Stream Processing Optimizations

Scott Schneider
IBM Thomas J. Watson Research Center
New York, USA

Martin Hirzel
IBM Thomas J. Watson Research Center
New York, USA

Buğra Gedik
Computer Engineering Department
Bilkent University
Ankara, Turkey

Agenda

- 9:00-10:30
  - Overview and background (40 minutes)
  - Optimization catalog (50 minutes)

- 11:00-12:30
  - SPL and InfoSphere Streams background (25 minutes)
  - Fission (40 minutes)
  - Open research questions (25 minutes)

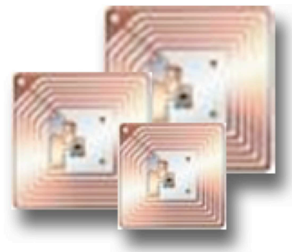
DEBS'13 Tutorial: Stream Processing Optimizations

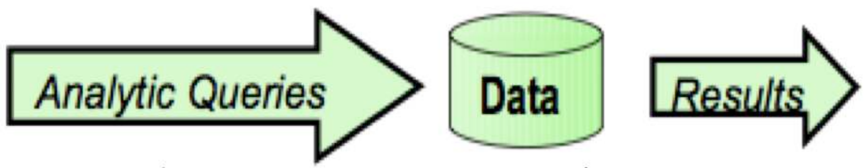
Scott Schneider, Martin Hirzel, and Buğra Gedik
Acknowledgements: Robert Soulé, Robert Grimm, Kun-Lung Wu

Part 1: Overview and Background

Stream Processing

- Streaming sources are plenty
  - Volume, Velocity, Variety
- Online analysis is paramount
  - Quickly process and analyze data, derive insights, and take timely action

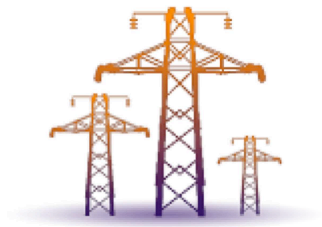




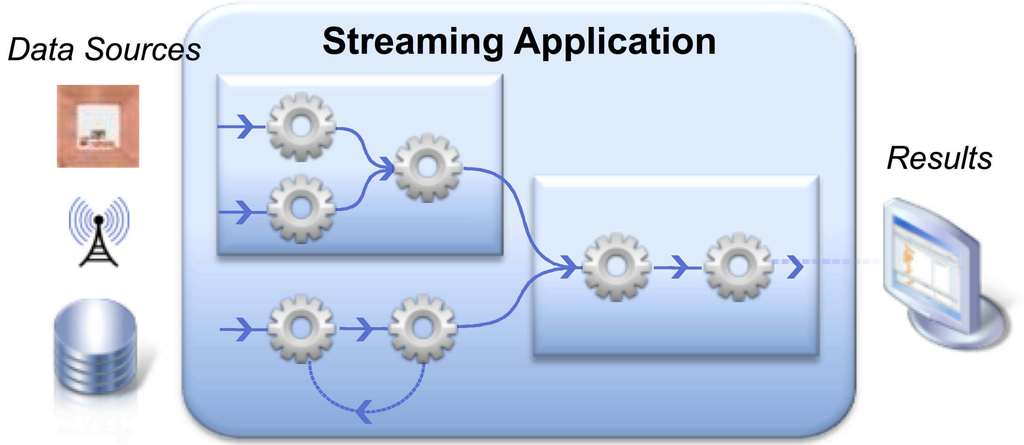
Telco analyses streaming network data to reduce hardware costs by 90%

Utility avoids power failures by analysing 10 PB of data in minutes

Hospital analyses streaming vitals to detect illness 24 hours earlier

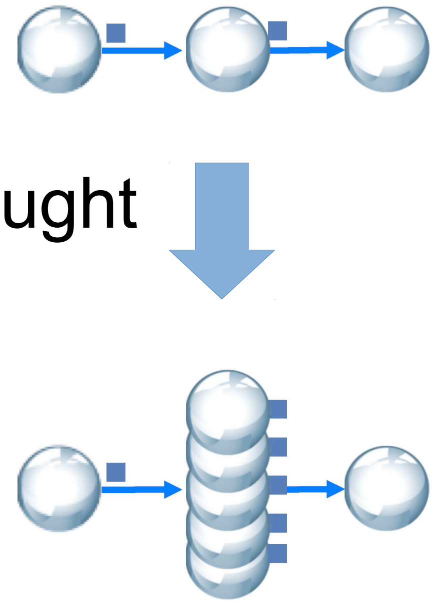


Catalog of Streaming Optimizations



- Streaming applications: graph of streams and operators
- Performance is an important requirement
- Different communities → different terminology
  - e.g. operator/box/filter; hoisting/push-down
- Different communities → different assumtions
  - e.g. acyclic graphs/arbitrary graphs; shared memory/distributed
- Catalouge of optimizations
  - Uniform terminology
  - Safety & profitability conditions
  - Interactions among optimizations

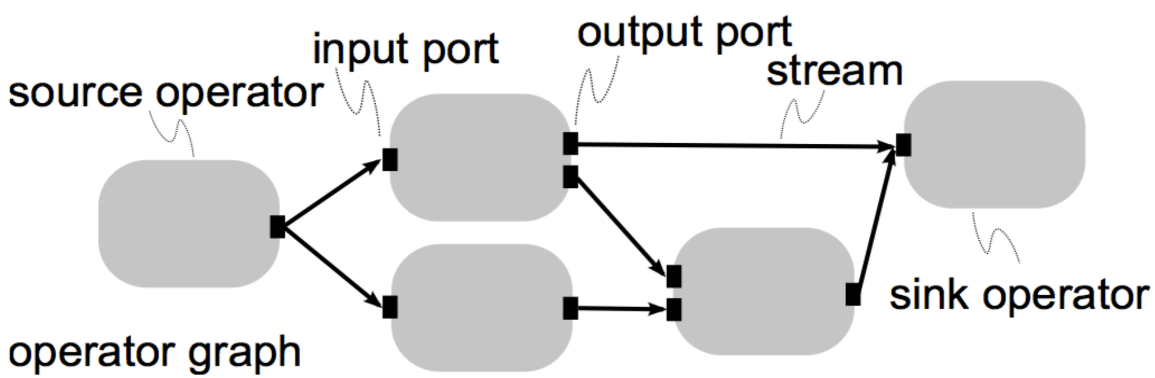
Fission Optimization



- High throughput processing is a critical requirement
  - Multiple cores and/or host machines
  - System and language level techniques
- Application characteristics limit the speedup brought by optimizations
  - pipeline depth (# of ops), filter selectivity
- Data parallelism is an exception
  - number of available cores (can be scaled)
- **Fission**
  - Data parallelism optimization in streaming applications
  - How to apply transparently, safely, and adaptively?

Background

- *Operator graph*
  - Operators connected by streams
- *Stream*
  - A series of data items
- *Data item*
  - A set of attributes
- *Operator*
  - Generic data manipulator
  - Has input and output *ports*
  - Streams connect output ports to input ports
    - FIFO semantics
  - *Source* operator, no input ports
  - *Sink* operator, no output ports
- Operator *firing*
  - Perform processing, produce data items



State in Operators

- *Stateful* operators
  - Maintain state across firings
  - E.g., *deduplicate*: pass data items not seen recently
- *Stateless* operators
  - Do not maintain state across firings
  - E.g., *filter*: pass data items with values larger than a threshold
- *Partitioned stateful* operators
  - Maintain independent state for non-overlapping sub-streams
  - These sub-streams are identified by a *partitioning attribute*
  - E.g.: For each stock symbol in a financial trading stream, compute the volume weighted average price over the last 10 transactions.
    The partitioning attribute: stock symbol.

Selectivity of Operators

- Selectivity
  - the number of data items produced per data item consumed
  - e.g., selectivity=0.1 means
    - 1 data item is produced for every 10 consumed
  - used in establishing safety and profitability
- Dynamic selectivity
  - selectivity value is
    - *not known at development time*
    - *can change at run-time*
  - e.g., data-dependent filtering, compression, or aggregates on time-based windows

Selectivity Categories

- Selectivity categories (singe input/output operators)
  - *Exactly-once* (=1): one in; one out [always]
  - *At-most-once* (≤1): one in; zero or one out [always]
  - *Prolific* (≥1): one in; one, or more out [sometimes]
- *Synchronous data flow (SDF) languages*
  - Assume that the selectivity of each operator is fixed and known at compile time
  - Provide good optimization opportunities at the cost of reduced application flexibility
  - Typically used for signal processing applications
- Unlike SDF, we assume dynamic selectivity
  - Support general-purpose streaming
- Selectivity categories are used to fine-tune optimizations

Streaming Programming Models

Synchronous

- Static selectivity
  - e.g., 1 : 3

```
for i in range(3):
    result = f(i)
    submit(result)
```

  - In general, *m* : *n* where *m* and *n* are statically known
  - Always has static schedule

Asynchronous

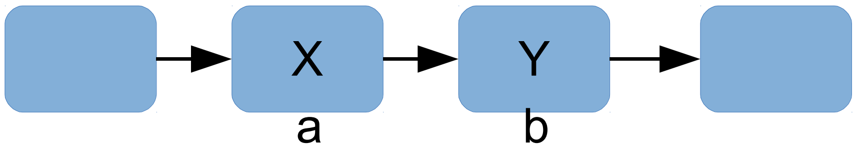
- Dynamic selectivity
  - e.g., 1 : [0,1]

```
if input.value > 5:
  submit(result)
```

  - In general, 1 : *
  - In general, schedules cannot be static

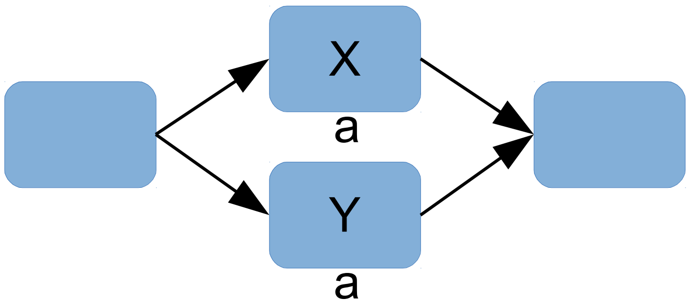
Flavors of Parallelism

- There are three main forms of parallelism in streaming applications
  - Pipeline, task, and data parallelism



pipeline

an operator processes a data item at the same time its upstream operator processes the next data item

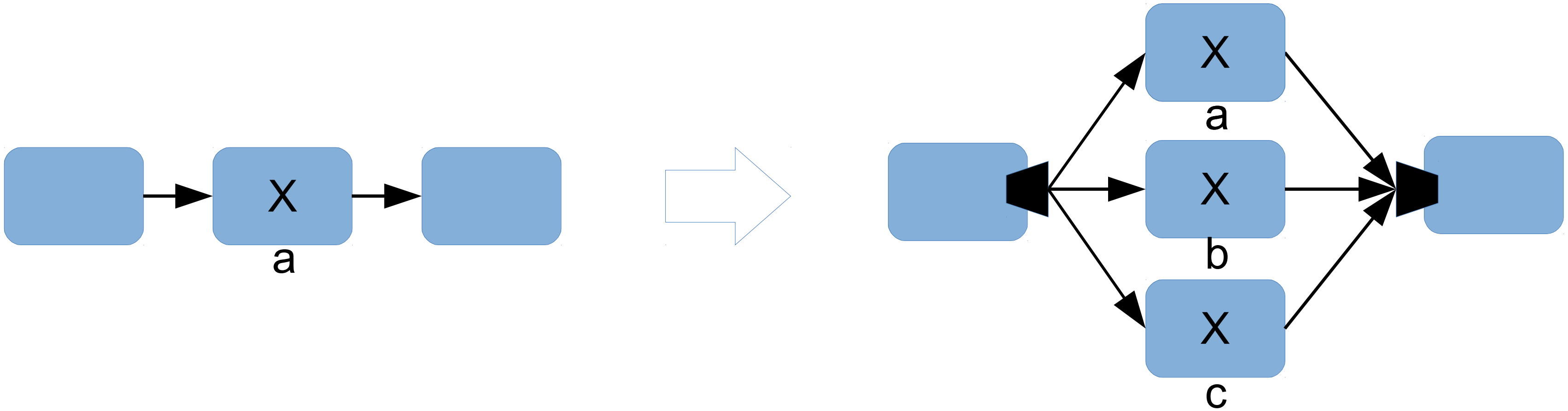


task

different operators process a data item produced by their common upstream operator, at the same time

- Pipeline and task parallelism are inherent in the graph

Data Parallelism



different data items from the same stream are processed by the replicas of an operator, at the same time

- Data parallelism needs to be extracted from the application
  - Morph the graph
    - Split: distribute to replicas
    - Replicate: do data parallel processing
    - Merge: put results back together
- Requires additional mechanisms to preserve application semantics
  - Maintaining the order of tuples
  - Making sure state is partitioned correctly

Safety and Profitability

- *Safety*: an optimization is *safe* if applying it is guaranteed to maintain the semantics
  - State (stateless & partitioned stateful)
    - Parallel region formation, splitting tuples
  - Selectivity
    - Result ordering, splitting and merging tuples
- *Profitability*: an optimization in profitable if it increases the performance (throughput)
  - Transparency: Does not require developer input
  - Adaptivity: Adapt to resource and workload availability

Adaptive Optimization

- When the workload increases, more resources should be requested
- In the context of data parallelism
  - How many parallel channels to use at a given time
- Maintaining SASO properties is a challenge
  - **S**tability: do not oscillate wildly
  - **A**ccuracy: eventually find the most profitable operating point
  - **S**ettling time: quickly settle on an operating point
  - **O**vershoot: steer away from disastrous settings

Publications

- M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. **A catalog of stream processing optimizations**. Technical Report RC25215, IBM Research, 2011. Conditionally accepted to ACM Computing Surveys, minor revisions pending.
- S. Schneider, M. Hirzel, B. Gedik, and K-L. Wu. **Auto-Parallelizing Stateful Distributed Streaming Applications**, International Conference on Parallel Architectures and Compilation Techniques (PACT), 2012.
- R. Soulé, M. Hirzel, B. Gedik, and R. Grimm. **From a Calculus to an Execution Environment for Stream Processing**, International Conference on Distributed Event Based Systems, ACM (DEBS), 2012.
- Y. Tang and B. Gedik. **Auto-pipelining for Data Stream Processing**, Transactions on Parallel and Distributed Systems, IEEE (TPDS), ISSN: 1045-9219, DOI: 10.1109/TPDS.2012.333, 2012.
- H. Andrade, B. Gedik, K-L. Wu, and P. S. Yu. **Processing High Data Rate Streams in System S**, Journal of Parallel and Distributed Computing - Special Issue on Data Intensive Computing, Elsevier (JPDC), Volume 71, Issue 2, 145-156, 2011.
- R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, H. Andrade, K-L. Wu, and B. Gedik. **COLA: Optimizing Stream Processing Applications Via Graph Partitioning**, International Middleware Conference, ACM/IFIP/USENIX (Middleware), 2009.
- B. Gedik, H. Andrade, and K-L. Wu. **A Code Generation Approach to Optimizing High-Performance Distributed Data Stream Processing**, International Conference on Information and Knowledge Management, ACM (CIKM), 2009.
- S. Schneider, H. Andrade, B. Gedik, A. Biem, and K-L. Wu. **Elastic Scaling of Data Parallel Operators in Stream Processing**, International Parallel and Distributed Processing Symposium, IEEE (IPDPS), 2009.

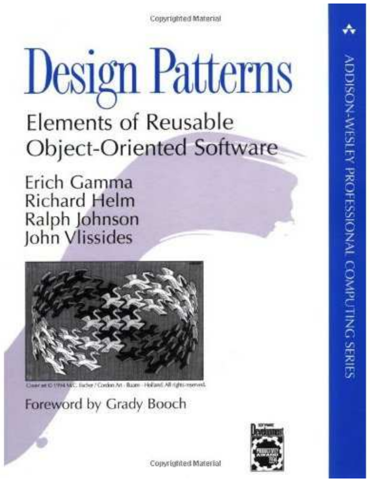
- **SPL Language Reference**. IBM Research Report RC24897, 2009.

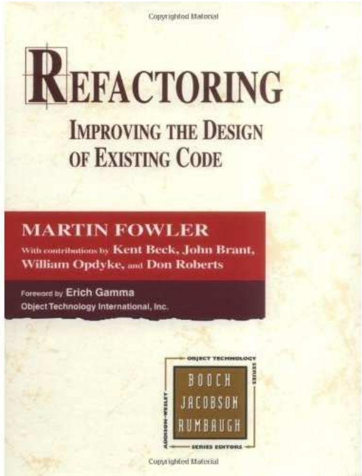
DEBS’13 Tutorial: Stream Processing Optimizations

Scott Schneider, Martin Hirzel, and Buğra Gedik
Acknowledgements: Robert Soulé, Robert Grimm, Kun-Lung Wu

Part 2: Optimization Catalog

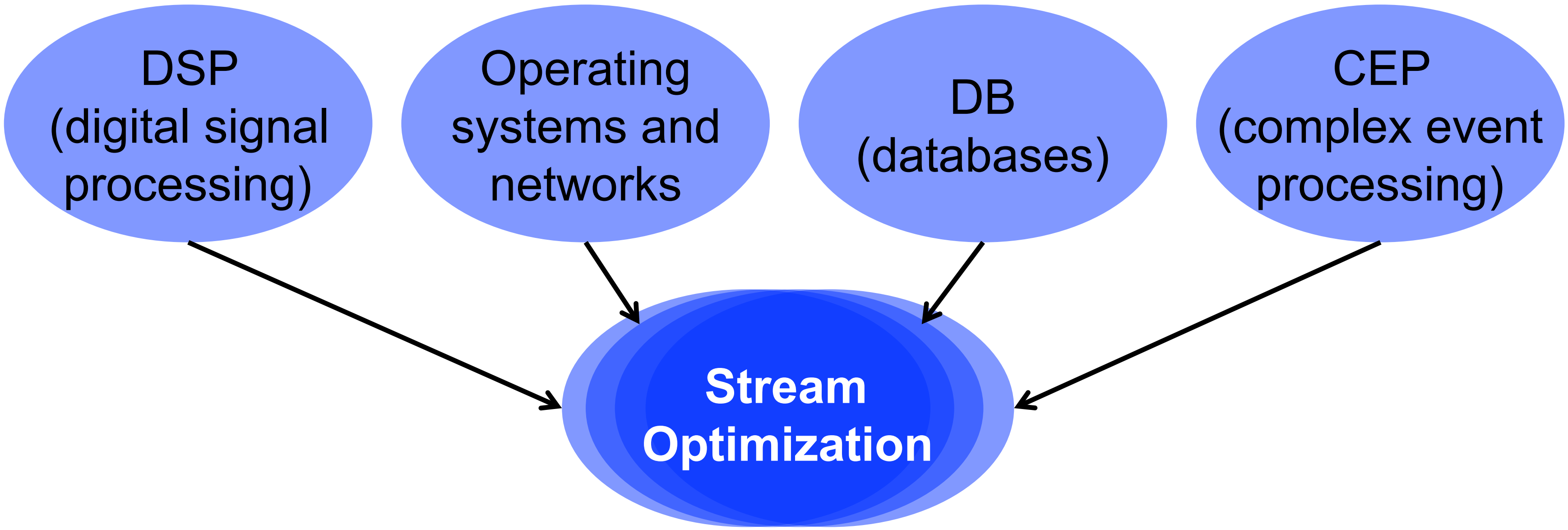
Motivation





- Catalog = survey, but organized as easy reference
- Use cases:
  - User: understand optimized code; hand-implement optimizations
  - System builder: automate optimizations; avoid interference with other features
  - Researcher: literature survey (see paper); open research issues

Stream Optimization Literature



Conflicting terminology

- Operator = filter = box = stage = actor = module
- Data item = tuple = sample
- Join = relational vs. any merge
- Rate = speed vs. selectivity

Unstated assumptions

- Missing safety conditions
- Missing profitability trade-offs
- Any graph vs. forest vs. single-entry, single-exit region
- Shared-memory vs. distributed

Optimization Name

Key idea.

Graph before → Graph after

Safety

- Preconditions for correctness

Variations

- Most influential published papers

Profitability

Throughput (higher is better)

- Micro-benchmark
- Runs in SPL
- Relative numbers
- Error bars are standard deviation of 3+ runs

Central trade-off factor

Dynamism

- How to optimize at runtime

List of Optimizations

Graph changed	*Graph unchanged*
Operator reordering	Placement
Redundancy elimination	Load balancing
Operator separation	State sharing
Fusion	Batching
Fission	Algorithm selection
	Load shedding

Semantics unchanged

Semantics changed

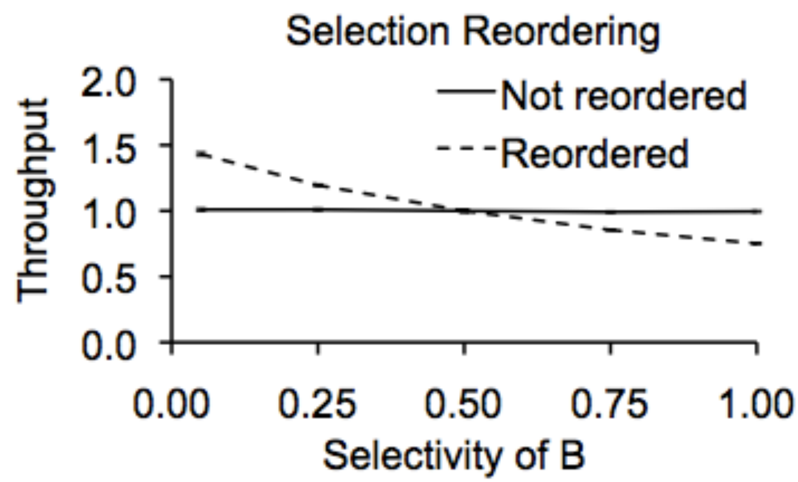
Operator Reordering

Change the order in which operators appear in the graph.

Safety

- Commutative
- Attributes available

Profitability



Variations

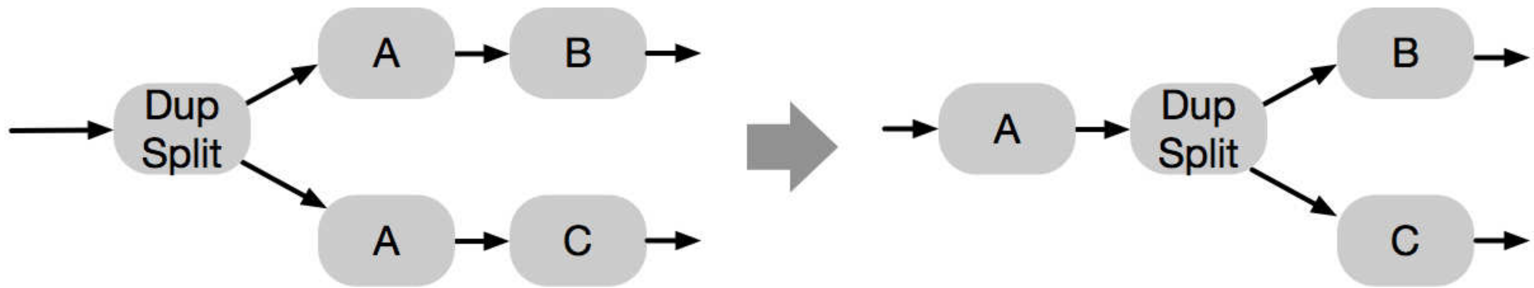
- Algebraic
- Commutativity analysis
- Synergies, e.g. fusion, fission

Dynamism

- Eddy

Redundancy Elimination

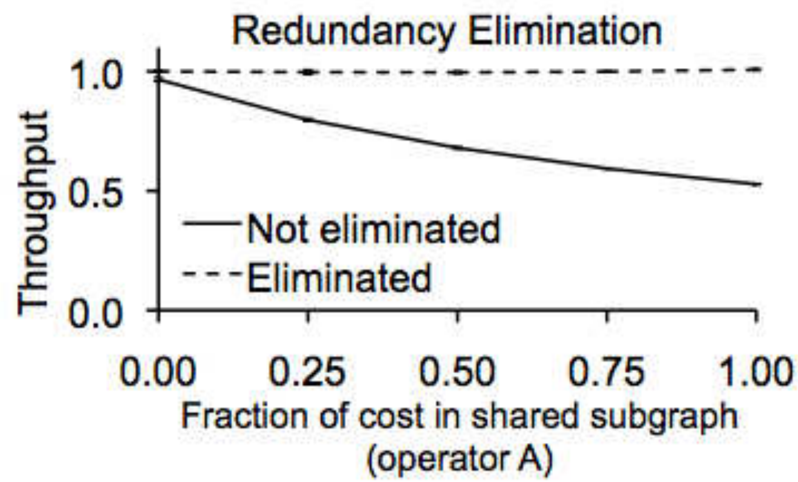
Eliminate operators that are redundant in the graph.



Safety

- Same algorithm
- Data available

Profitability



Variations

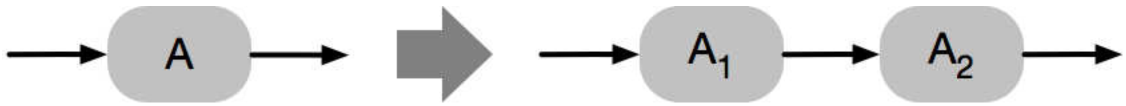
- Many-query optimization
- Eliminate no-op
- Eliminate idempotent operator
- Eliminate dead subgraph

Dynamism

- In many-query case:
  share at submission time

Operator Separation

Separate an operator into multiple constituent operators.



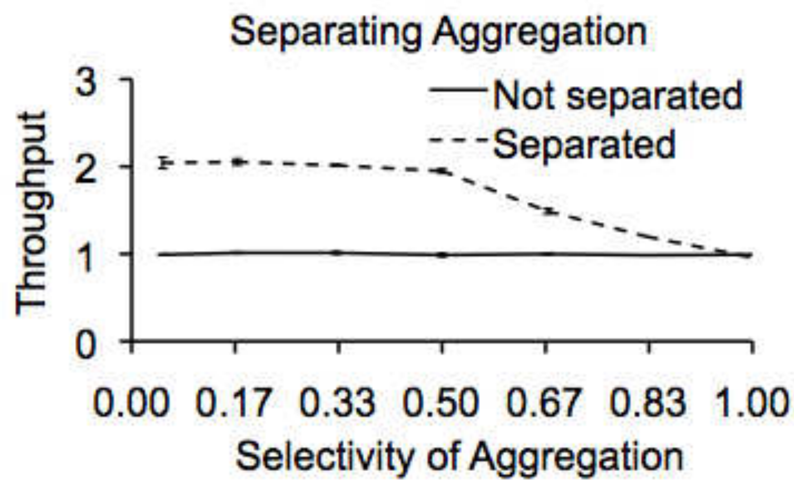
Safety

- Ensure $A_1(A_2(s)) = A(s)$

Variations

- Algebraic
- Using special API
- Dependency analysis
- Enables reordering

Profitability



Dynamism

- N/A

Fusion

Fuse multiple separate operators into a single operator.

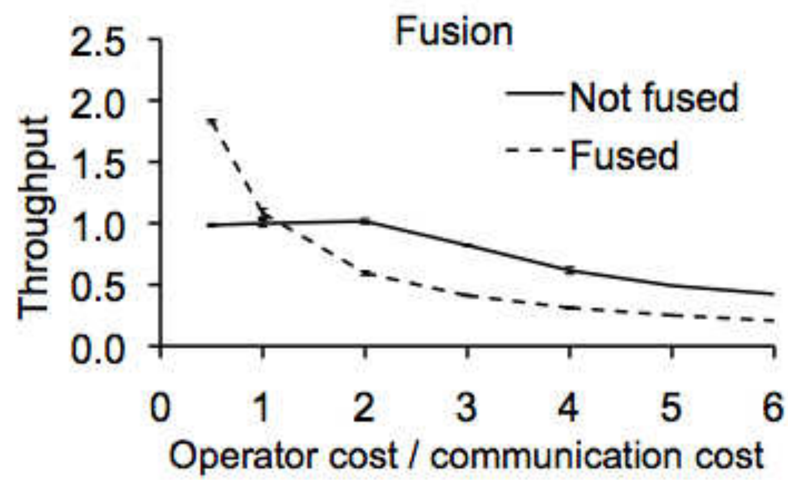
Safety

- Have right resources
- Have enough resources
- No infinite recursion

Variations

- Single vs. multiple threads
- Fusion enables traditional compiler optimizations

Profitability

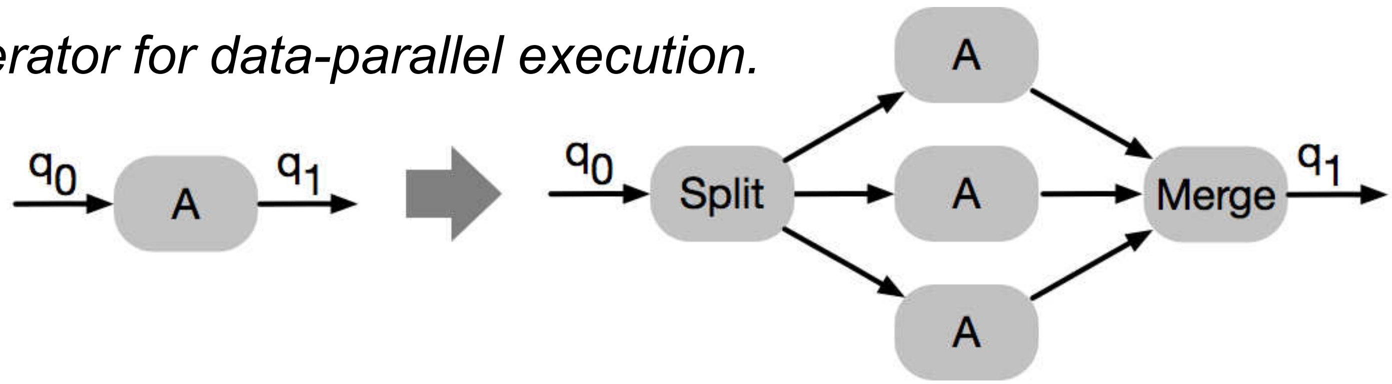


Dynamism

- Online recompilation
- Transport operators

Fission

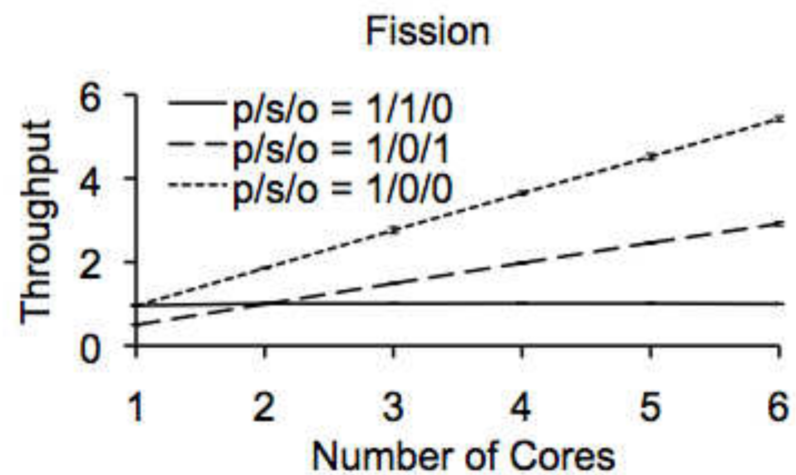
Replicate an operator for data-parallel execution.



Safety

- No state or disjoint state
- Merge in order, if needed

Profitability



Variations

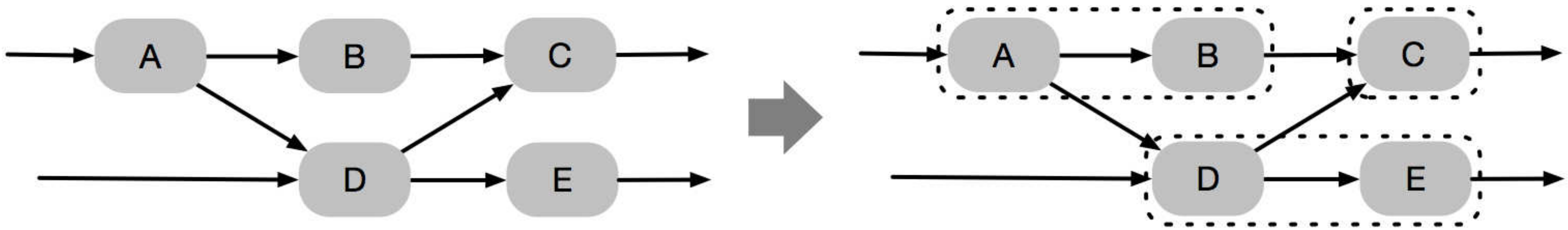
- Round-robin (no state)
- Hash by key (disjoint state)
- Duplicate

Dynamism

- Elastic operators (learn width)
- STM (resolve conflicts)

Placement

Place the logical graph onto physical machines and cores.



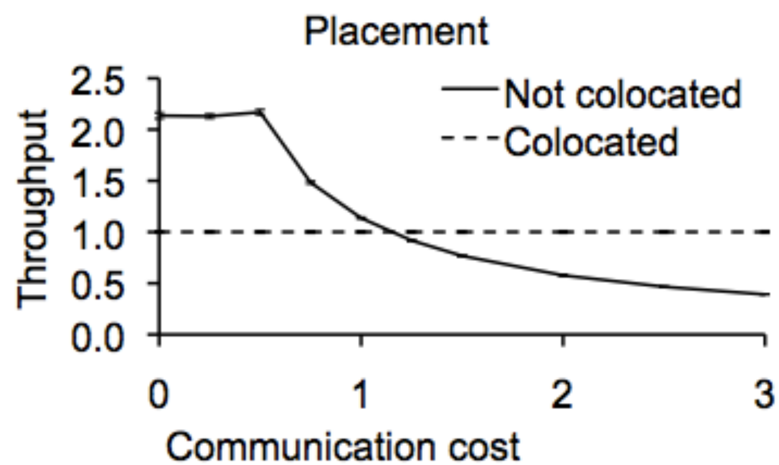
Safety

- Have right resources
- Have enough resources
- Obey license/security
- If dynamic, need migratability

Variations

- Based on host resources vs. network resources, or both
- Automatic vs. user-specified

Profitability

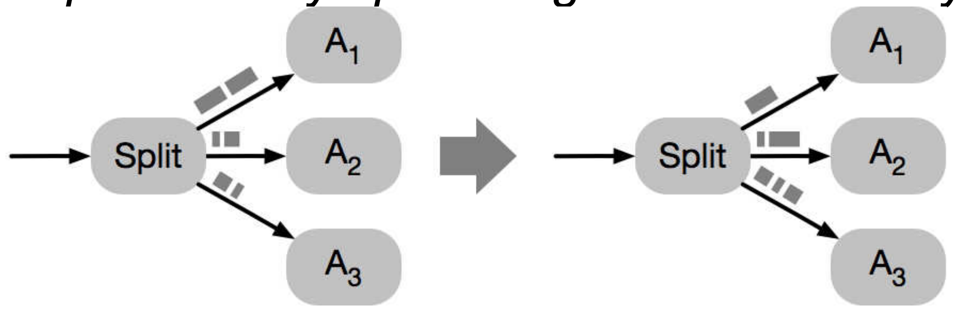


Dynamism

- Submission-time
- Online, via operator migration

Load Balancing

Avoid bottleneck operators by spreading the work evenly.



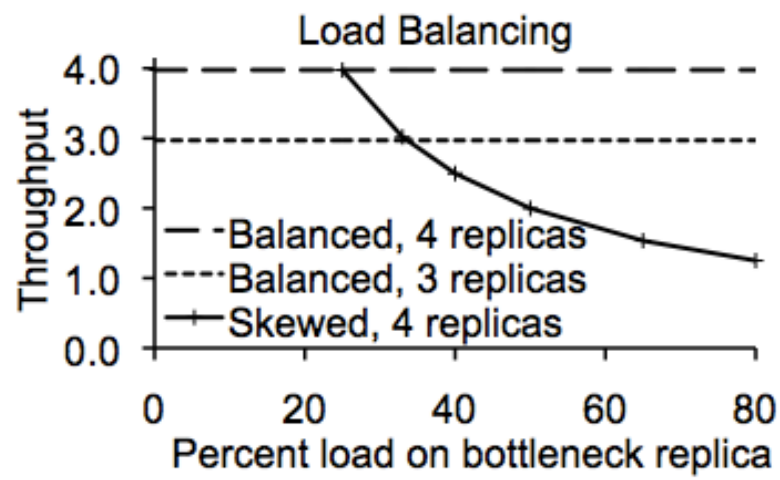
Safety

- Avoid starvation
- Ensure each worker is equally qualified
- Establish placement safety

Variations

- Balancing work while placing operators
- Balancing work by re-routing data

Profitability

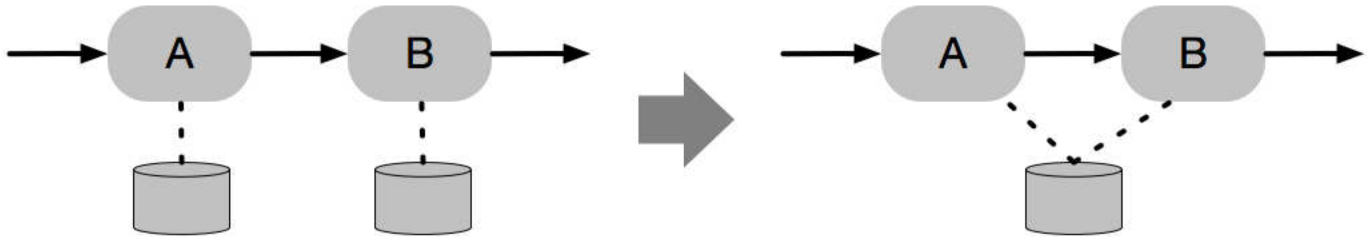


Dynamism

- Easier for routing than placement

State Sharing

Share identical data stored in multiple places in the graph.



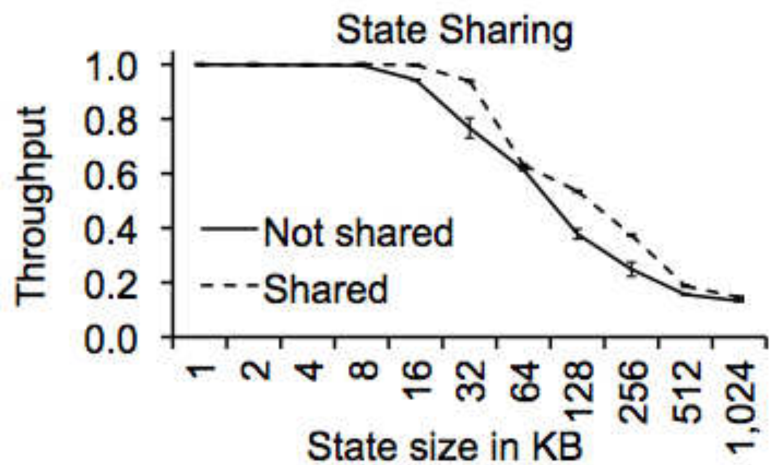
Safety

- Common access (usually: fusion)
- No race conditions
- No memory leaks

Variations

- Sharing queues
- Sharing windows
- Sharing operator state

Profitability



Dynamism

- N/A

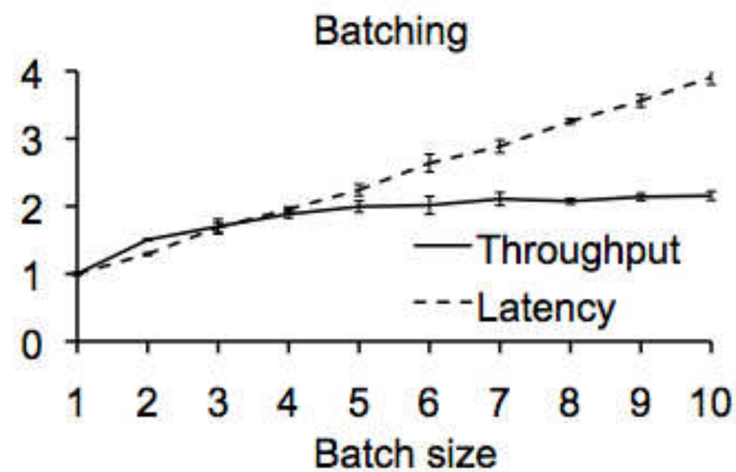
Batching

Communicate or compute over multiple data items as a unit.

Safety

- No deadlocks
- Satisfy deadlines

Profitability



Variations

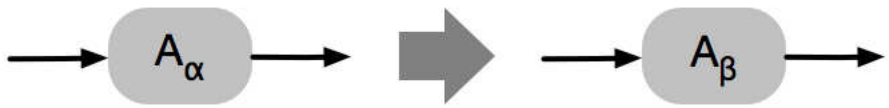
- Batching enables traditional compiler optimizations

Dynamism

- Batching controller
- Train scheduling

Algorithm Selection

Replace an operator by a different operator.



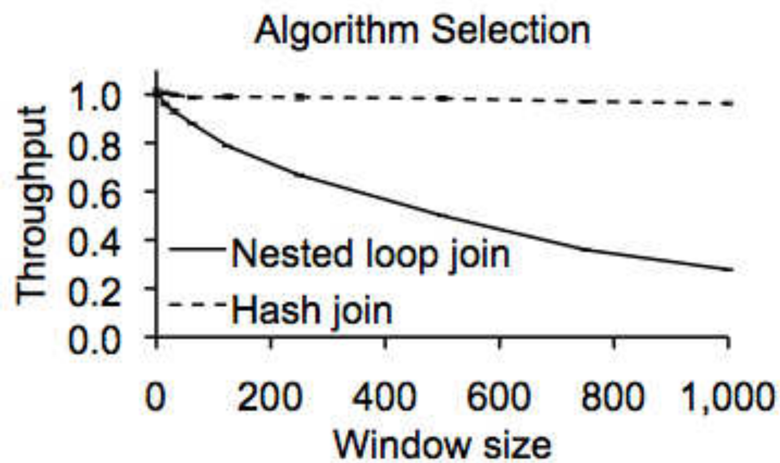
Safety

- $A_\alpha(s) \cong A_\beta(s)$
- May not need to be safe

Variations

- Algebraic
- Auto-tuners
- General vs. specialized

Profitability



Dynamism

- Compile both versions, then select via control port

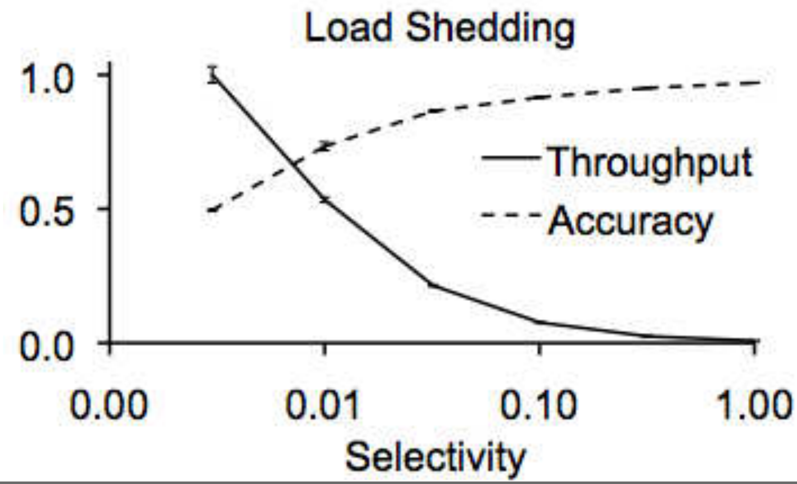
Load Shedding

Degrade gracefully during overload situations.

Safety

- By definition, not safe!
- QoS trade-off

Profitability



Variations

- Filtering data items (variations: where in graph)
- Algorithm selection

Dynamism

- Always dynamic

To Learn More

- DEBS’13 proceedings:
  “Tutorial: Stream Processing Optimizations”
- “A Catalog of Stream Processing Optimizations”, Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. IBM Research Report RC25215, 28 September 2011.
- “A Catalog of Stream Processing Optimizations”, Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. ACM Computing Surveys (CSUR). Conditionally accepted, minor revisions pending.

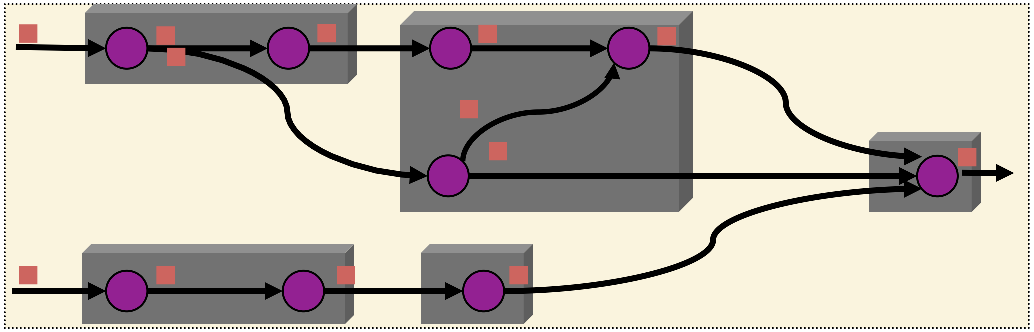
DEBS’ 13 Tutorial: Stream Processing Optimizations

Scott Schneider, Martin Hirzel, and Buğra Gedik
Acknowledgements: Robert Soulé, Robert Grimm, Kun-Lung Wu

Part 3: InfoSphere Streams Background

Streams Programming Model

- Streams applications are data flow graphs that consist of:
  - **Tuples**: structured data item
  - **Operators**: reusable stream analytics
  - **Streams**: series of tuples with a fixed type
  - **Processing Elements**: operator groups in execution



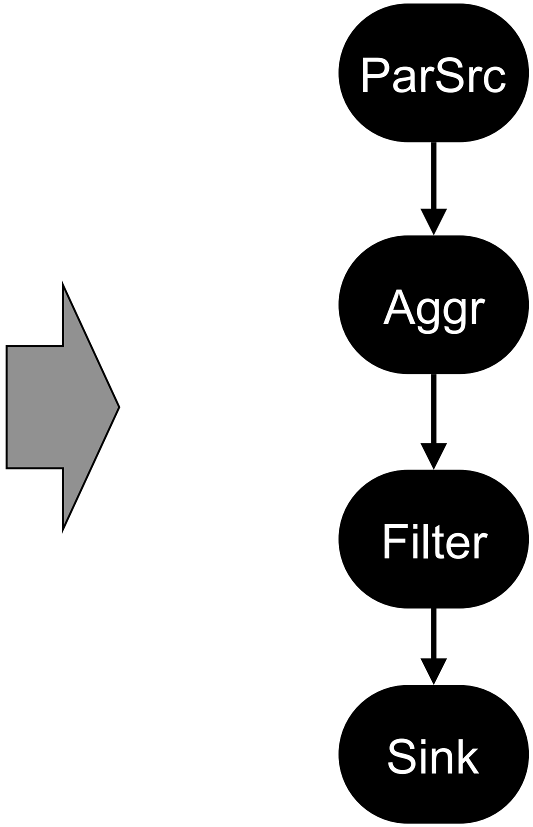
Streams Processing Language

```
composite Main {
type
  Entry = int32 uid, rstring server,
          rstring msg;
  Sum = uint32 uid, int32 total;
graph
  stream<Entry> Msgs = ParSource() {
    param servers: "logs.*.com";
          partitionBy: server;
  }

  stream<Sum> Sums = Aggregate(Msgs) {
    window Msgs: tumbling, time(5),
                 partitioned;
    param partitionBy: uid;
  }

  stream<Sum> Suspects = Filter(Sums) {
    param filter: total > 100;
  }

  () as Sink = FileSink(Suspects) {
    param file: "suspects.csv";
  }
}
```



SPL: Immutable by Default

```
stream<Data> Out = Custom(In) {
  logic state: int32 factor_ = 42;
  onTuple In: {
    submit({result=In.val*factor_}, Out);
  }
}
```

immutable by default

straight-forward to statically determine this is a stateless operator

```
stream<Data> Out = Custom(In) {
  logic state: mutable int32 count_ = 0;
  onTuple In: {
    ++count_;
    submit({count=count_}, Out);
  }
}
```

explicitly mutable

straight-forward to statically determine this is a statelful operator

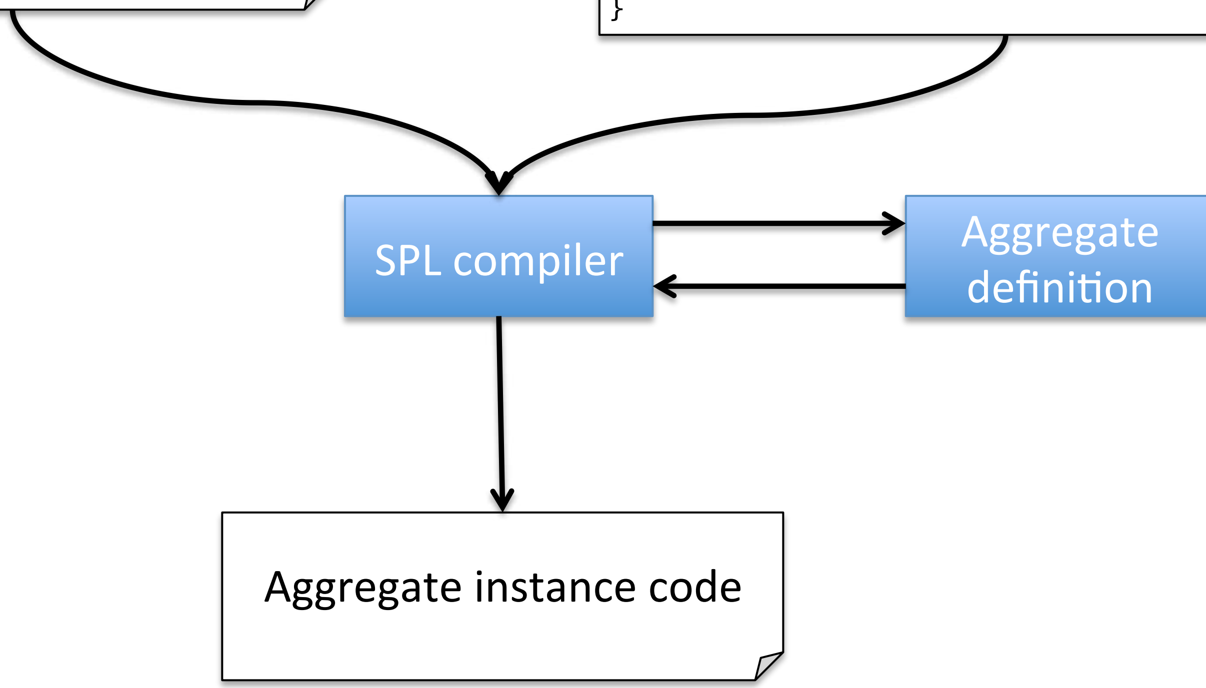
SPL: Generic Primitive Operators

an Aggregate invocation

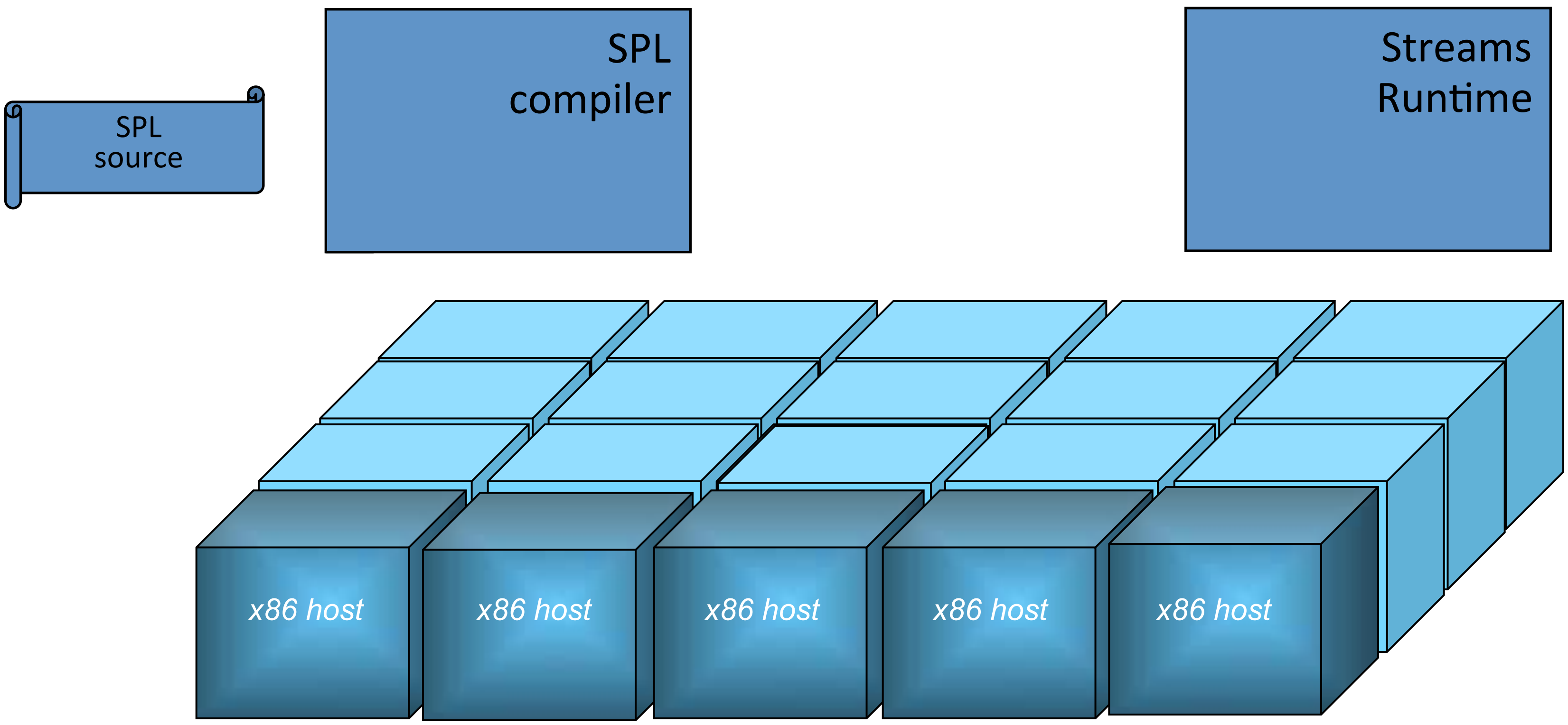
```
stream<Sum> Sums = Aggregate(Msgs) {
  window Msgs: tumbling, time(5),
               partitioned;
  param partitionBy: uid;
}
```

the Aggregate operator model

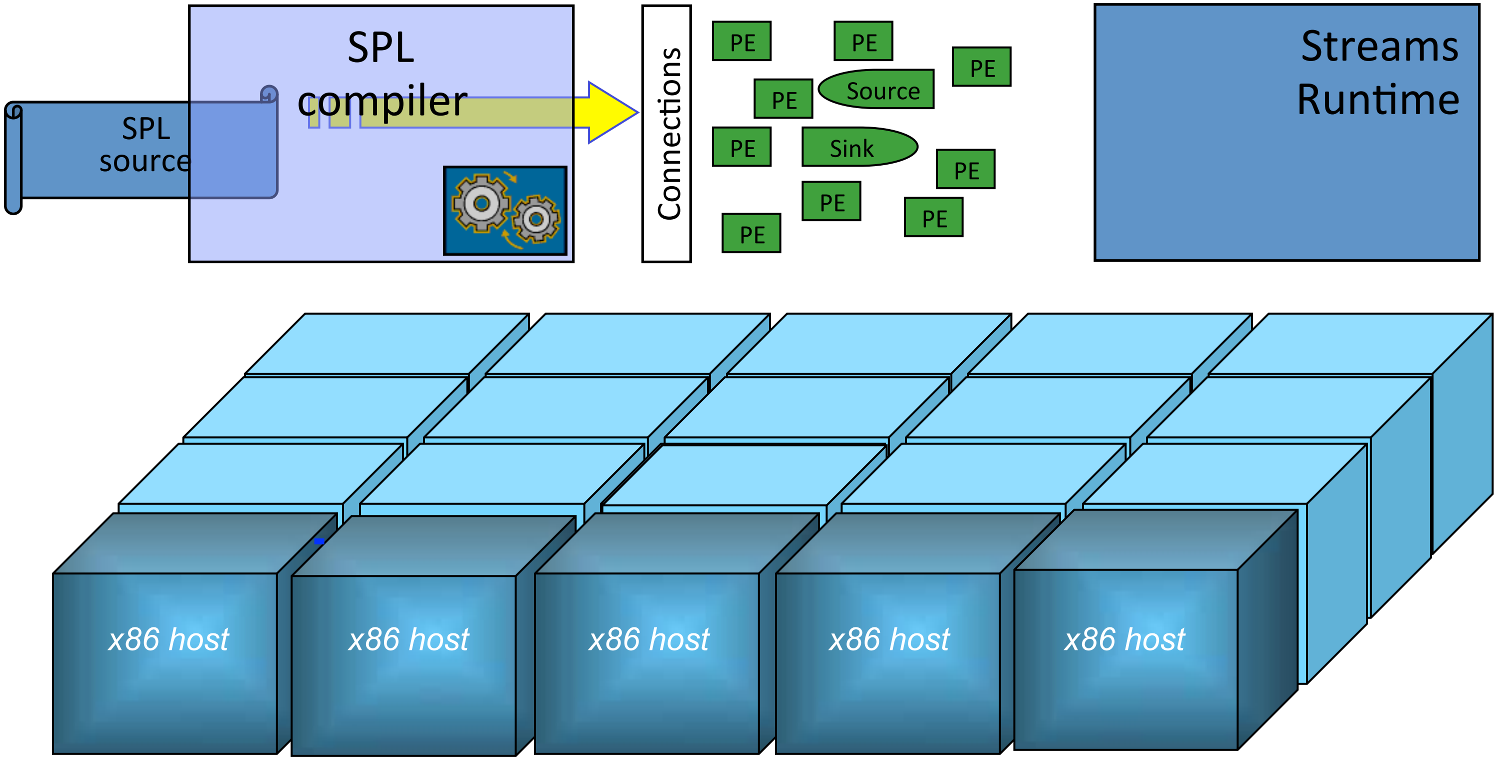
```
{Aggregate
  {parameters {groupBy optional Expression}
              {partitionBy optional Expression}}
  {inputPorts 1 required windowed}
  {outputPorts 1 required}
}
```



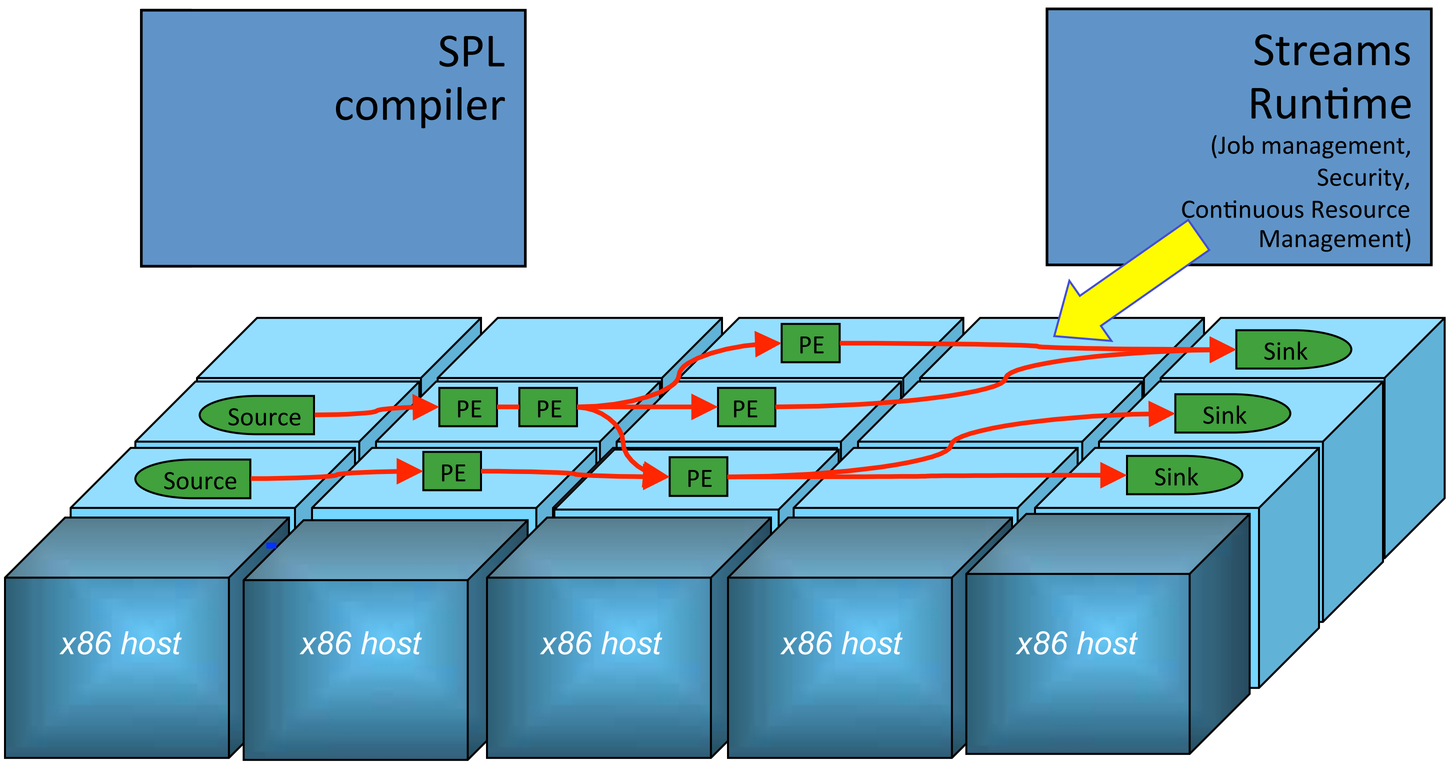
Source → Compilation → Execution



Source ➔ Compilation ➔ Execution



Source → Compilation → Execution



8

DEBS’ 13 Tutorial: Stream Processing Optimizations

Scott Schneider, Martin Hirzel, and Buğra Gedik
Acknowledgements: Robert Soulé, Robert Grimm, Kun-Lung Wu

Part 4: Fission Deep Dive

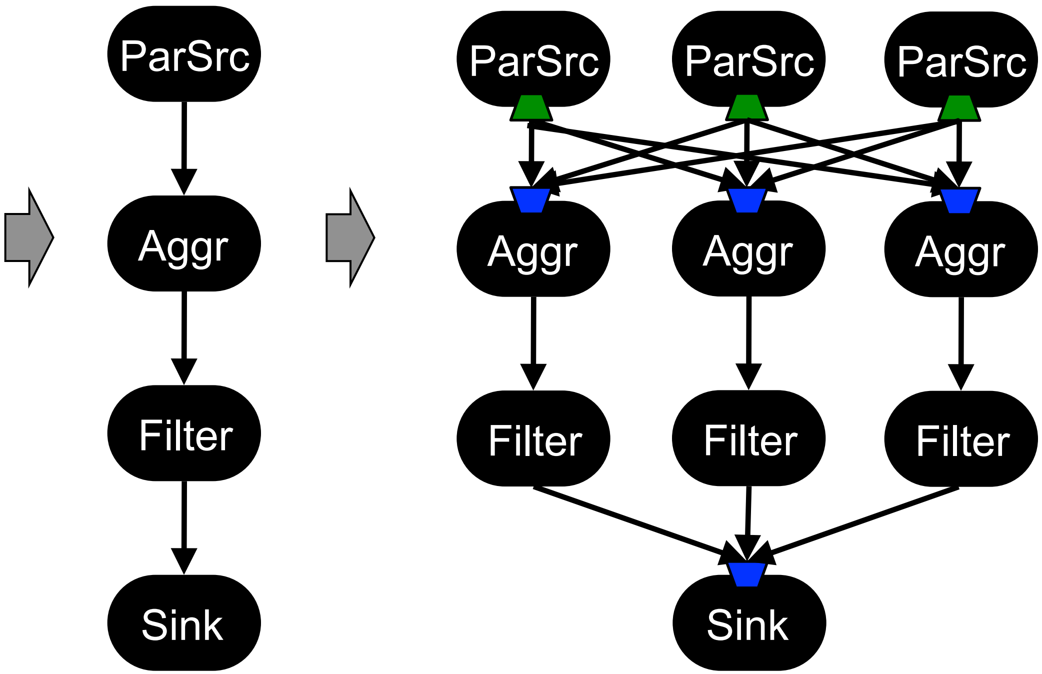
Fission Overview

```
composite Main {
type
  Entry = int32 uid, rstring server,
          rstring msg;
  Sum = uint32 uid, int32 total;
graph
  stream<Entry> Msgs = ParSource() {
    param servers: "logs.*.com";
          partitionBy: server;
  }

  stream<Sum> Sums = Aggregate(Msgs) {
    window Msgs: tumbling, time(5),
                 partitioned;
    param partitionBy: uid;
  }

  stream<Sum> Suspects = Filter(Sums) {
    param filter: total > 100;
  }

  () as Sink = FileSink(Suspects) {
    param file: "suspects.csv";
  }
}
```



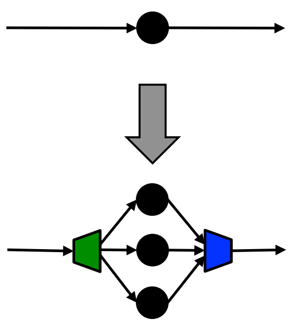
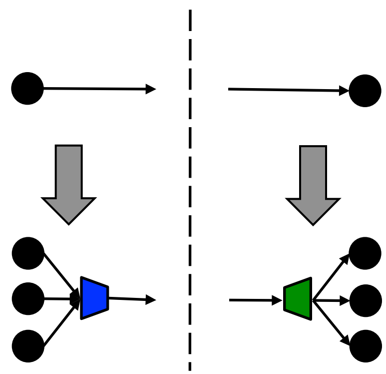
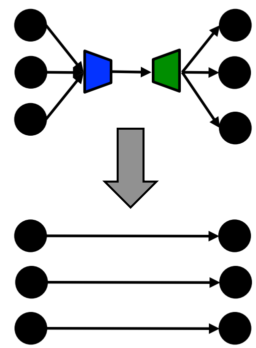
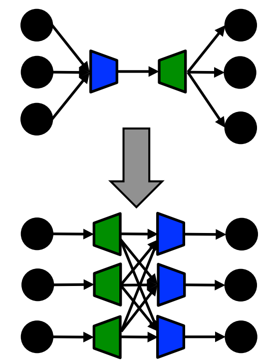
Technical Overview

Compiler:
- Apply parallel transformations
- Pick routing mechanism (e.g., hash by key)
- Pick ordering mechanism (e.g., seq. numbers)

ADL

Runtime:
- Replicate segment into channels
- Add split/merge/shuffle as needed
- Enforce ordering

Transformations

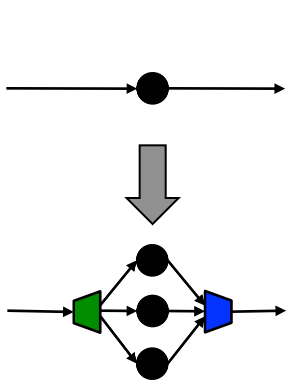
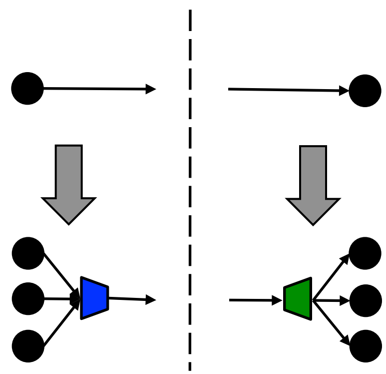
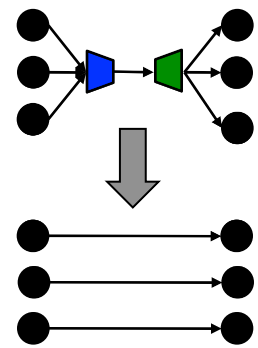
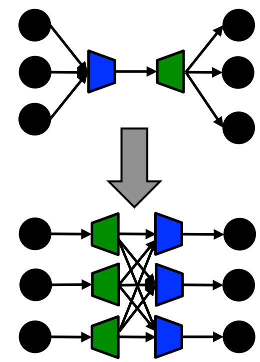
Parallelize non-source/sink	Parallelize sources and sinks	Combine parallel regions	Rotate merge and split
			
	Examples: •OPRA source •Database sink		Also known as "shuffle"

Do all of the above as much as possible,
wherever it is safe to do so.

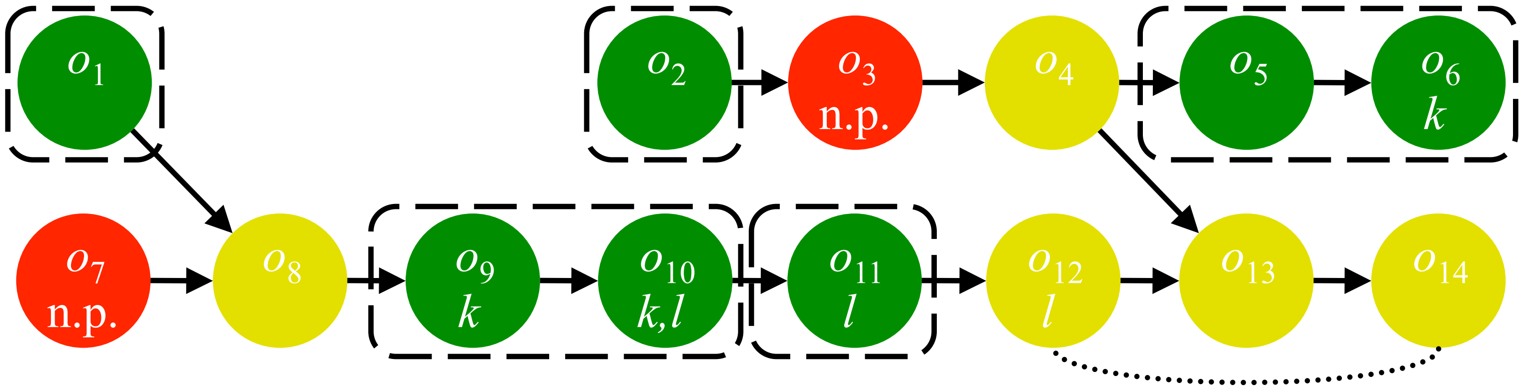
Core Challenges

- State
  - **Problem**: No shared memory between channels (partitioned local state)
  - **Solution**: Only parallelize if stateless or partitioned (i.e., separate state into channels by keys)
- Order
  - **Problem**: Race conditions between channels (independent threads of control)
  - **Solution**: Only parallelize if merge can guarantee same tuple order as without parallelization

Safety Conditions

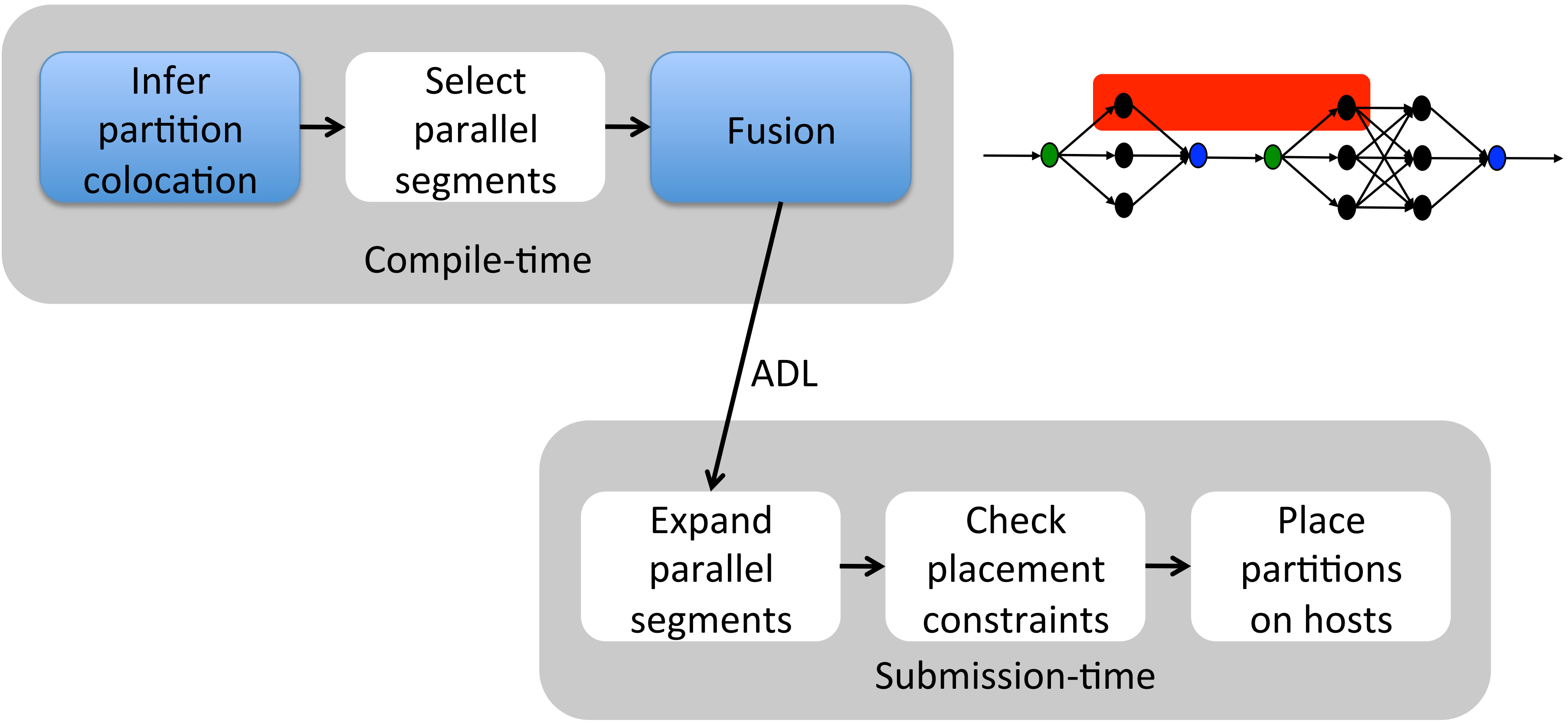
Parallelize non-source/sink	Parallelize sources and sinks	Combine parallel regions	Rotate merge and split
			
• stateless *or* partitioned state • simple chain	• stateless *or* partitioned state	• stateless *or* • compatible keys • forwarding	• incompatible keys • selectivity ≤ 1

Select Parallel Segments

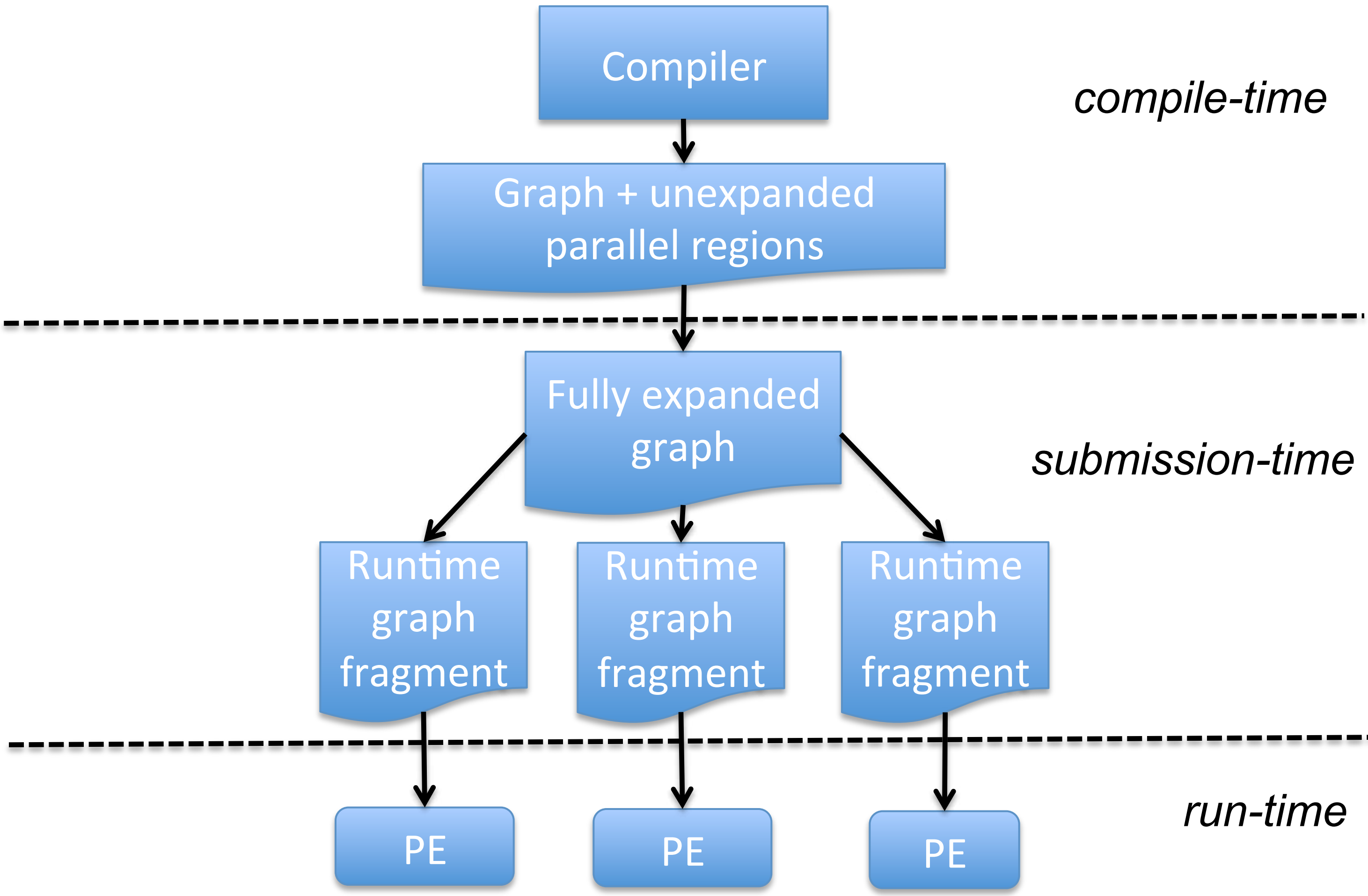


- Can't parallelize
  - Operators with >1 fan-in or fan-out
  - Punctuation dependecy later on
- Can't add operator to parallel segment if
  - Another operator in segment has co-location constraint
  - Keys don't match

Constraints & Fusion



Compiler *to* Runtime

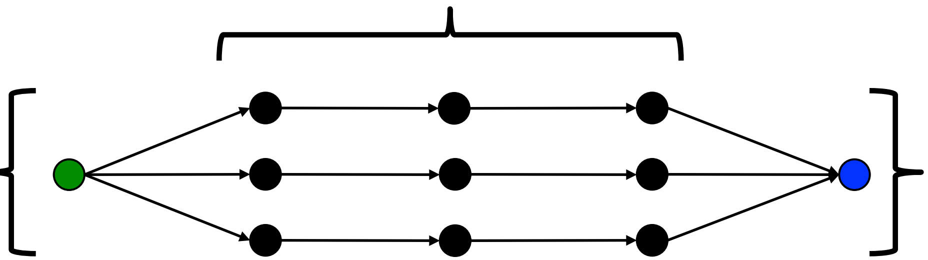


Runtime

	state	selectivity		
		gaps	dups	ratio
round-robin	✗	✗	✗	1 : 1
seqno	*partitioned*	✗	✗	1 : 1
strict seqno & pulse	*partitioned*	✓	✗	1 : [0,1]
relaxed seqno & pulse	*partitioned*	✓	✓	1 : [0,∞]

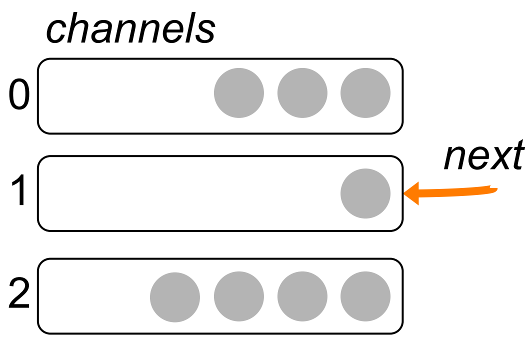
Operators in parallel segments:
- Forward seqno & pulse

Split:
- Insert seqno & pulse
- Routing

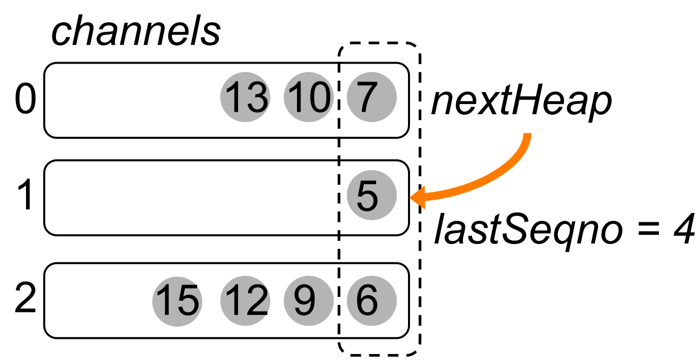


Merge:
- Apply ordering policy
- Remove seqno (if there) and drop pulse (if there)

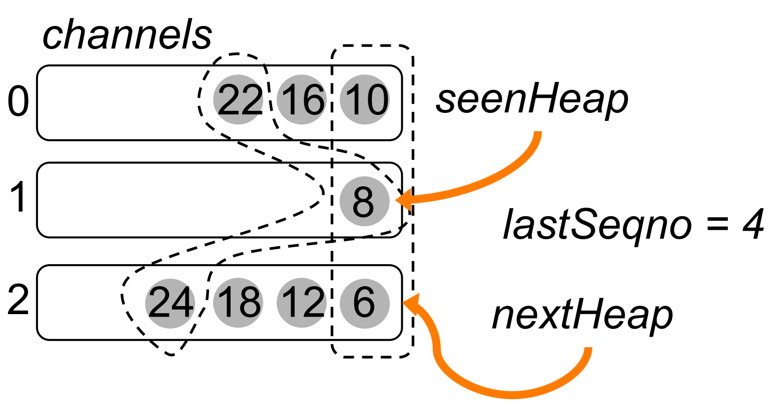
Merger Ordering



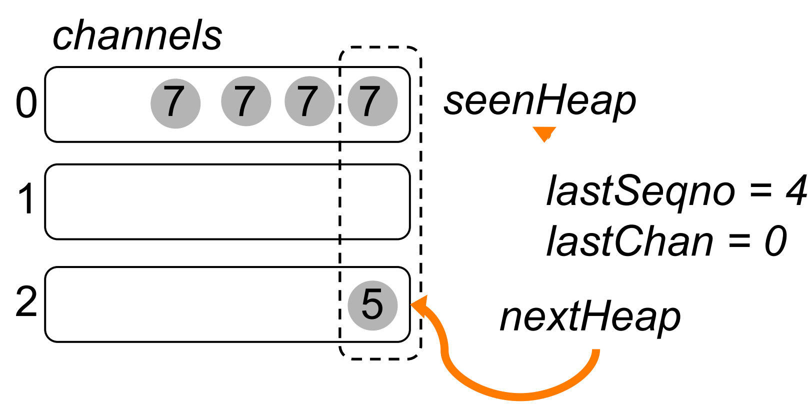
Round-Robin



Sequence Numbers

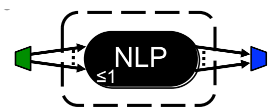


Strict Sequence Number & Pulses

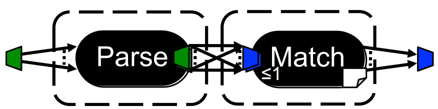


Relaxed Sequence Number & Pulses

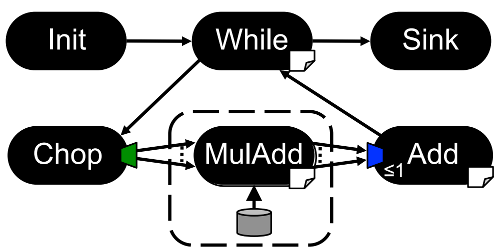
Application Kernel Performance



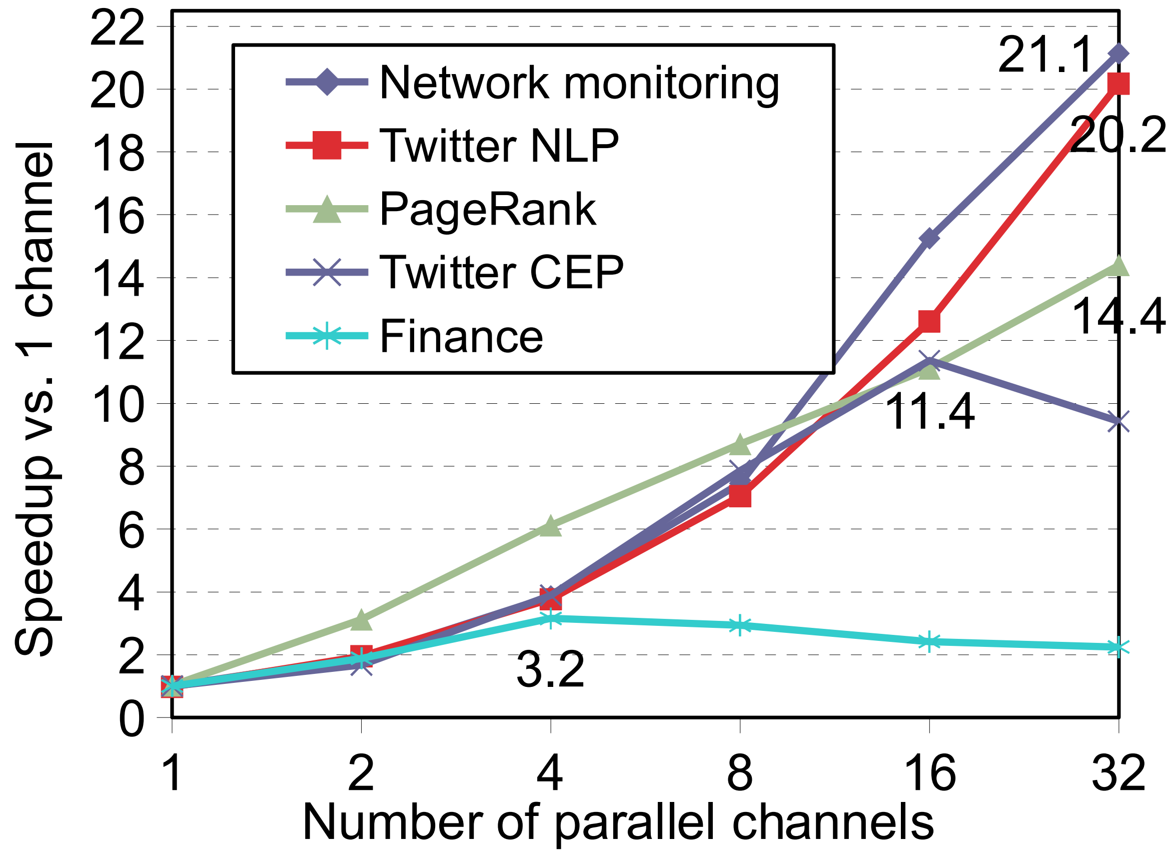
(c) Twitter NLP

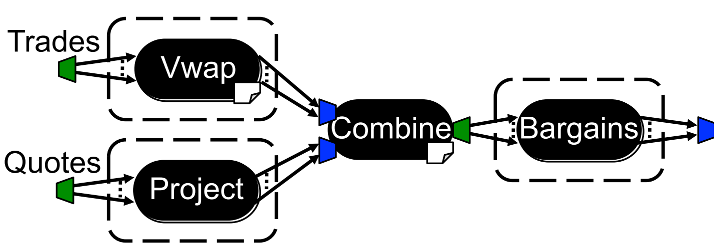


(d) Twitter CEP

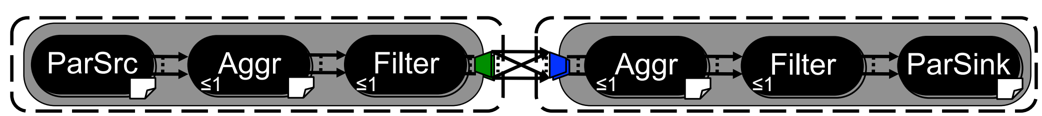


(b) PageRank



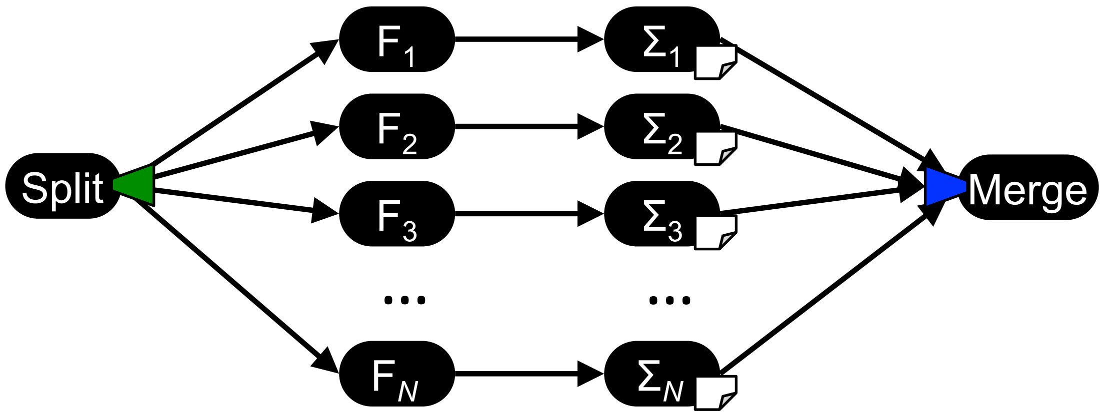


(e) Finance



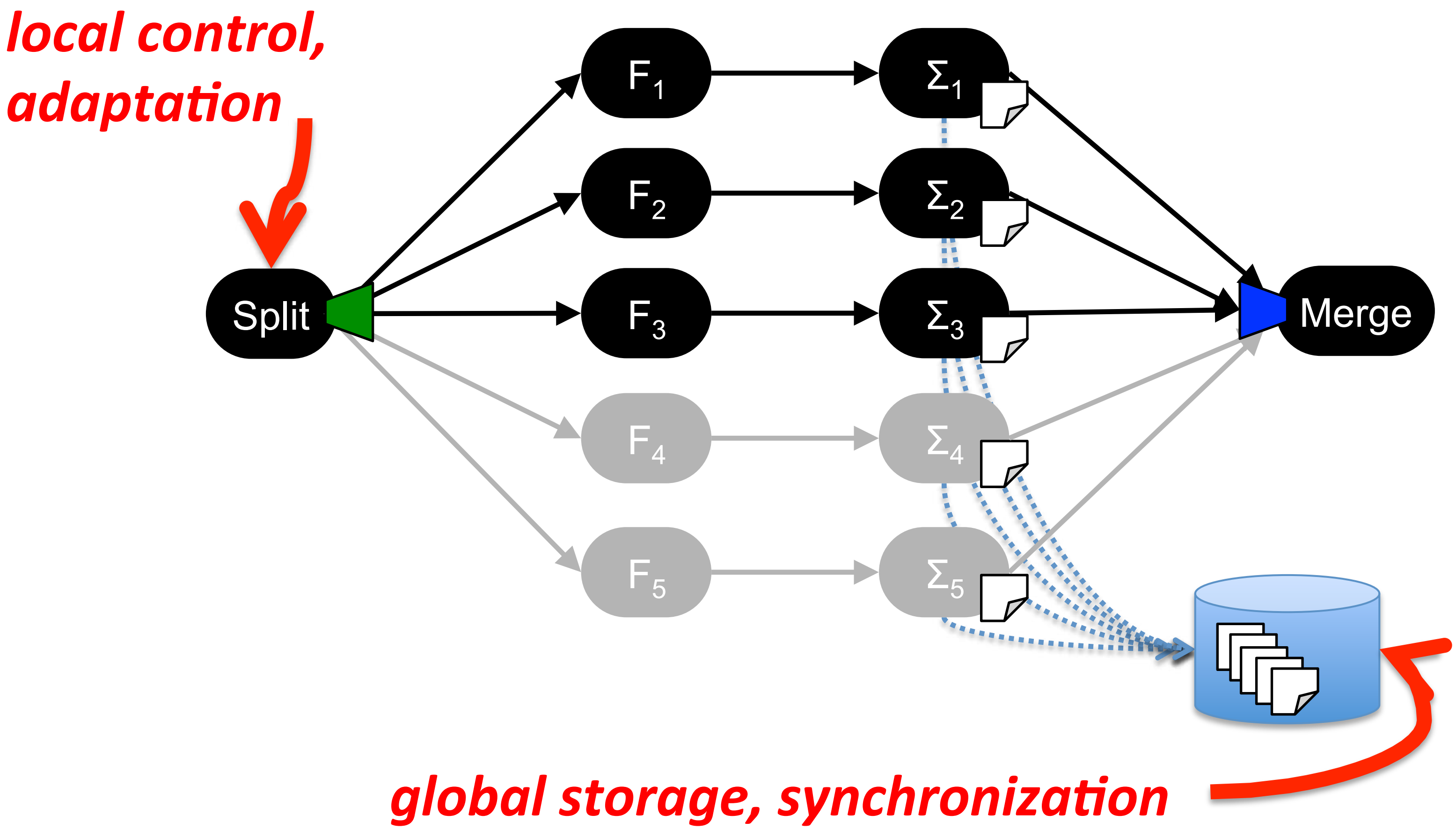
(a) Network monitoring

Elasticity: The Problem



- What is *N*? We want to:
  - find it dynamically, at runtime
  - automatically, with no user intervention
  - in the presence of stateless and partitioned stateful operators
  - maximize throughput

Elasticity: Solution Sketch



DEBS'13 Tutorial: Stream Processing Optimizations

Scott Schneider, Martin Hirzel, and Buğra Gedik
Acknowledgements: Robert Soulé, Robert Grimm, Kun-Lung Wu

Part 6: Open Research Questions

Programming Model Challenges

High-level
Easy to use
Optimizable

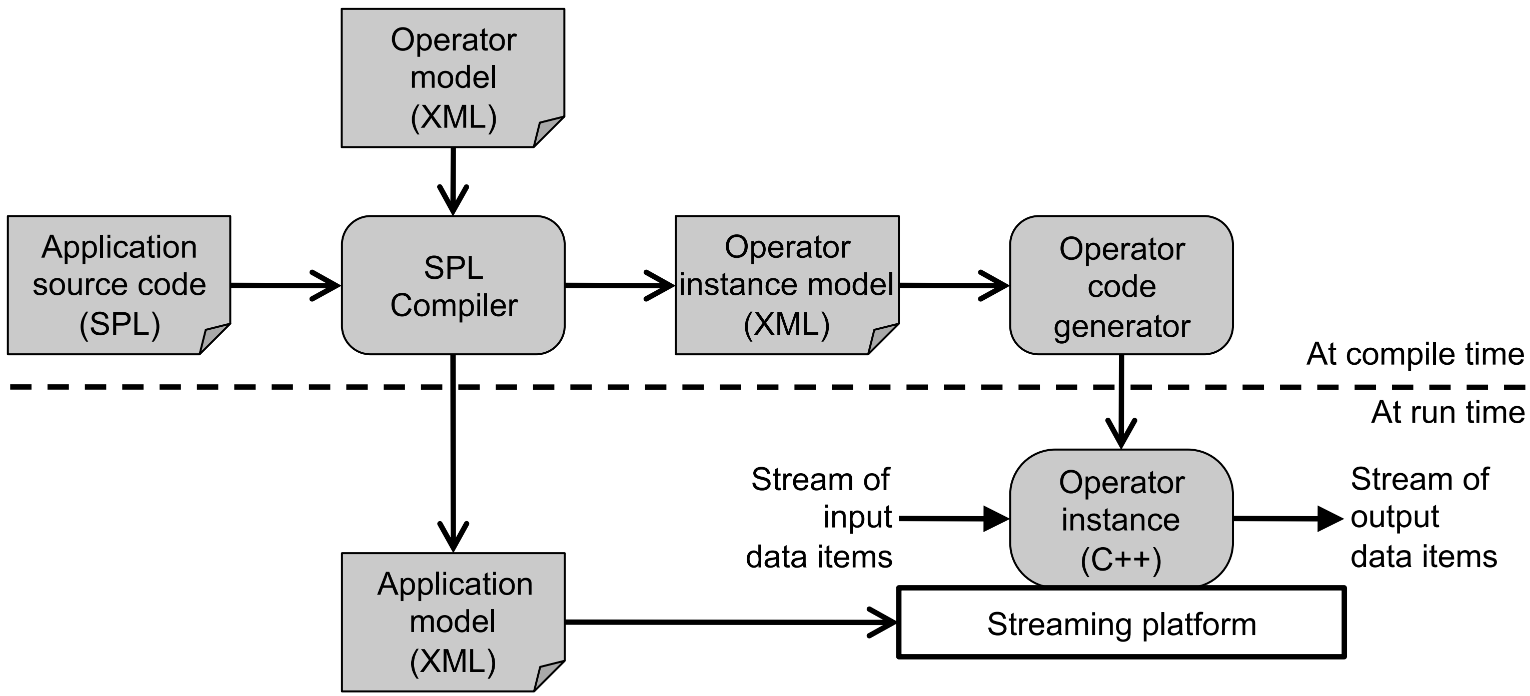
CEP patterns
StreamDatalog
StreamSQL
StreamIt (MIT)
Graph GUI
SPL
Java API
Annotated C
C/Fortran

Low-level
General
Predictable

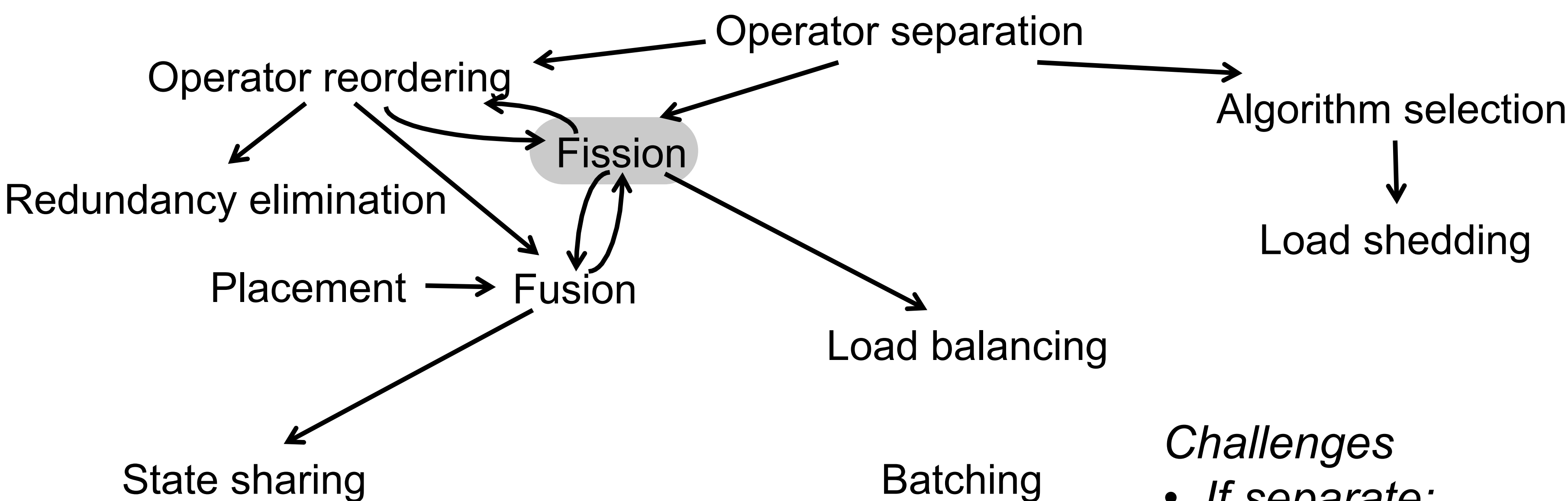
Other challenges

- *Foreign code*
- *Familiarity*

Interaction of SPL and C++



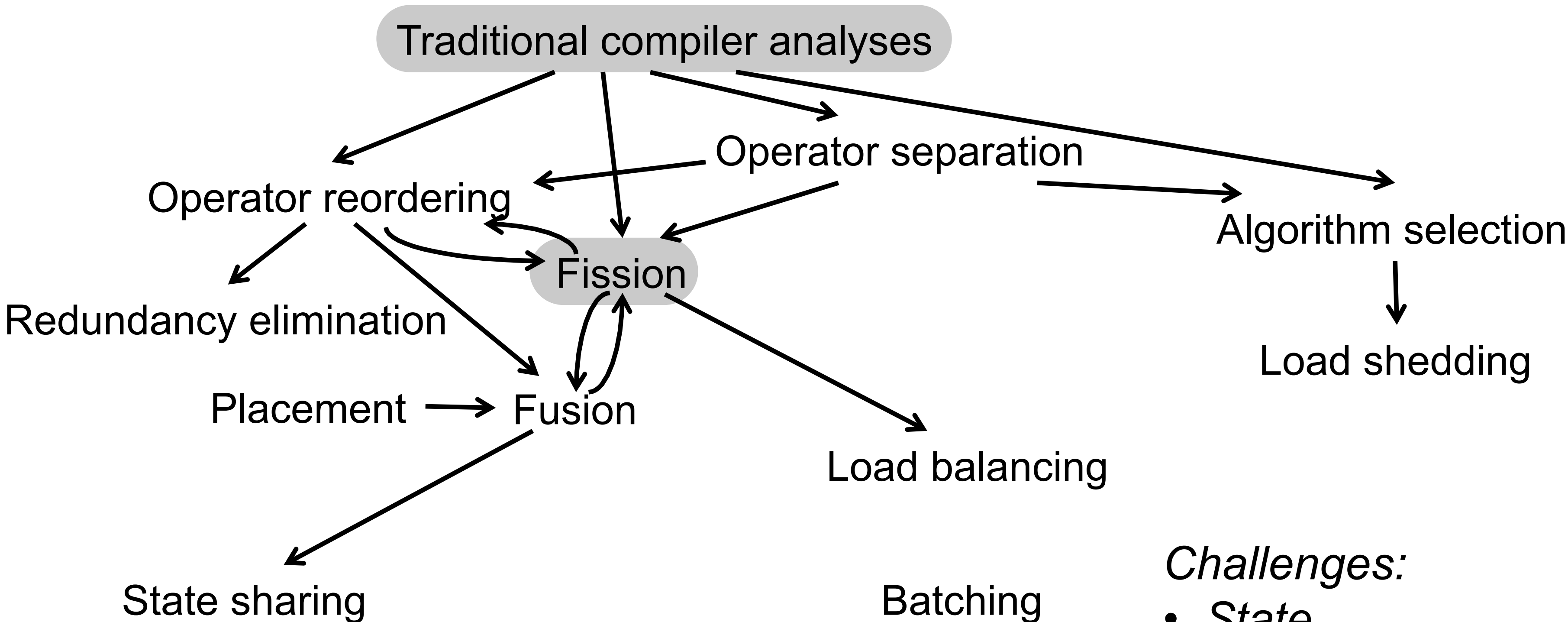
Optimization Combination



Challenges

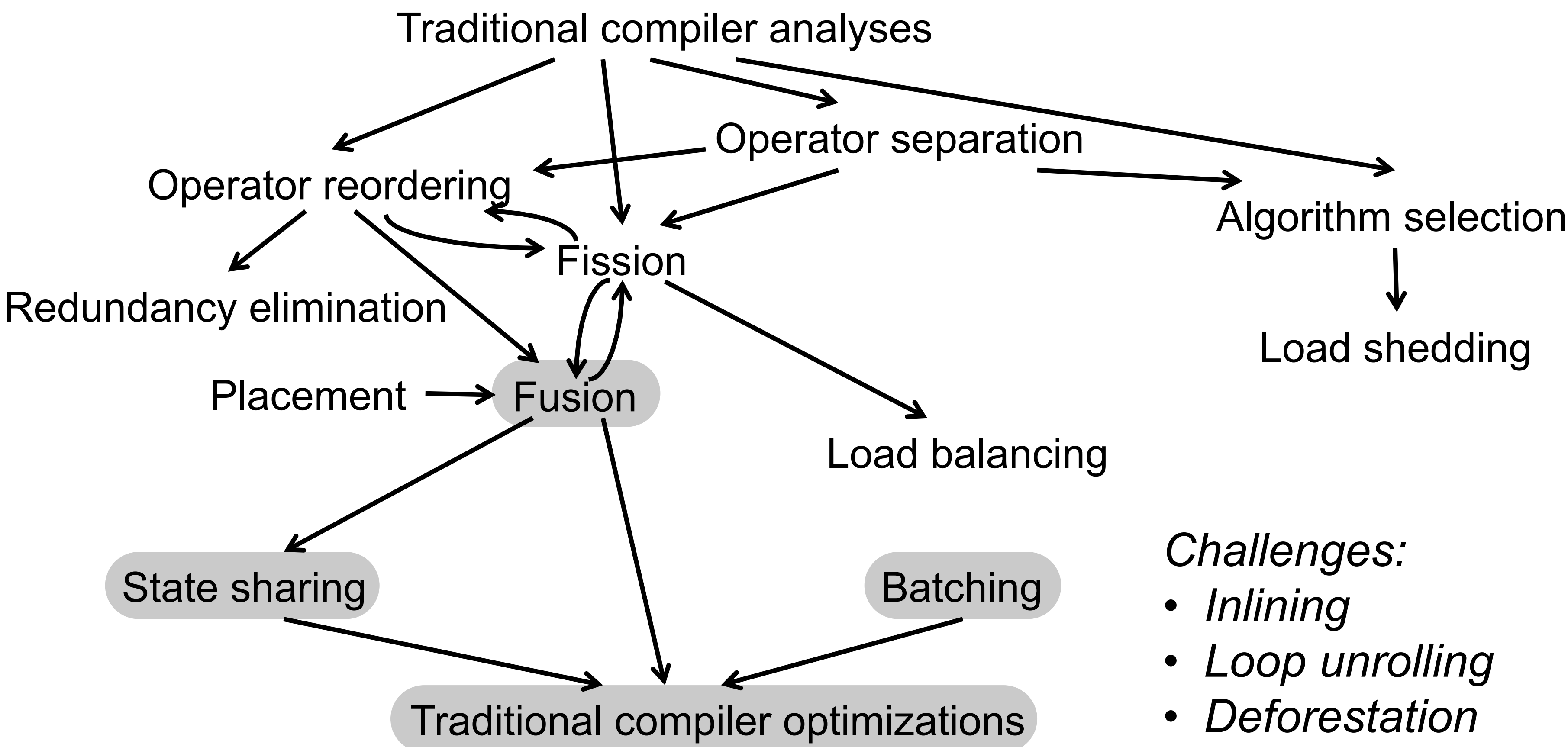
- *If separate: order*
- *If combined: profitability model*

Interaction with Traditional Compiler Analysis



Challenges:
- *State*
- *Ordering*
- *Selectivity*
- *Key forwarding*

Interaction with Traditional Compiler Optimizations



Challenges:
- *Inlining*
- *Loop unrolling*
- *Deforestation*
- *Scalarization*

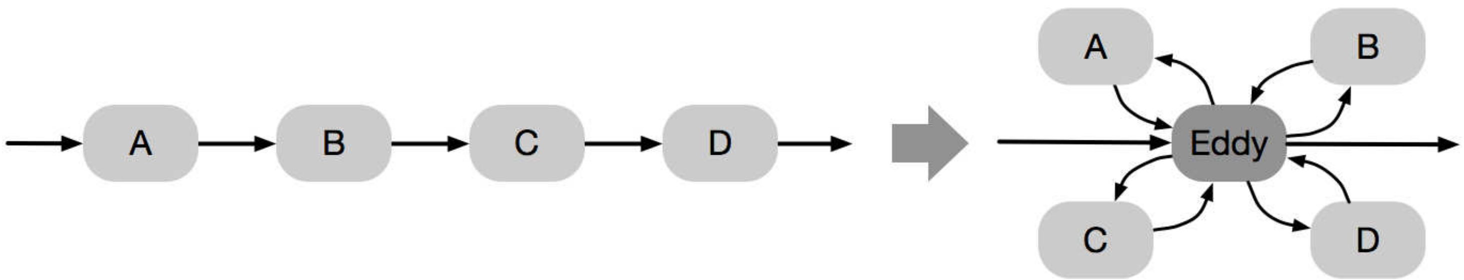
Dynamic Optimization

Compile time	*Submission time*	*Runtime discrete*	*Runtime continuous*
Operator separation	Redundancy elimination	Load balancing	Operator reordering
Fusion	Fission		Batching
State sharing	Placement		Load shedding
Algorithm selection			

Other challenges:

- *Settling*
- *Accuracy*
- *Stability*
- *Overshoot*

Dynamic Operator Reordering



Approach: Emulate graph change via data-item routing.
Example: Eddies [Avnur, Hellerstein SIGMOD'00]

Benchmarks

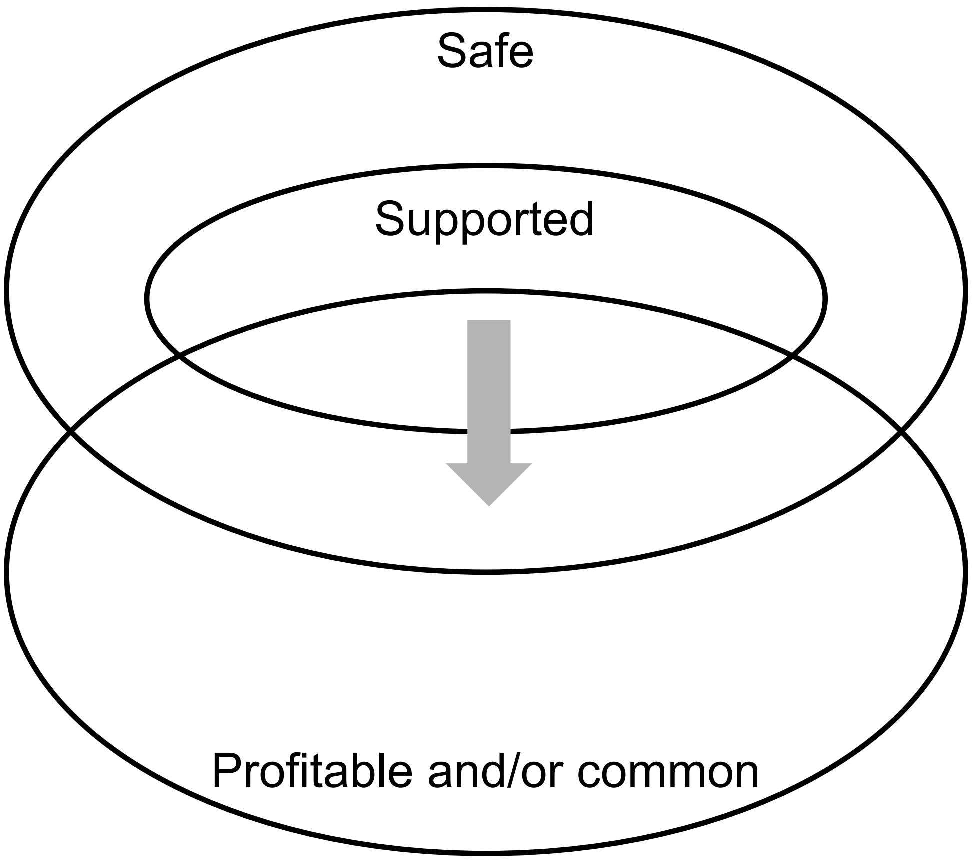
Wish List

- Representative
  - … of real code
  - … of real inputs
- Fast enough to conduct many experiments
- Fully automated / scripted
- Self-validating
- Open-source with industry-friendly license

Literature

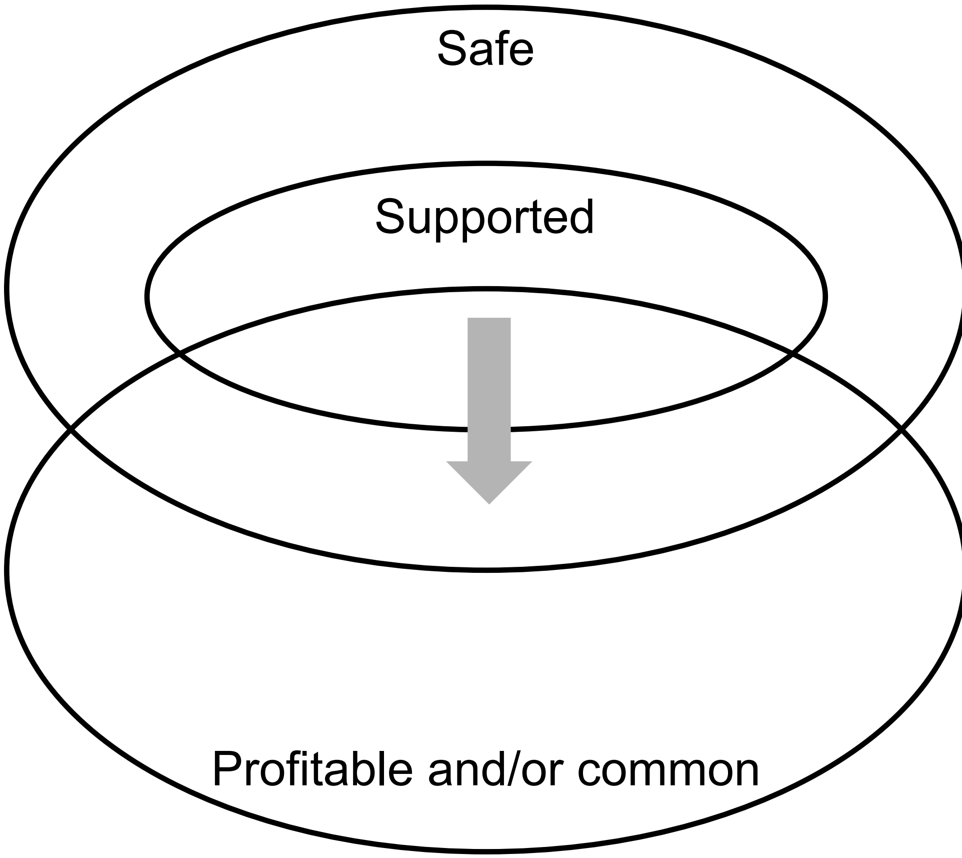
- LinearRoad [Arasu et al. VLDB'04]
- BiCEP [Mendes, Bizarro, Marques TPC TC'09]
- StreamIt [Thies, Amarasinghe PACT'10]

Generality of Optimizations



Challenges
- *Expand "Supported"*
- *In the right direction*

Generality of Fission



State	*Ordering*	*Topology*	*User code*
Stateless	Static selectivity	Single operator	Built-in operators
Partitioned stateful		Simple pipeline	Streaming language
Arbitrary stateful	Dynamic selectivity	Arbitrary subgraph	Foreign language

Challenges

- *Expand "Supported"*
- *In the right direction*