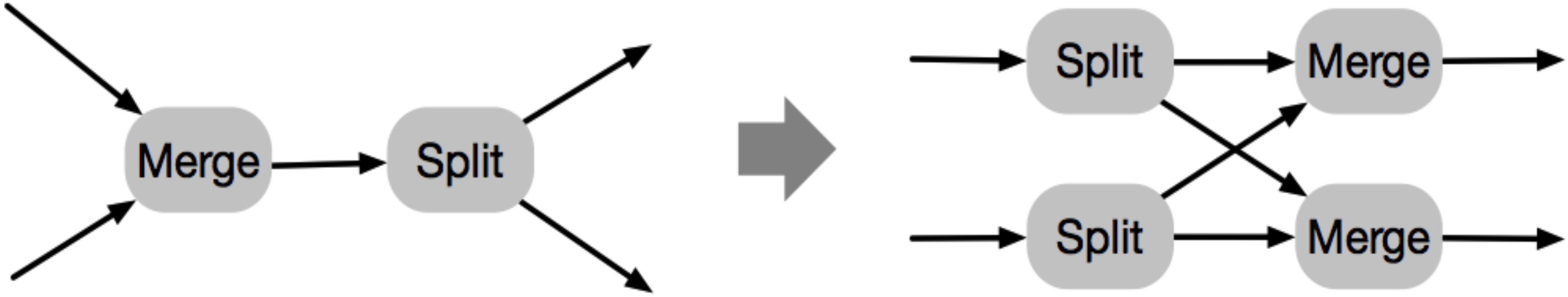
Copyright 2023 – Deepak Turaga

Safety of Operator Reordering

- Ensure attribute availability
  - The second operator should only rely on the attributes of the data that are already available before the first operator
  - i.e., the set of attributes that B reads from a data item must be disjoint from the set of attributes that A writes to a data item.
- Ensure commutativity
  - The order in which the operations are applied does not matter
  - E.g. filtering and projection
  - In the presence of attribute availability, a sufficient condition for commutativity is if A and B are stateless

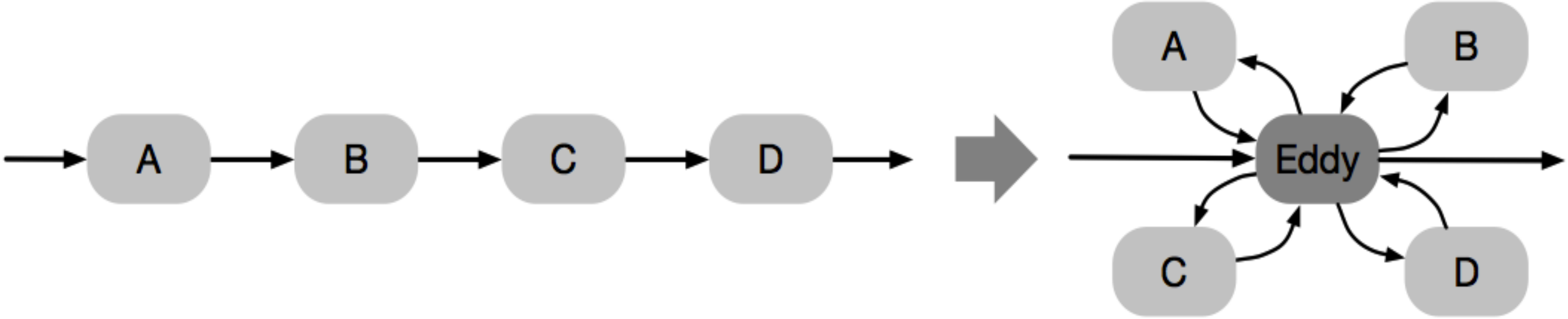
Copyright 2023 – Deepak Turaga

Variant: Split-Merge Rotation



- Move split before the merge
- This is often called a *shuffle*
- The reordered version is more scalable
  - You can increase the number of merged/splitted streams
- Used in M/R systems as well

Discussion: Dynamic Reordering



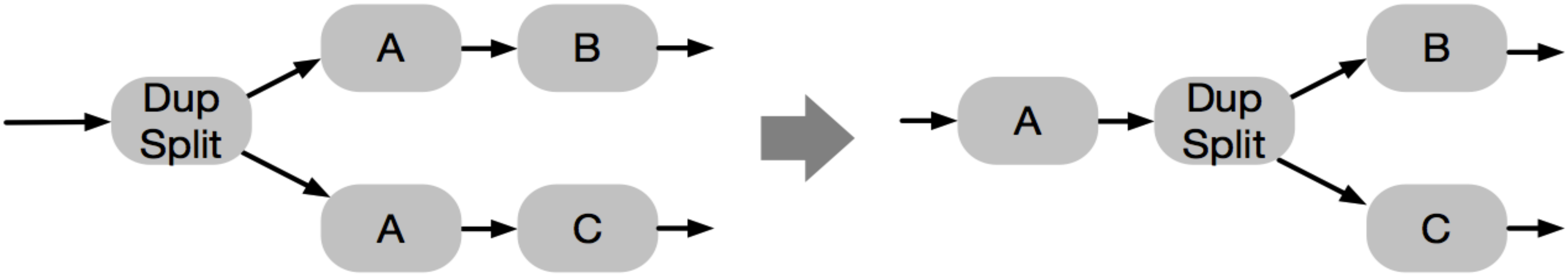
- The Eddy operator
  - Dynamically route the tuples
  - Has to keep track of progress
  - Can determine the order based on current selectivities
    - Selectivity may not be known at compile-time or can change at runtime
  - A weakness: Assumes selectivities are independent of the order

Copyright 2023 – Deepak Turaga

Redundancy Elimination

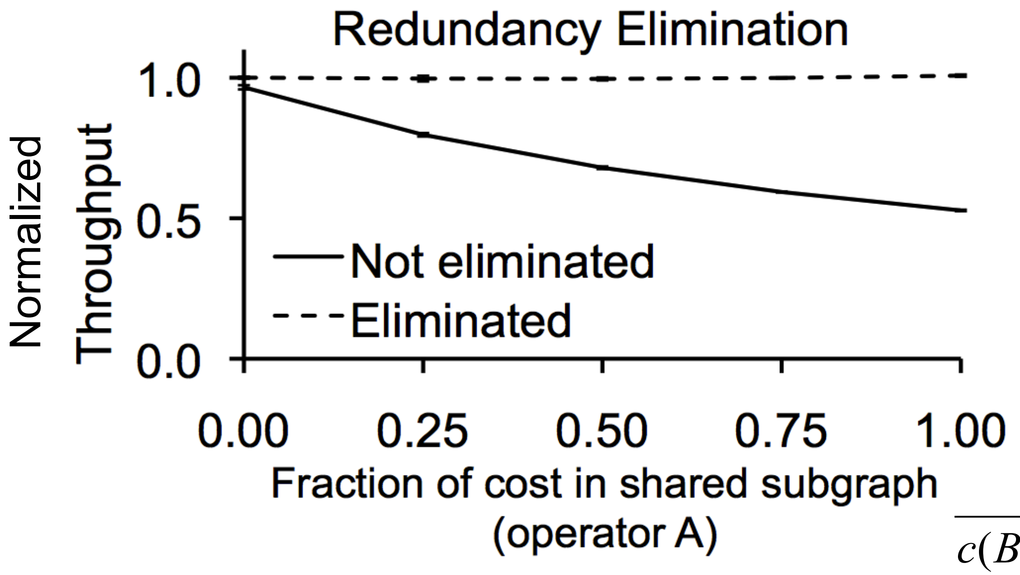
- Also known as
  - Subgraph sharing, multi-query optimization

- *Eliminate redundant computations.*



Profitability of Redundancy Elimination

- In general, removing redundancy does not hurt performance
- It improves performance when
  - The shared subgraph does significant work
- Before elimination: $c(B)+c(C)+c(A)+c(A)$
- After elimination: $c(B)+c(C)+c(A)$



$$\frac{c(A)}{c(B)+c(C)+c(A)}$$

Safety of Redundancy Elimination

- Ensure same algorithm
  - The redundant operators must perform the same operation
  - E.g. Same code, or algebraic equality
- Ensure combinable state
  - Assume the operator counts the number of tuples and we have a content based split that uses a partition by attribute
    - The result before and after redundancy elimination will be different, even though the operators do the same thing
  - If the shared operator is stateless, then no issue.
  - If stateful, then the state maintained over the original stream must be independent for individual sub-streams.

Copyright 2023 – Deepak Turaga

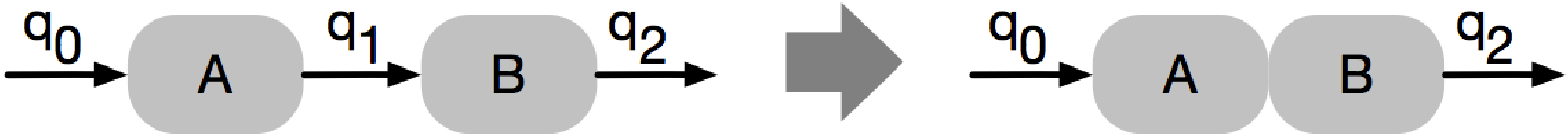
Dynamic Redundancy Elimination

- Static redundancy elimination
  - No-ops, dead-subgraphs
  - Shared subgraphs
    - if applications are all available
- Dynamic redundancy elimination
  - Needed when multi-tenancy is present
  - Applications can be submitted at different times
  - Multi-query optimization at run-time (submit/cancel)

Copyright 2023 – Deepak Turaga

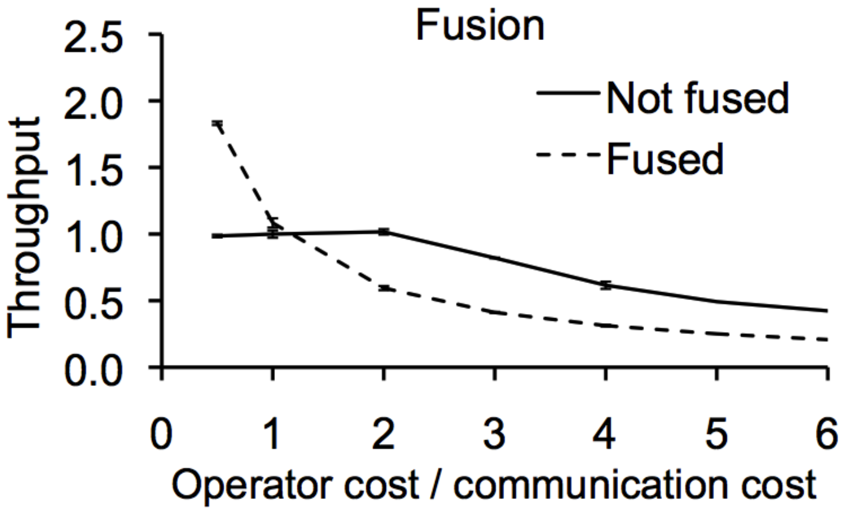
Fusion

- Also known as
  - – Superbox scheduling
- *Avoid the overhead of data serialization and transport.*



Profitability of Fusion

- Fusion trades communication cost against pipeline parallelism
- If the cost of one of the operators is small (relative to communication cost)
  - Better if we fuse them (avoid the overheads)



$$\frac{c(A)+c(B)}{c(Comm)}$$

Not fused setting:
1. Operator cost < communication cost, then bounded by communication cost.
2. After that goes down with operator cost

Fused setting:
1. Goes down with operator cost (but fused operator has twice the cost of unfused)

Copyright 2023 – Deepak Turaga

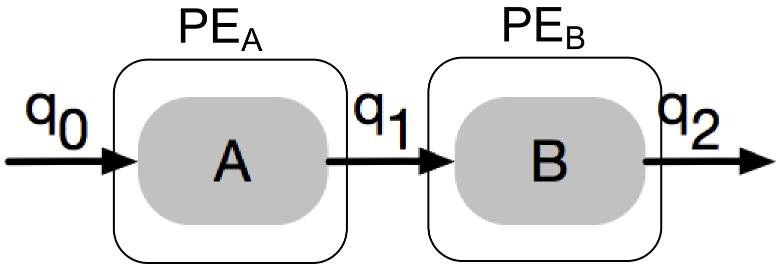
Illustration: Fusion in the SPL Language

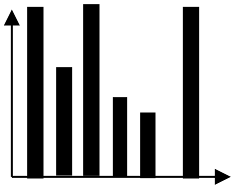
- Compiler can fuse in different modes
- Default Fusion
  - `sc -p FDEF -M <Namespace>::<MainCompositeName>`
- Optimized Fusion
  - `sc -p FOPT -M <Namespace>::<MainCompositeName>`
- Need to run in profile mode
  - `sc -P <Nsamples> -S <samplingrate> -p FOPT -M <Namespace>::<MainCompositeName>`
  - Store `Nsamples` from which profiles are computed using reservoir sampling
  - Sample every `samplingrate` (≤1) tuples – affects performance
  - Submit and cancel job to compute profiles

Copyright 2023 – Deepak Turaga

Profile Mode

- Run each operator inside its own PE
- Collect and compute statistics
  - Compute resources used
    - Memory, CPU, disk
  - Communication costs
- Determine optimum fusion assuming certain available compute from each core
  - Multi-objective optimization
  - Re-optimize
- Notes:
  - Measuring statistics has performance overheads
  - Need dedicated cores when profiling





Distribution of CPU usage

Estimated from `Nsamples` samples, each of which is measured every `1/samplingrate` tuples

Copyright 2023 – Deepak Turaga

Operator → PE: config placement Clause

- Partitioning allows you to specify operators that are to be fused together or kept apart
  - Uses `config placement` clause
- `partitionColocation`
  - Operator instances with the same partitionColocationvalue must run in the same partition
- `partitionIsolation`
  - Operator instance has a partition by itself
  - No other operator instances can run in that partition
- `partitionExlocation`
  - Operator instances with the same partitionExlocation value must run in different partitions

Operator → PE: Partitions

```
Stream <> S1 = MyOp1 {…
config placement: partitionColocation(“fuseMe");
}
Stream <> S2 = MyOp2 {…
config placement: partitionColocation(“fuseMe"),
  partitionExlocation(“dontFuse");
}
Stream <> S3 = MyOp3 {…
config placement: partitionIsolation;
}
Stream <> S4 = MyOp4 {…
config placement: partitionExlocation(“dontFuse")
}
```

Copyright 2023 – Deepak Turaga

Safety of Fusion

- Ensure resource kinds
  - All resources needed by A and B are available on a single process/host
- Ensure resource amounts
  - Total amount of resources required are available on a single process/host
- Avoid infinite recursion
  - Cycles in the stream graph can result in stack overflow

Copyright 2023 – Deepak Turaga

Dynamic Fusion

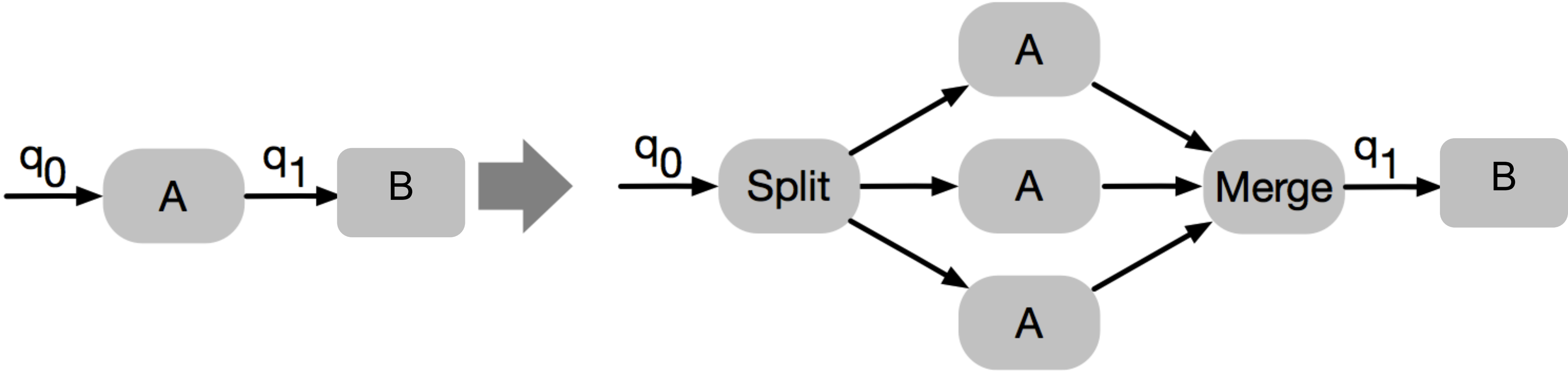
- Typically done at compile-time
  - Fusion across hosts at runtime requires heavy machinery
- Dynamic fusion using multiple threads in the same process
  - See “Auto-Pipelining for Data Stream Processing”
  - Dynamic profiling
  - Automatic thread/buffer insertion

Copyright 2023 – Deepak Turaga

Fission

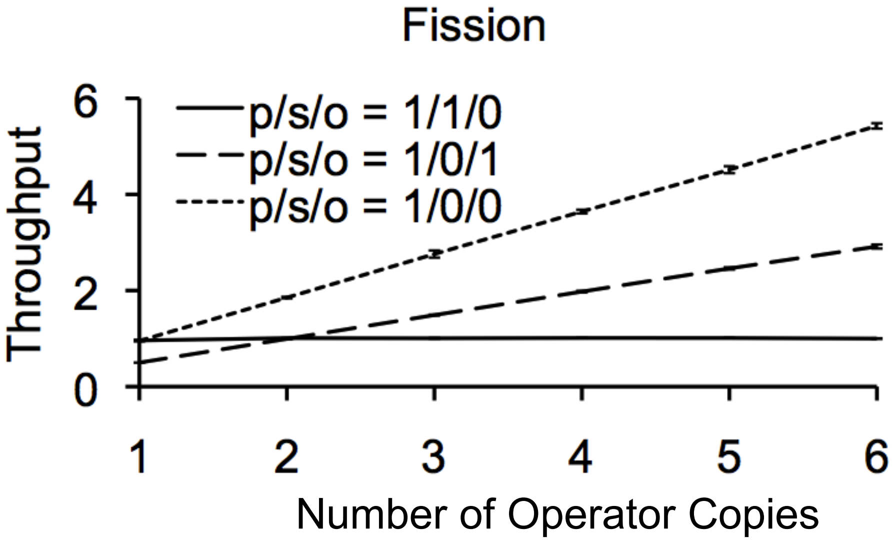
- Also known as
  - Partitioning, data parallelism, replication

- *Parallelize computations.*



Profitability of Fission

- Fission is profitable if the replicated operator is costly enough to be the bottleneck of the system
- Splits and merges can introduce overhead
- p/s/o: cost of the parallel part (A) / cost of the sequential part (B) / overhead



p/s/o (1/1/0): No matter how much we speed up parallel part, sequential part will be bottleneck
p/s/o (1/0/0): Speeds up linearly with number of copies
p/s/o (1/0/1): Until overhead is met (i.e. 2) worse than no parallel

Copyright 2023 – Deepak Turaga

Safety of Fission

- If there is state, it needs to be partitioned stateful
  - Need to use a splitting strategy that assigns tuples to parallel channels based on the partition by attribute
- If ordering is required, merge in order
  - Need to use sequence numbers to re-order tuples at the merge
  - If there are selective operators, blockage can happen
    - Use pulses to address this

Copyright 2023 – Deepak Turaga

Dynamic Fission

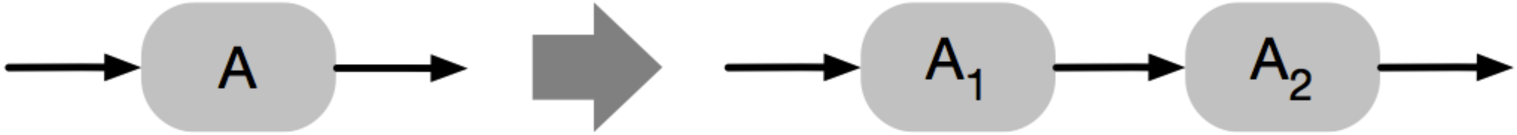
- Dynamism is possible
  - Can increase/decrease the number of parallel channels based on the workload availability
- A number of challenges exist
  - Detecting bottlenecks
    - Congestion and throughput
    - Local vs remote congestion
  - Locating a good operating point
    - Providing SASO properties: Settling time / Accuracy / Stability / Overshoot
  - Migration of state for partitioned stateful operators
    - Consistent hashing

Copyright 2023 – Deepak Turaga

Operator Separation and Pipelines

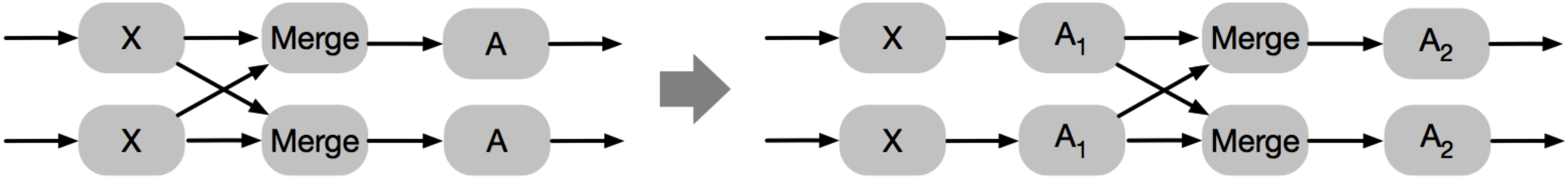
- Also known as
  - Decoupled Software Pipelining

- *Separate operators into smaller computational steps (inverse of Fusion)*

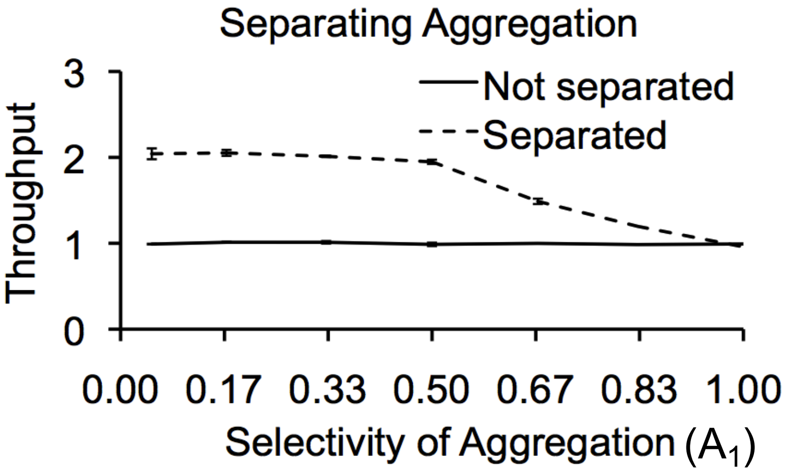


Profitability of Operator Separation

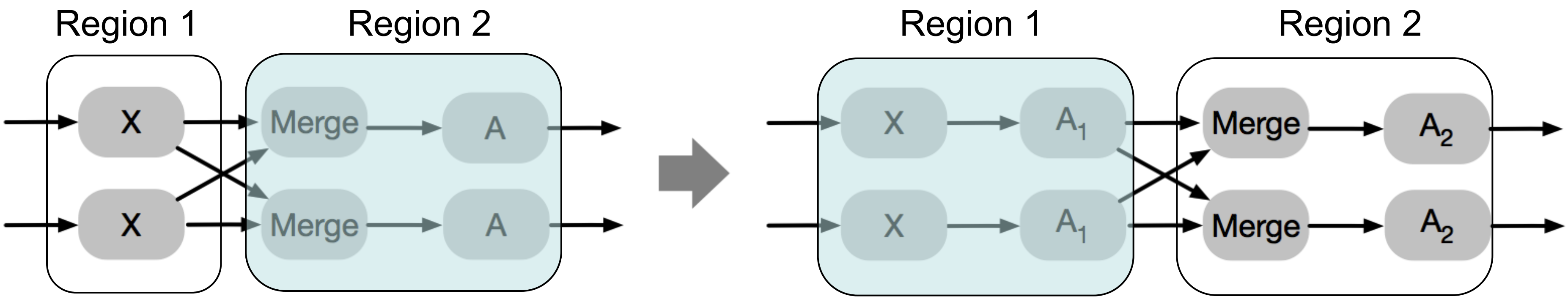
- This is an enabler optimization
  - Operator reordering often benefits from separation
  - Fission benefits from separation
- Example: Combiners in M/R style shuffles



Assume $c(Merge) = 0.5$, $c(A) = 0.5$, $c(X)\sim 0$
Assume A performs filtering (by sentiment and price)
Can do pre-aggregation A1 before merge
Assume $c(A_1)=0.5$, $c(A_2) = 0.5$



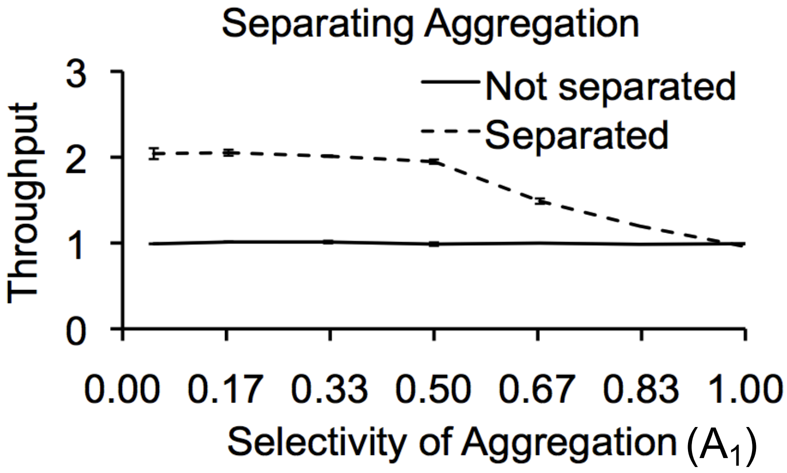
Profitability of Operator Separation

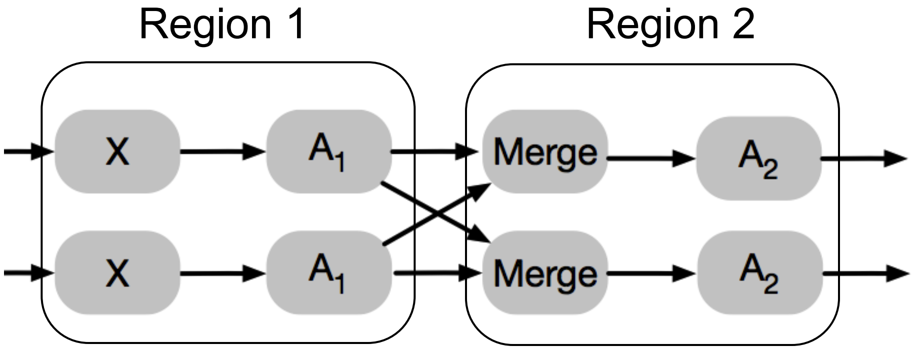


Bottleneck

$s(A_1)\leq 0.5$

Bottleneck





$s(A_1)=1$

As before split

Safety of Operator Separation

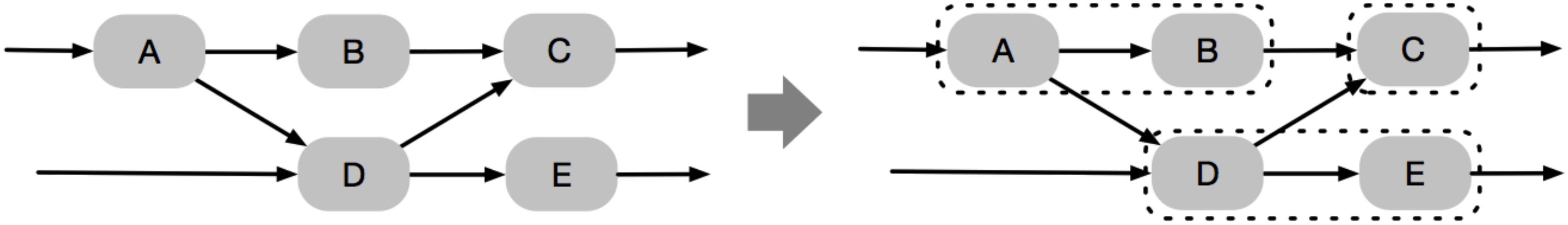
- Ensure that the combination of the separated operators is equivalent to the original operator
  - $A_2(A_1(s)) = A(s)$
  - This is easier to establish in the relational domain
    - If A is a selection op and the selection predicate uses logical conjunction, A_1 and A_2 can be selections on the conjuncts
    - If A is a projection that assigns multiple attributes, then A_1 and A_2 can be projections that assign the attributes separately.
    - If A is an idempotent aggregation, then A_1 and A_2 can simply be the **same** as A itself.

Idempotence: property of certain operations that can be applied multiple times without changing the result

Copyright 2023 – Deepak Turaga

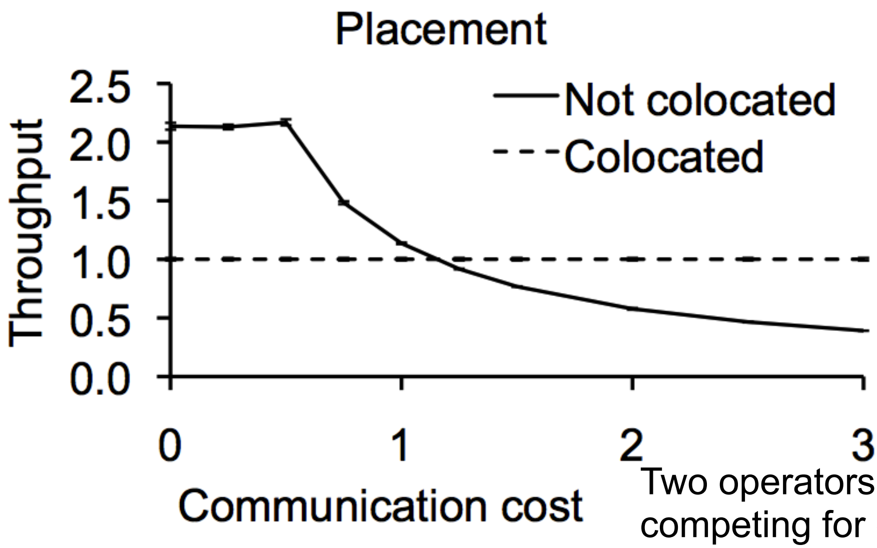
Placement

- *Assigning processing elements to hosts.*



Profitability of Placement

- Placement trades communication cost against resource utilization (disk/memory/CPU)
  - Can use more than one host, if you pay the cost of intra-host transfers



Two operators competing for disk I/O

Assumption: No CPU contention, only resource contention is disk

When communication overhead low, much better to place on different machine

Otherwise degrades with communication cost

Can we generalize to cases with contention for other resources?

Placement with Scheduler

- The Streams job scheduler selects the host (node) where each PE in a submitted job is to be placed
  - Done so in accordance with any requirements specified in the application
- The host is selected from a set of candidate hosts
- Makes an attempt to balance the processing load of the hosts
  - Takes into account their current load
- What does this mean?
  - You can let the system balance the workload for you
- User can provide directives on placement to scheduler

Copyright 2023 – Deepak Turaga

Placement: config Clause

```
Streams<uint32 val> Fil = Filter(Beat) {
  logic state : mutable blooean sw = false;
  onTuple Beat : sw = !sw;
  param filter : sw;
  config placement: host(“streams.ibm.com”);
}
```

Format	Description
`host(“10.4.40.24”);`	User picks host by IP address
`host(“streams.ibm.com”);`	User picks host by name
`host(MyPool[5]);`	User picks host at a fixed offset from the pool
`host(MyPool);`	System picks host from the pool at runtime

Copyright 2023 – Deepak Turaga

Placement: Host Pools

The main composite's config clause is used to create node pools
Node pools can be defined with specific associated hosts
Can use the `createPool` sub-config clause to create the pool at runtime

```
composite Main {
  graph
    operator1 {}
    operator2 {}
  config hostPool:
    MyPool= ["ibmclass.ibm.com", "10.8.5.6"],
    P1= createPool({size=5u, tags=["res","prod"]}, Sys.Shared),
    P2= createPool({size=10u, tags=[“my"]}, Sys.Exclusive);
}
```

Node pools can be defined with specific associated hosts

Other Placement Directives

- `hostColocation`
  - Operator instances with the same hostColocationvalue will run on the same host

    ```
    config placement : hostColocation(“myHost");
    ```
- `hostIsolation`
  - Operator instance belongs to a partition that has a host of its own
  - Other operator instances can be in the partition
  - But no other partition can run on that host

    ```
    config placement : hostIsolation;
    ```
- `hostExlocation`
  - Any operator instances that have the same hostExlocation value must run on different hosts

  ```
  config placement : hostExlocation(“notMyHost");
  ```

Safety of Placement

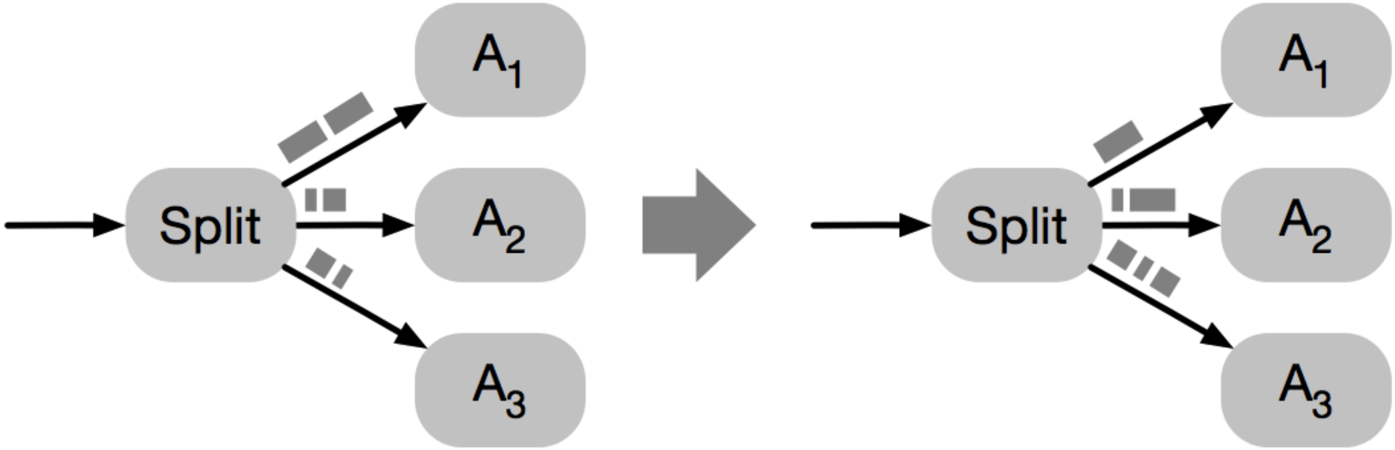
- Ensure resource kinds
  - Placement is safe if each host has the right resources for all the operators placed on it.
- Ensure resource amounts
  - Total amount of resource required by the co-placed operators should not exceed the amount of resources available on a single host
- If placement is dynamic, move only relocatable operators
  - Moving state and ensuring no tuple loss is an issue
  - Operators with OS resources such as sockets and files may be designated as non-movable.

Dynamic Placement

- Placement decisions are often made during submission time
- Some placement algorithms continue to be active after the job starts, to adapt to changes in load or resource availability
  - As discussed, this poses additional safety issues
    - State migration, tuple loss

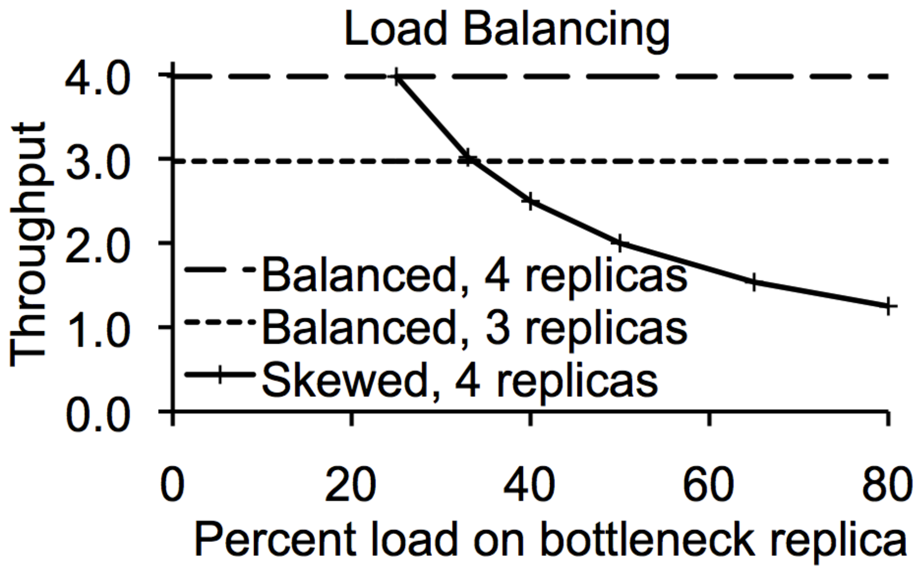
Load-balancing for Data Parallel Splits

- *Distribute workload evenly across resources*
  - *Data parallel processing*



Profitability of Load-balancing

- Load balancing is profitable if it compensates for skew
- Skew can be in the
  - data: some partitions have higher volume than others
  - resources: some channels have more capacity than others



Split operator that streams data to 3 or 4 replicated operators. With perfect load balancing, throughput is linear in replicas. Without load balancing, there is skew, and throughput is bounded by whichever replica receives the most load

Safety of Load-balancing

- Avoid starvation
  - Every data item eventually gets processed
- Each worker is qualified
  - A data item can only be routed to a worker that is capable of handling it
    - E.g. In partitioned stateful parallelism, tuples from the same partition should go to the same place

Copyright 2023 – Deepak Turaga

Variations in Load-balancing

- Balancing load during operator placement
  - Host resources are utilized evenly
  - During submission, prefer placement on hosts that are less utilized
- Balance load during fission
  - Apply fission and then decide how much load each parallel channel gets
  - i.e., combined fission and load balance

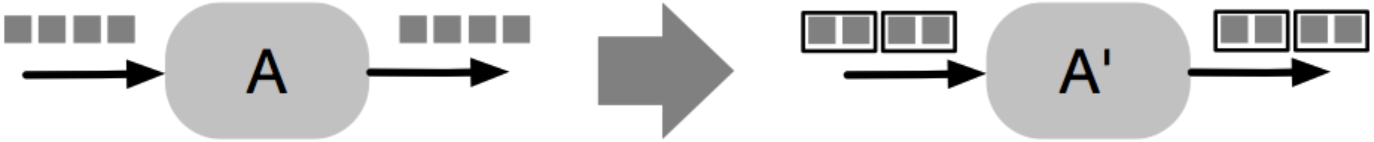
Dynamic Load Balancing

- We discussed two forms of load balancing
  - Routing based: Usually done dynamically
    - Requires splitter level decisions at runtime
  - Placement based: Typically static
    - Requires moving operators at runtime
      - Could take long time, may not be safe

Copyright 2023 – Deepak Turaga

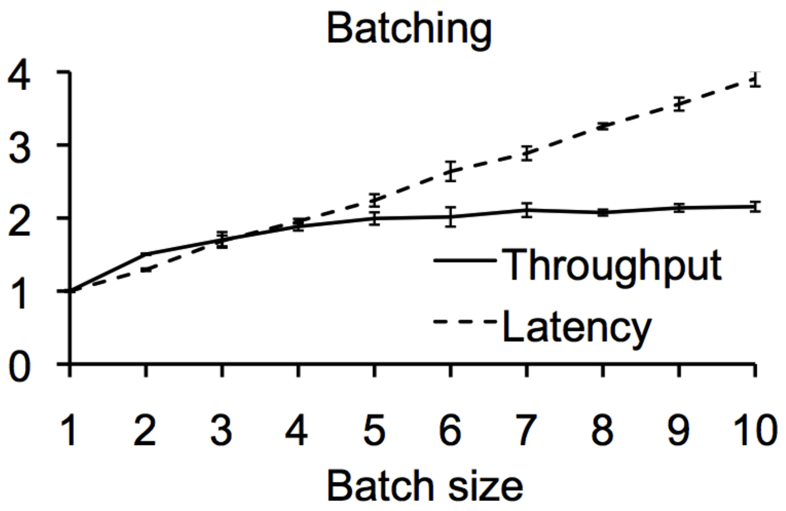
Batching

- Also known as
  - Train scheduling, execution scheduling
- *Process multiple data items in a single batch*
  - *Like RDDs*



Profitability of batching

- Batching trades throughput for latency
- Can reduce operator firing and communication costs
- The amortizable costs can include
  - Scheduling cost, context switch cost, synchronization cost, instruction cache costs, data cache costs



Assume $c(A) = c(Tuple) + c(overhead)$
With batching n tuples
$c(A') = n \times c(Tuple) + c(overhead)$
Amortize overhead costs
Latency: delay between when tuple arrives to when it is transmitted. *Different for different tuples in the batch.* Maximum delay scales linearly with batch size.

Safety of Batching

- Avoid deadlock
  - In the presence of feedback loops
  - In the presence of shared locks
- Satisfy deadlines
  - Real-time constraints on the per-tuple delay
  - QoS (quality of service) constraints, where delay impacts the QoS value
    - E.g. maintaining frame rates to avoid jitter in video processing

Copyright 2023 – Deepak Turaga

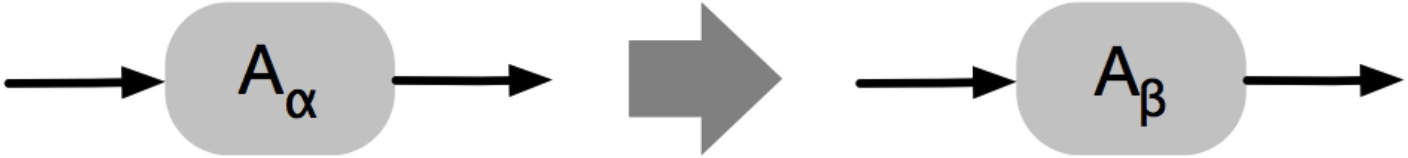
Dynamism

- The batch size can be set
  - Statically
    - StreamIt uses instruction cache vs data cache tradeoff to set the batch sizes
  - Dynamically
    - Aurora uses train scheduling to minimize context switching costs
    - Staged Event-Driven Architecture (SEDA) uses the tradeoff between latency and adaptivity to set the buffer sizes

Copyright 2023 – Deepak Turaga

Algorithm Selection

- Also known as
  - translation to physical query plan
- *Use a faster algorithm for implementing an operator.*



Copyright 2023 – Deepak Turaga

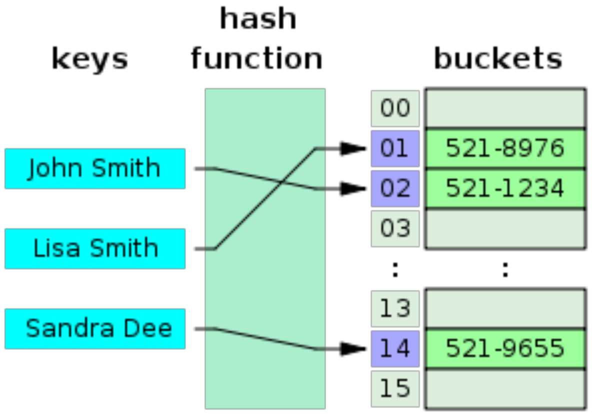
Profitability of algorithm selection

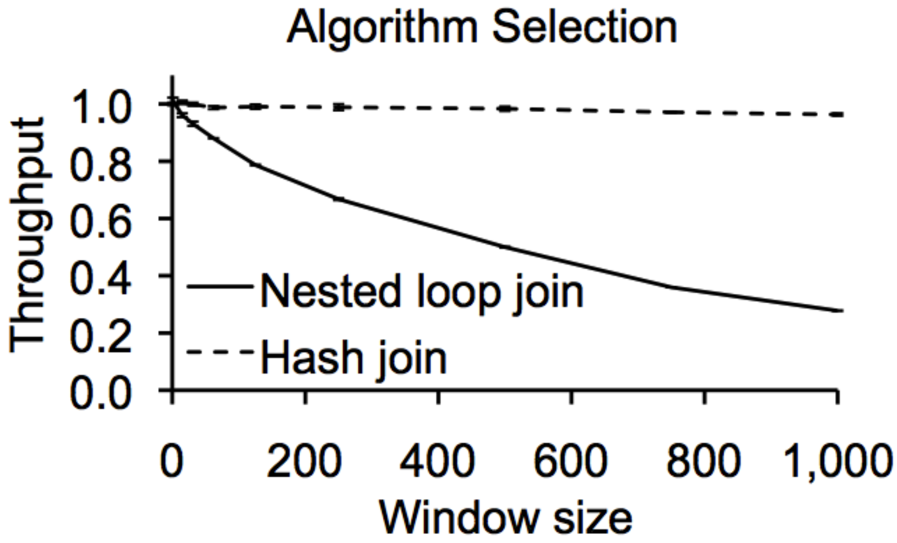
- It is profitable if a costly operator is replaced with a cheap one
- One algorithm may not be superior in all cases
  - E.g. An algorithm that works faster for small tuples vs. an algorithm that works faster for larger tuples
  - E.g. An algorithm that works faster but uses more memory.
- Example: Join
  - A hash-based implementation is almost always faster (unless the window is very small) than loop join (linear scan)
  - A hash-based implementation is suitable only for equi-joins

Copyright 2023 – Deepak Turaga

Profitability of algorithm selection

- Loop join: For each tuple iterate through the tuples in window of other stream to find match
- Hash join: Create a hash table for the stream window. Apply hash function to new tuple on other stream to find match





Aside on Hash Functions

```
function Hash(key)
  return key mod PrimeNumber
end

Additive Hash
ub4 additive(char *key, ub4 len, ub4 prime)
{
  ub4 hash, i;
  for (hash=len, i=0; i<len; ++i)
    hash += key[i];
  return (hash % prime);
}

Rotating Hash
ub4 rotating(char *key, ub4 len, ub4 prime)
{
  ub4 hash, i;
  for (hash=len, i=0; i<len; ++i)
    hash = (hash<<4)^(hash>>28)^key[i];
  return (hash % prime);
}
```

```
Bernstein's hash
ub4 bernstein(ub1 *key, ub4 len, ub4 level)
{
  ub4 hash = level;
  ub4 i;
  for (i=0; i<len; ++i) hash = 33*hash + key[i];
  return hash;
}

If your keys are lowercase English words, this
will fit 6 characters into a 32-bit hash with no
collisions
```

Several cryptographic hash functions, e.g. SHA2

Safety of algorithm selection

- Safety
  - Ensure same behavior
  - Be aware of variations
- Variations
  - Physical query plans
  - Auto-tuners
    - Empirical optimization
  - Different semantics
    - Load shedding via cheaper but less accurate algorithms

Copyright 2023 – Deepak Turaga

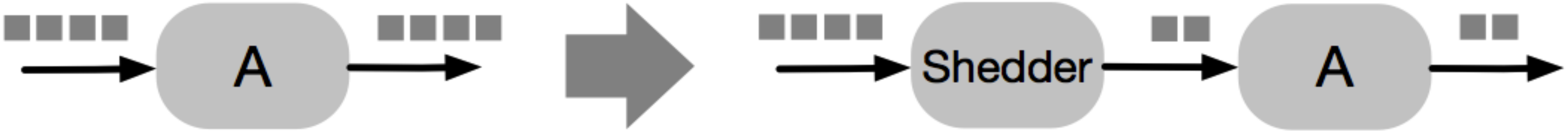
Dynamics of algorithm selection

- Dynamism
  - Algorithm selection can be dynamic
- Both algorithms are provisioned and the right one is selected at runtime
  - Performed via dynamic routing
- Change algorithm operation via parameter choice
  - E.g. size of ensemble (IMARS example)

Copyright 2023 – Deepak Turaga

Load-shedding

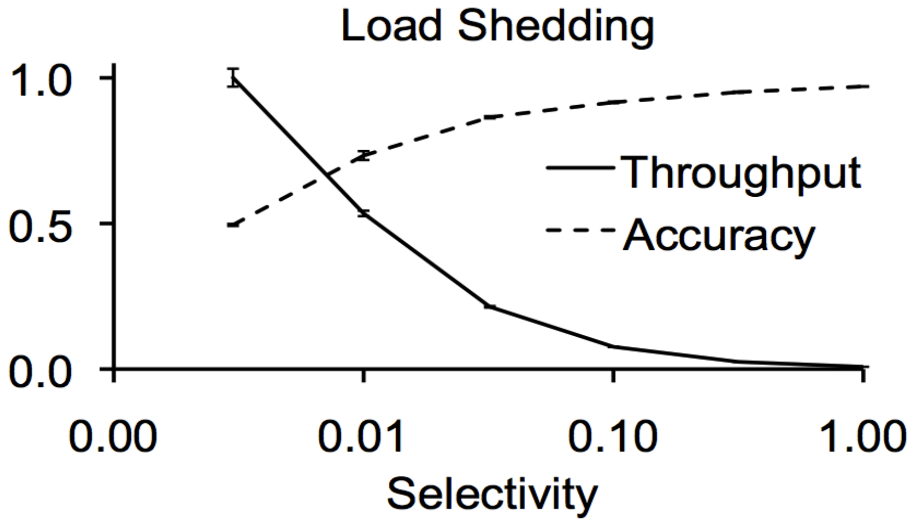
- Also known as
  - Admission control, graceful degradation
- *Degrade gracefully when overloaded*.



Profitability of load-shedding

- Load shedding improves throughput at the cost of accuracy
- Consider an aggregator-like algorithm that constructs a histogram
  - Sampling can be used effectively
  - Reducing the rate to one tenth may have a negligible impact on the histogram accuracy

Euclidean distance between histogram at full accuracy, and histogram at reduced accuracy



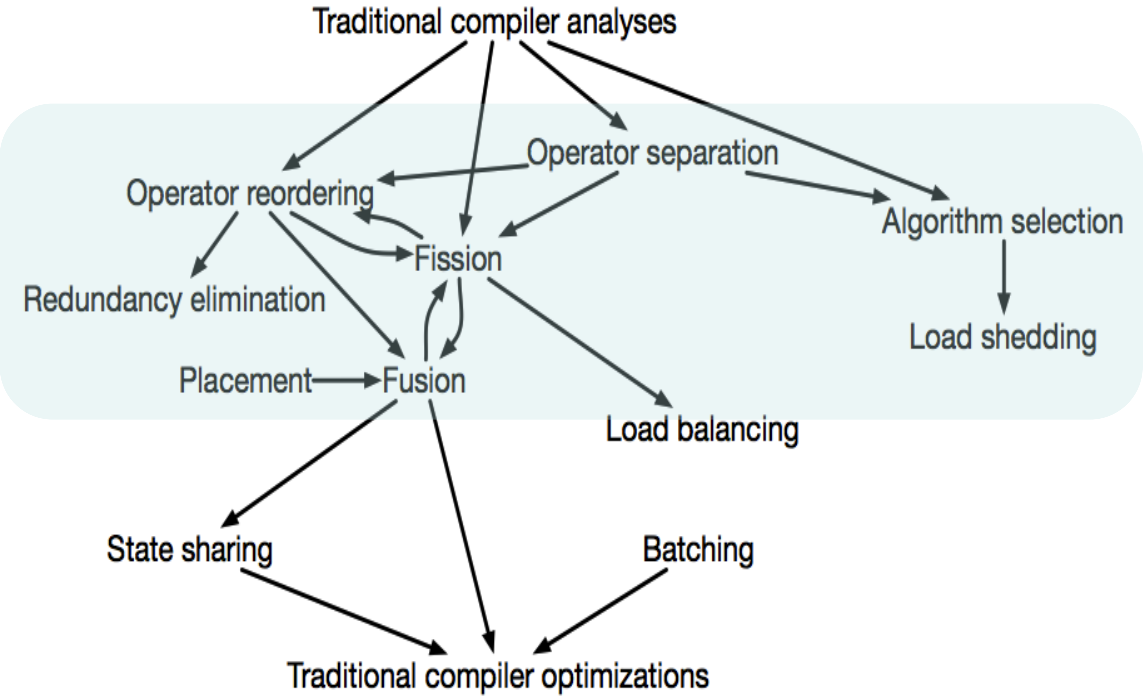
Safety and Dynamism of load-shedding

- Load shedding, by definition, is not safe
  - Ideally, the reduction in the quality should be acceptable
  - Major area of research
- Load shedding is always dynamic
  - Change the selectivity based on current load

Copyright 2023 – Deepak Turaga

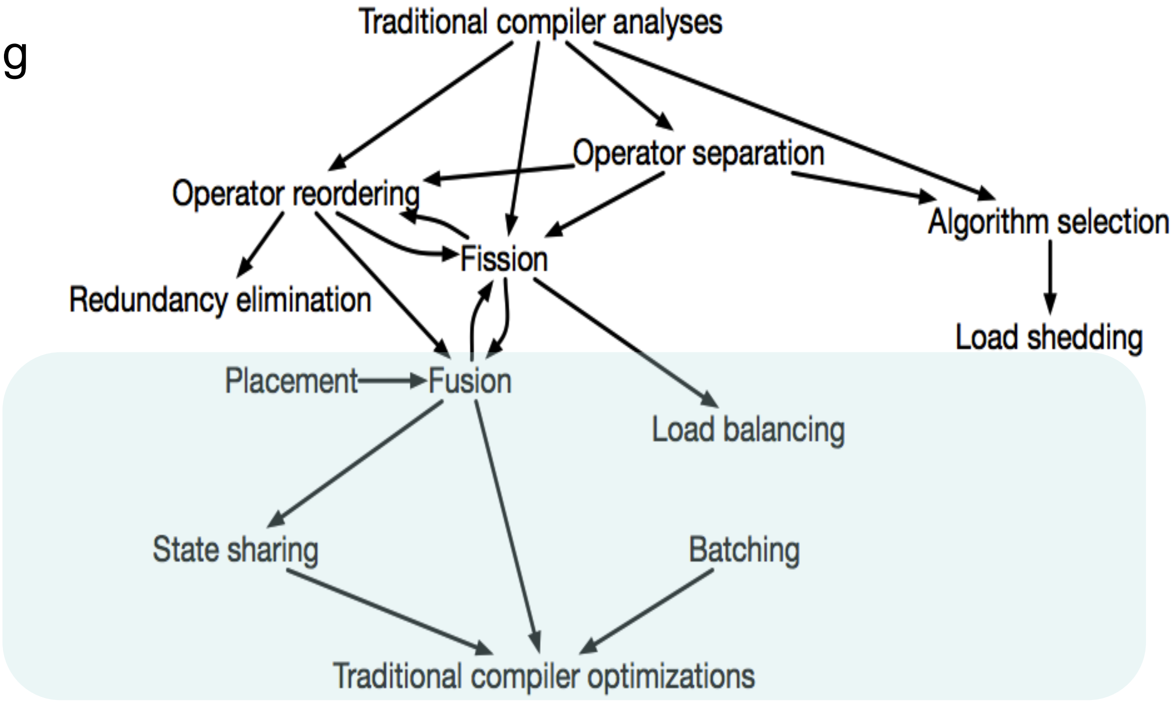
Interactions between Optimizations

1. Primary enablers are operator separation and operator reordering
2. Circular enablement between operator reordering and fission
3. Circular enablement between fission and fusion
4. Fission makes it easier to balance load



Interactions between Optimizations

1. Fusion enables function inlining
2. State sharing enables scalar replacement (replace arrays/structs with scalars), if the compiler can statically determine the size of a communication queue based on fixed data rates, and can determine the index of data access for each iteration
3. Batching enables loop unrolling, software pipelining, and loop optimizations

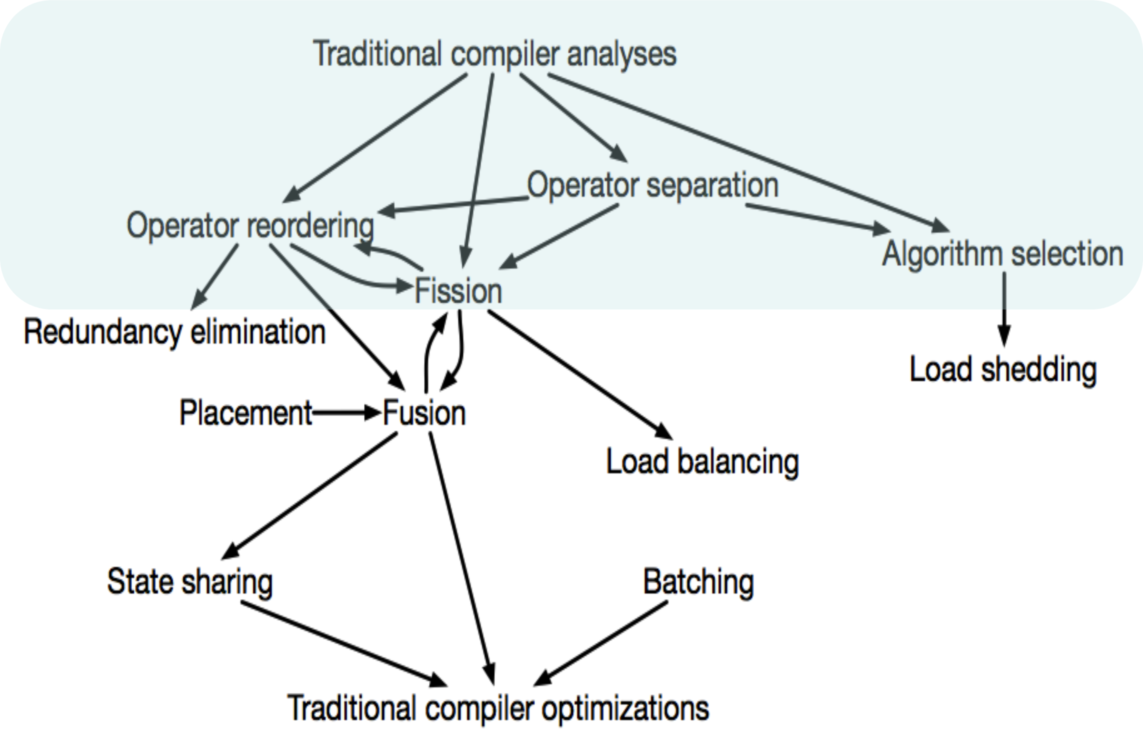


http://www.cs.uiuc.edu/class/fa06/cs498dp/notes/optimizations.pdf

Copyright 2023 – Deepak Turaga

Interactions between Optimizations

1. Operator reordering can be enabled by commutativity analysis
2. Operator separation can be supported by compiler analysis
3. Fission can also be supported by compiler analysis
4. Algorithm selection can be supported by worst-case execution time analysis



Discussion on Optimization

- Metrics for evaluation
  - many ways to measure whether a streaming optimization was profitable
  - throughput, latency, quality of service (QoS), accuracy, power, and system utilization
- Need standard benchmarks for streaming workloads
  - Stanford stream query repository including Linear Road [Arasu et al. 2006],
  - BiCEP benchmarks [Mendes et al. 2009]
  - StreamIt benchmarks [Thies and Amarasinghe 2010]
  - *Project idea*? ☺

Arasu, A., Babu, S., and Widom, J. 2006. The CQL continuous query language: semantic foundations and query execution. The VLDB Journal 15, 2 (June), 121–142.
Mendes, M. R. N., Bizarro, P., and Marques, P. 2009. A performance study of event processing systems. In TPC Technology Conference on Performance Evaluation & Benchmarking (TPC TC). 221–236.
Thies, W. and Amarasinghe, S. 2010. An empirical characterization of stream programs and its implications for language and compiler design. In Parallel Architectures and Compilation Techniques (PACT). 365–376.

Copyright 2023 – Deepak Turaga

Wrap-up on Optimization

- Non-trivial to optimize performance
  - Compiler and programming language support important
  - Often requires manual tuning and experimentation
  - Lots of dynamics (compute resources, operator performance, data characteristics and workload)
  - Several different competing objectives
- Guiding principle
  - Keep operators small and lightweight and explore many knobs
- Impact on fault tolerance
  - Many optimizations are orthogonal to whether or not the system is fault tolerant.
- Centralized versus distributed optimization
  - Assumptions on shared memory, or other resources may not be valid

Copyright 2023 – Deepak Turaga