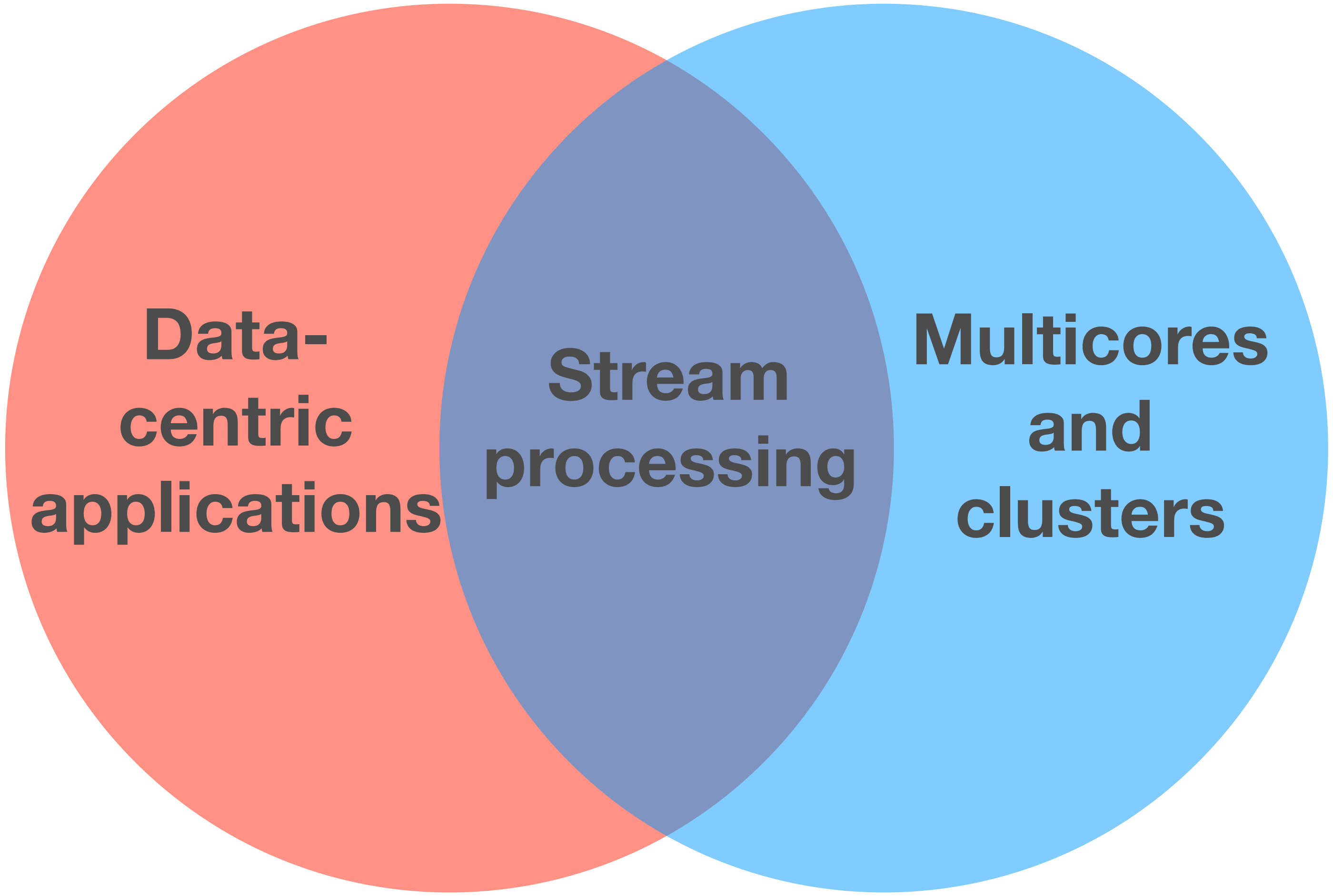
Reusable Software Infrastructure for Stream Processing

Robert Soulé
New York University
Thesis Defense

Stream Processing Is Everywhere

- Netflix accounts for ~30% of downstream internet traffic.
- Algorithmic trading accounts for 50-60% of all trades in the U.S.
- A streaming application can predict the onset of sepsis in premature babies 24 hours sooner than experienced ICU nurses.

At the Intersection of Two Trends



Languages and optimizations need to adapt

Streaming Languages and Optimizations

Streaming Languages	Streaming Optimizations
CQL, StreamIt, Sawzall, Hancock, Gigascope, Lime, etc.	Fusion, fission, placement, reordering, etc.
Represent an application as a graph of streams and operators	Maximize utilization of available resources
Tailored to the needs of a particular application domain	Often re-write the data-flow graph

Stream Processing Needs Infrastructure

- Benefits of a *intermediate language (IL)* are well known
  - Increase portability
  - Share optimizations
- Streaming needs its own intermediate language
  - Need to reason across machines
  - Support different optimizations

Hypothesis

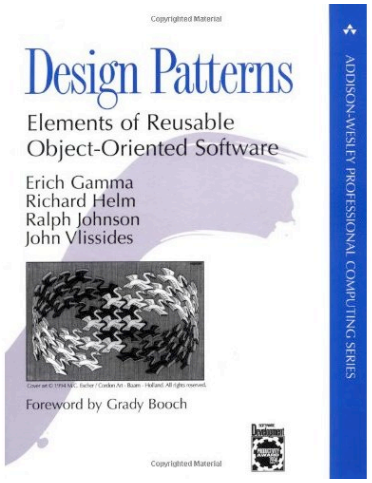
An intermediate language designed to meet the requirements of stream processing can serve as a common substrate for optimizations; assure implementation correctness; and reduce overall implementation effort.

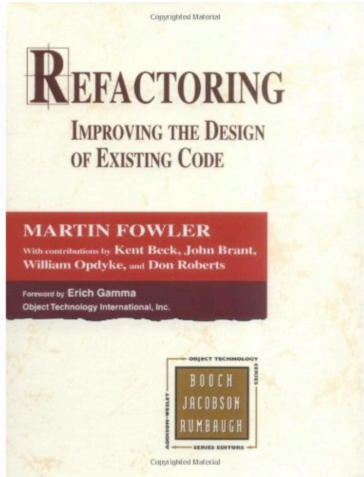
Thesis Components

- A **catalog of streaming optimizations** identifies the requirements for a streaming IL
- A **minimal calculus** provides a general, formal semantics and enables reasoning about correctness
- An **intermediate language** provides a practical realization of the calculus

Optimizations Catalog

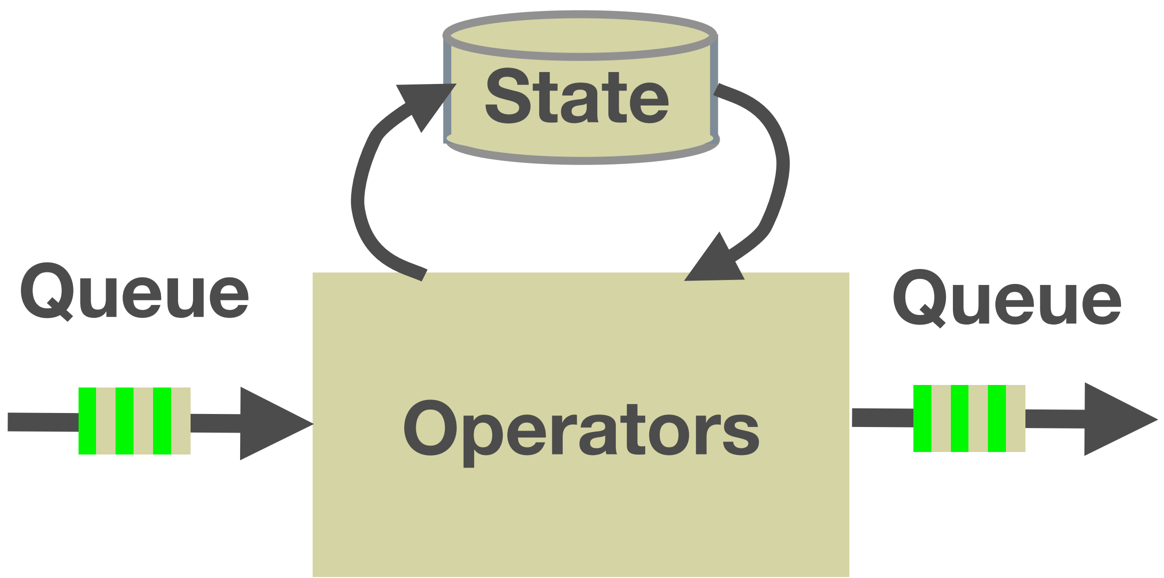
A catalog, but organized as a reference.





- Resolves conflicting terminology (e.g. kernel = operator = box)
- Makes assumptions explicit (e.g. stream graph is a forrest)
- Identifies the requirements for implementing optimizations

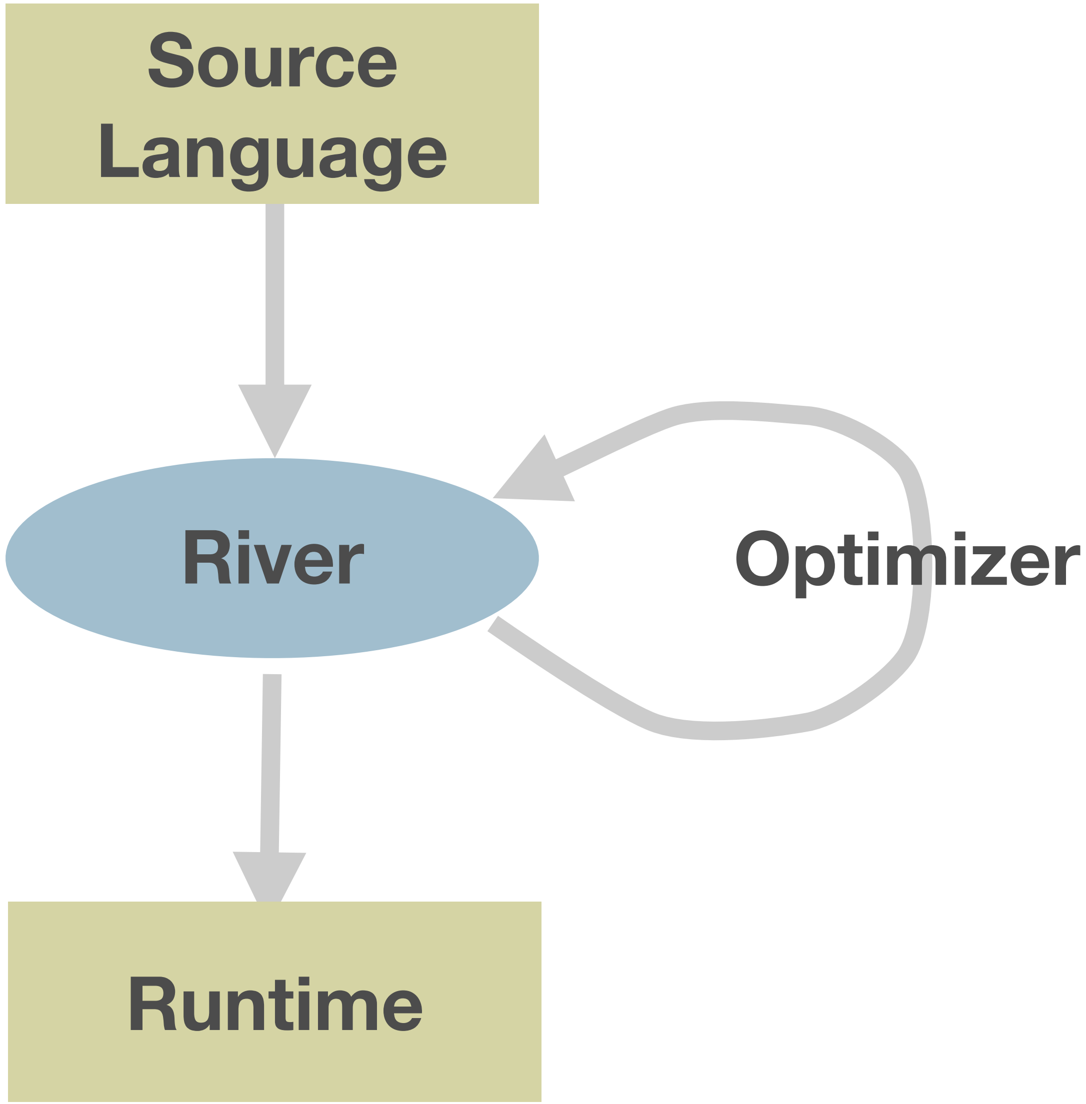
Brooklet Calculus



- Names operators and queues: fundamental components
- Explicit state and communication: need machinery
- Non-deterministic execution: reality of distributed systems
- Establishes a formal foundation for an IL

River IL

- Decouples front-ends from optimizations: portability and reuse
- Concretizes Brooklet: operator implementations, concurrent execution, back-pressure
- Modular parsers, type-checkers, code generators
- Practical IL for streaming with a formal semantics



Evaluation

Condition	Experiment
Meets the requirements of stream processing	Front-ends for CQL, StreamIt, Sawzall and benchmark applications
Serves as a common substrate for optimization	Operator fusion, fission, and placement optimizations
Assures implementation correctness	Formal translations of three languages, Safety proofs for three optimizations
Reduces overall implementation effort	Language agnostic optimizations applied to benchmarks illustrates reuse

Contributions

- A systematic exploration of the requirements for a streaming IL
- A formal foundation for the design of an IL
- An IL with a rigorously defined semantics that decouples front-ends from optimizations
- The first formal semantics for Sawzall
- The first distributed implementation of CQL

Outline of This Talk

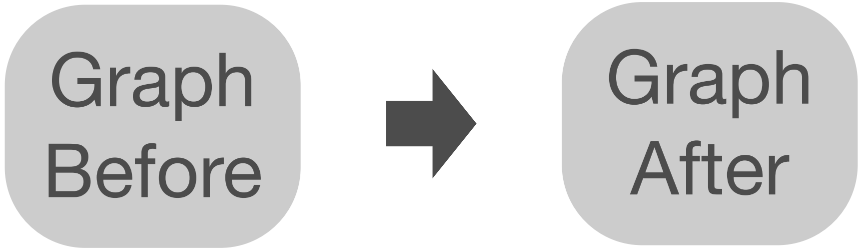
- A Catalog of Streaming Optimizations
- The Brooklet Core Calculus
- River: From a Calculus to an Execution Environment
- Related Work
- Outlook and Conclusions

Optimizations Catalog

Identifying the Requirements for a Streaming IL

Optimization Name

Key Idea



Items highlighted in red will be addressed in this talk

Safety

- Preconditions for correctness

Profitability

Variations

- Most influential published papers

Dynamism

- How to optimize at runtime

List of Optimizations

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

Operator Reordering

Move more selective operators upstream to filter data early.

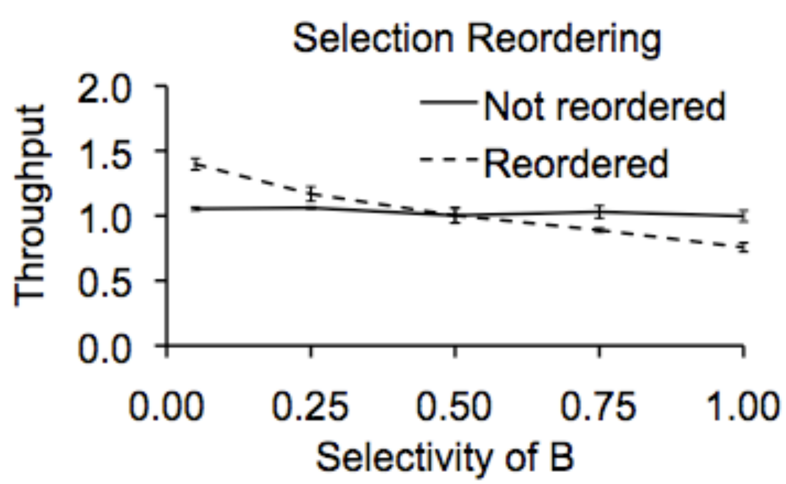
Safety

- Commutative
- Attributes available

Variations

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

Profitability

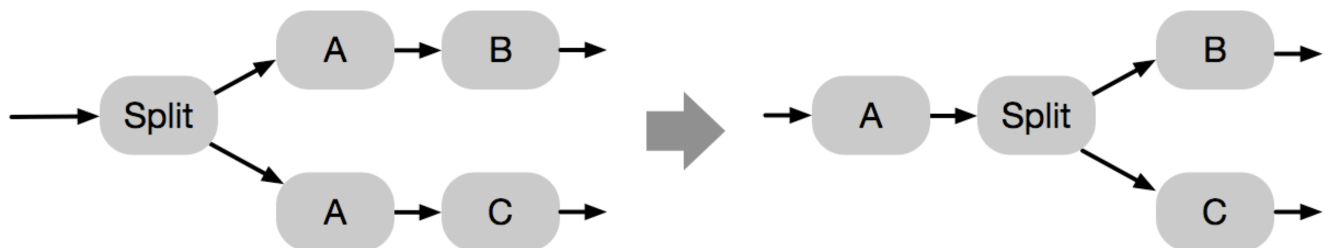


Dynamism

- **Eddy**

Redundancy Elimination

Combine or remove redundant operators.



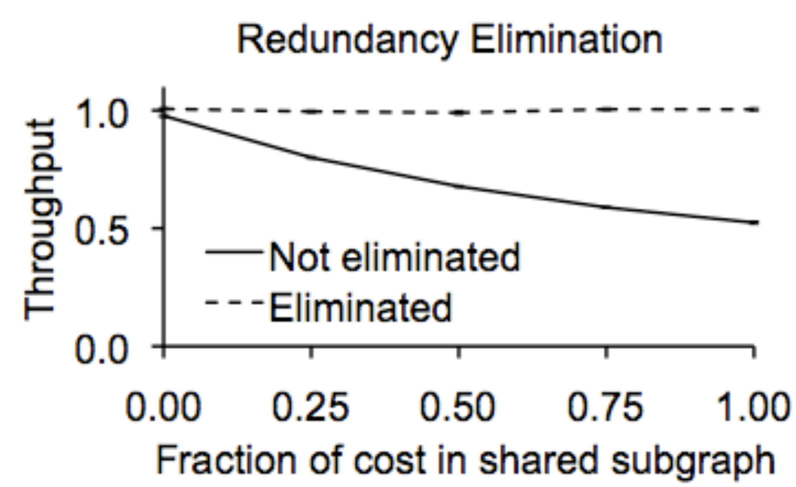
Safety

- Same algorithm
- Data available

Variations

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

Profitability

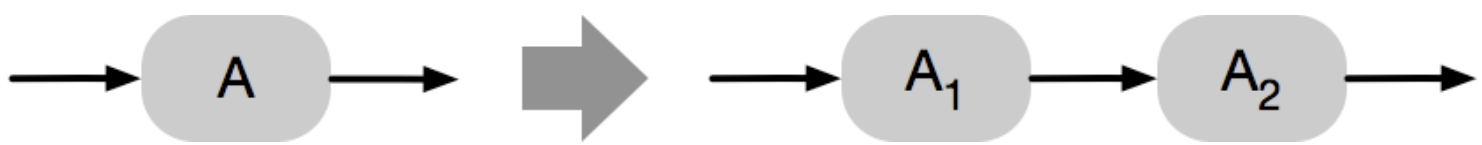


Dynamism

- In many-query case: share at submission time

Operator Separation

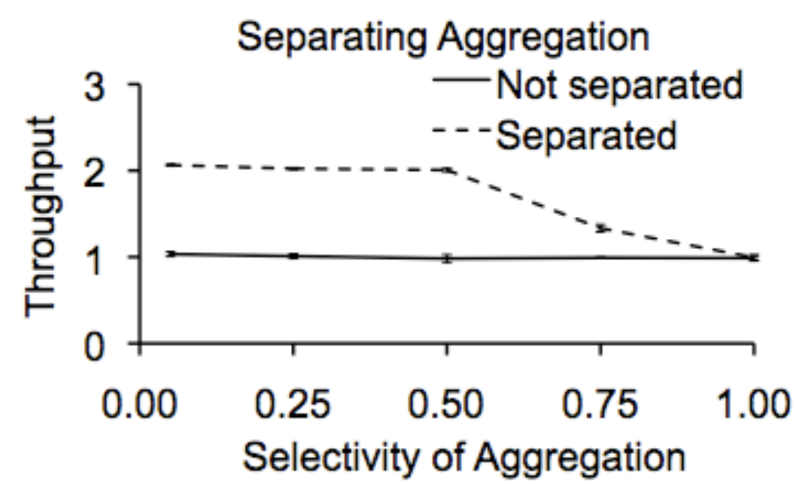
Break coarse-grained operators into finer steps.



Safety

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

Profitability



Variations

- Algebraic
- Using special API
- Dependency analysis
- Enable Reordering

Dynamism

- N/A

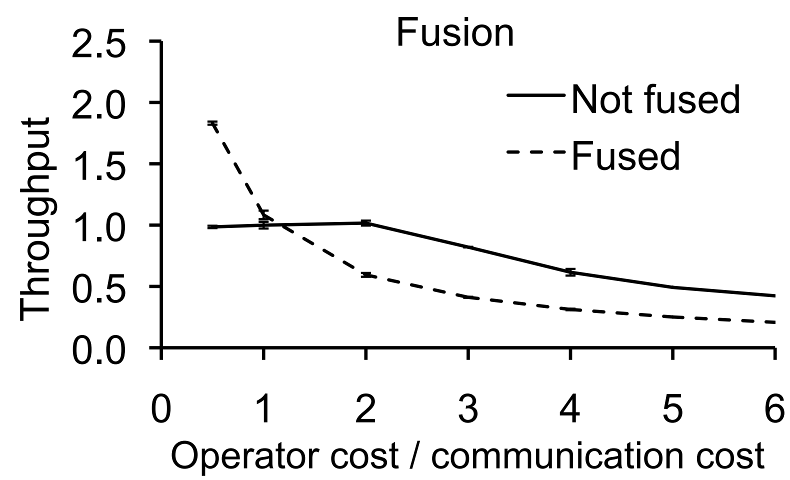
Fusion

Avoid the overhead of data serialization and transport.

Safety

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

Profitability



Variations

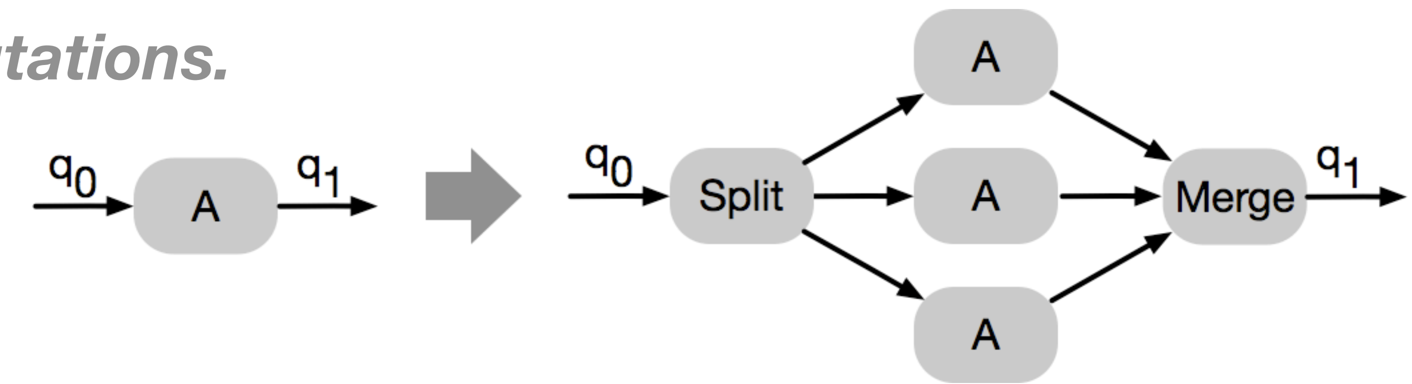
- Single vs. multiple threads
- Fusion enables traditional compiler optimizations

Dynamism

- Online recompilation
- Transport operators

Fission

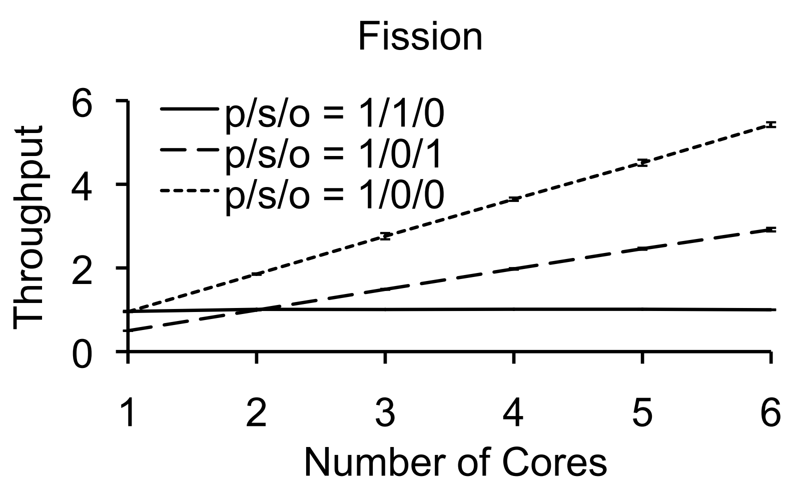
Parallelize computations.



Safety

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

Profitability



Variations

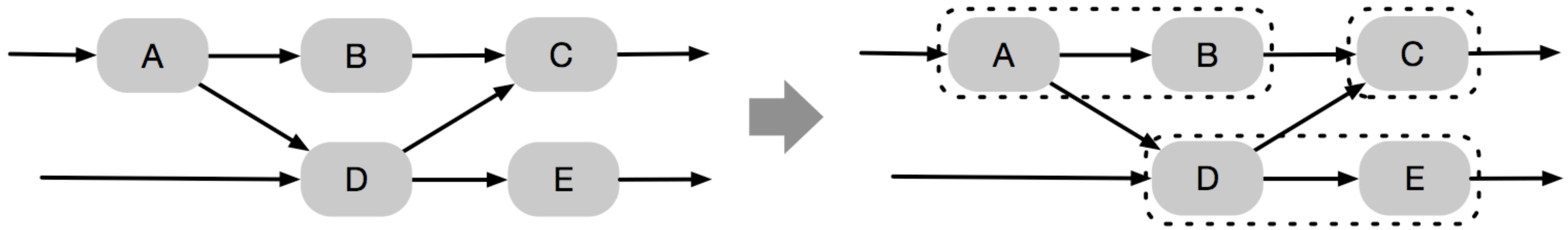
- Round-robin (no state)
- Hash by key (disjoint state)
- Duplicate

Dynamism

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

Placement

Assign operators to hosts and cores.



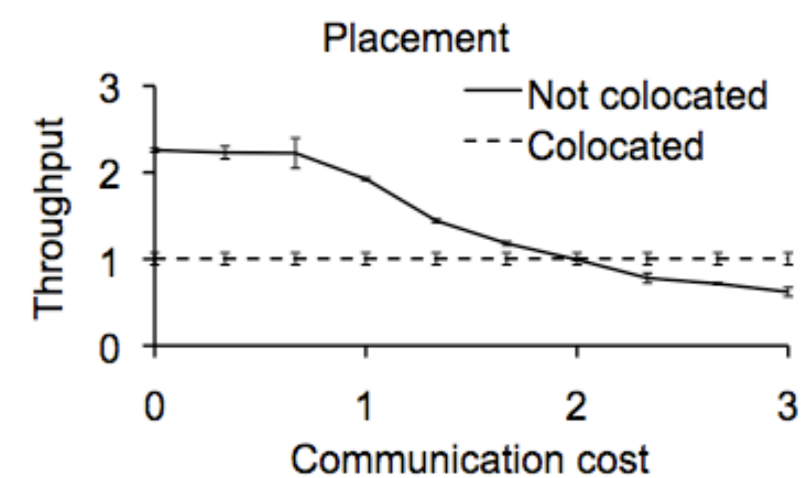
Safety

- 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

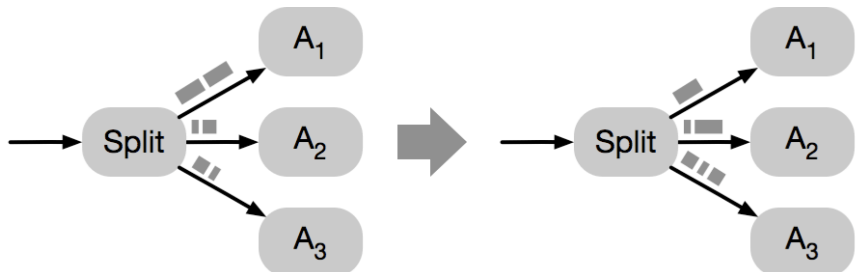


Dynamism

- Submission-time
- Online, via operator migration

Load Balancing

Distribute workload evenly across resources



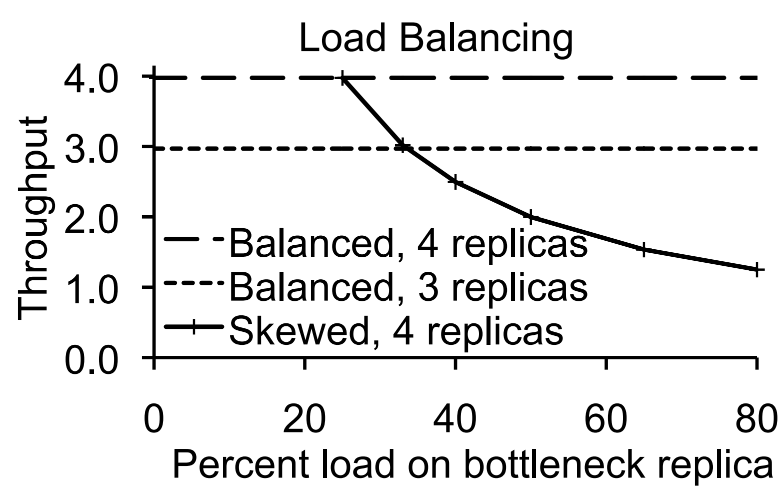
Safety

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

Variations

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

Profitability

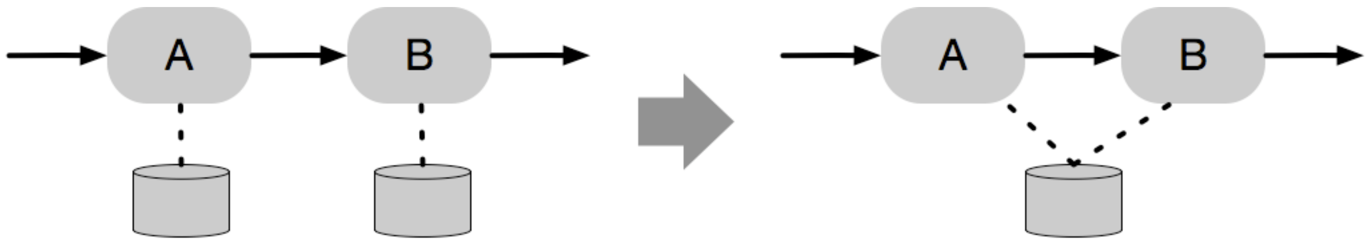


Dynamism

- Easier for routing than placement

State Sharing

Optimize for space by avoiding unnecessary copies of data.



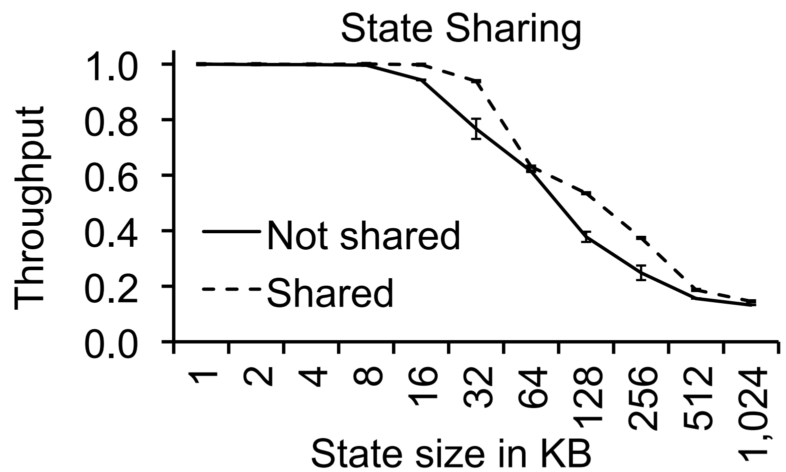
Safety

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

Variations

- Sharing **queues**
- Sharing **windows**
- Sharing **operator state**

Profitability



Dynamism

- N/A

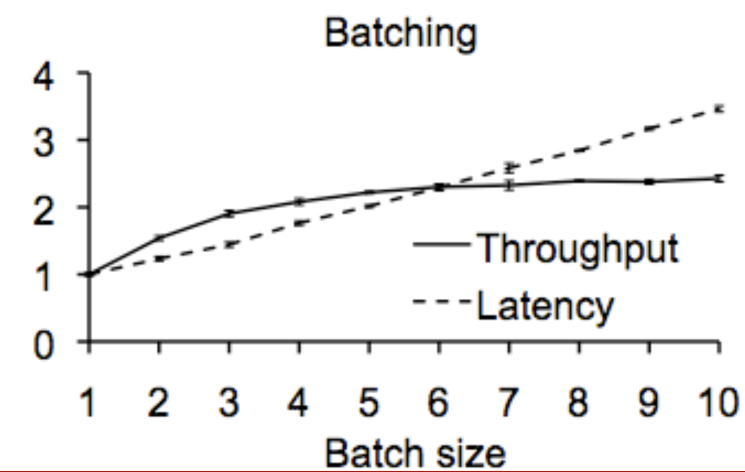
Batching

Process multiple data items in a single batch.

Safety

- No deadlocks
- Satisfy deadlines

Profitability



Variations

- **Batching enables traditional compiler optimizations**

Dynamism

- Batch controller
- Train scheduling

Algorithm Selection

Use a faster algorithm for implementing an operator.

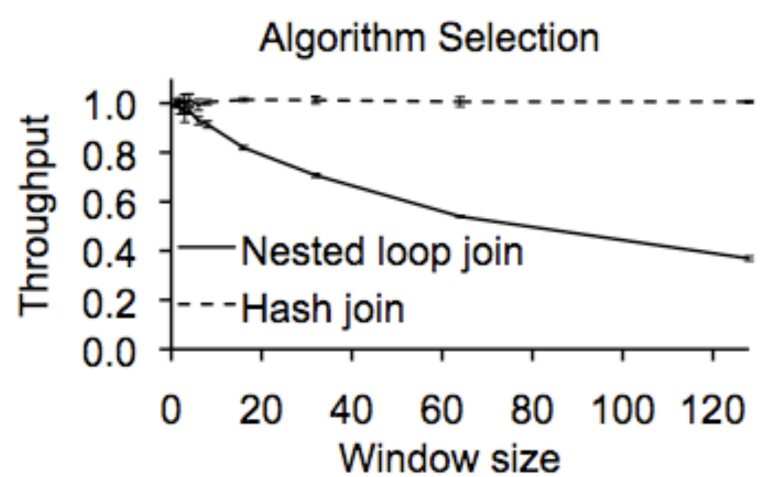
Safety

- $A_\alpha(s) \cong A_\beta(s)$
- **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 when overloaded.

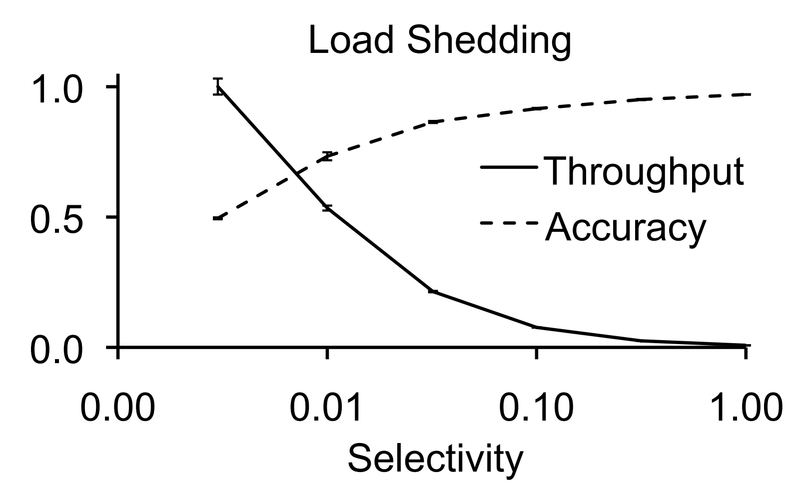
Safety

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

Variations

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

Profitability



Dynamism

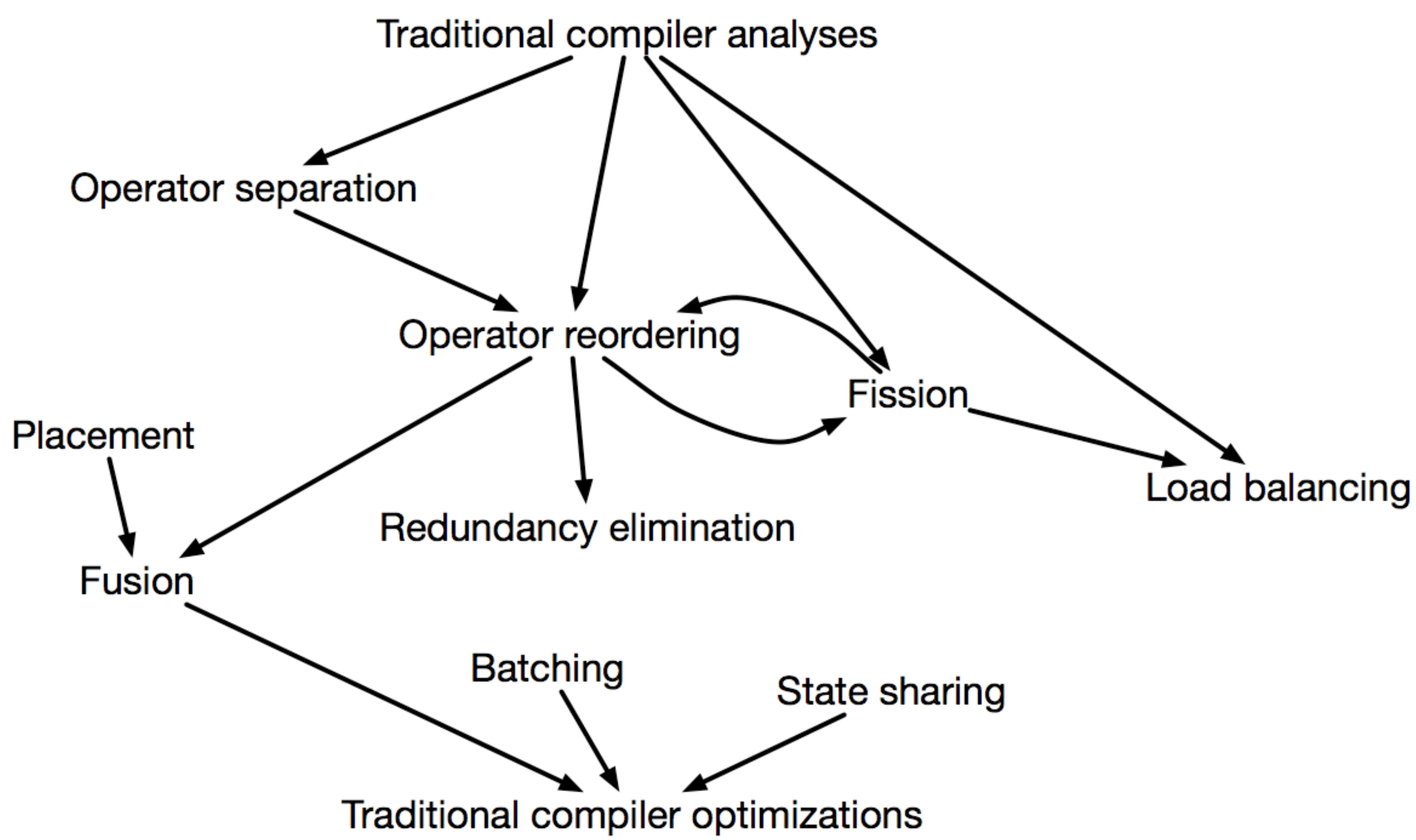
- **Always dynamic**

Optimizations Enable Optimizations

Traditional → Stream

Stream → Stream

Stream → Traditional



Languages Enable Optimizations

High-level
Easy to use
Optimizable

Mario
CEP patterns
StreamDatalog
StreamSQL
StreamIt
Graph GUI
SPL
Java API
Annotated C
C/Fortran

Low-level
General
Predictable

Hand-Optimized vs. Auto-Optimization

Hand-Optimized

- Experts can get better performance
- Better Control
- Generality
- Easier to build systems

Auto-Optimized

- Better out-of-the-box experience
- Portability
- Application code is less cluttered

Requirements for an IL

Observation	Conclusion
4/11 depend on the order that operators execute	IL should be explicit how determinism is enforced
5/11 modify the topology	IL needs to model communication
8/11 depend on state	IL needs to model state
9/11 have dynamic variations	IL needs to support dynamism
11/11 have a unique requirement	IL must be extensible

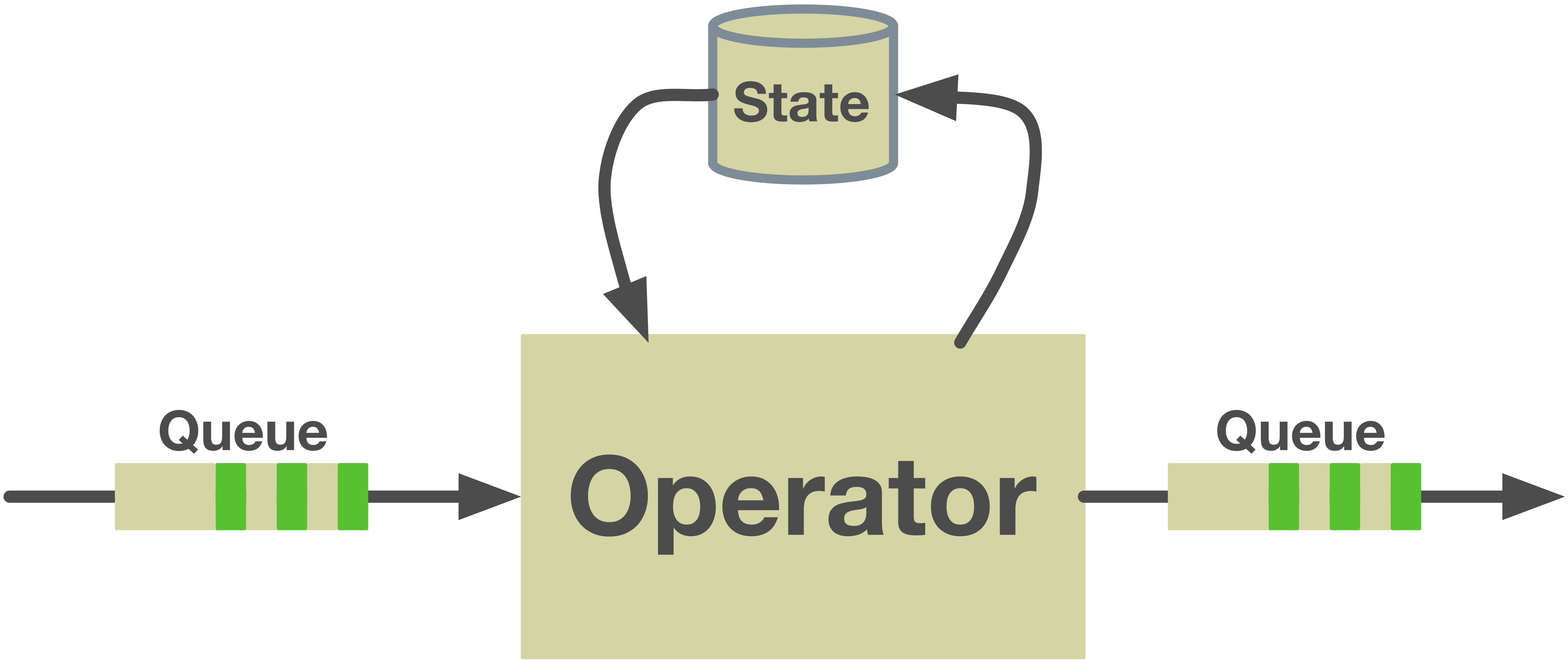
A Universal Calculus For Stream Processing

A formal foundation for a streaming IL

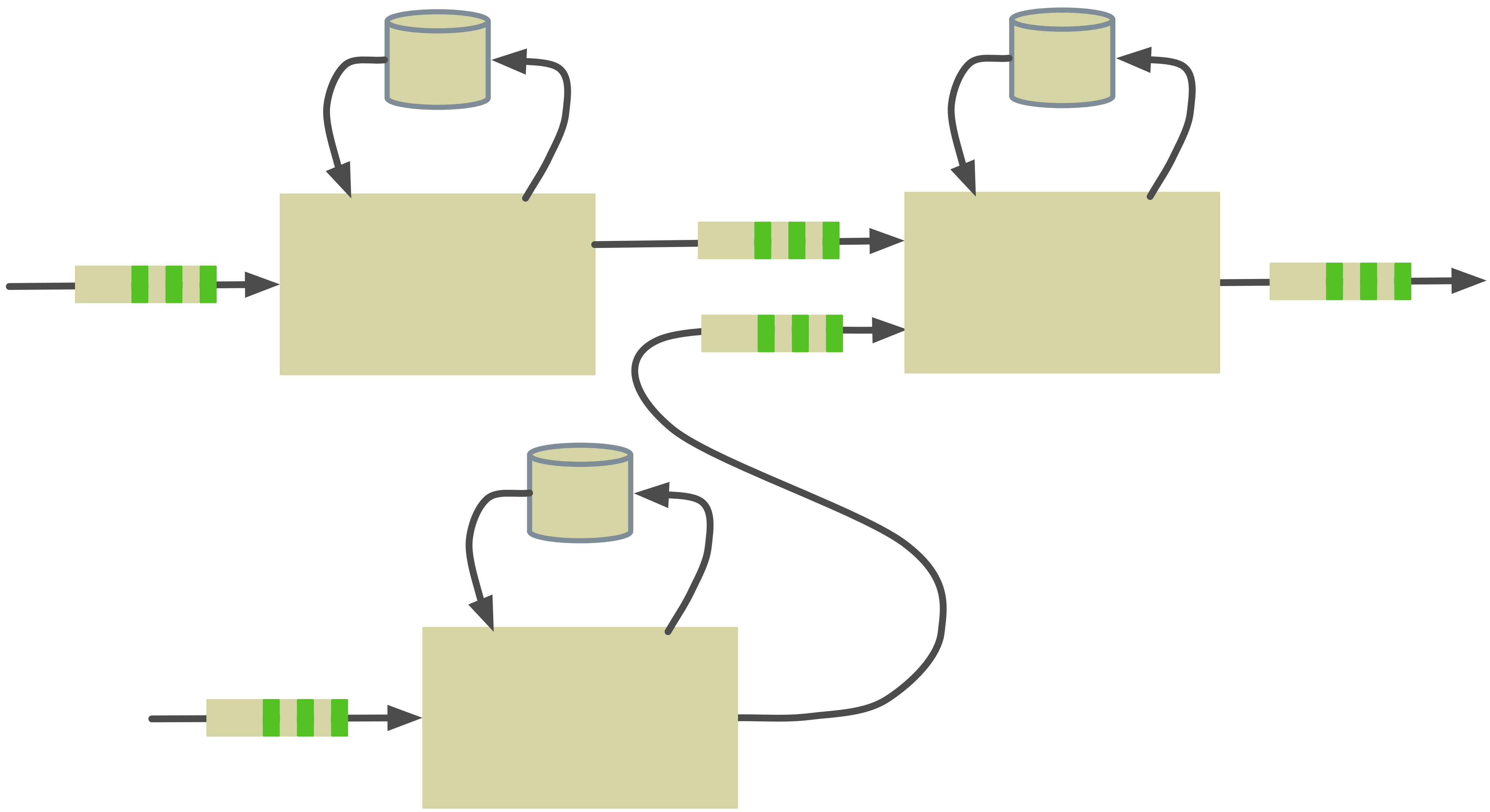
Design Goals

- Enable reasoning about correctness of optimizations
- Flexibility to represent diverse languages
- Formalize *three* of the requirements:
  - State, communication, and non-determinism
- Save dynamism for future work
- Extensibility is addressed in the IL

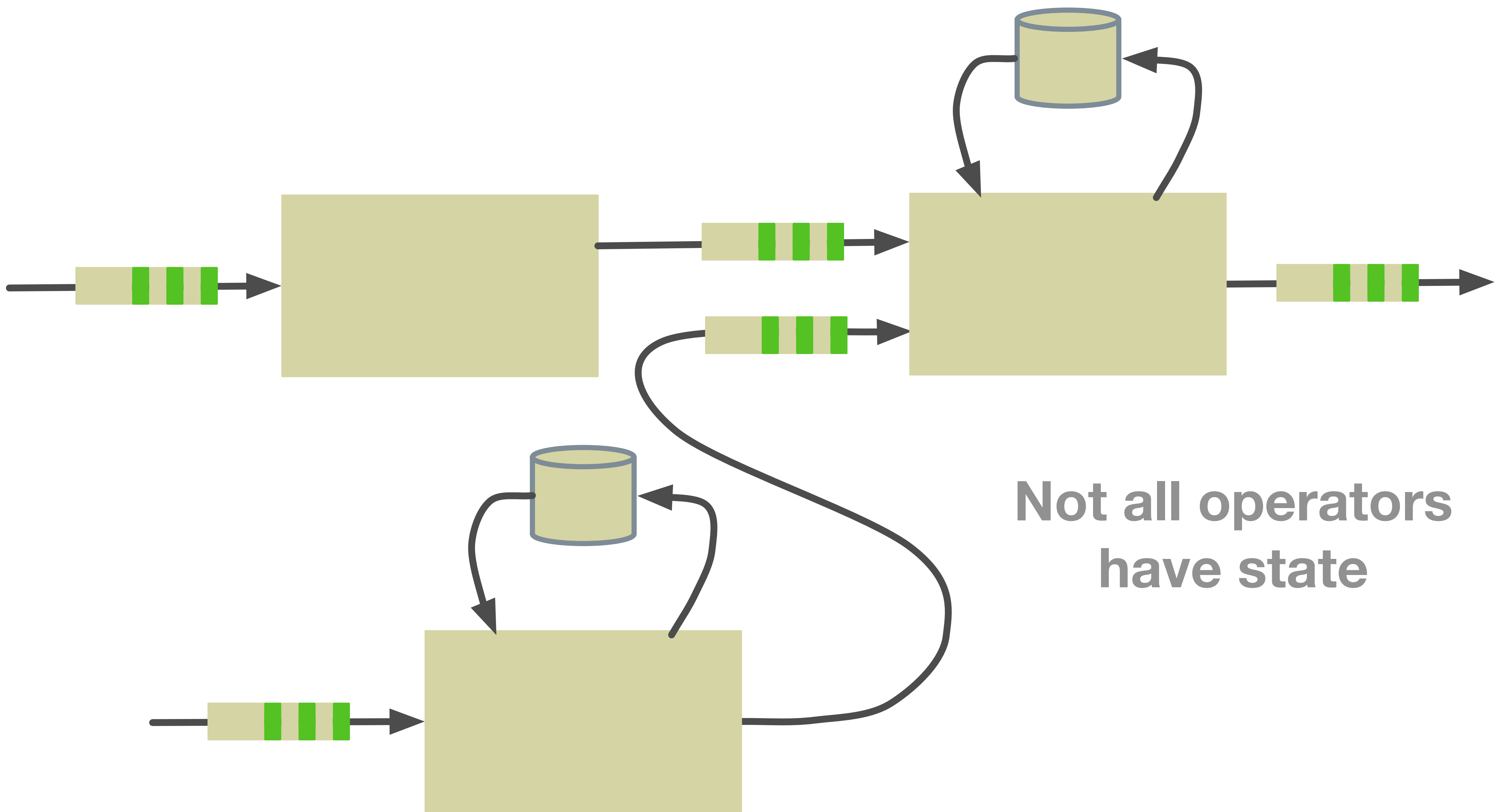
Elements of a Streaming App



Elements of a Streaming App

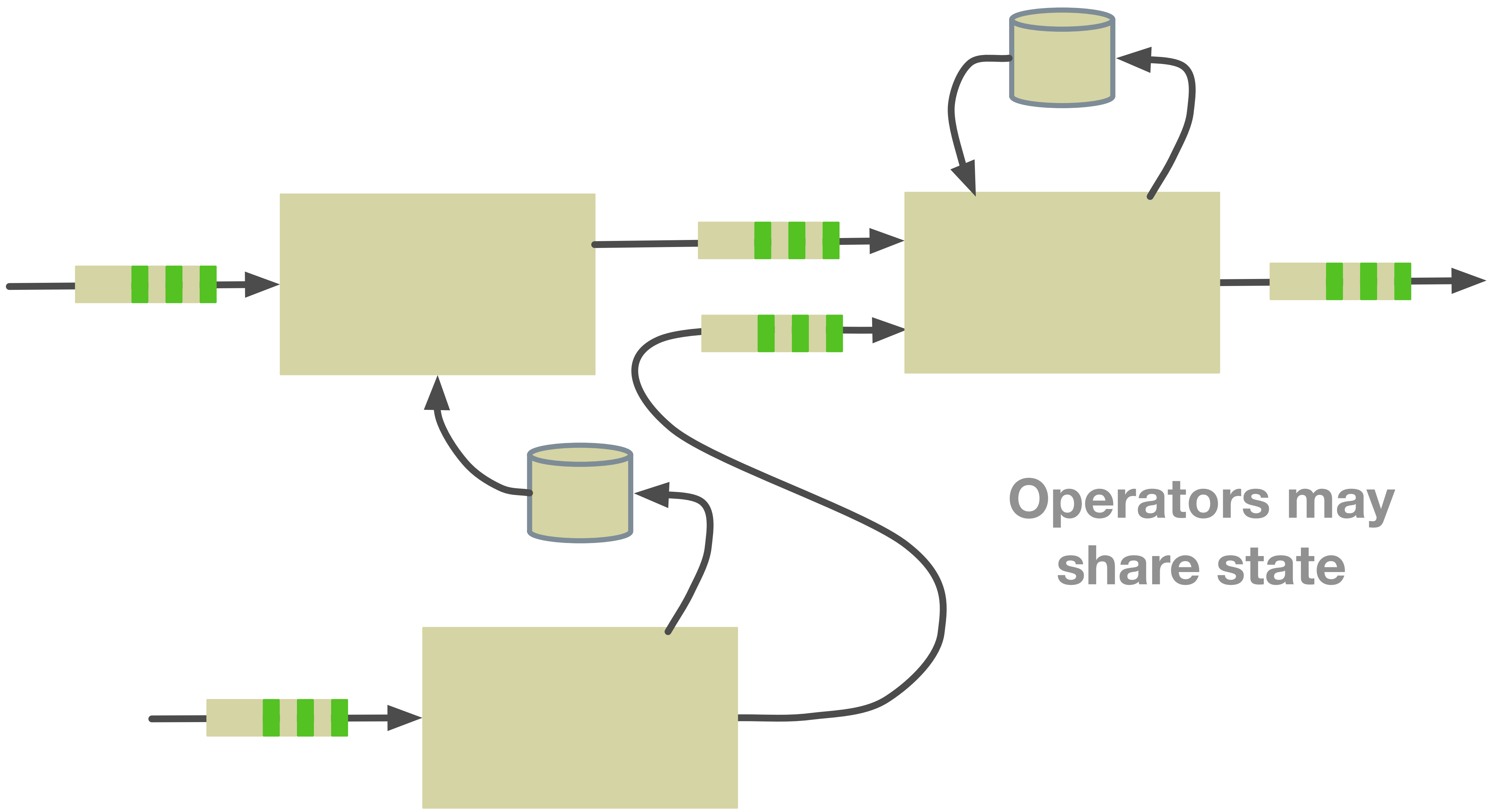


Elements of a Streaming App



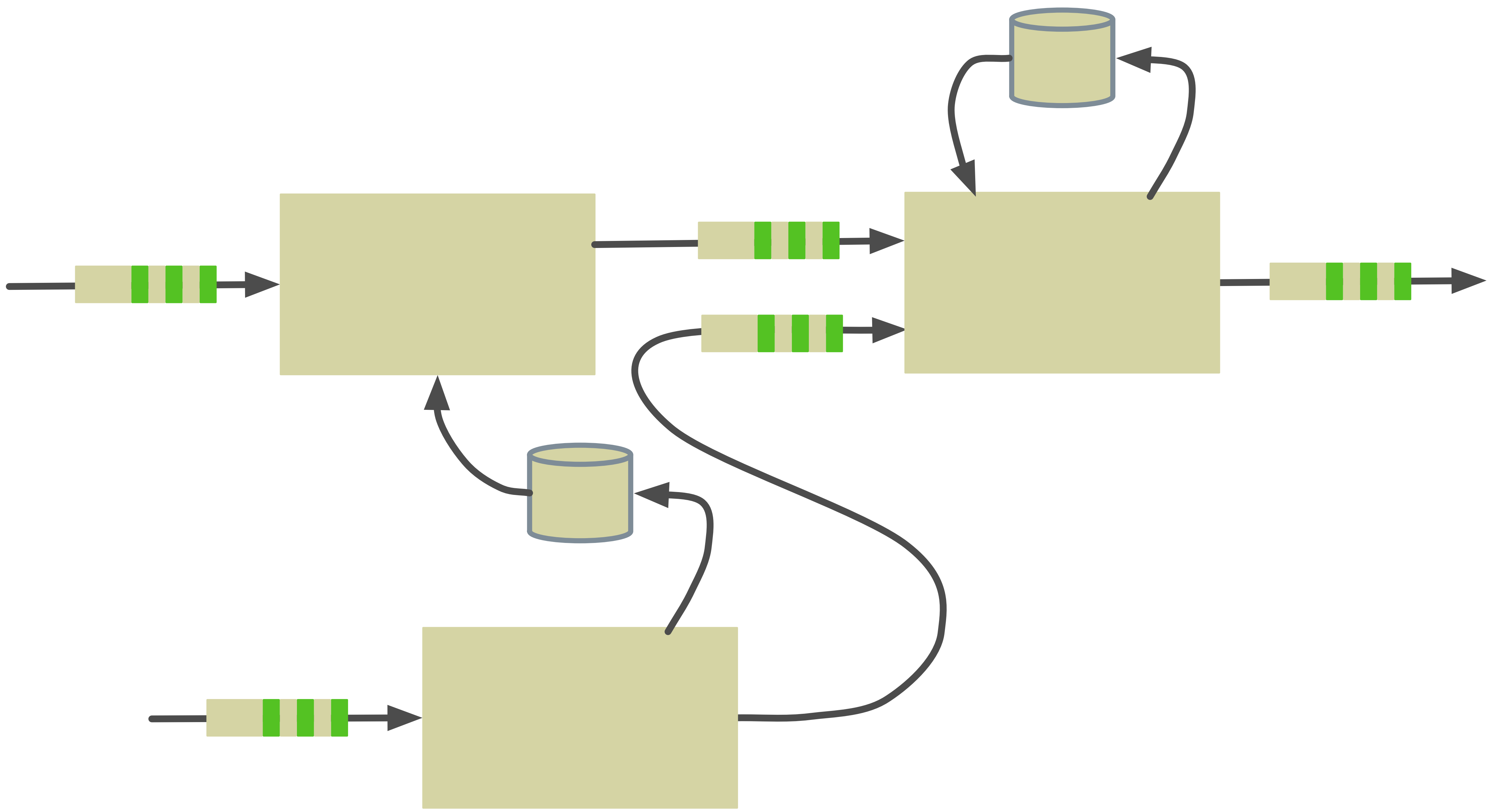
Not all operators have state

Elements of a Streaming App

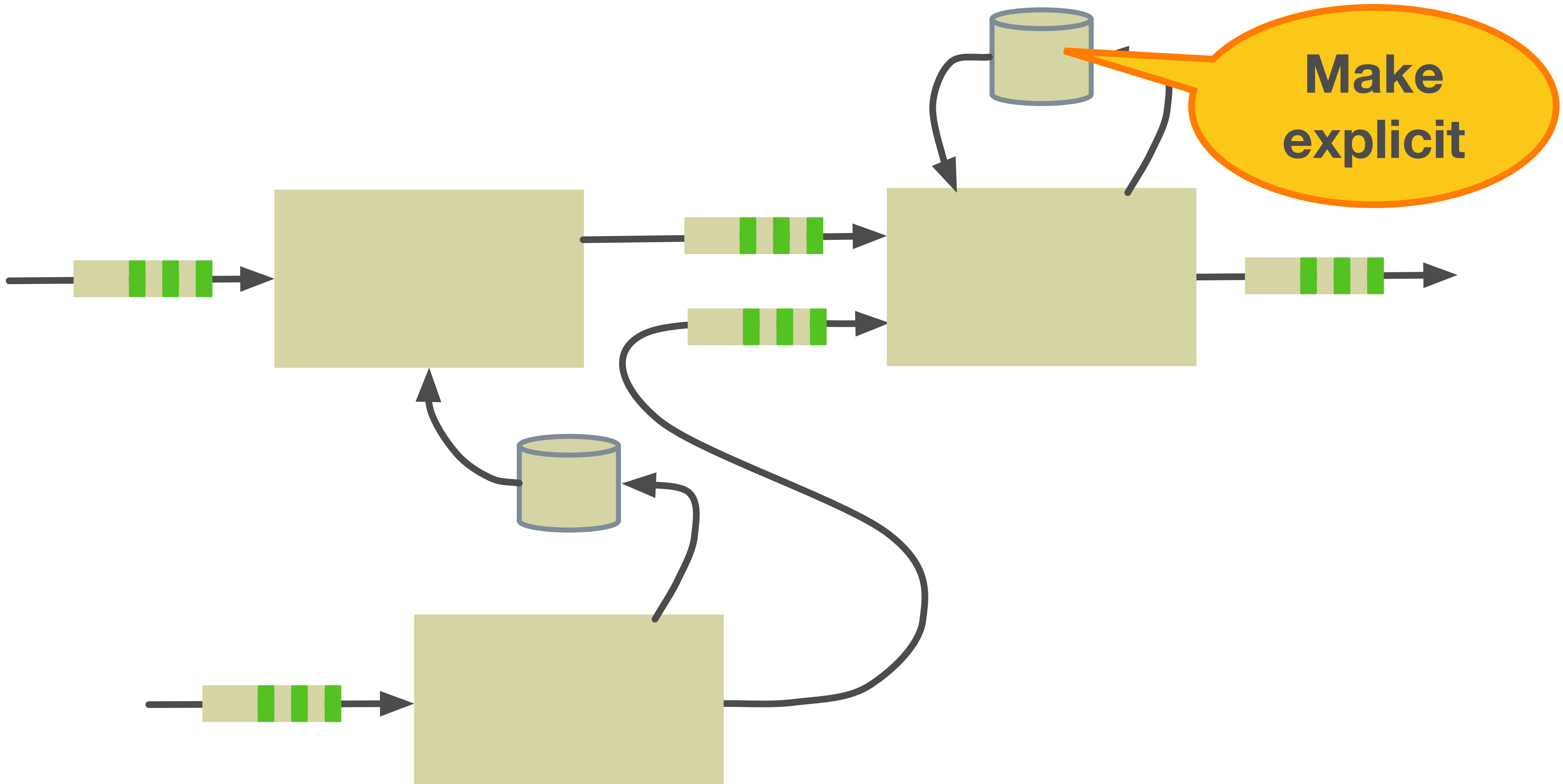


Operators may share state

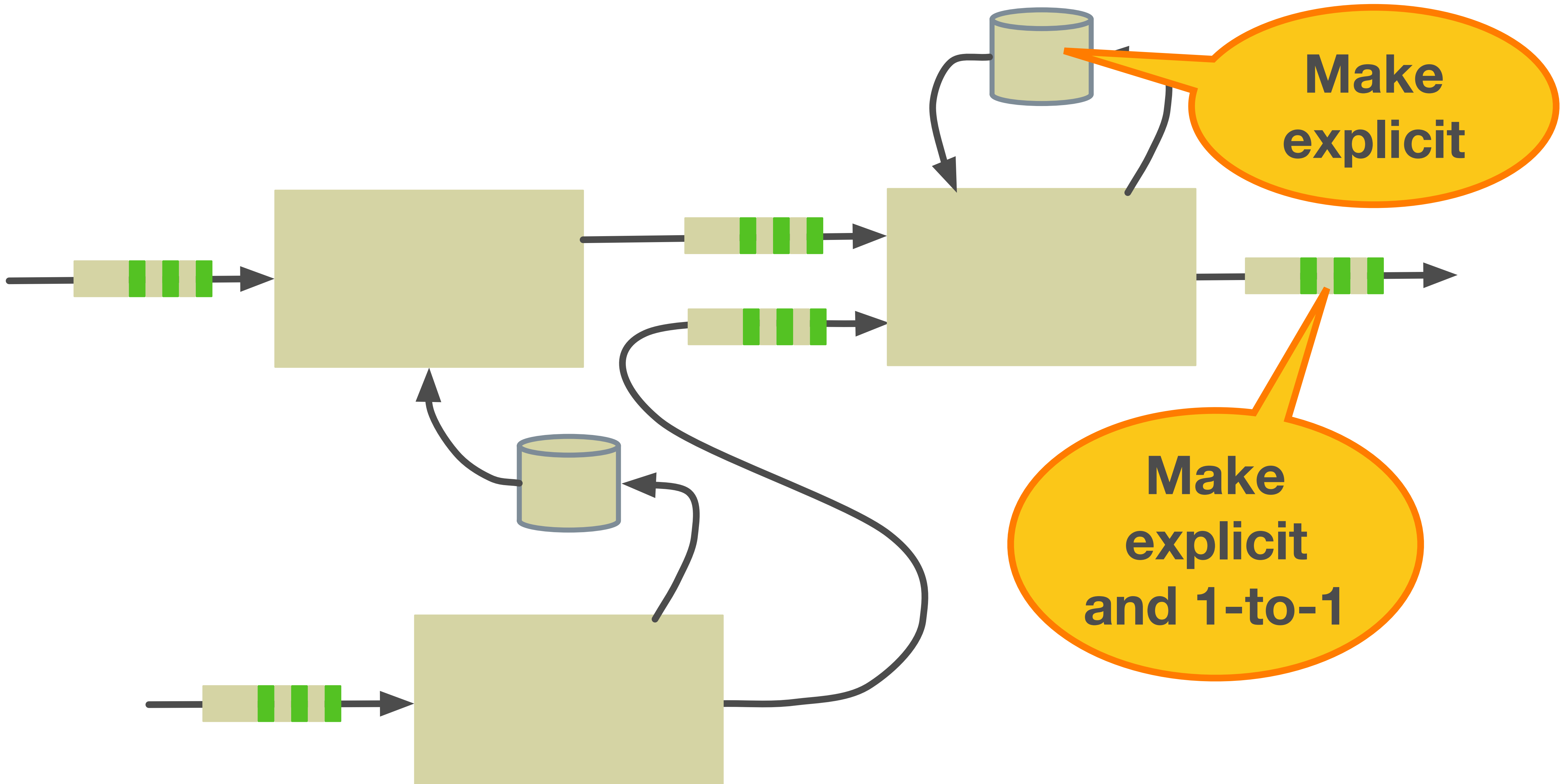
Requirements for Calculus



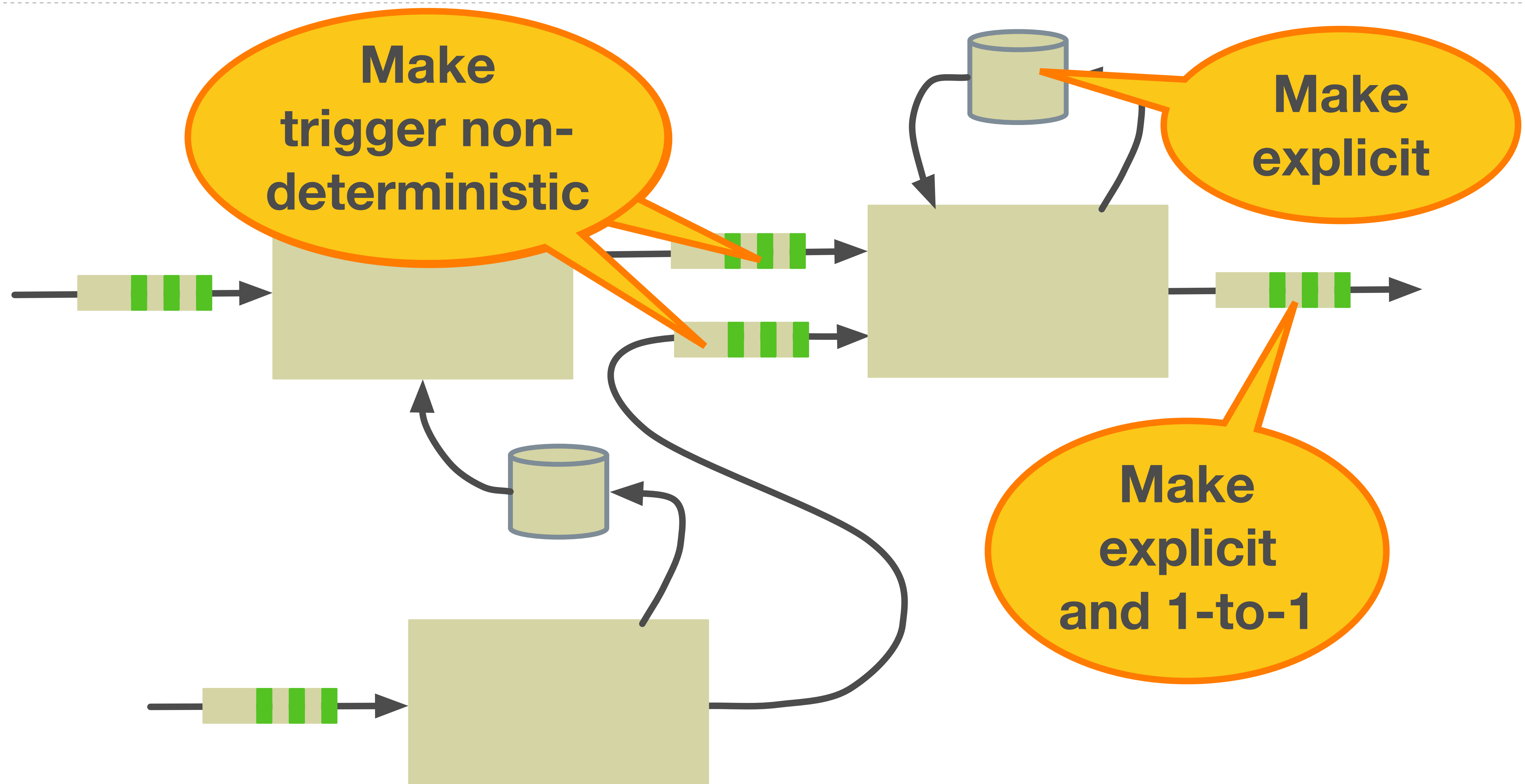
Requirements for Calculus



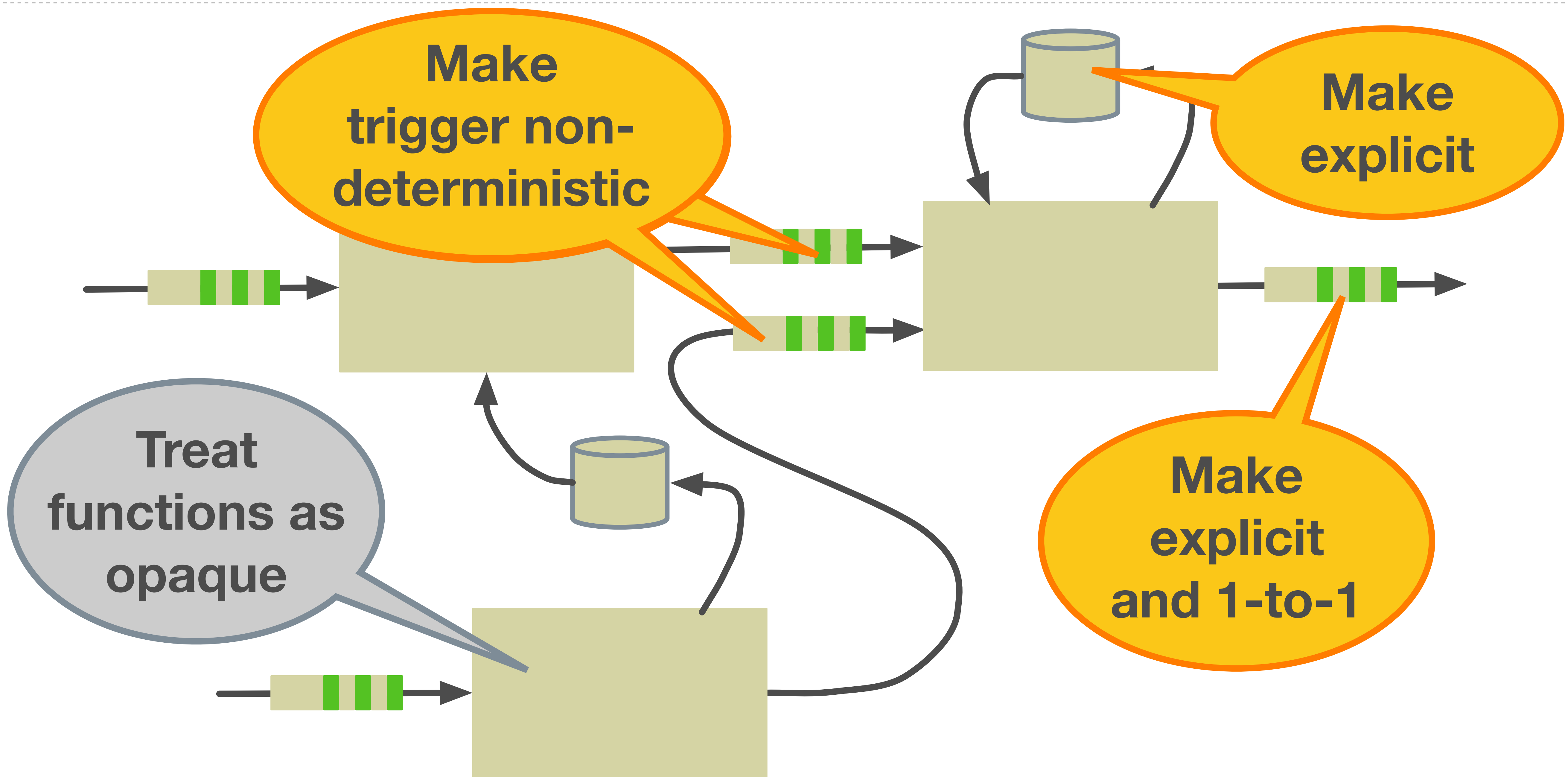
Requirements for Calculus



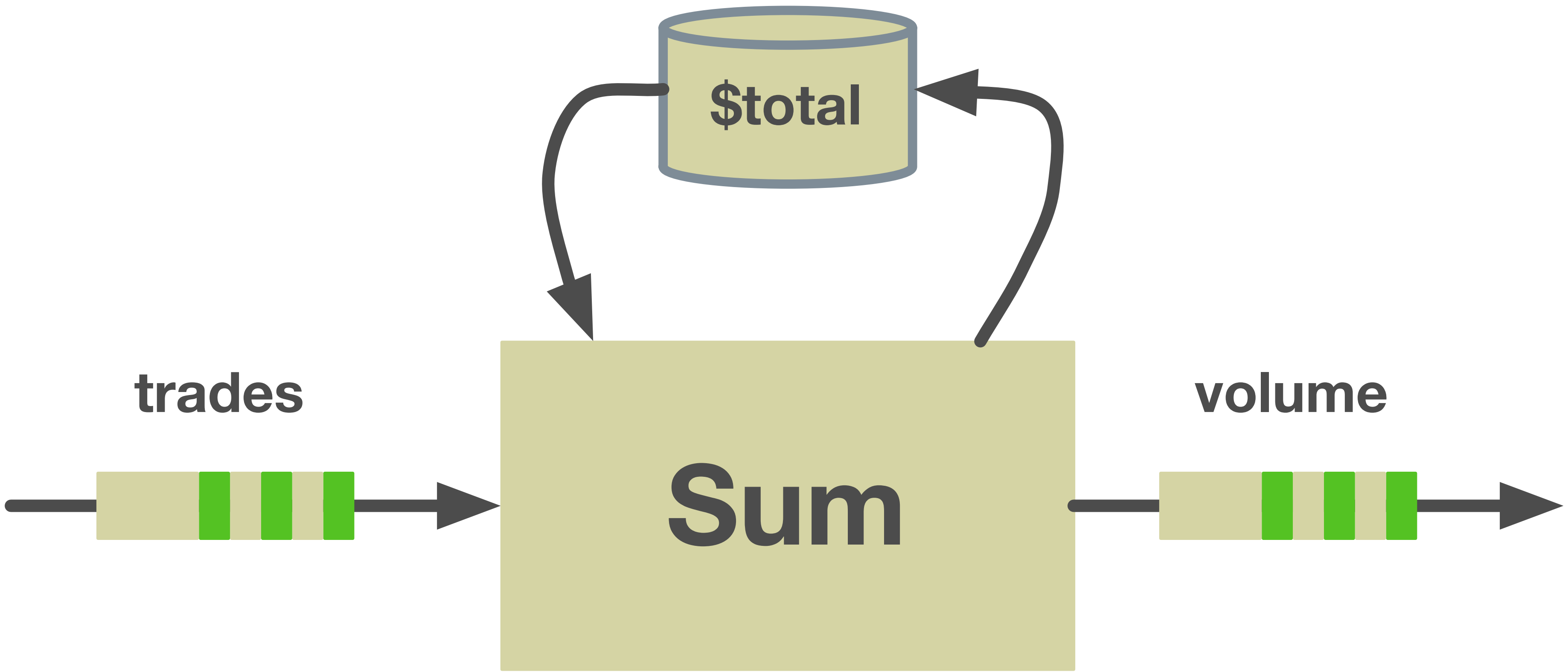
Requirements for Calculus



Requirements for Calculus

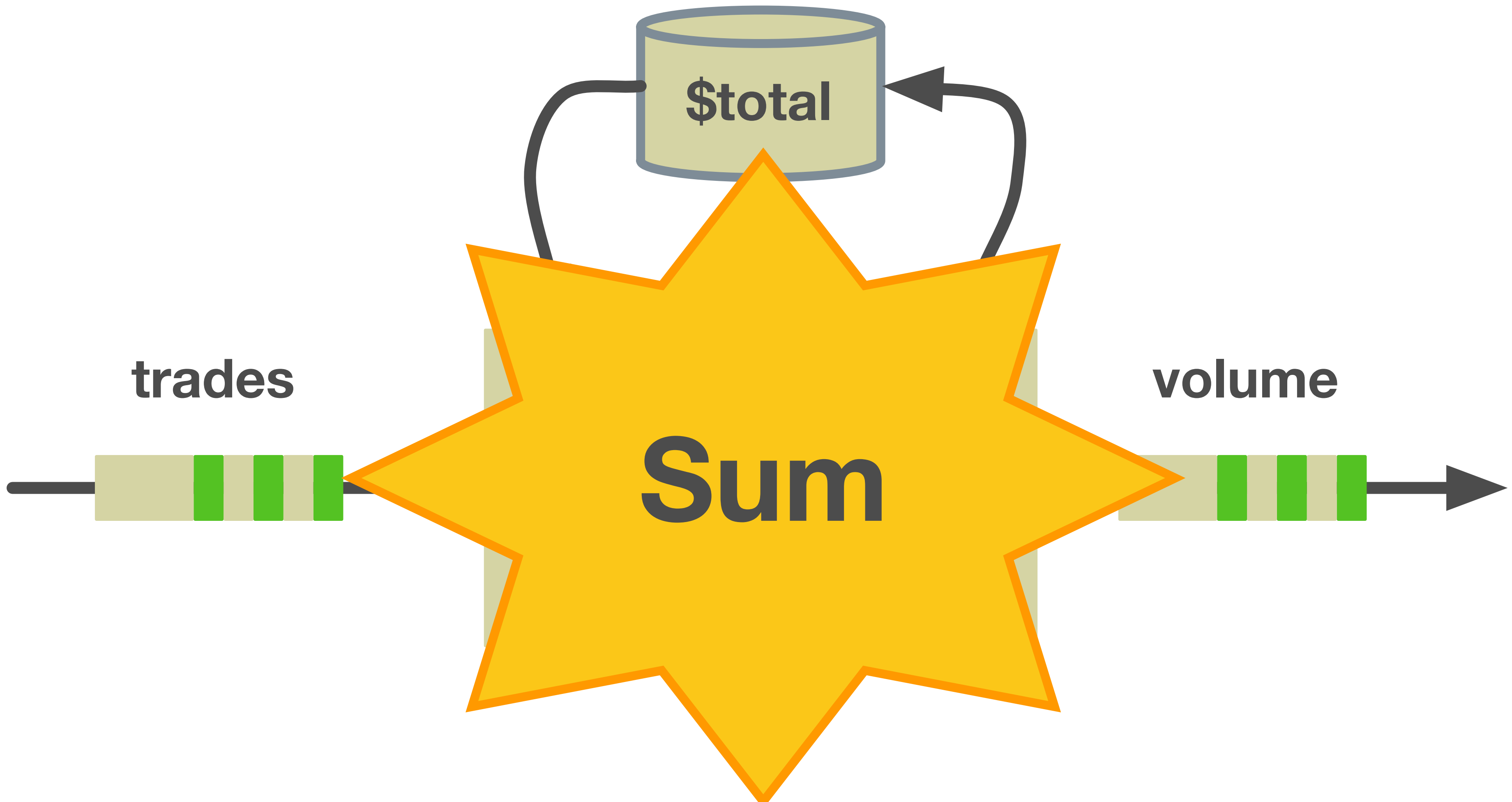


Brooklet Syntax



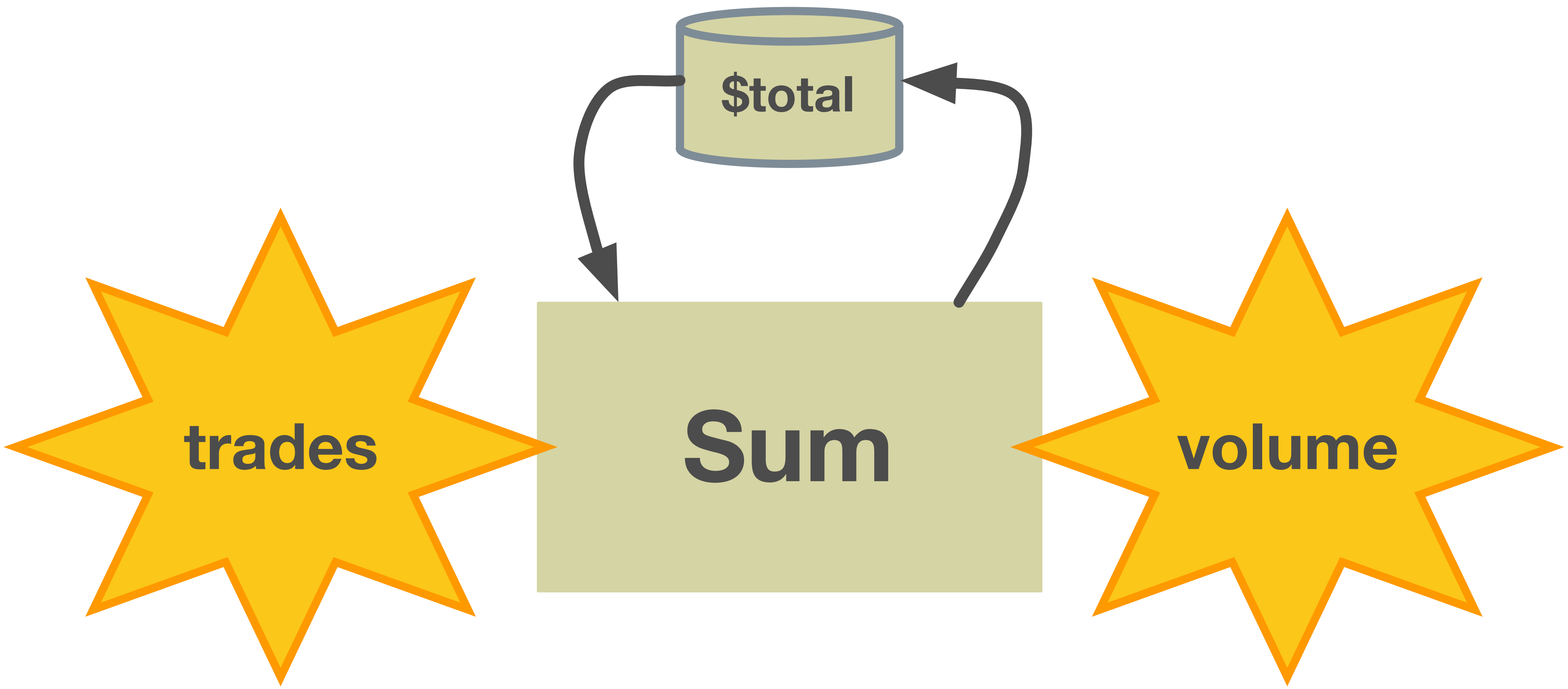
(volume, $total) ← Sum(trades, $total)

Function Environment



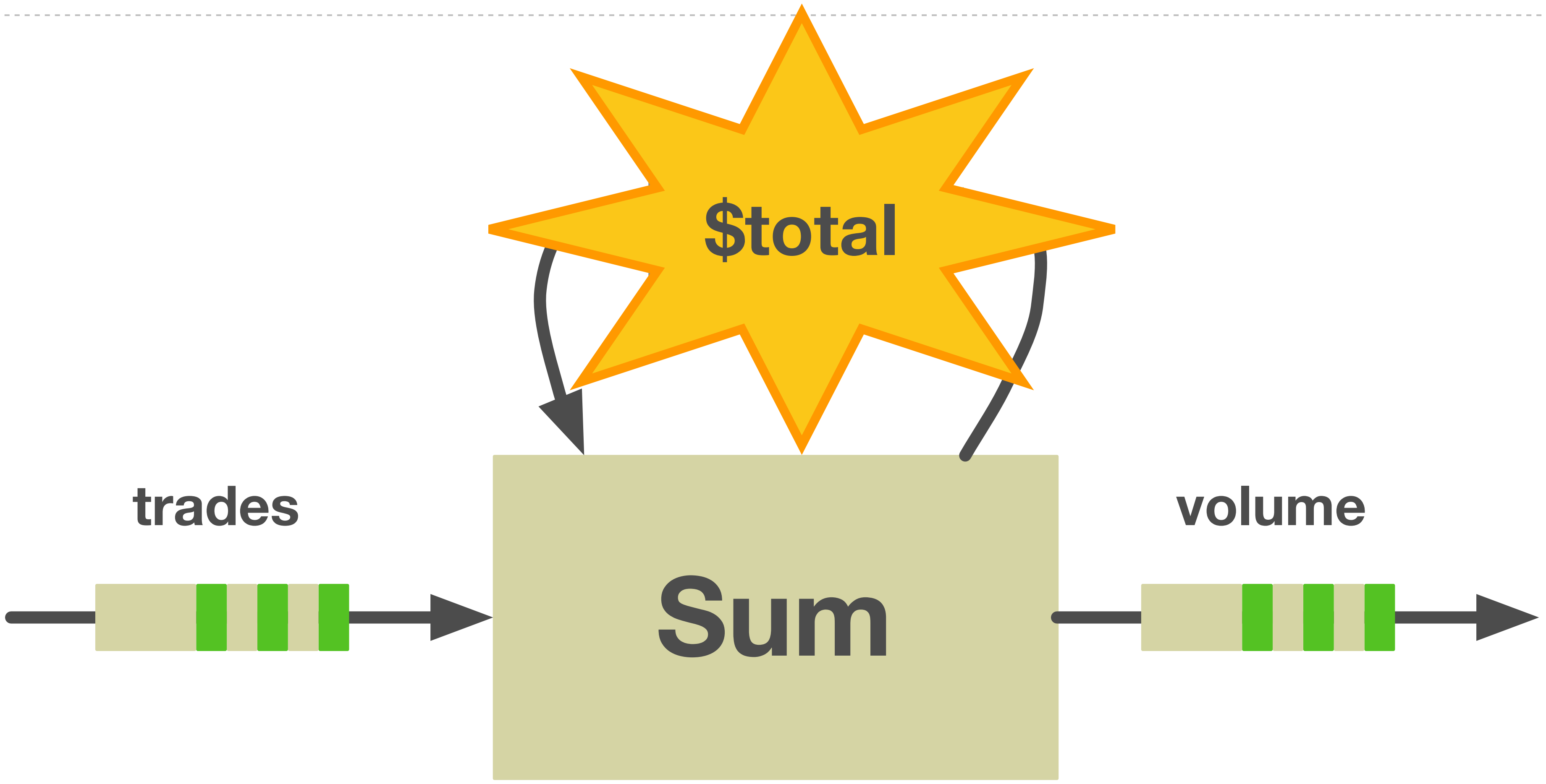
F: The function implementations

Queue Store



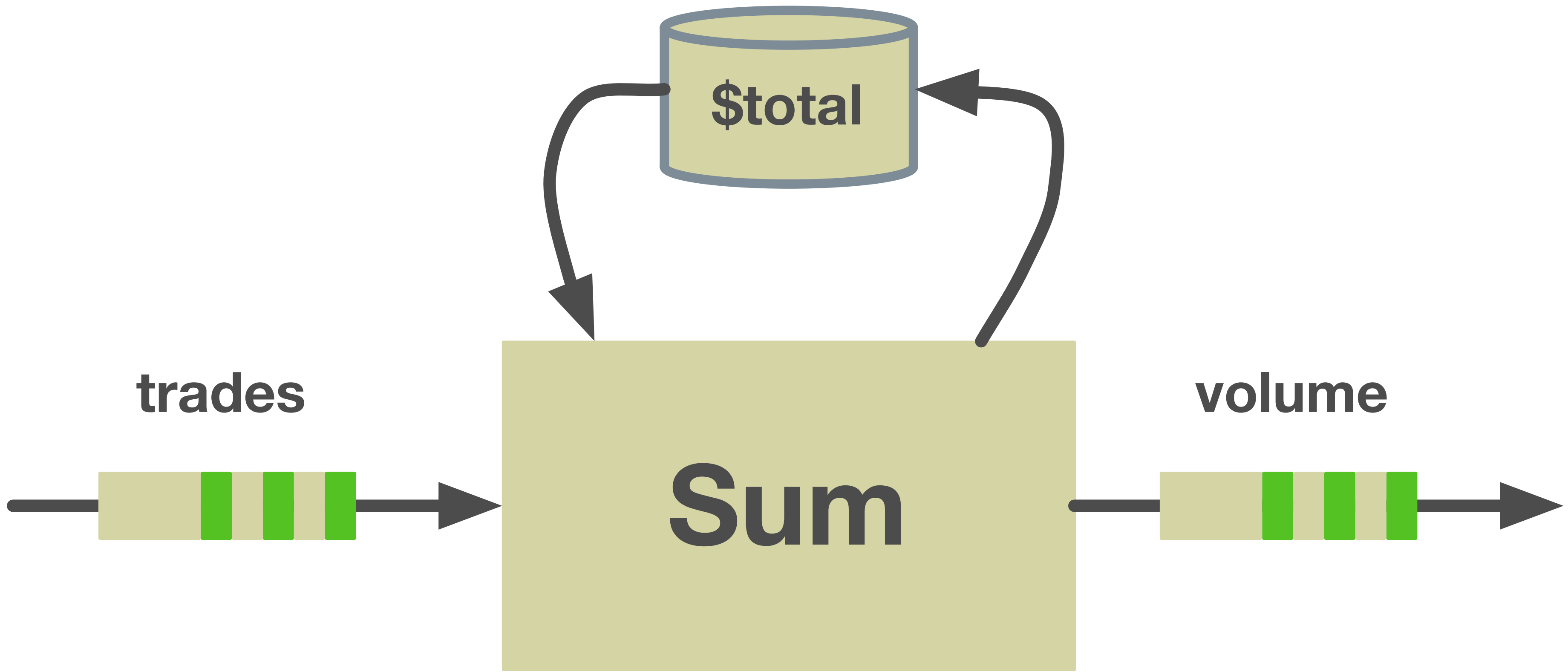
Q: The contents of the queues

Variable Store



V: The contents of the variables

Brooklet Operational Semantics



$$F \vdash \langle Q, V \rangle \rightarrow \langle Q', V' \rangle$$

Complete Calculus

Brooklet syntax:

P_b	::=	$out\ in\ \overline{op}$	*Brooklet program*
out	::=	`output` $\overline{q}$;	*Output declaration*
in	::=	`input` $\overline{q}$;	*Input declaration*
op	::=	($\overline{q}$, $\overline{v}$) $\leftarrow$ f ($\overline{q}$, $\overline{v}$);	*Operator*
q	::=	id	*Queue identifier*
v	::=	\$ id	*Variable identifier*
f	::=	id	*Function identifier*

Brooklet example: IBM market maker.

```
output result;
input bids, asks;
(ibmBids) ⟵ SelectIBM(bids);
(ibmAsks) ⟵ SelectIBM(asks);
($lastAsk)⟵ Window(ibmAsks);
(ibmSales)⟵ SaleJoin(ibmBids,$lastAsk);
(result,$cnt) ⟵ Count(ibmSales,$cnt);
```

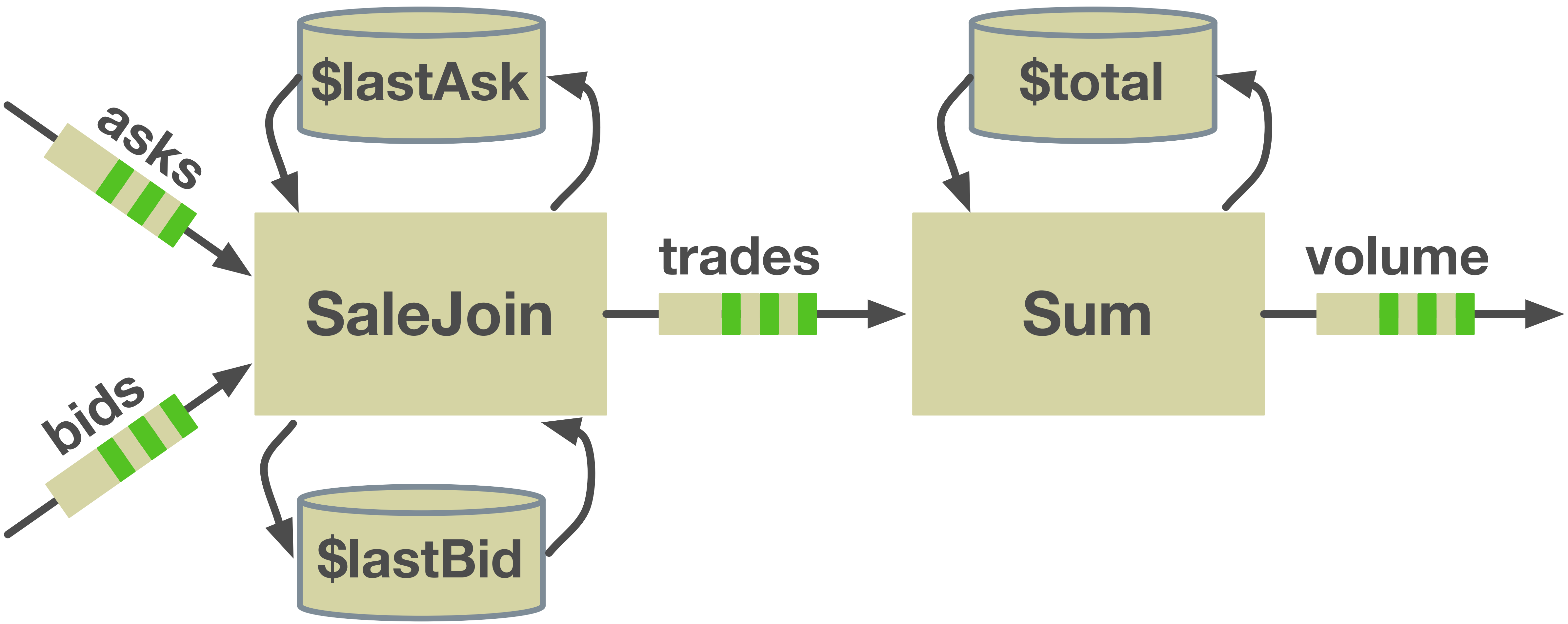
Brooklet semantics: $F_b \vdash \langle V, Q\rangle \longrightarrow \langle V', Q'\rangle$

$$\frac{\begin{array}{c} d, b = Q(q_i) \\ op = (\_,\_) \leftarrow f(\overline{q}, \overline{v}); \\ (\overline{b}', \overline{d}') = F_b(f)(d, i, V(\overline{v})) \\ V' = updateV(op, V, \overline{d}') \\ Q' = updateQ(op, Q, q_i, \overline{b}') \end{array}}{F_b \vdash \langle V, Q\rangle \longrightarrow \langle V', Q'\rangle} \quad \text{(E-FIREQUEUE)}$$

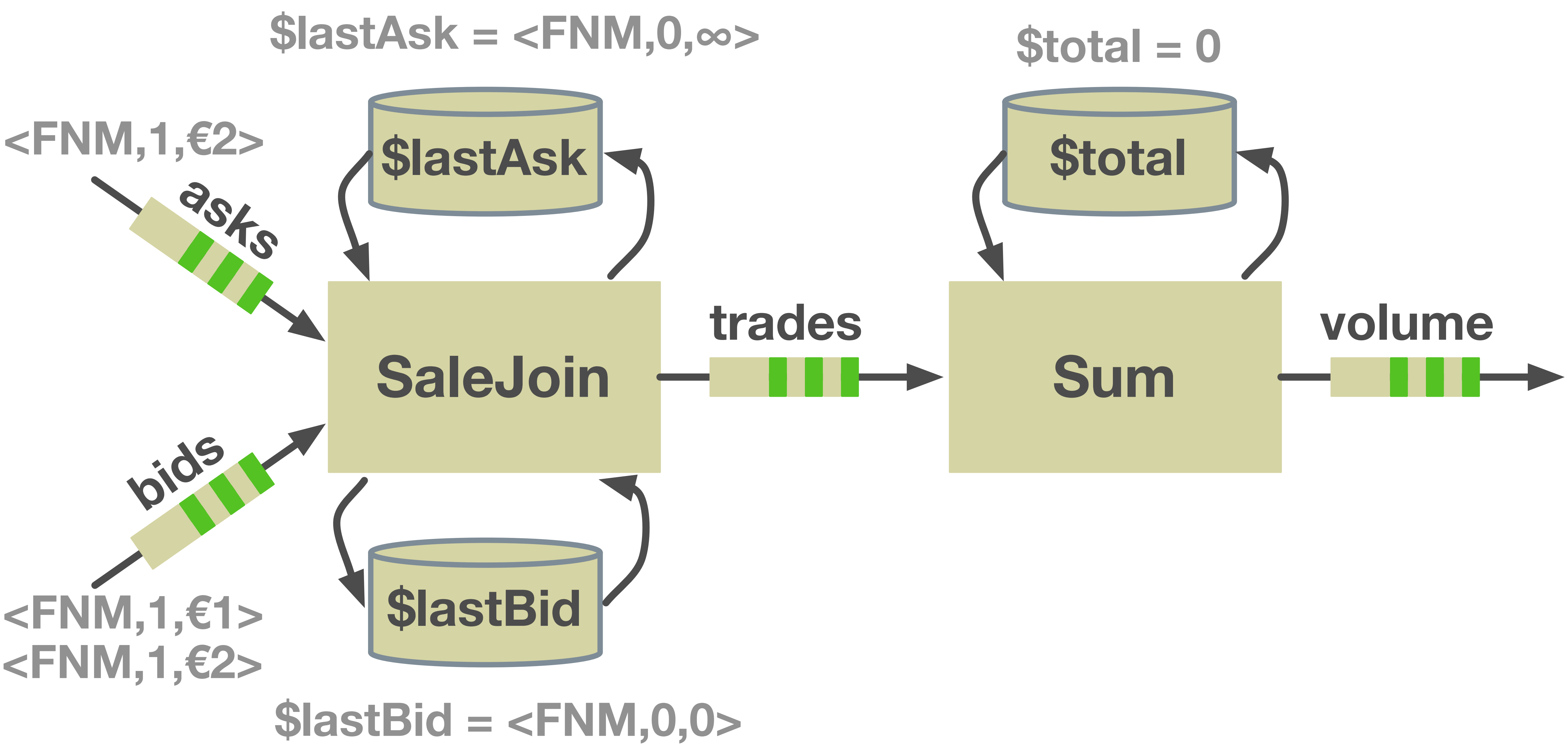
$$\frac{op = (\_, \overline{v}) \leftarrow f(\_,\_);}{updateV(op, V, \overline{d}) = [\overline{v} \mapsto \overline{d}]V} \quad \text{(E-UPDATEV)}$$

$$\frac{\begin{array}{c} op = (\overline{q}, \_) \leftarrow f(\_,\_); \\ d_f, b_f = Q(q_f) \\ Q' = [q_f \mapsto b_f]Q \\ Q'' = [\forall q_i \in \overline{q} : q_i \mapsto Q(q_i), b_i]Q' \end{array}}{updateQ(op, Q, q_f, \overline{b}) = Q''} \quad \text{(E-UPDATEQ)}$$

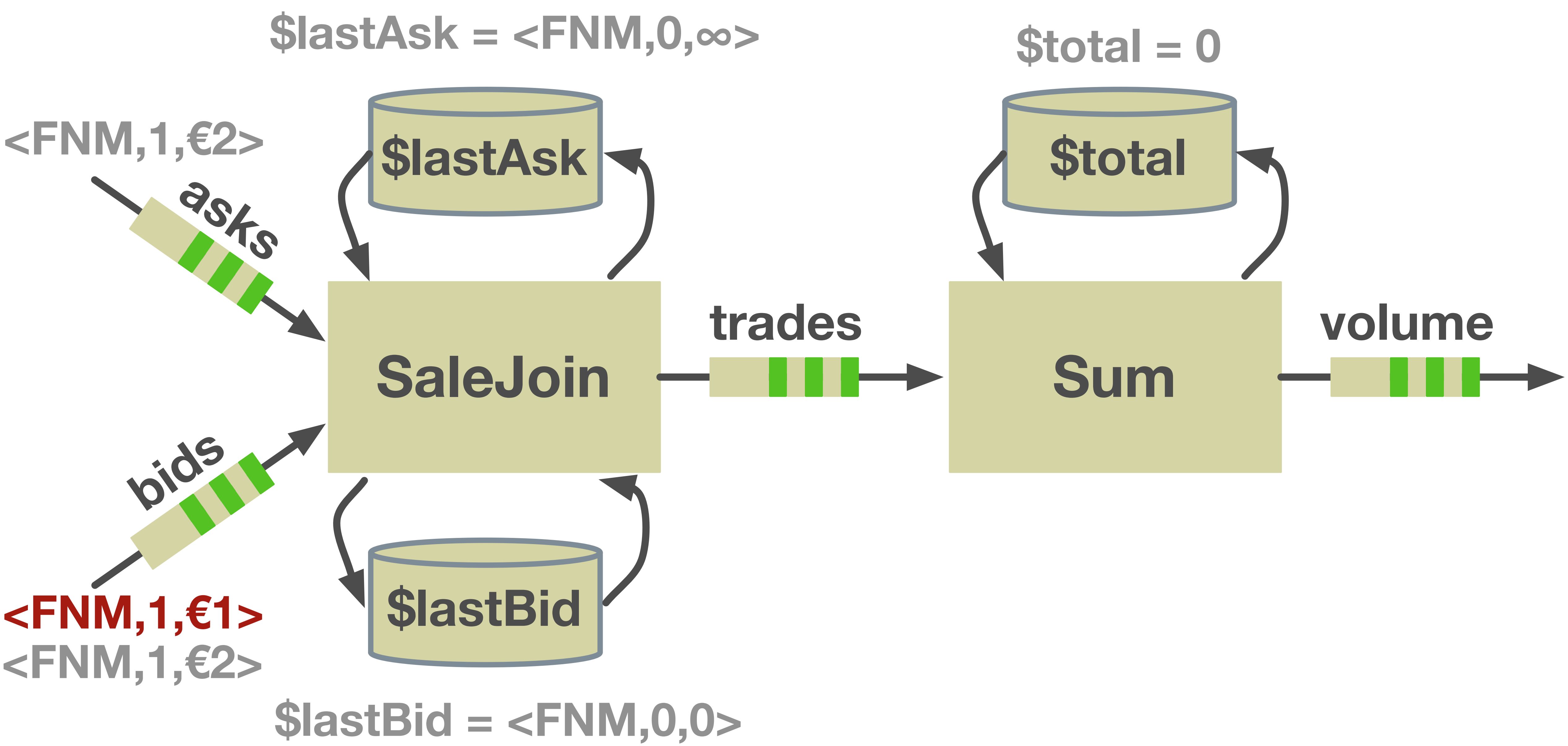
Example:
A Fannie Mae Bid/Ask Join



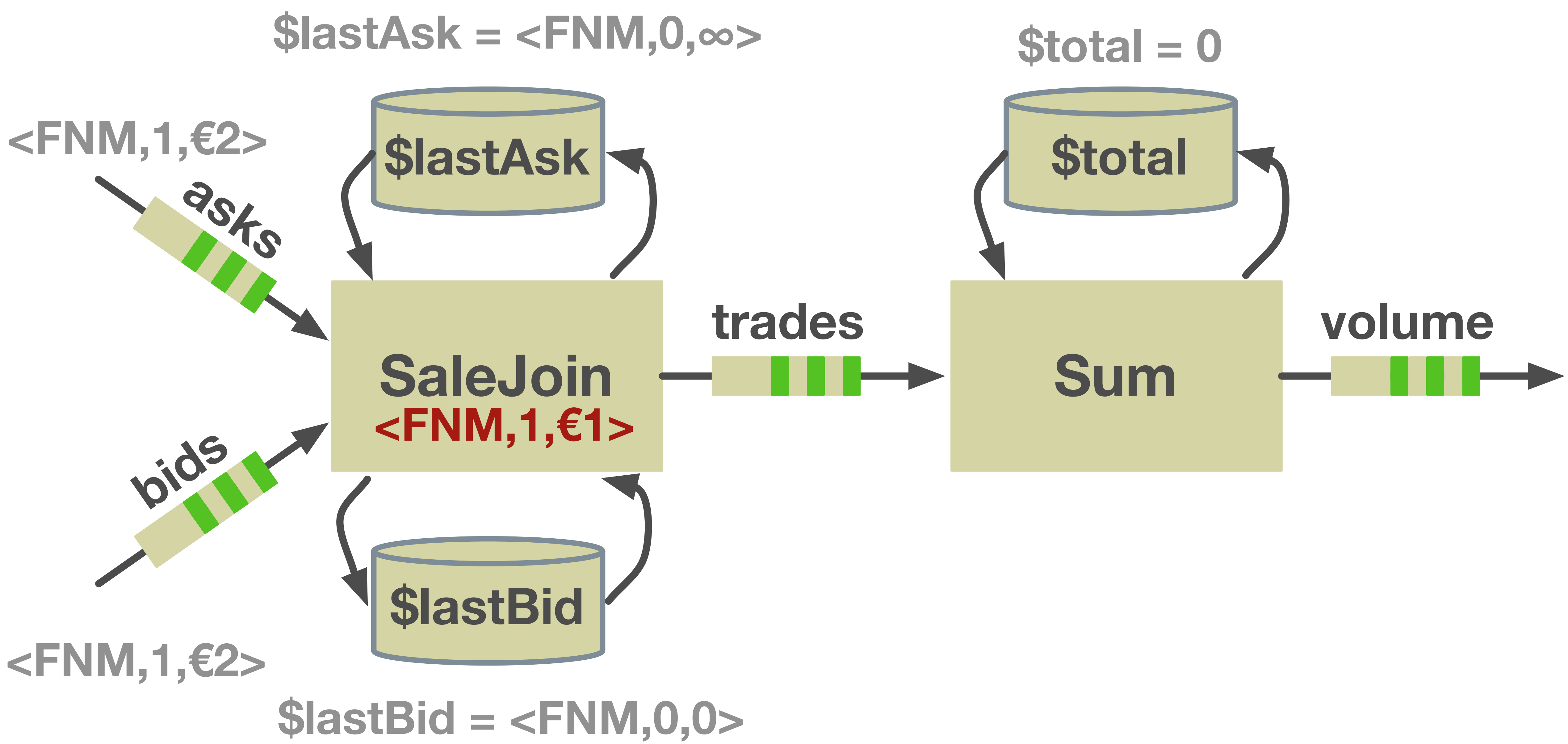
Example:
A Fannie Mae Bid/Ask Join



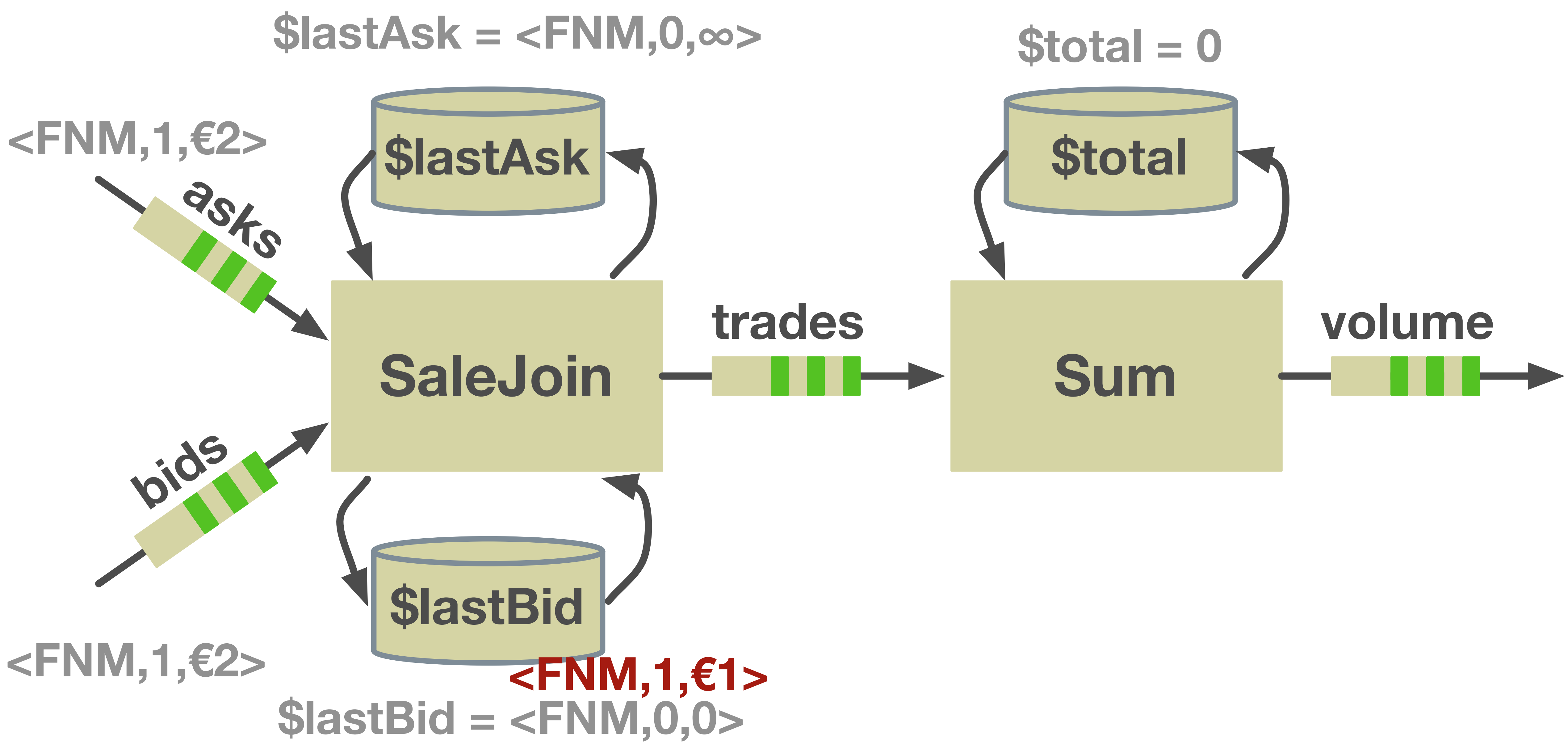
Example:
A Fannie Mae Bid/Ask Join



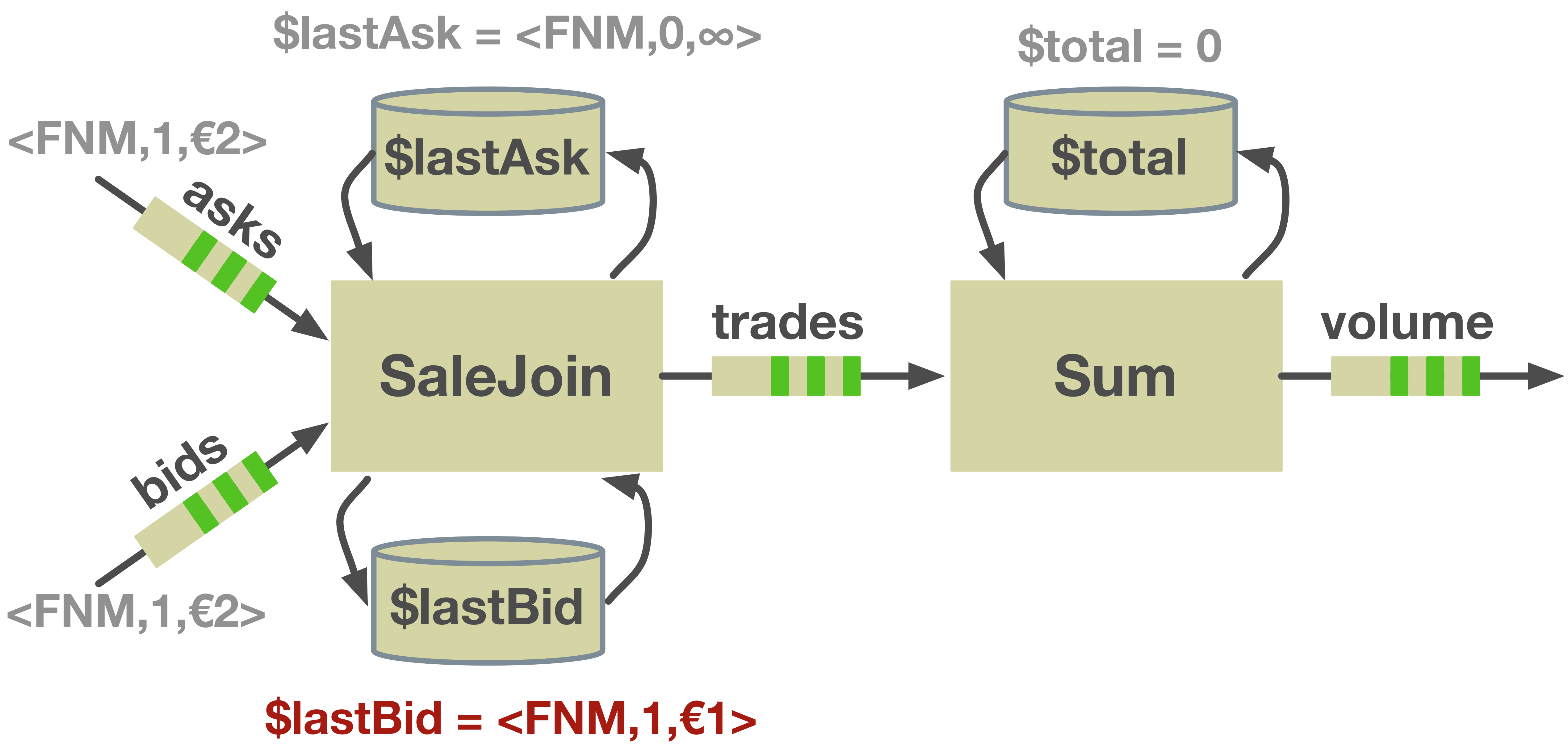
Example:
A Fannie Mae Bid/Ask Join



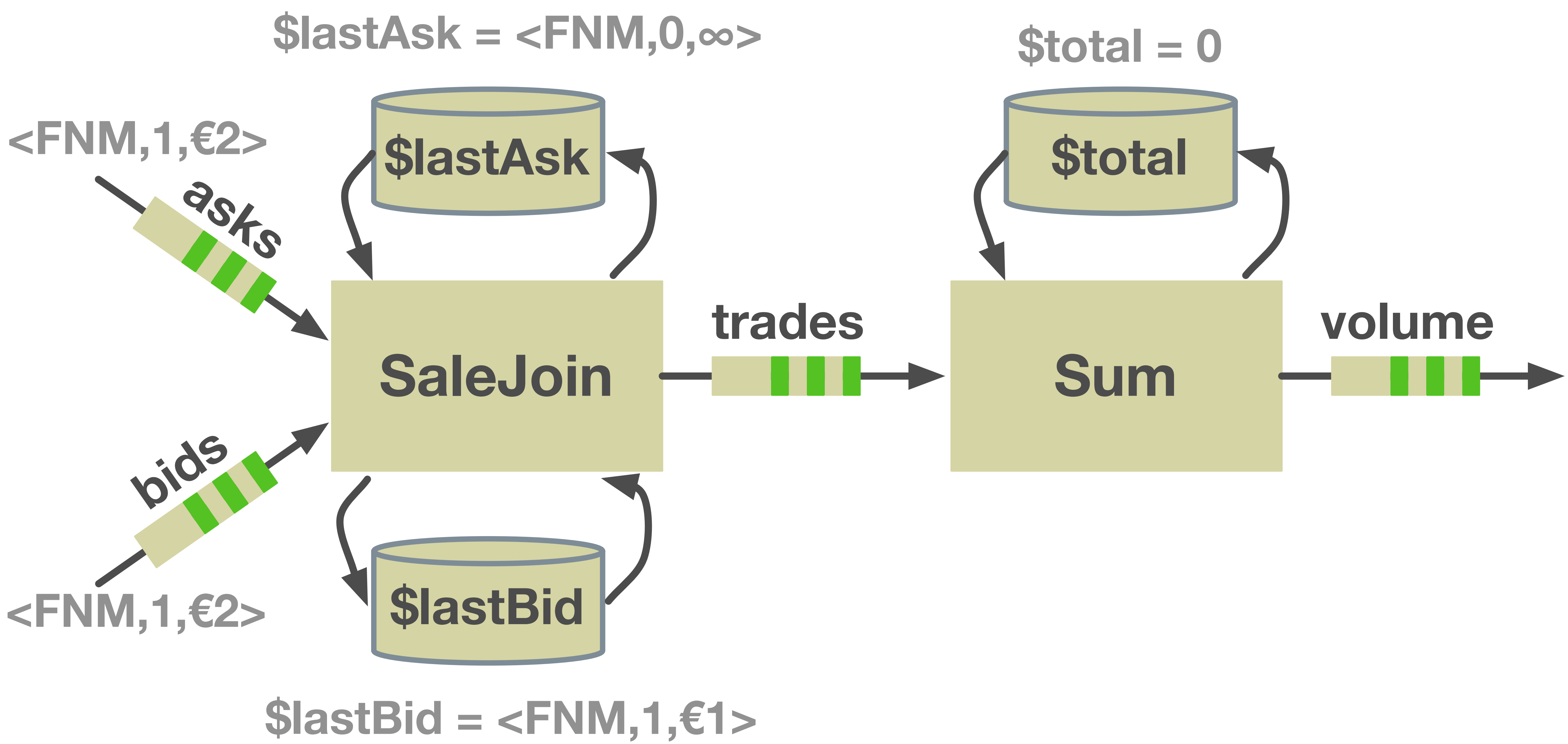
Example:
A Fannie Mae Bid/Ask Join



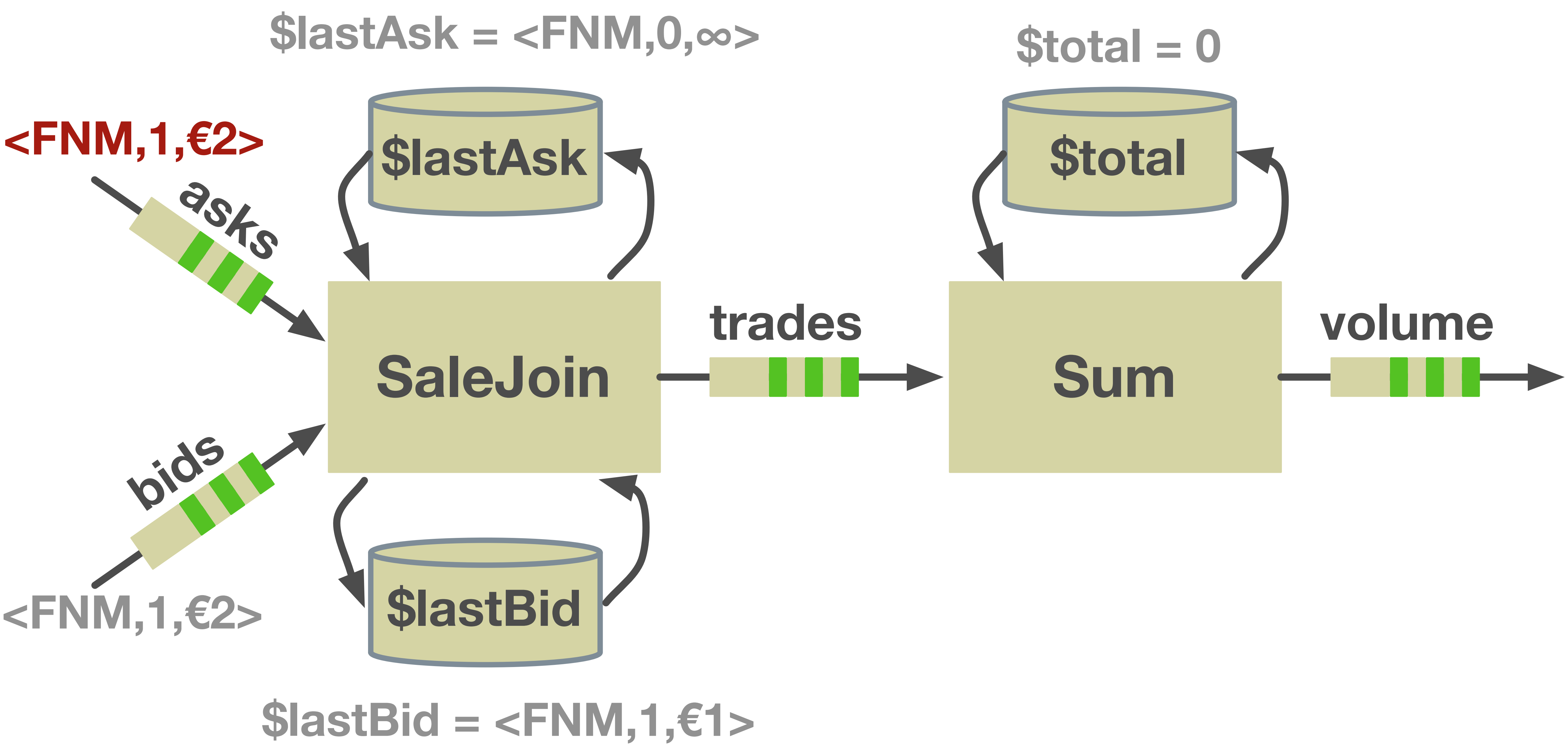
Example: A Fannie Mae Bid/Ask Join



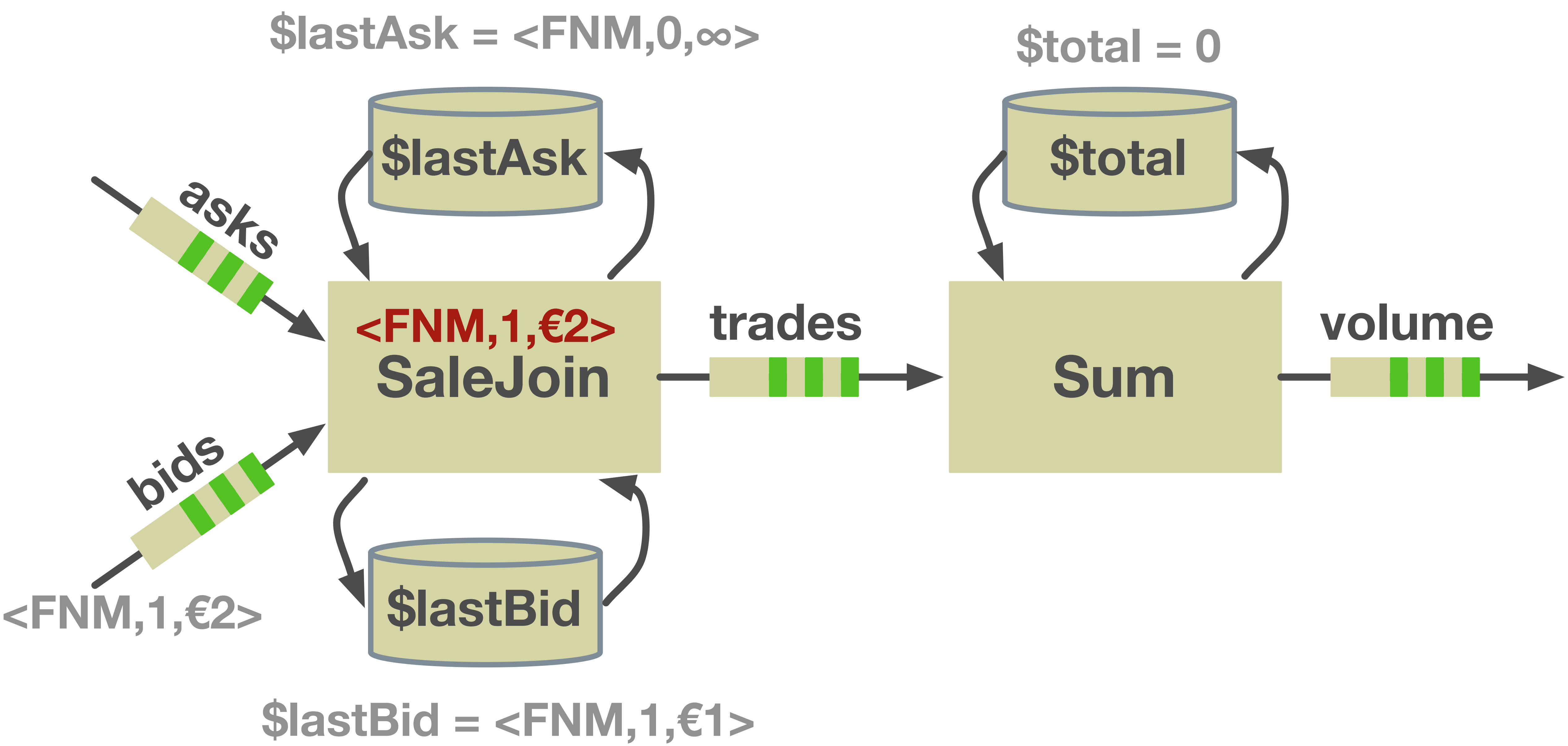
Example:
A Fannie Mae Bid/Ask Join



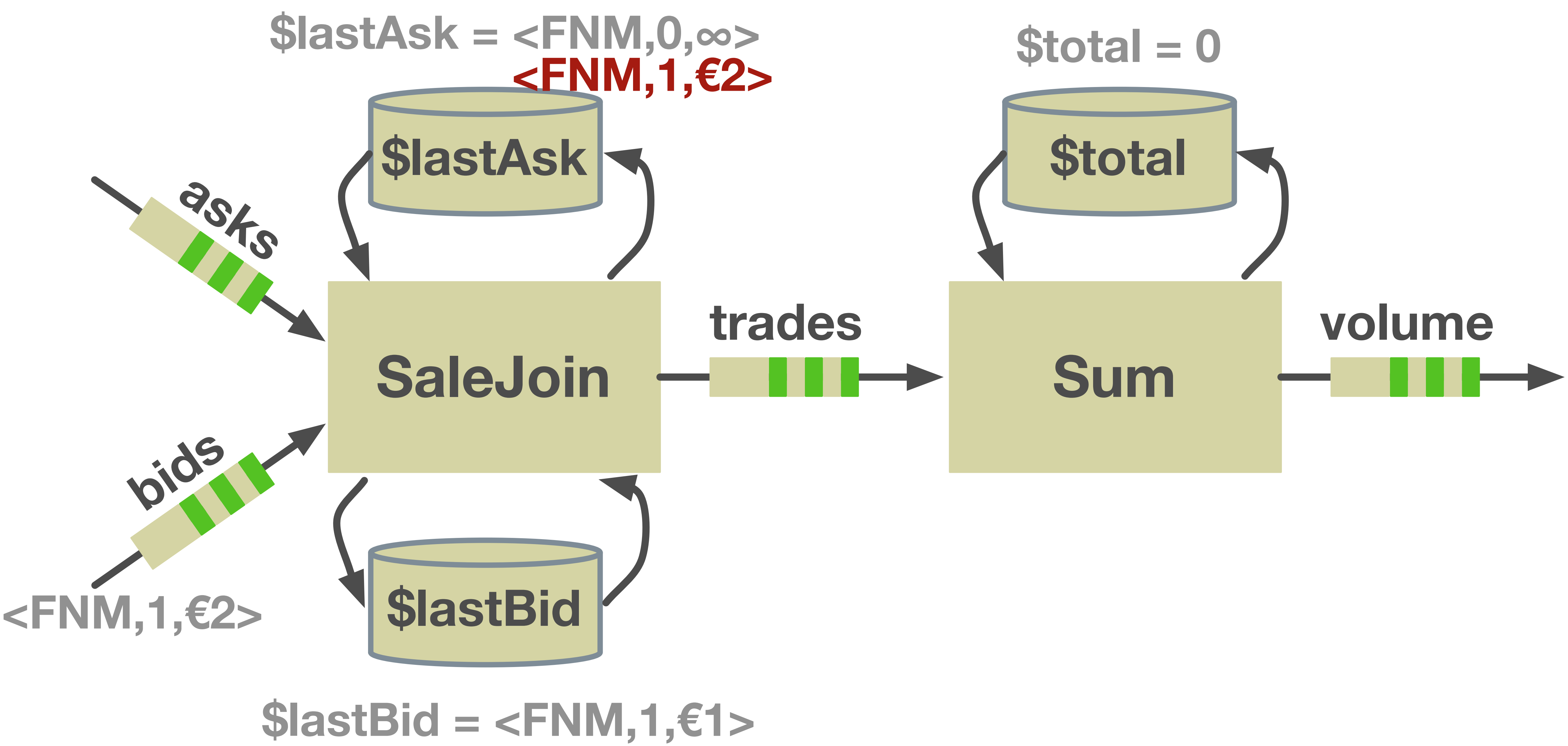
Example:
A Fannie Mae Bid/Ask Join



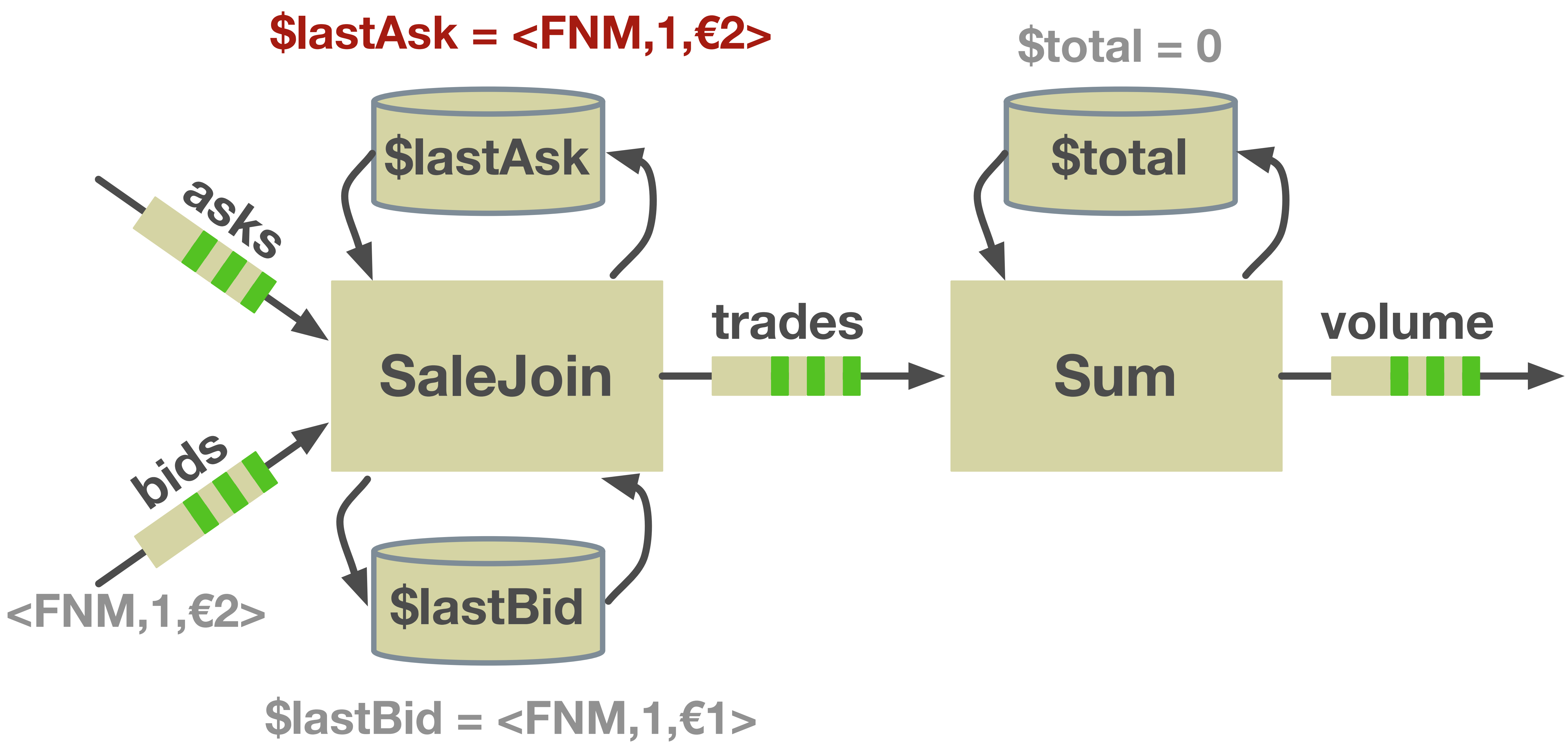
Example:
A Fannie Mae Bid/Ask Join



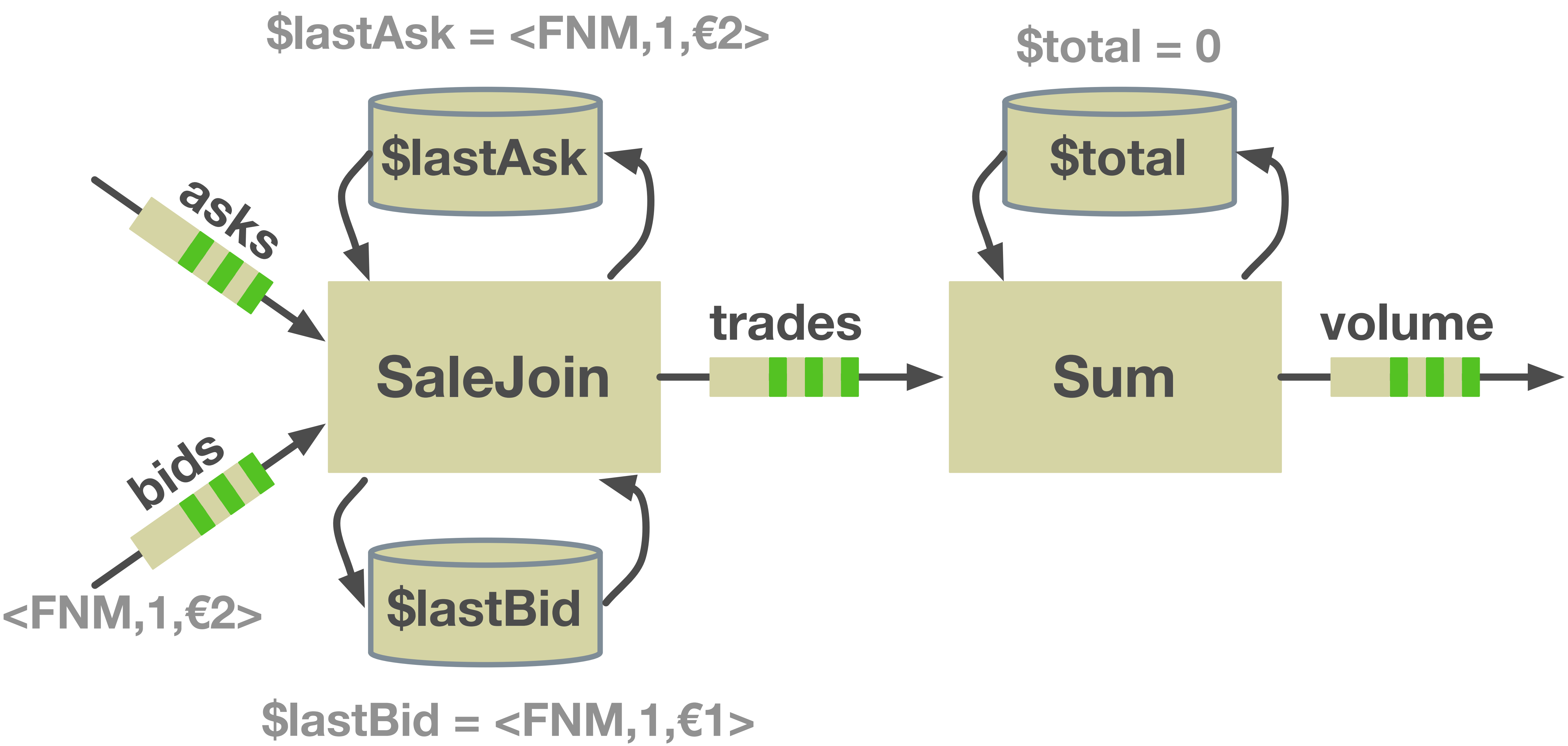
Example:
A Fannie Mae Bid/Ask Join



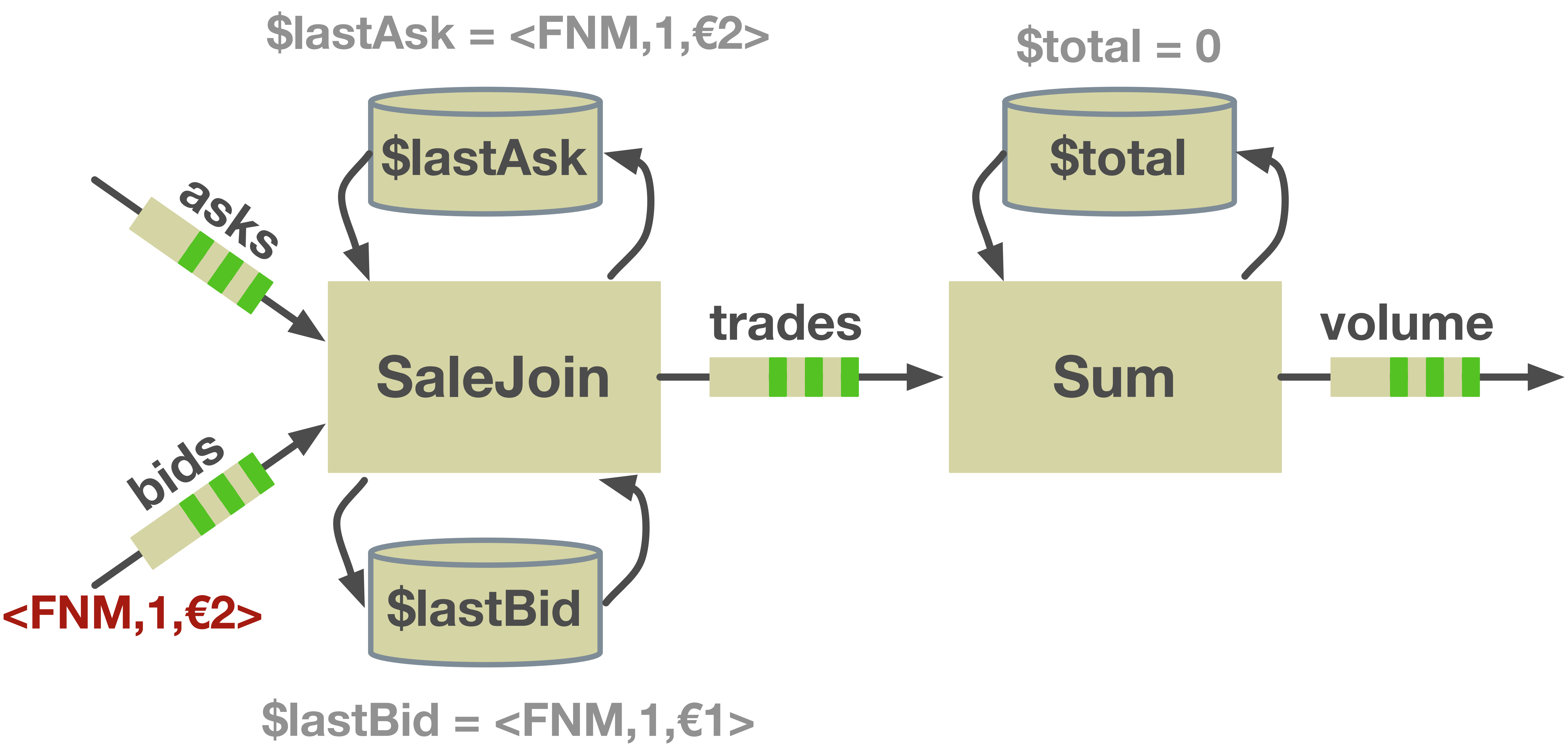
Example:
A Fannie Mae Bid/Ask Join



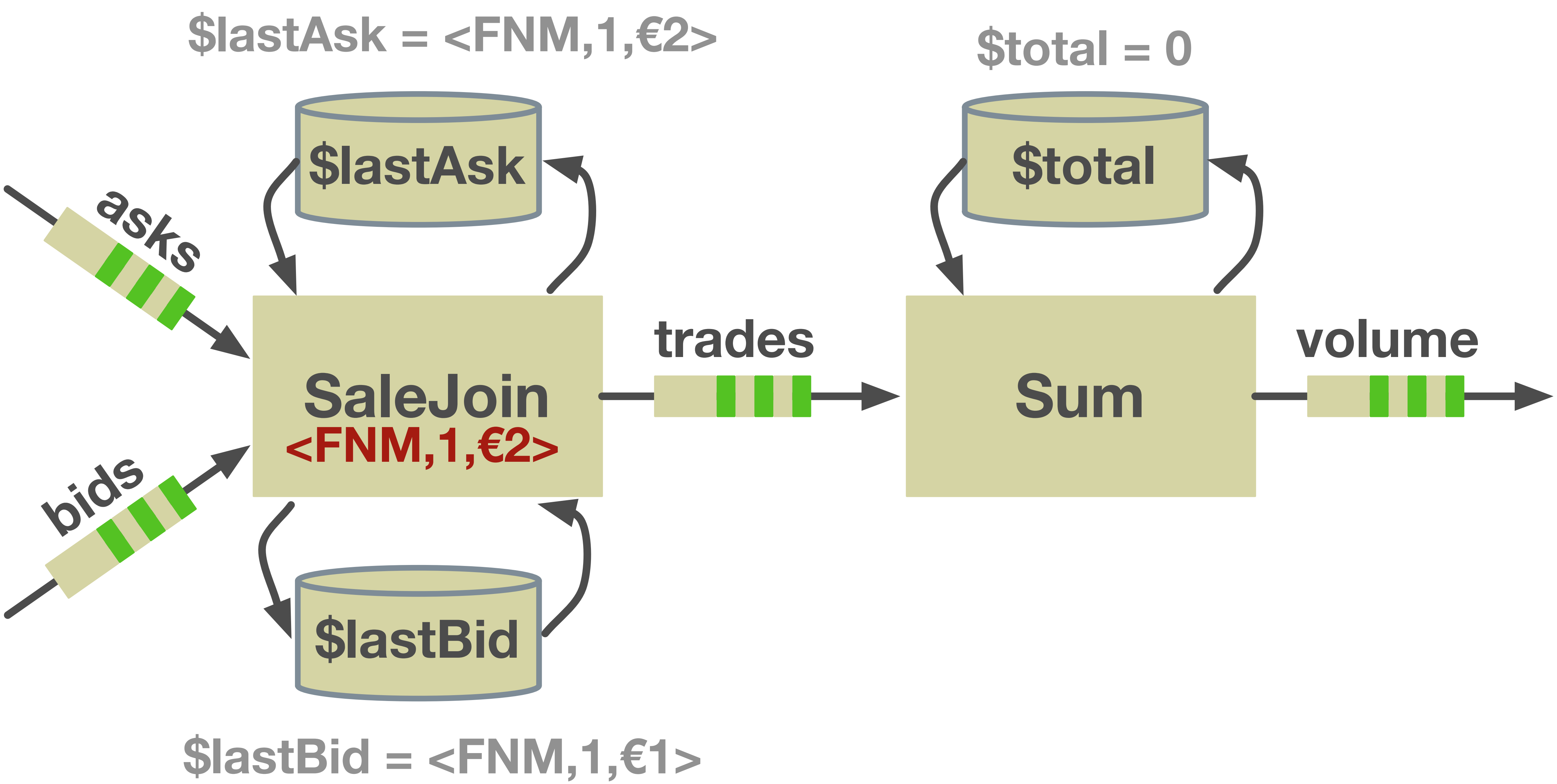
Example: A Fannie Mae Bid/Ask Join



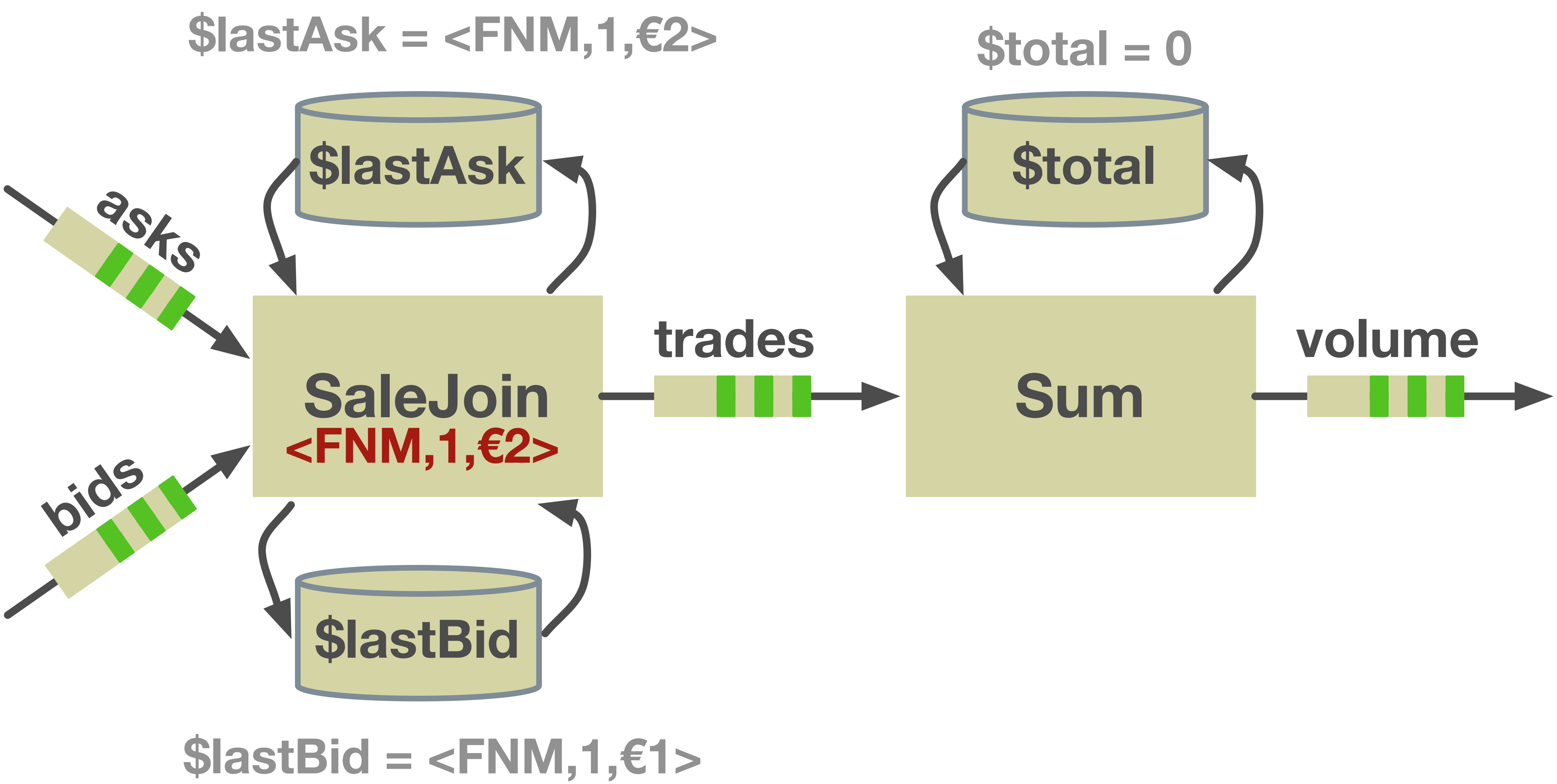
Example: A Fannie Mae Bid/Ask Join



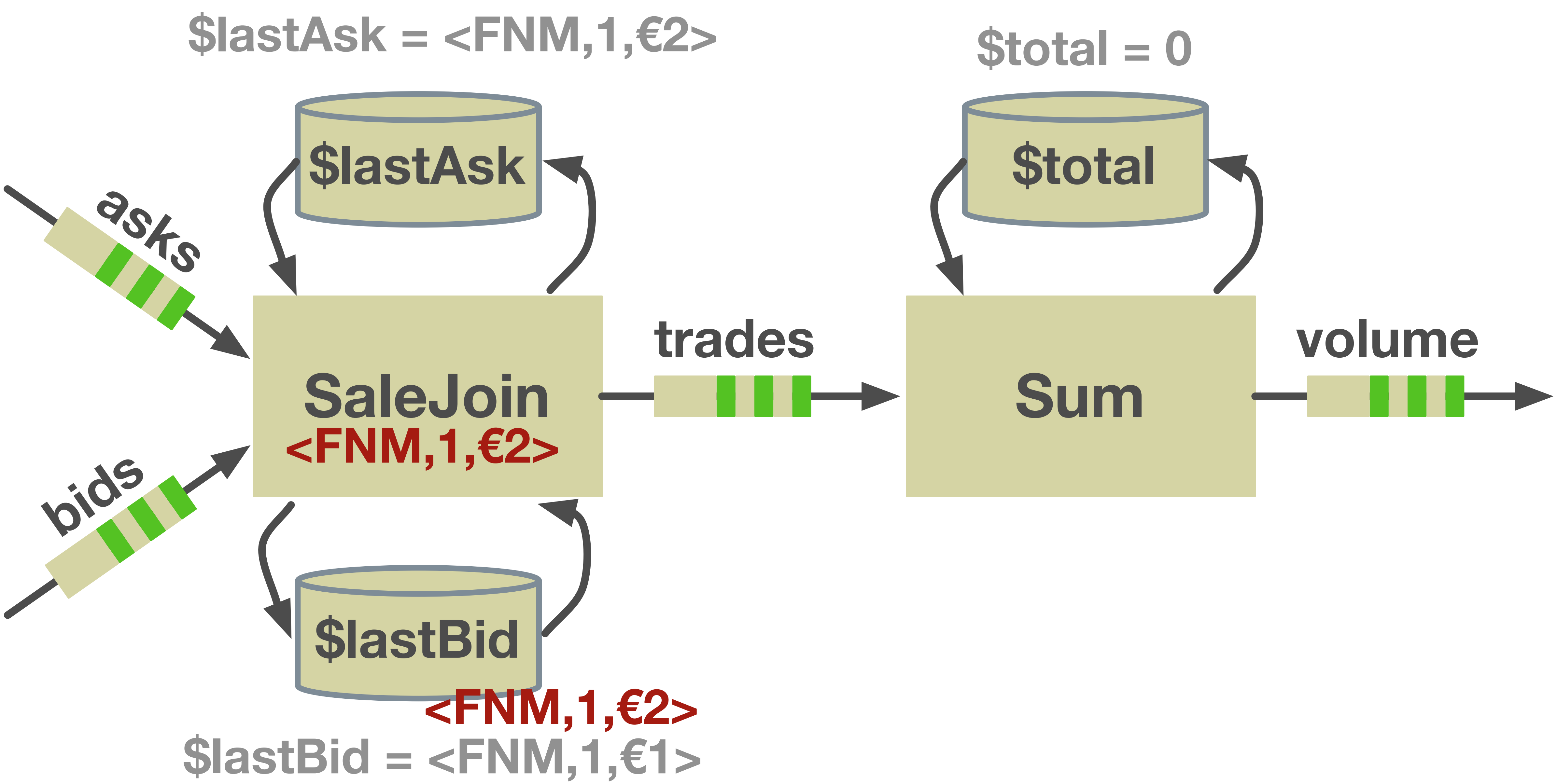
Example:
A Fannie Mae Bid/Ask Join



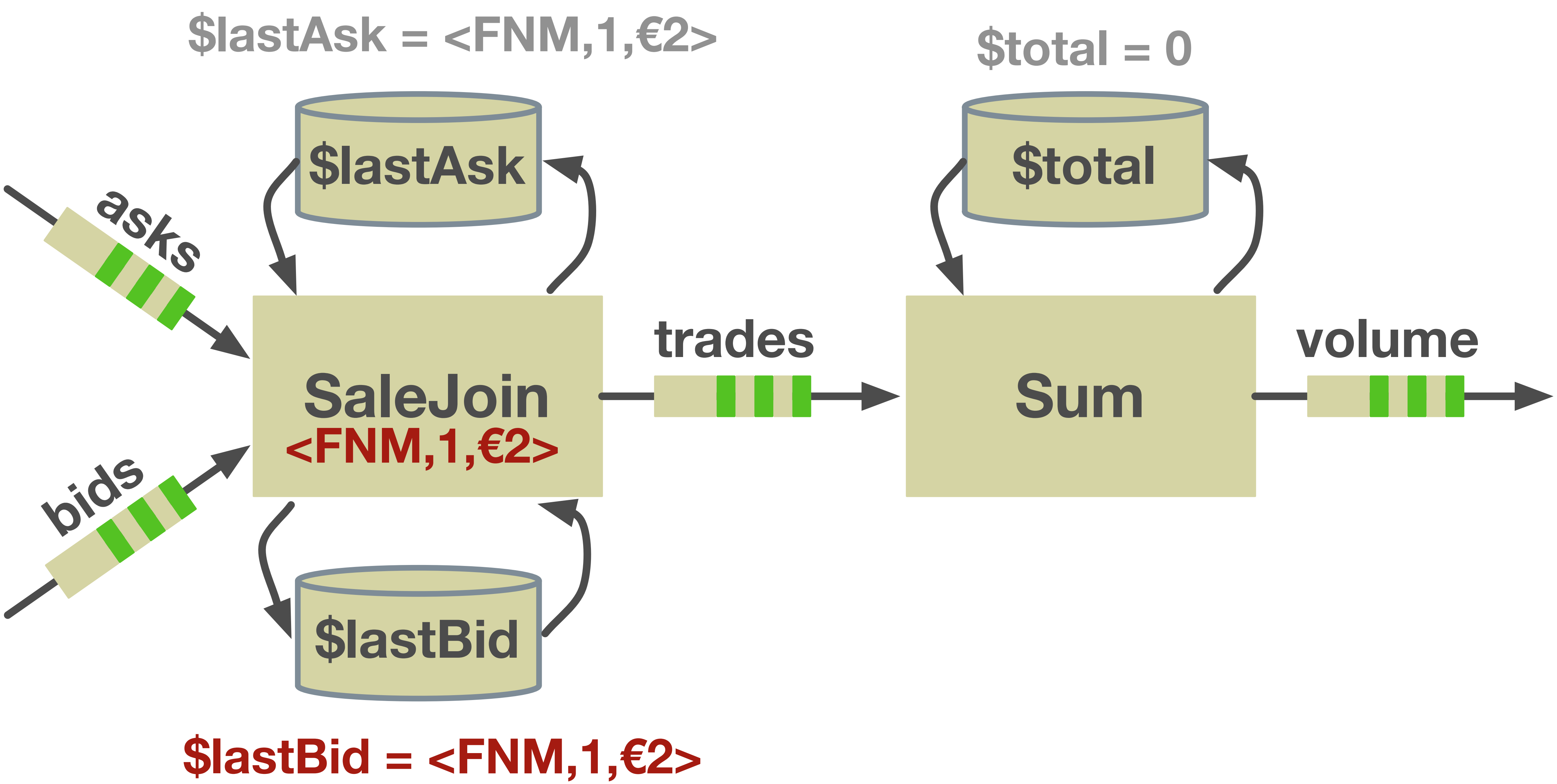
Example: A Fannie Mae Bid/Ask Join



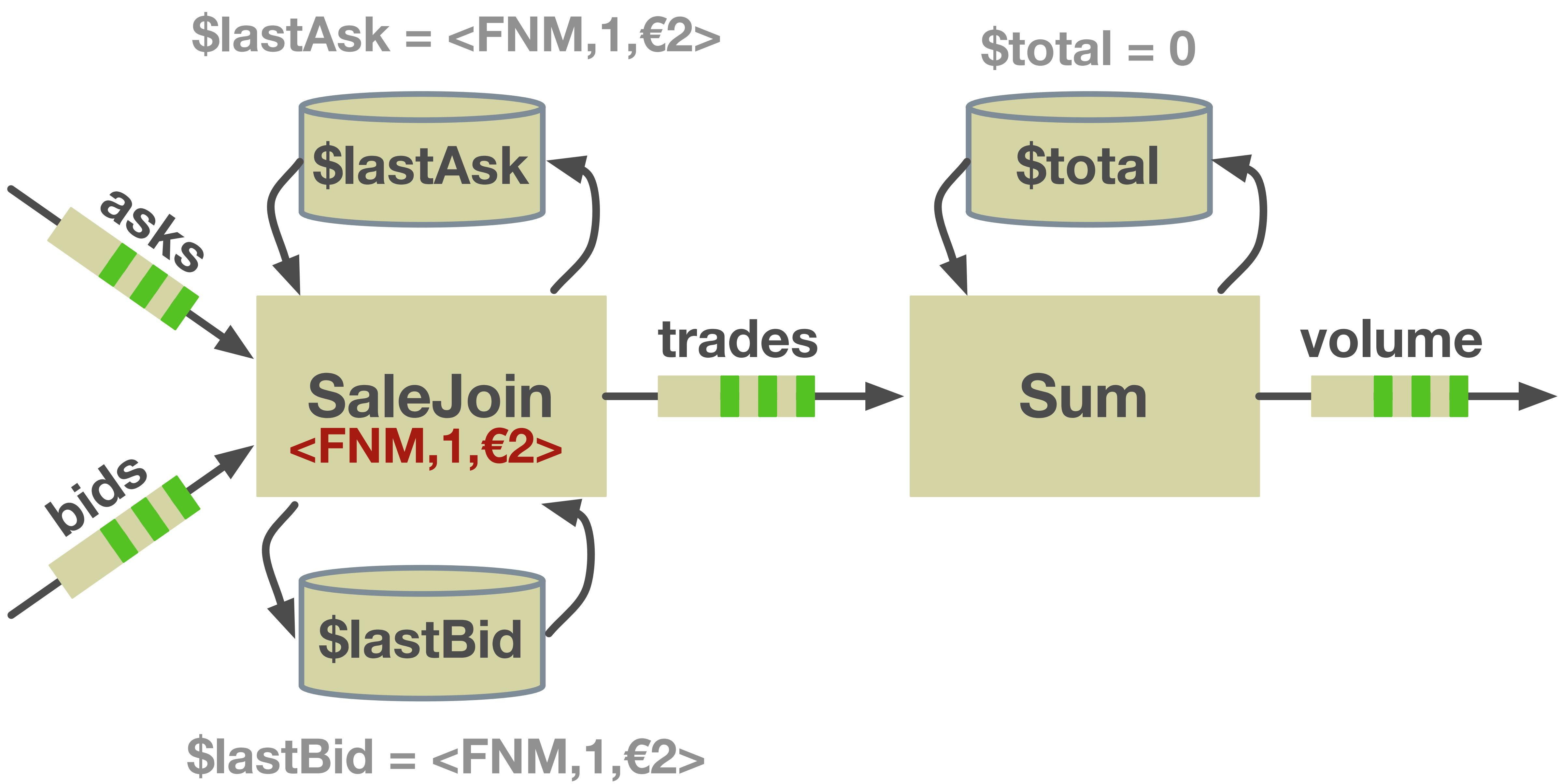
Example: A Fannie Mae Bid/Ask Join



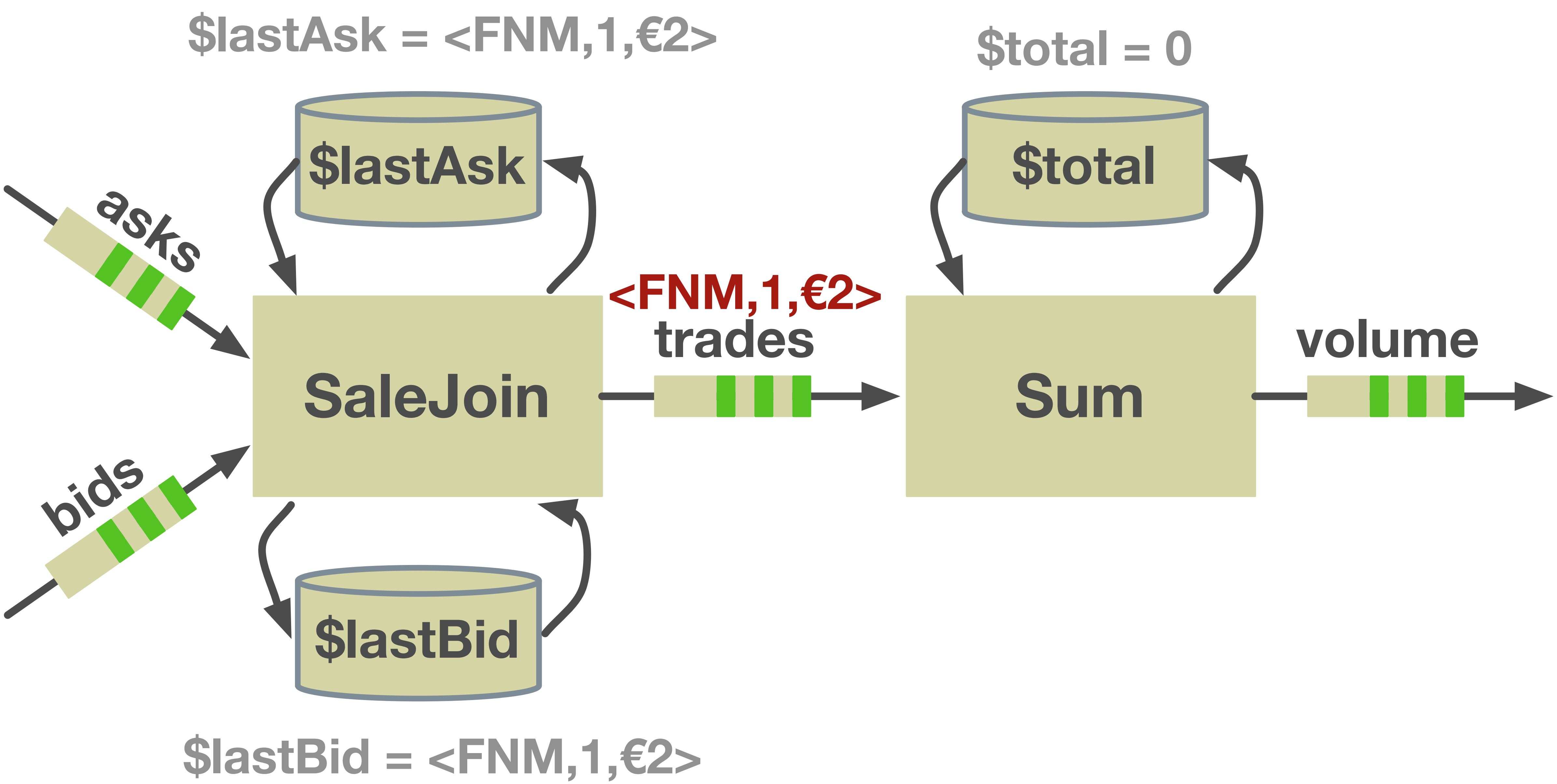
Example:
A Fannie Mae Bid/Ask Join



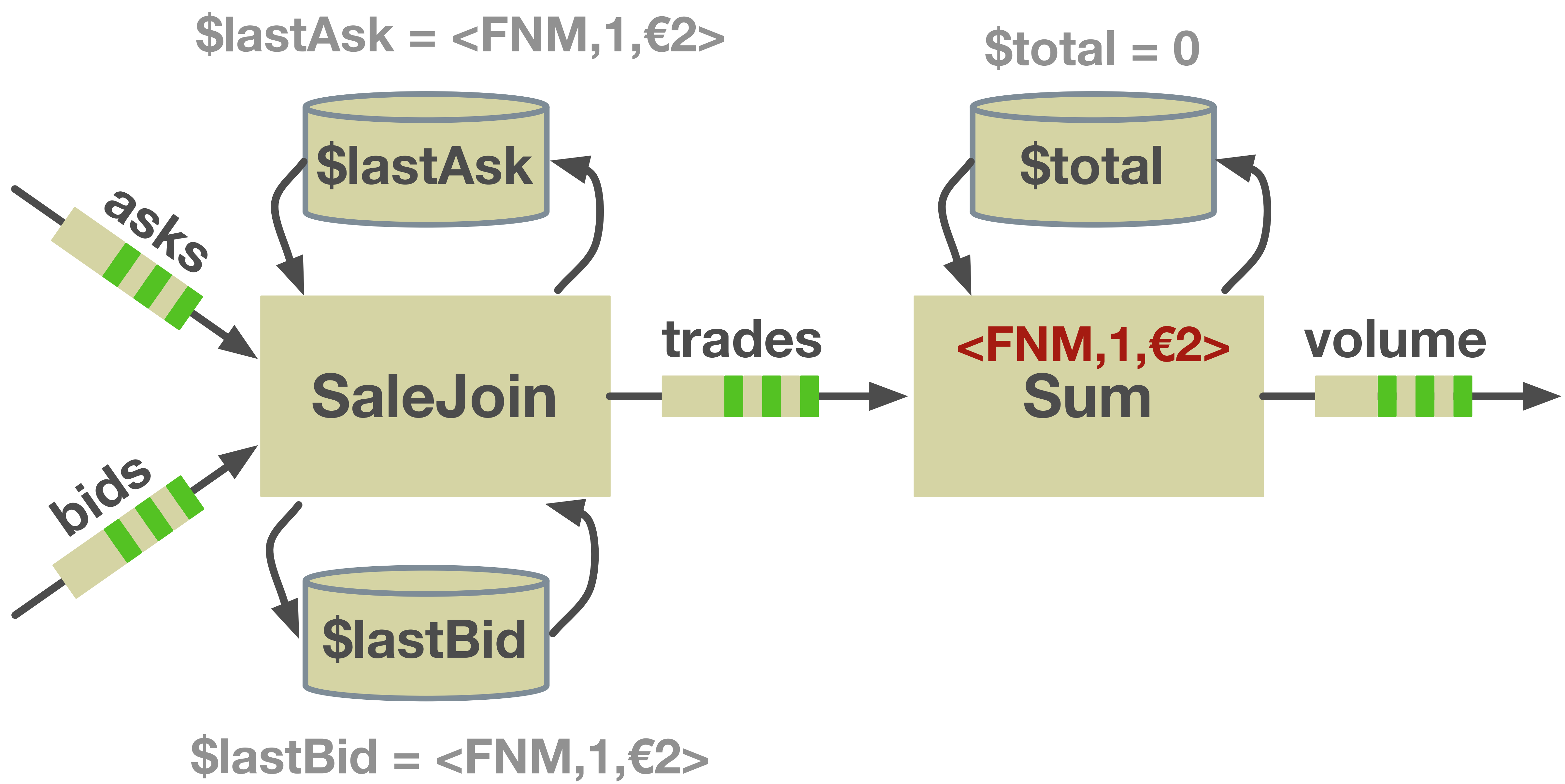
Example:
A Fannie Mae Bid/Ask Join



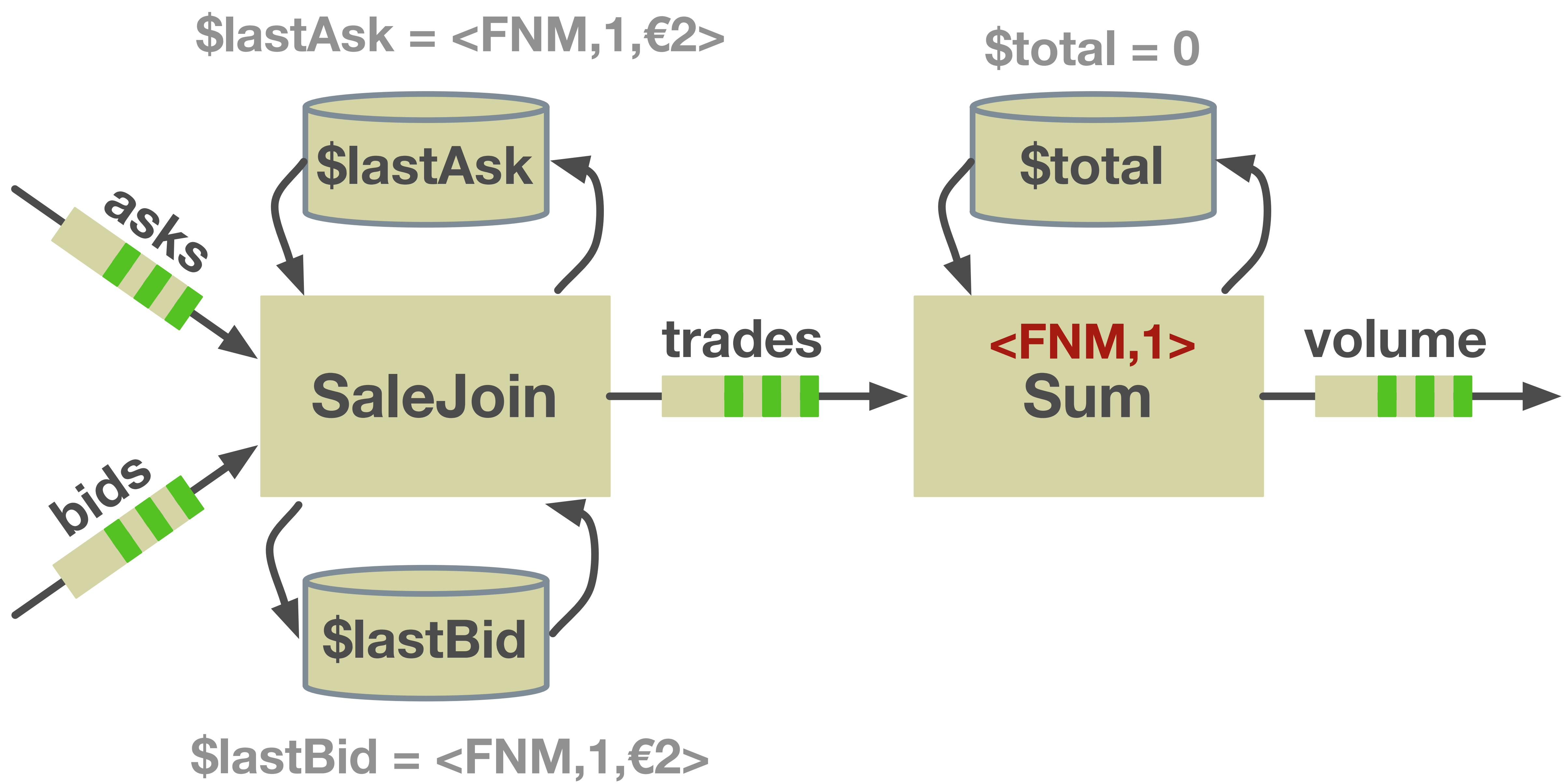
Example:
A Fannie Mae Bid/Ask Join



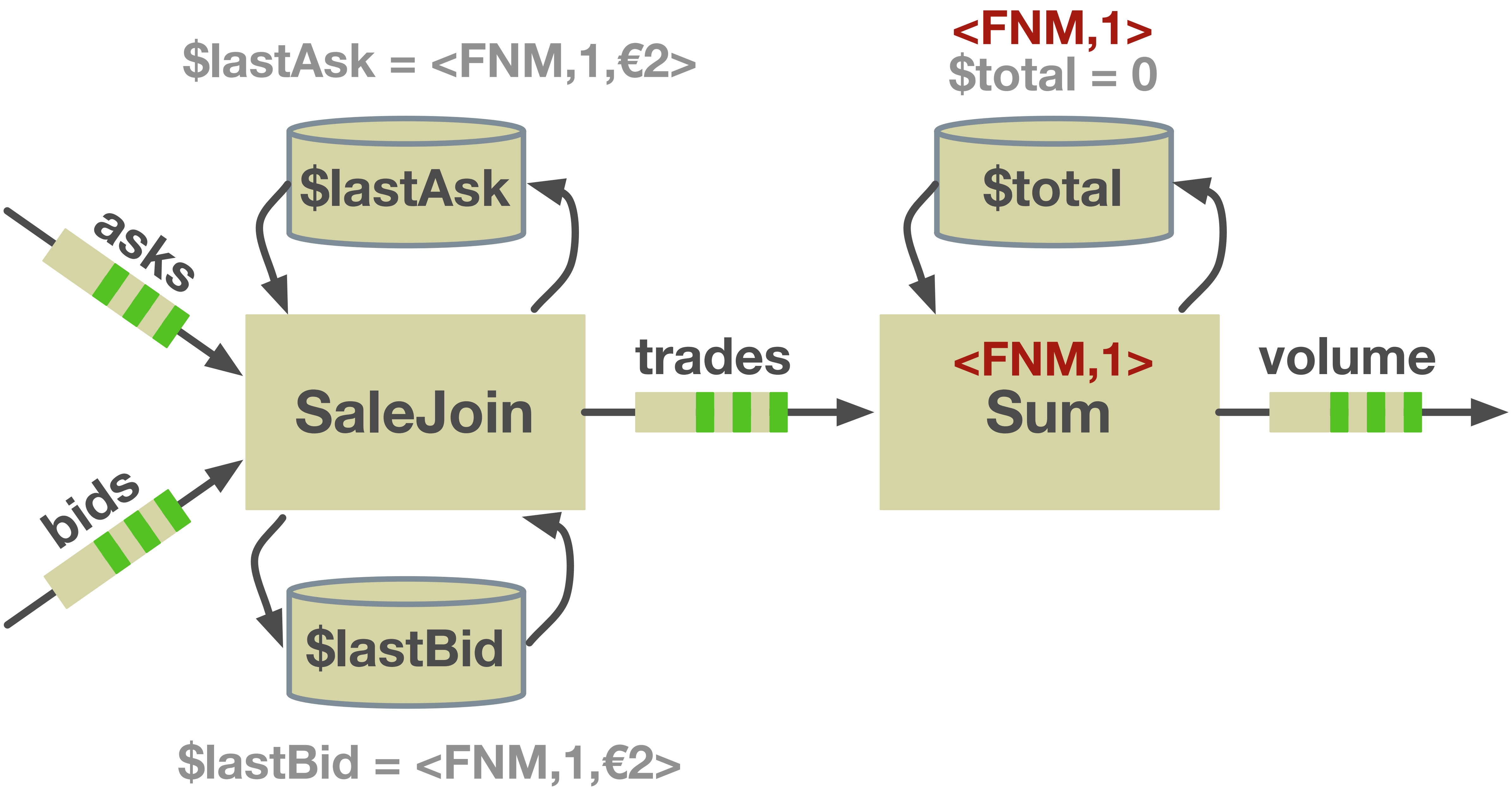
Example:
A Fannie Mae Bid/Ask Join



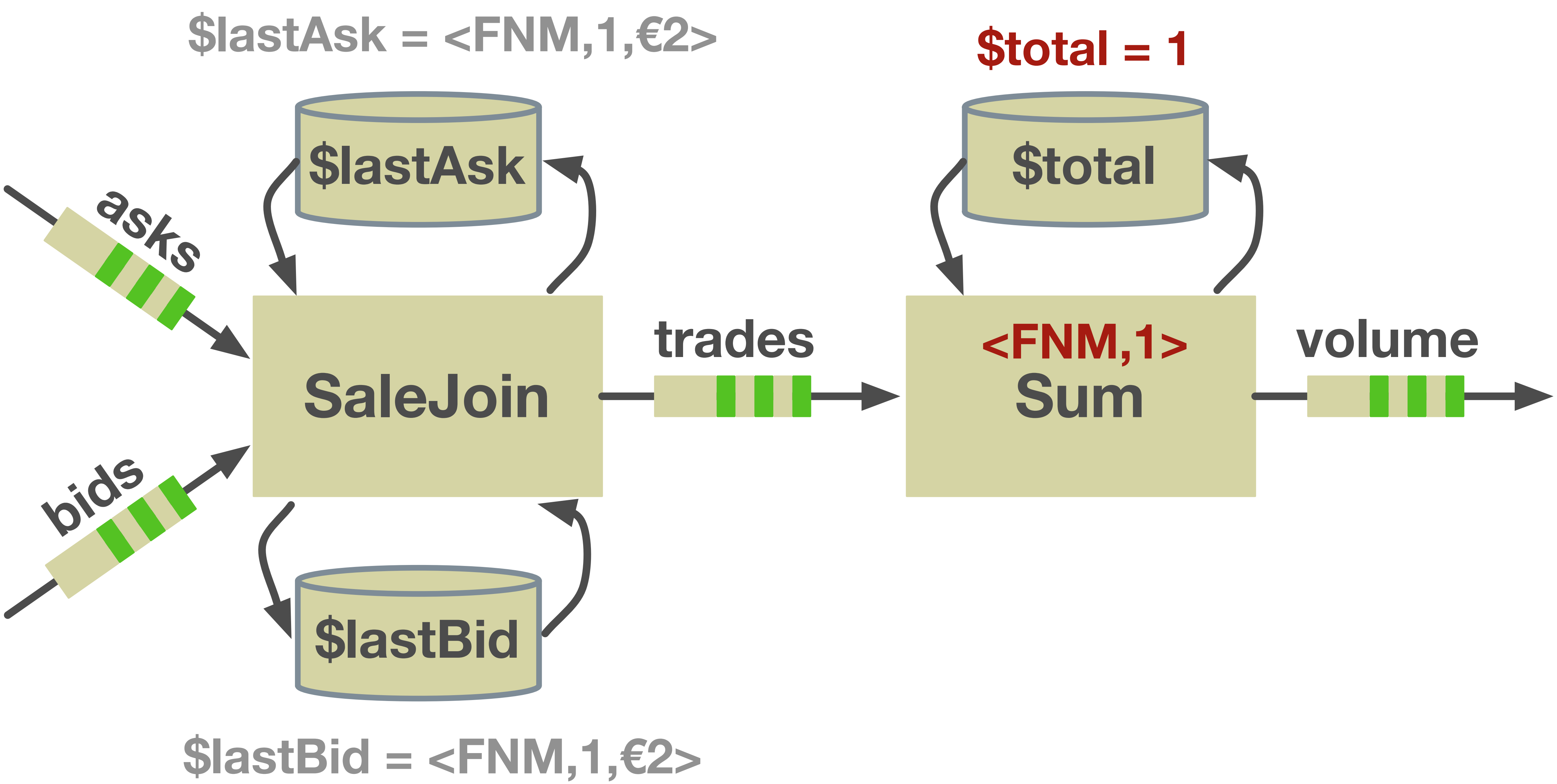
Example: A Fannie Mae Bid/Ask Join



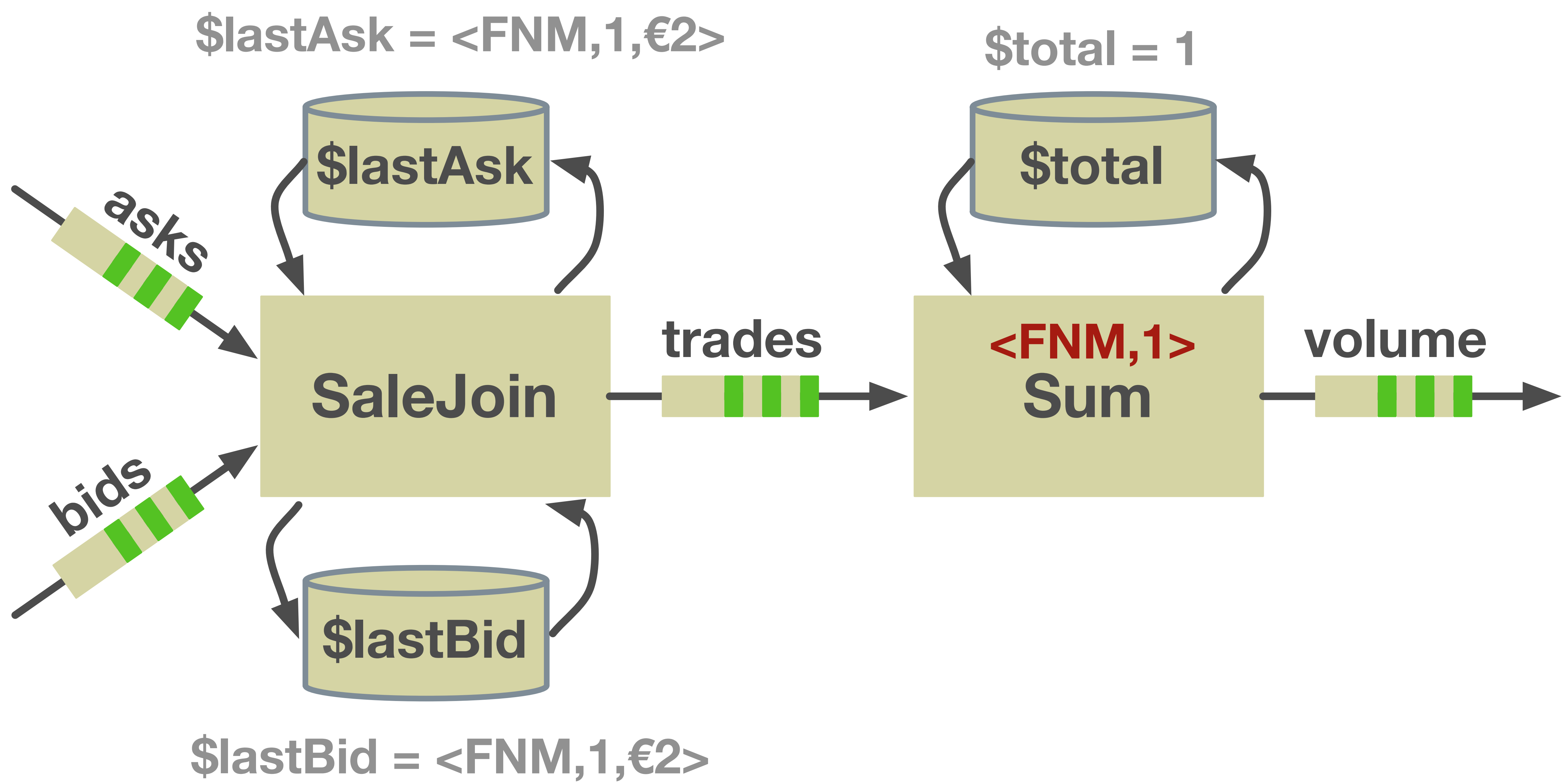
Example:
A Fannie Mae Bid/Ask Join



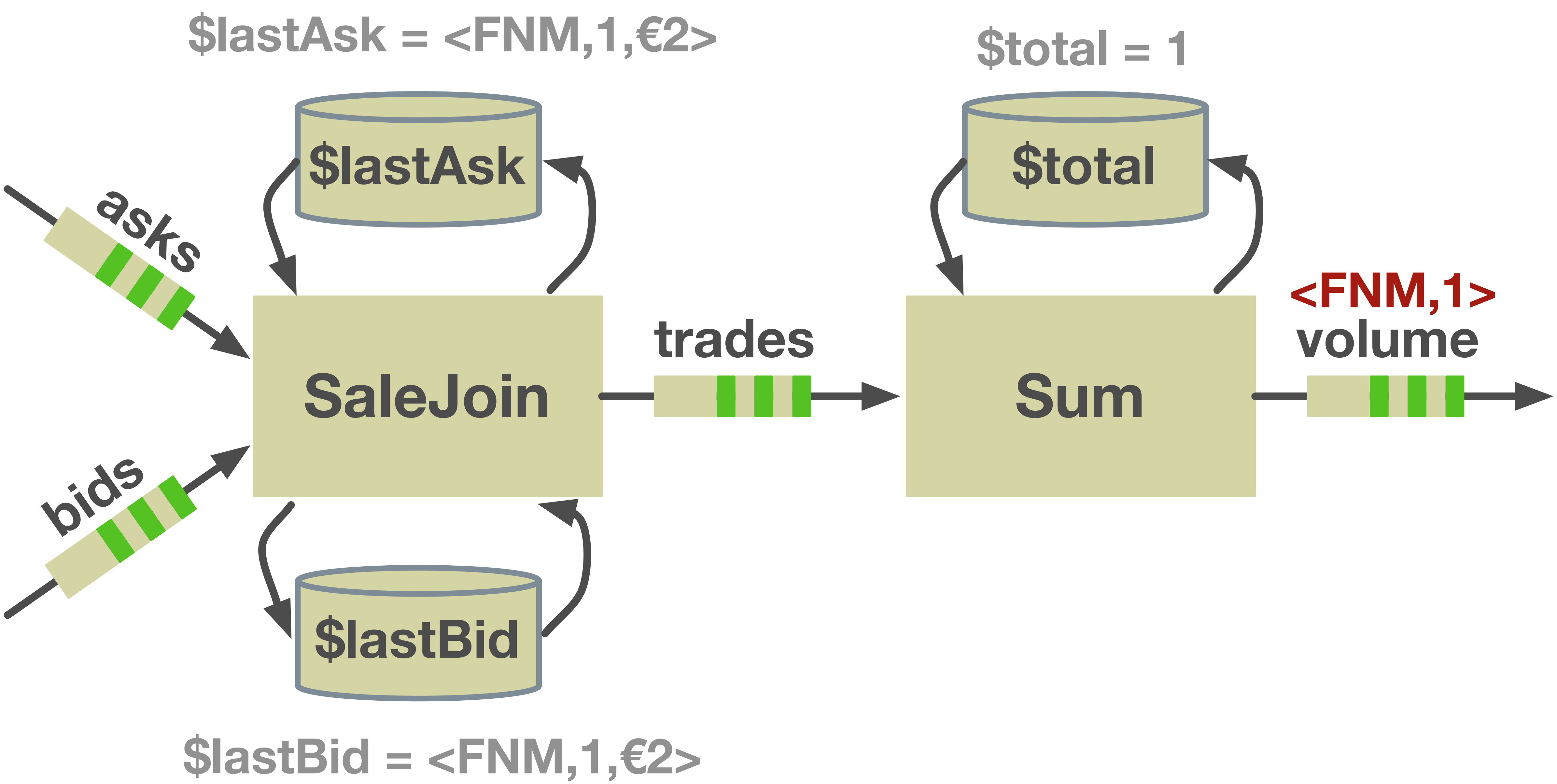
Example:
A Fannie Mae Bid/Ask Join



Example:
A Fannie Mae Bid/Ask Join



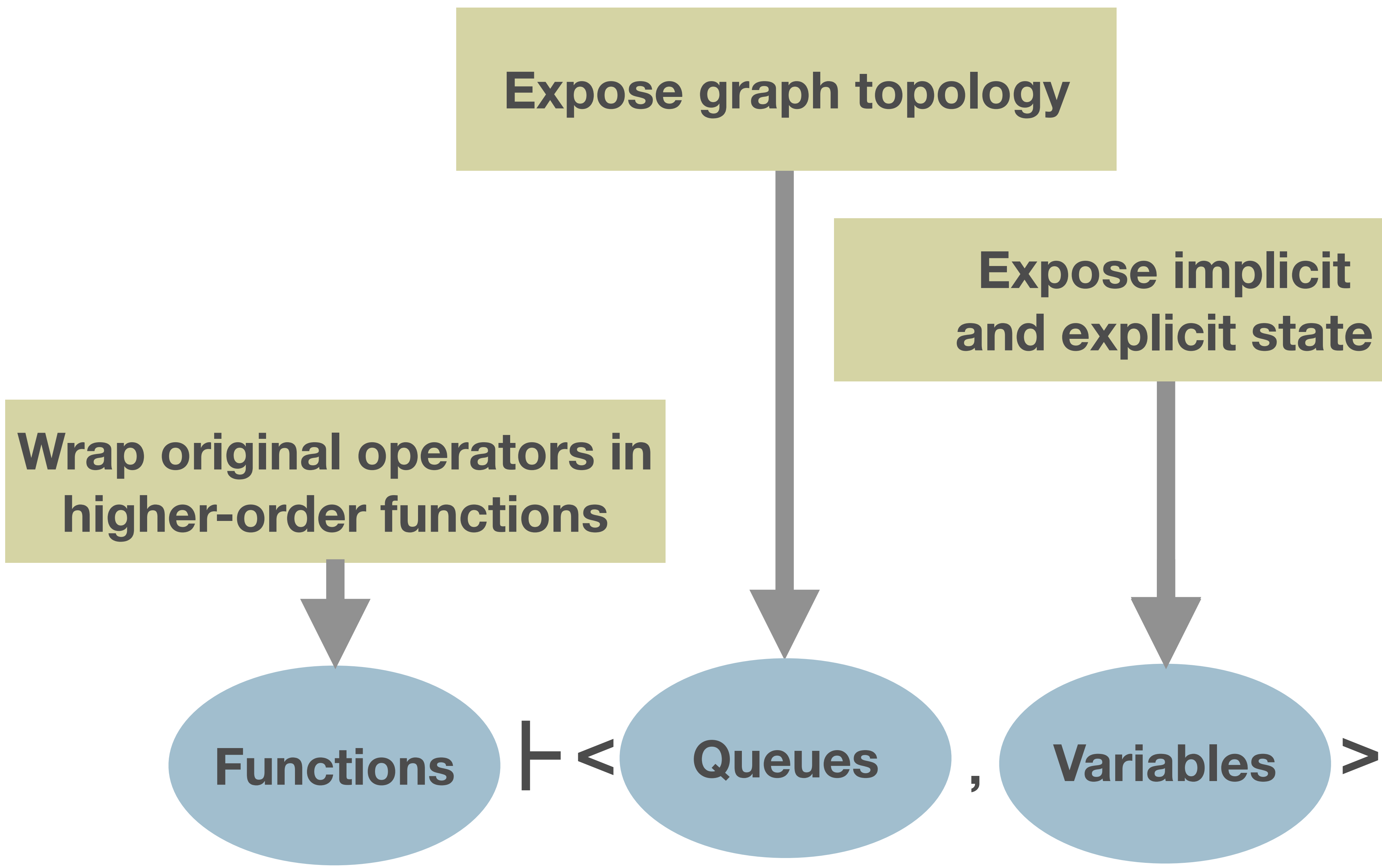
Example: A Fannie Mae Bid/Ask Join



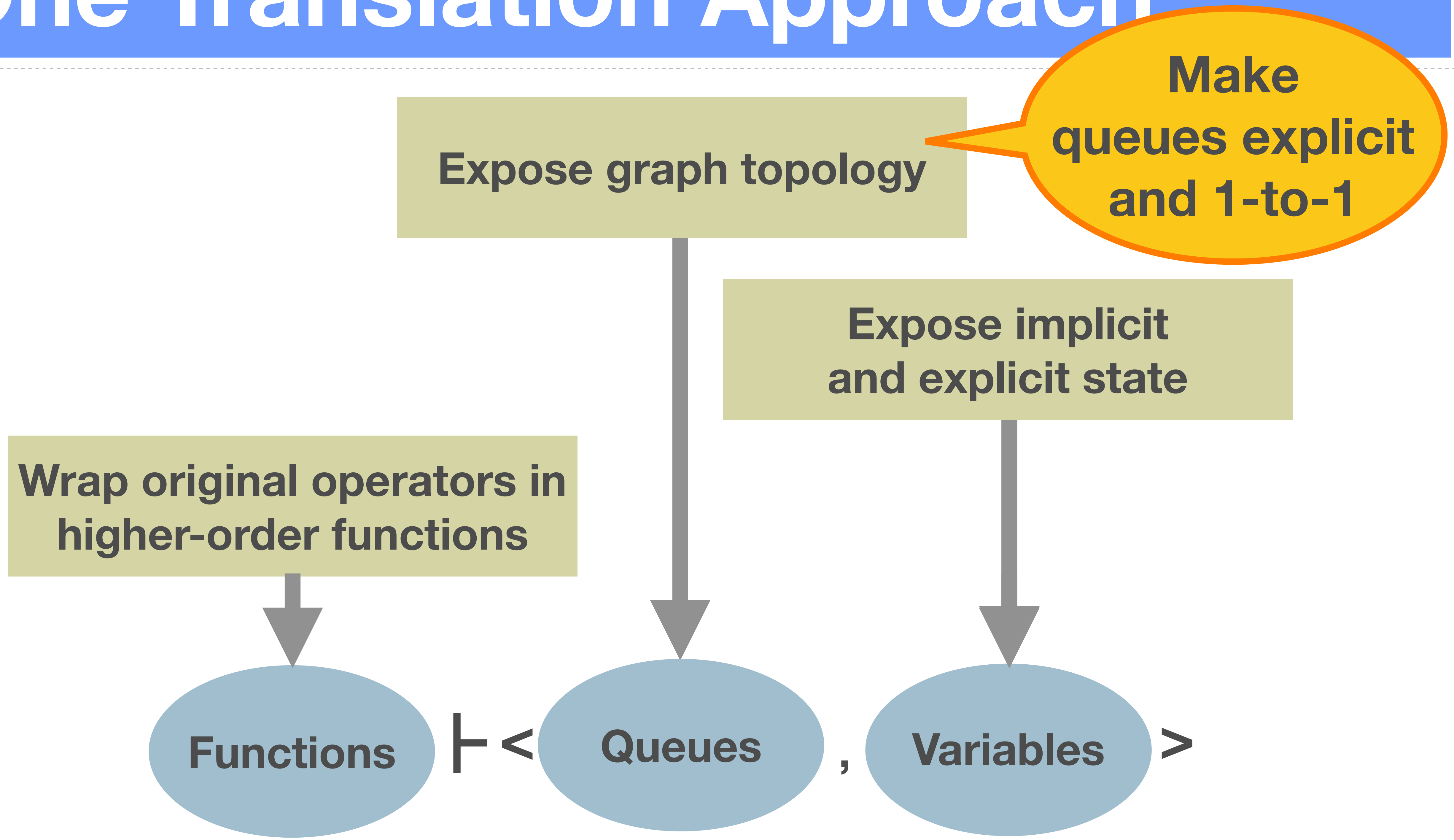
Translations

Demonstrating Brooklet's generality by translating three rather diverse streaming languages

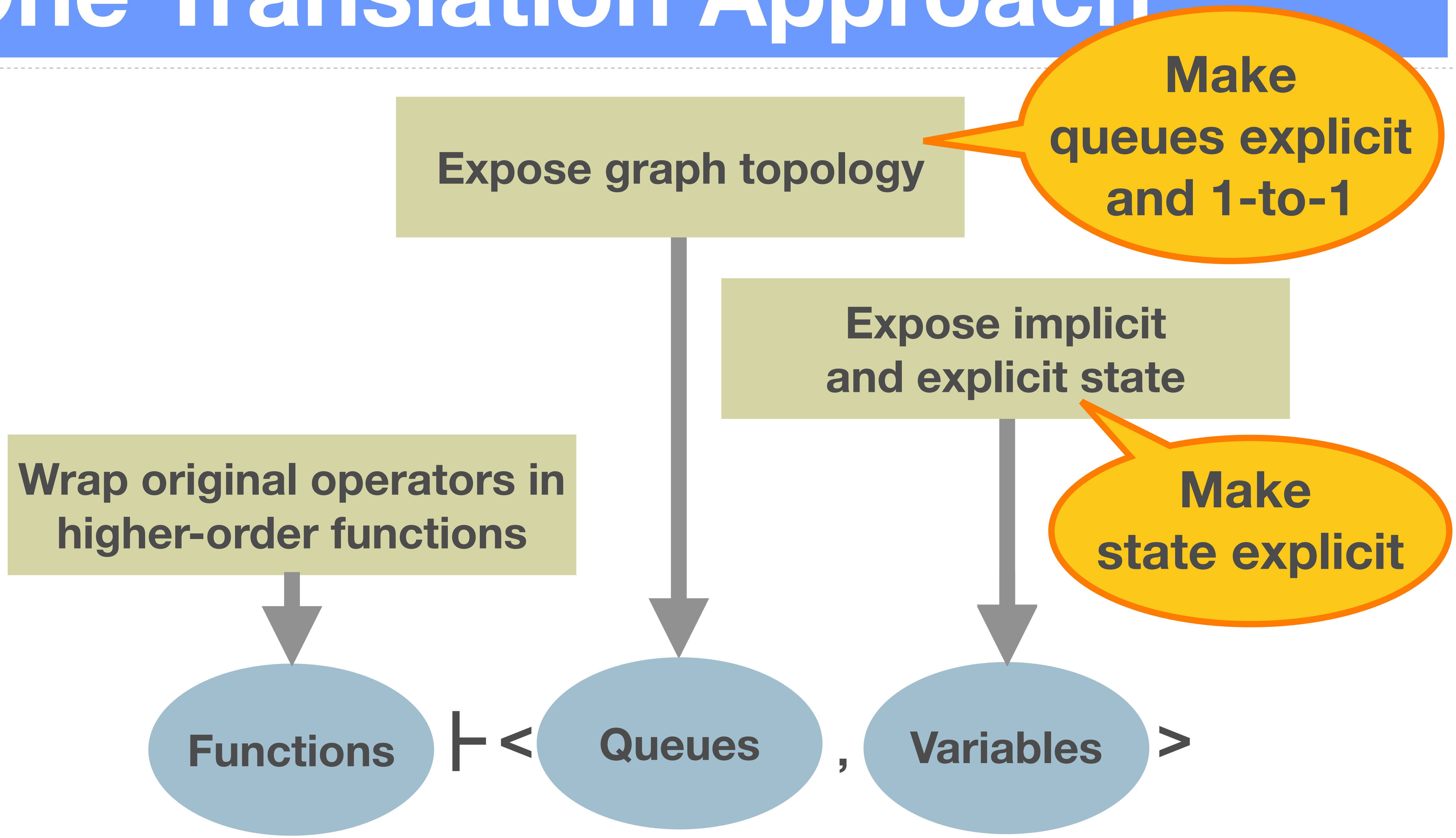
CQL, StreamIt, Sawzall: One Translation Approach



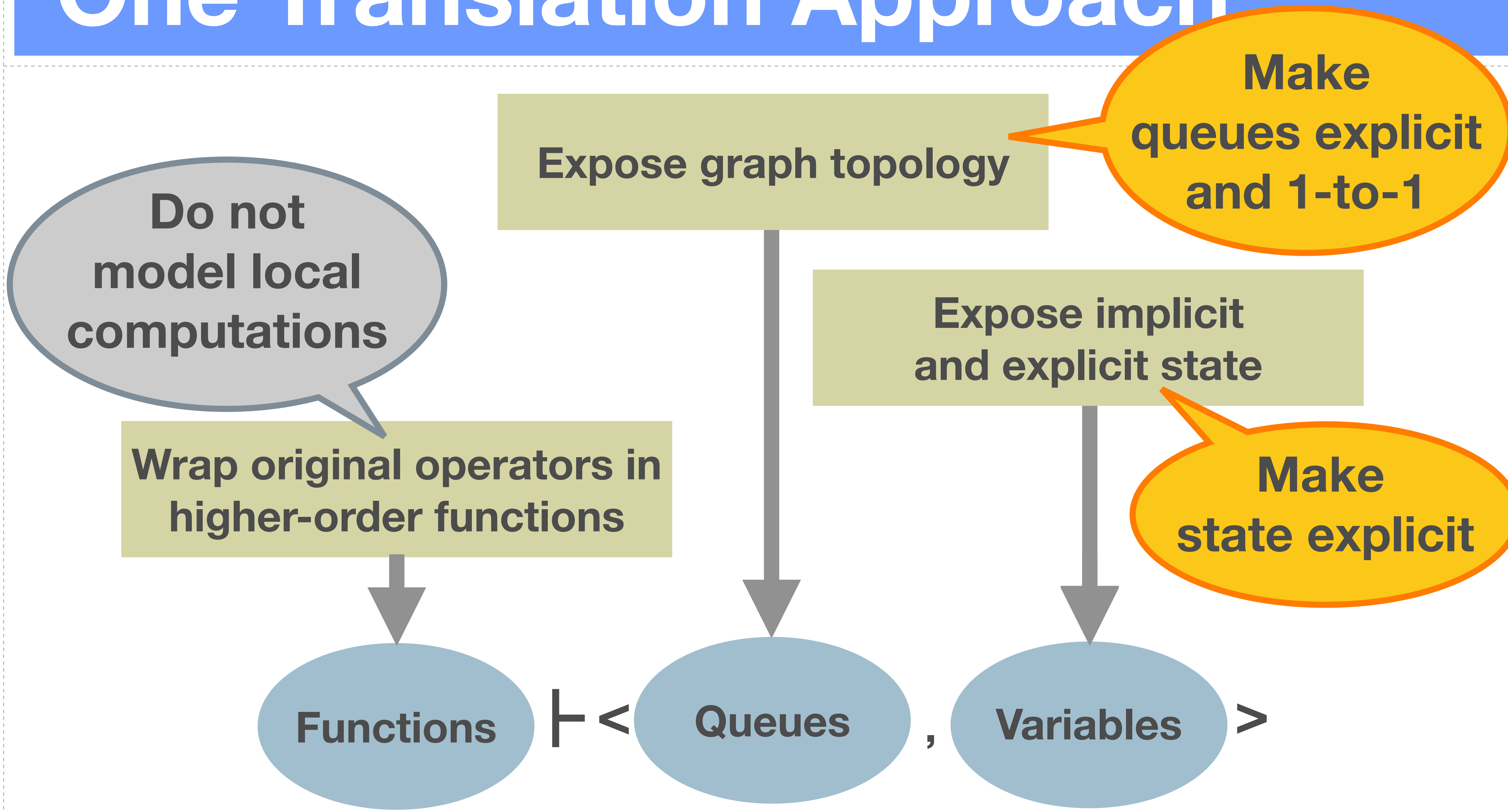
CQL, StreamIt, Sawzall: One Translation Approach



CQL, StreamIt, Sawzall: One Translation Approach



CQL, StreamIt, Sawzall: One Translation Approach

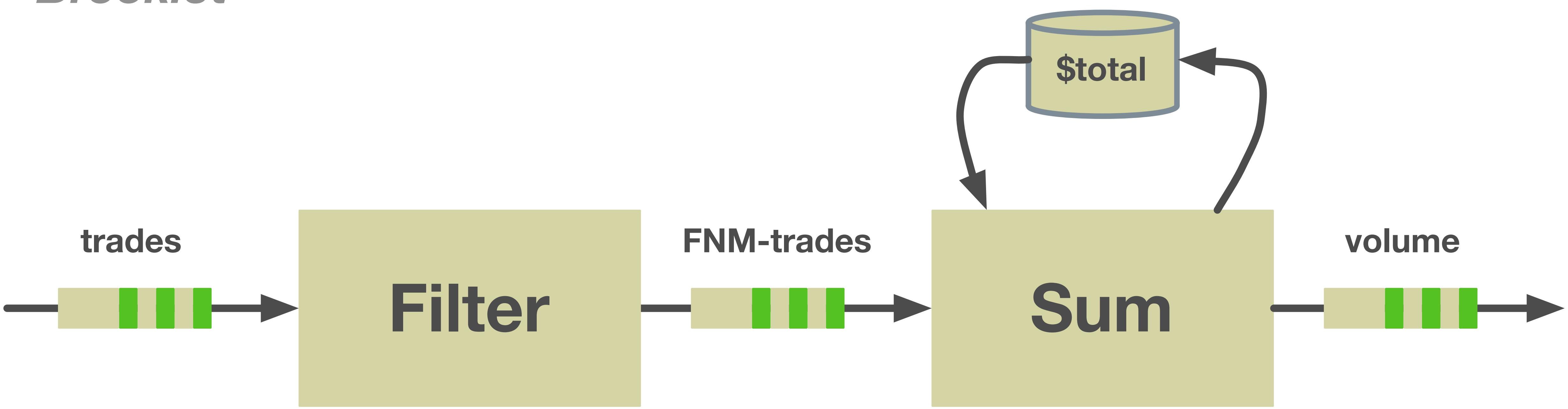


Example: CQL to Brooklet

select *Sum(shares)* from *trades*
where *trades.ticker* = *"FNM"*

CQL

Brooklet

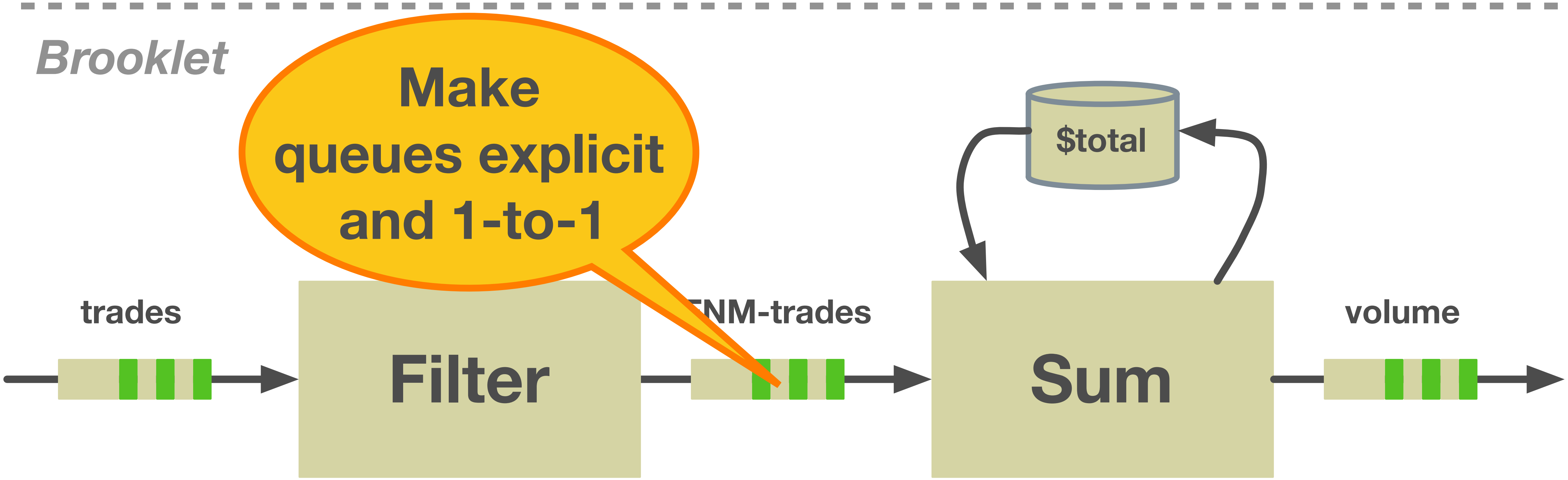


Example: CQL to Brooklet

select *Sum(shares)* from *trades*
where *trades.ticker* = *"FNM"*

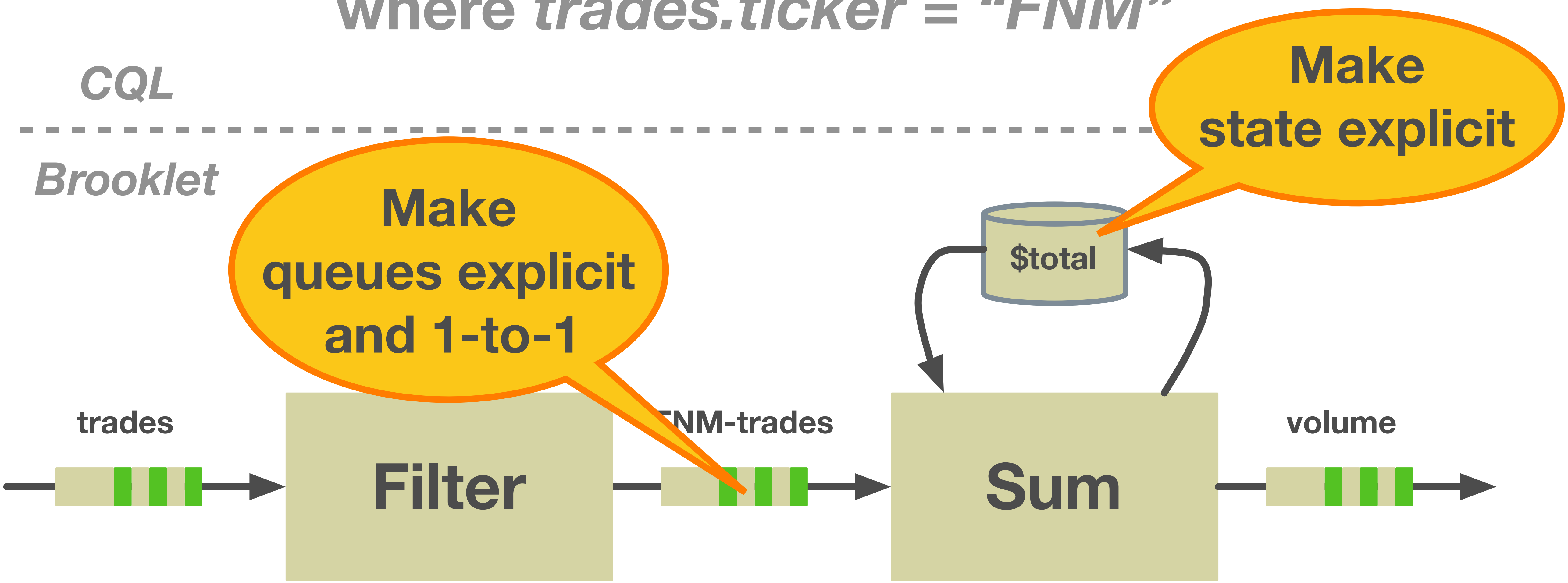
CQL

Brooklet



Example: CQL to Brooklet

select *Sum(shares)* from *trades*
where *trades.ticker* = *"FNM"*

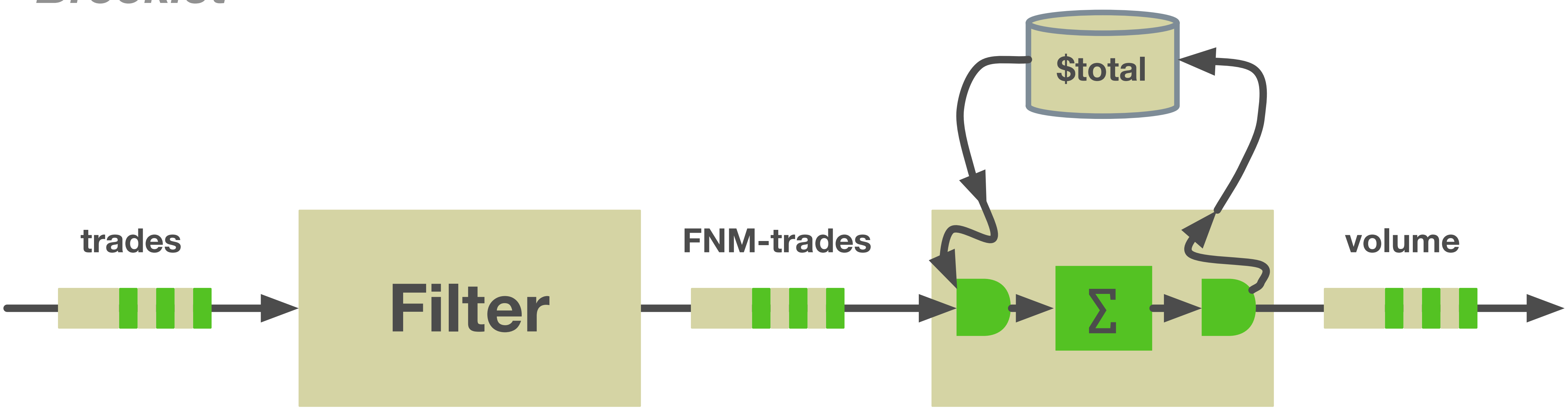


Example: CQL to Brooklet

select *Sum(shares)* from *trades*
where *trades.ticker* = *“FNM”*

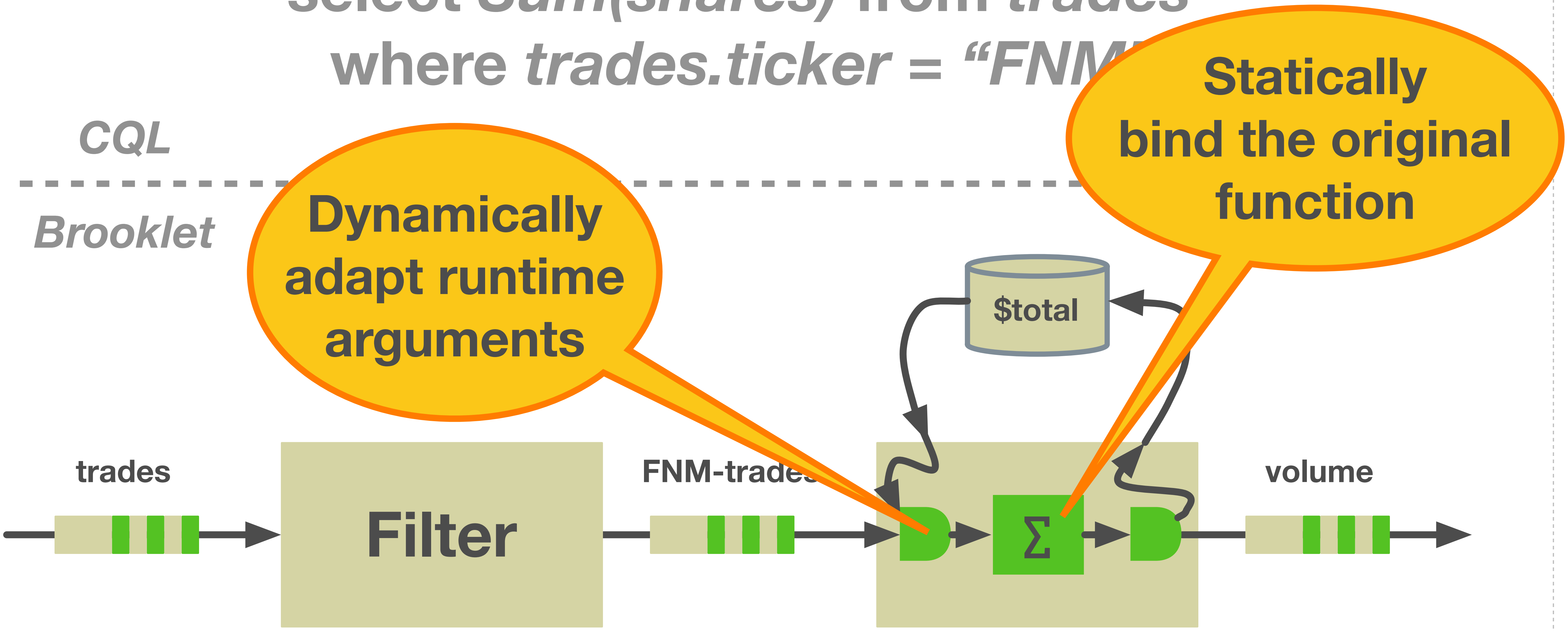
CQL

Brooklet

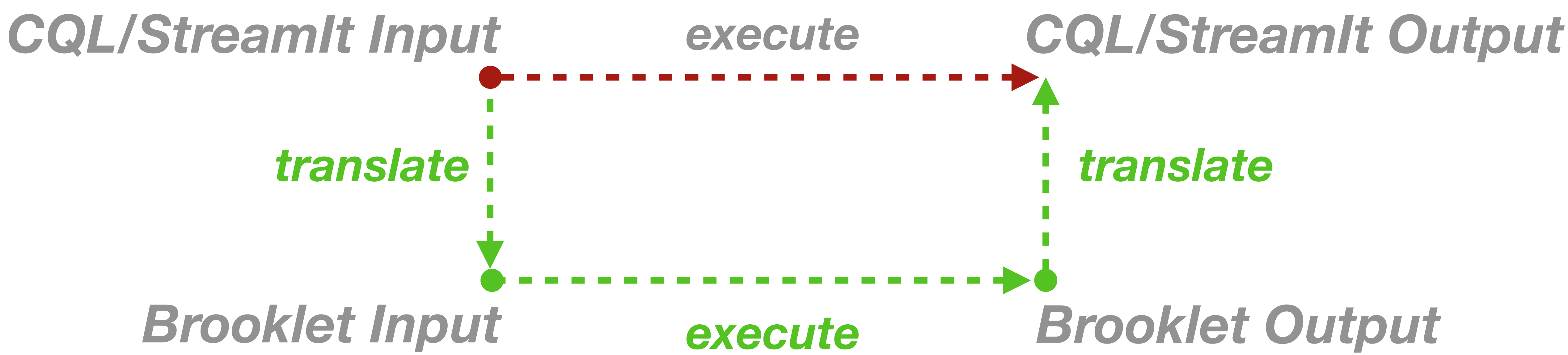


Example: CQL to Brooklet

select *Sum(shares)* from *trades*
where *trades.ticker* = *"FNM"*



Translation Correctness Theorem

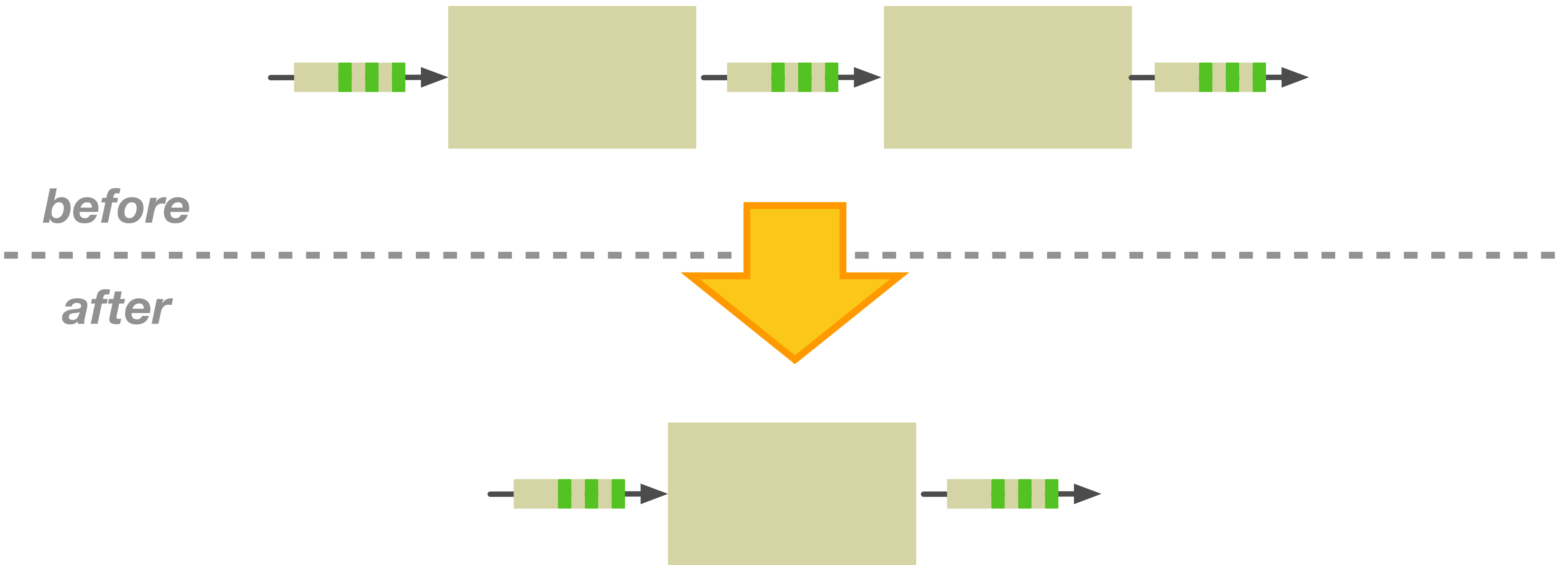


- Results under CQL and StreamIt semantics are the same as the results under Brooklet semantics after translation
- First formal semantics for Sawzall

Optimizations

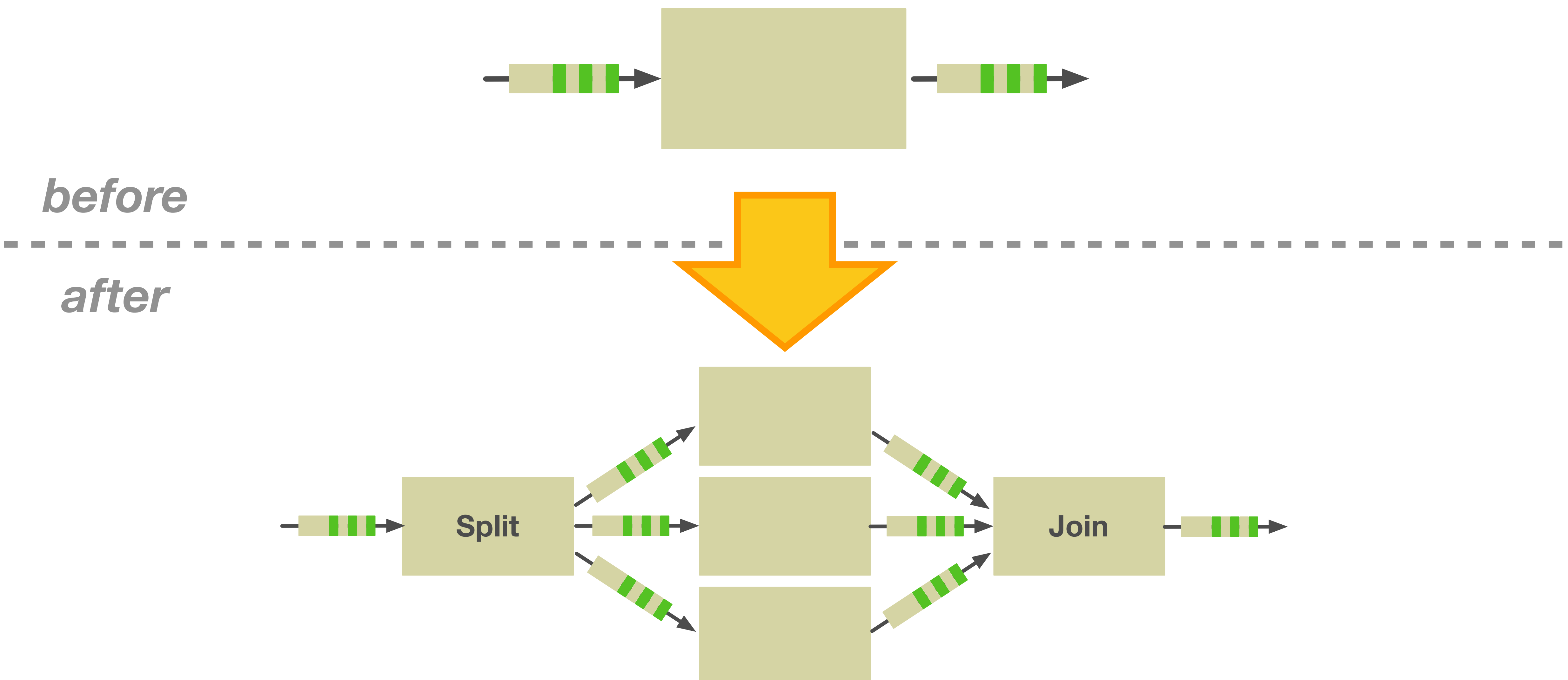
Demonstrating Brooklet's utility by realizing three essential optimizations

Operator Fusion: Eliminate Queueing Delays



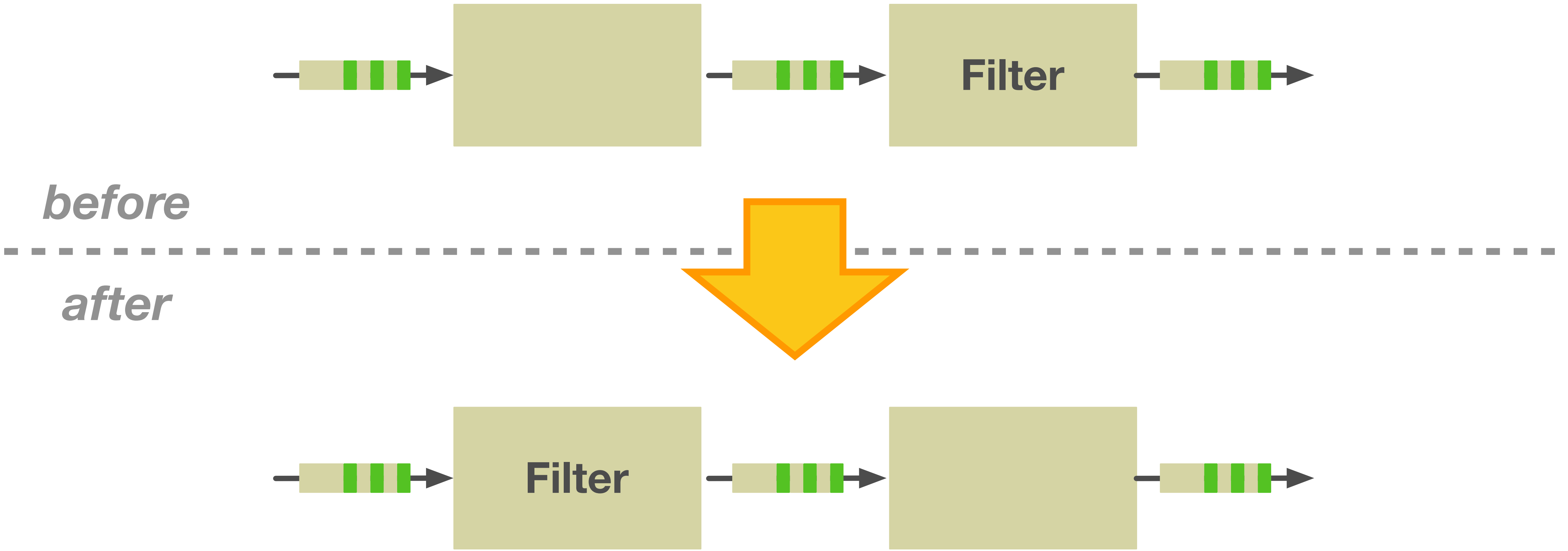
Look for connected operators, whose state isn't used anywhere else

Operator Fission: Process More Data in Parallel



Look for stateless operators

Operator Reordering: Filter Data Early



Look for operators whose read/write sets don't overlap [Ghelli et al., SIGMOD 08]

From a Calculus to an Intermediate Language

The River Intermediate Language

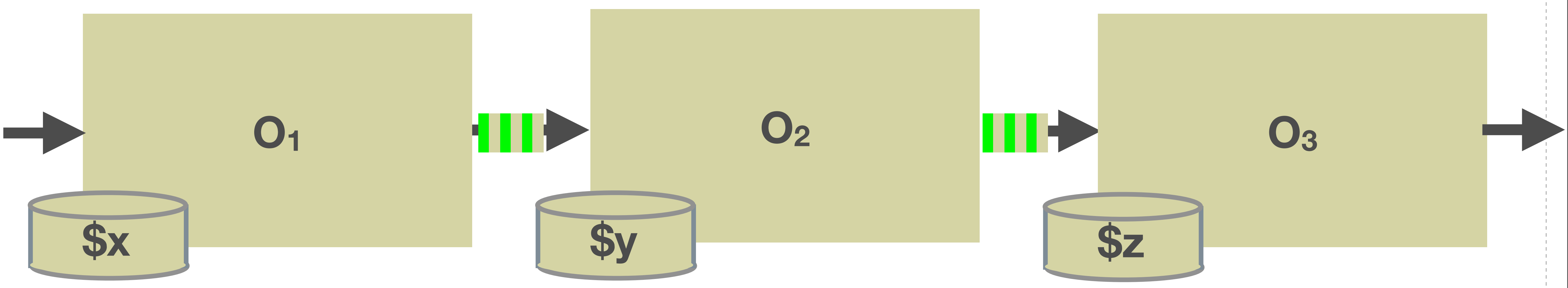
An Intermediate Language for Stream Processing

- Benefits of a VEE/IL are well known
  - Increase portability, share optimizations, etc.
- Streaming needs its own IL
  - Need to reason across machines, support different optimizations
- Brooklet serves as a solid foundation
  - Challenge: How to bridge the gap between theory and practice?

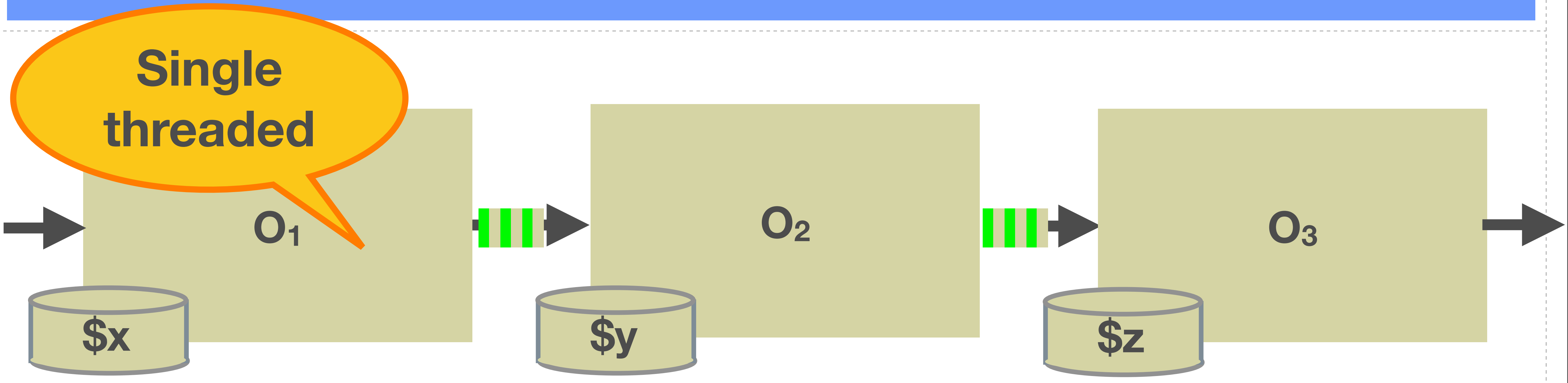
Make Abstractions Concrete

Brooklet	River
Sequence of atomic steps	Operators execute concurrently
Pure functions, state threaded through invocations	Stateful functions, protected with automatic locking
Non-deterministic execution	Restricted execution with bounded queues, and back-pressure
Opaque functions	Function implementations
No physical platform, independent from runtime	Abstract representation of runtime e.g. placement
Finite execution	Indefinite execution

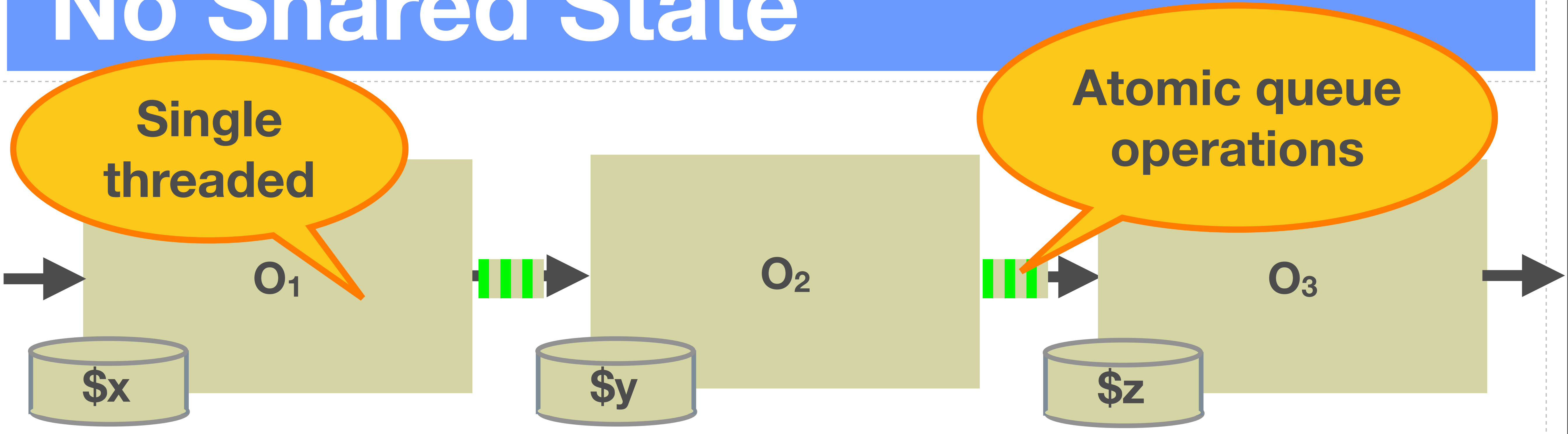
Concurrent Execution: No Shared State



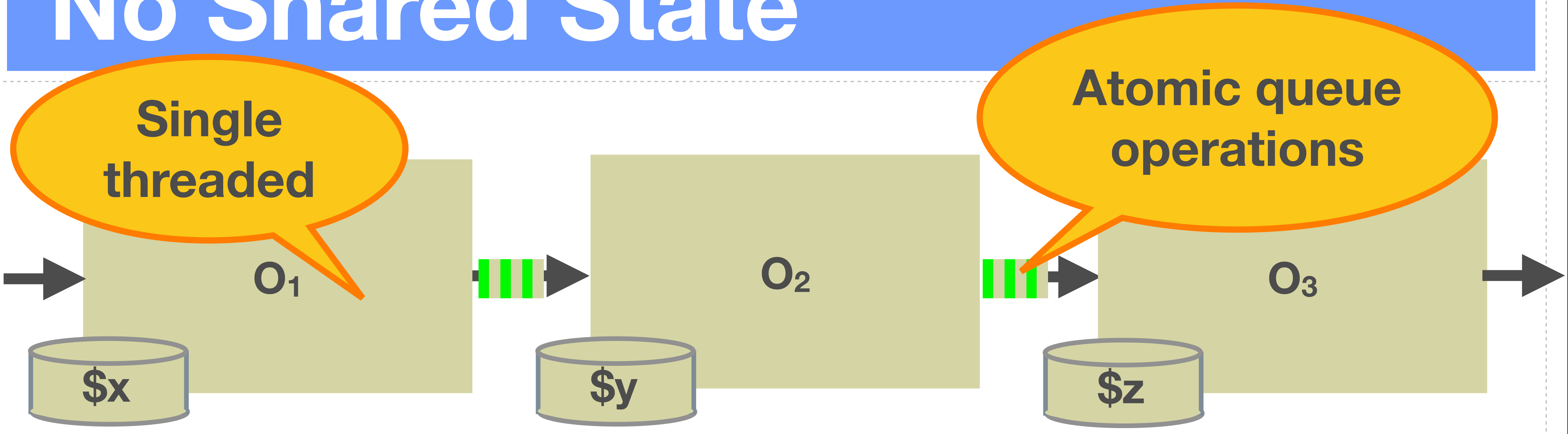
Concurrent Execution: No Shared State



Concurrent Execution: No Shared State

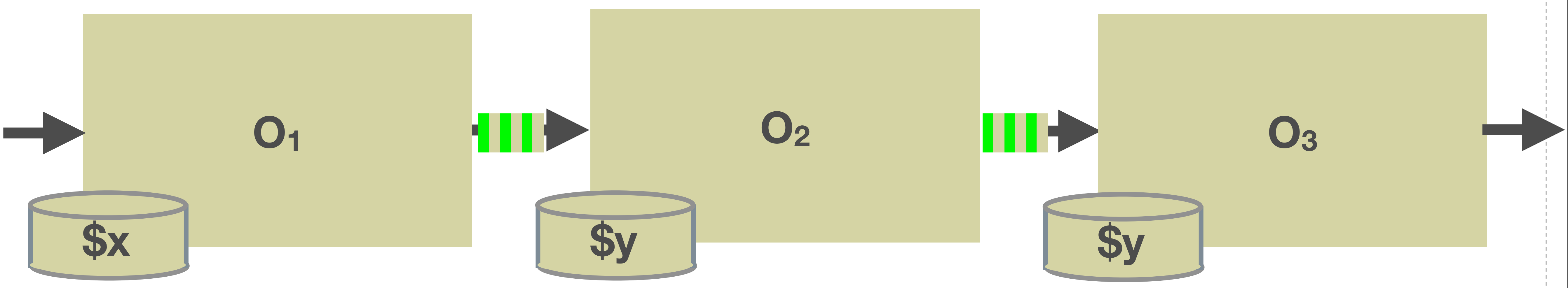


Concurrent Execution: No Shared State



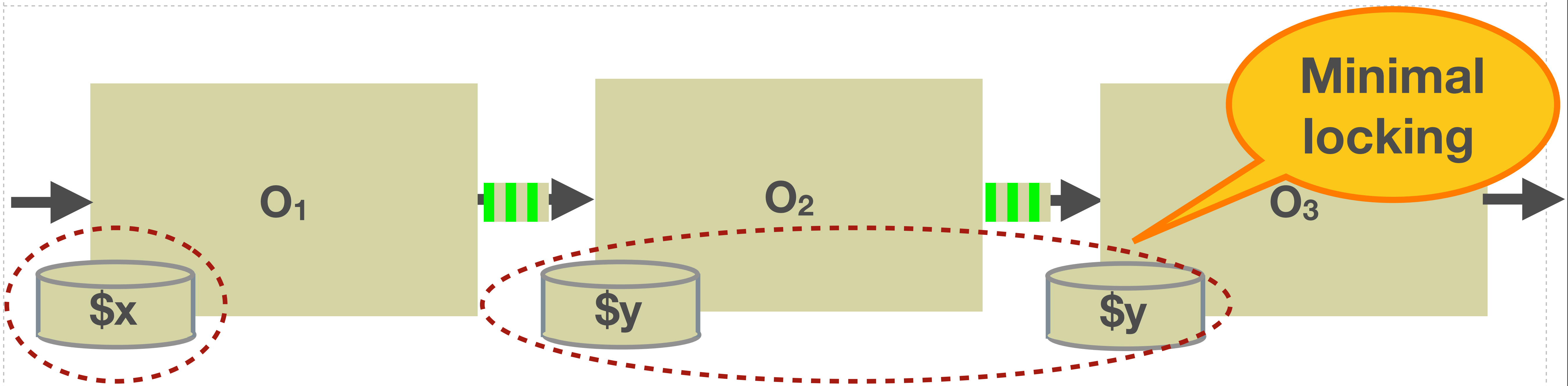
- Brooklet operators fire one at a time
- River operators fire concurrently
- For both, data must be available

Concurrent Execution: With Shared State



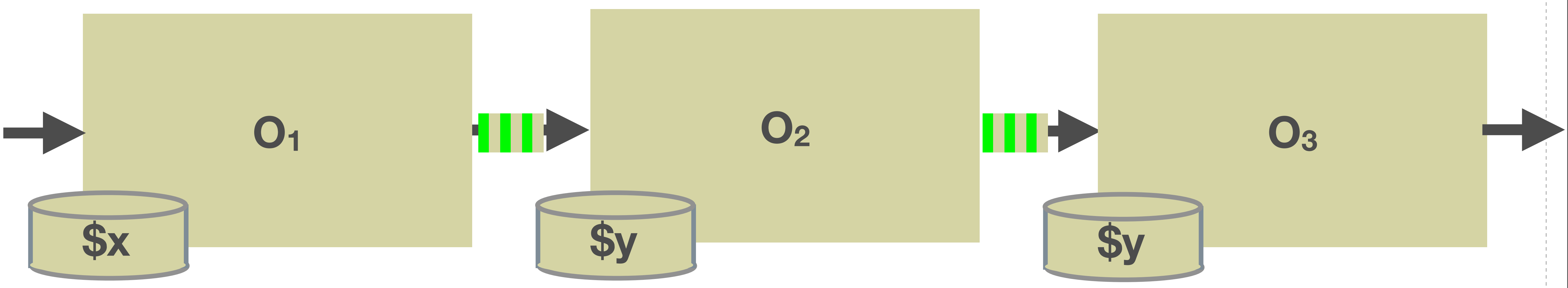
- Locks form equivalence classes over shared variables
- Every shared variable is protected by one lock
- Shared variables in the same class protected by same lock
- Locks acquired/released in standard order

Concurrent Execution: With Shared State



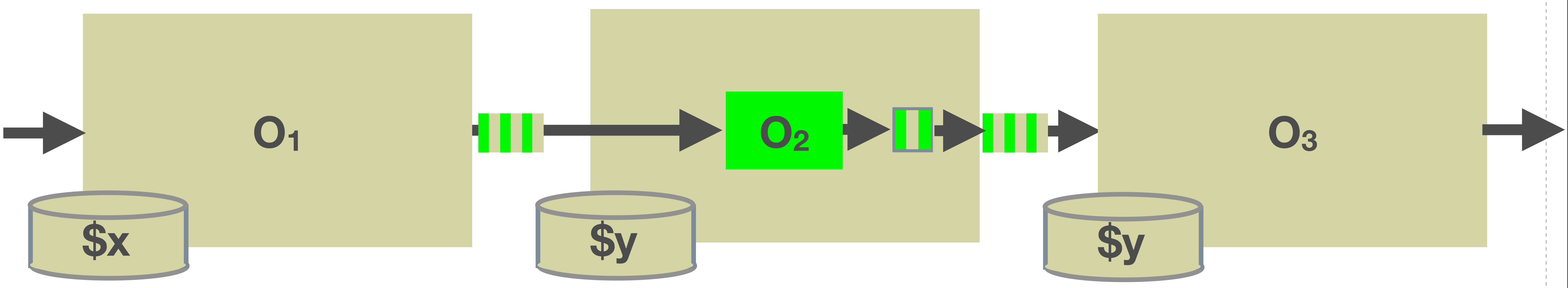
- Locks form equivalence classes over shared variables
- Every shared variable is protected by one lock
- Shared variables in the same class protected by same lock
- Locks acquired/released in standard order

Restricted Execution: Bounded Queues

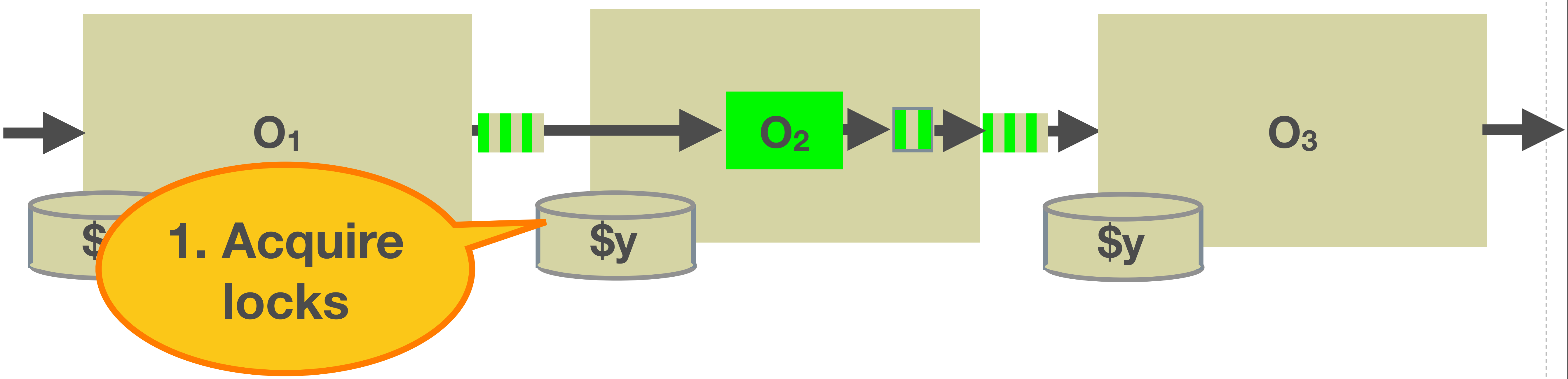


- Naïve approach: block when output queue is full
- If O_2 holds the lock on $x and blocks, O_3 cannot execute
- Deadlock!

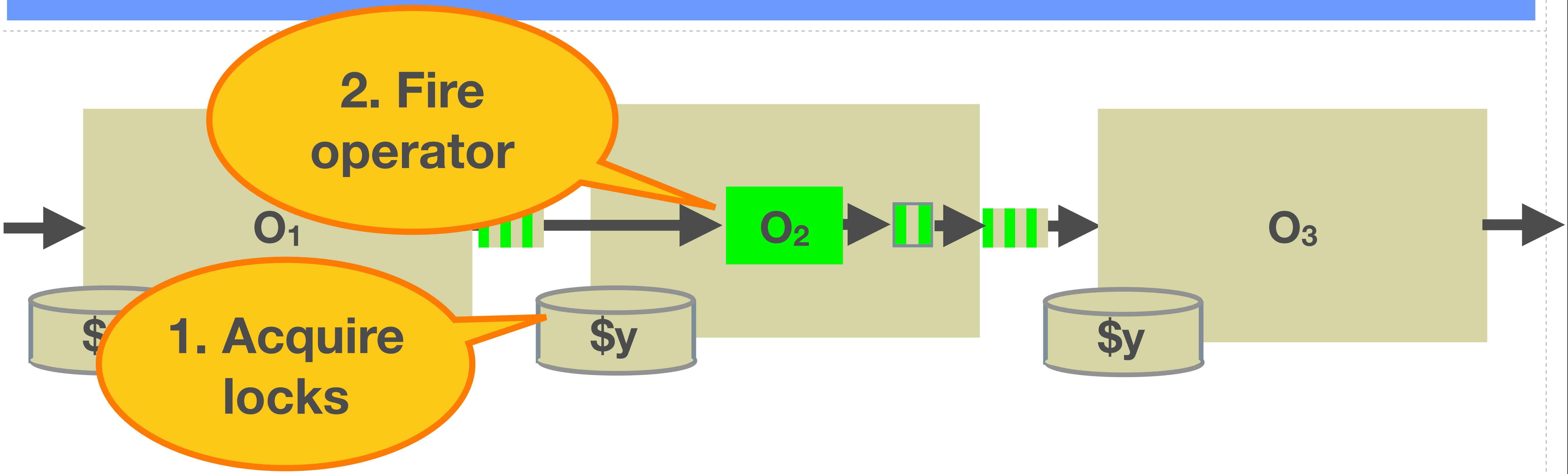
Restricted Execution: Safe Back-Pressure



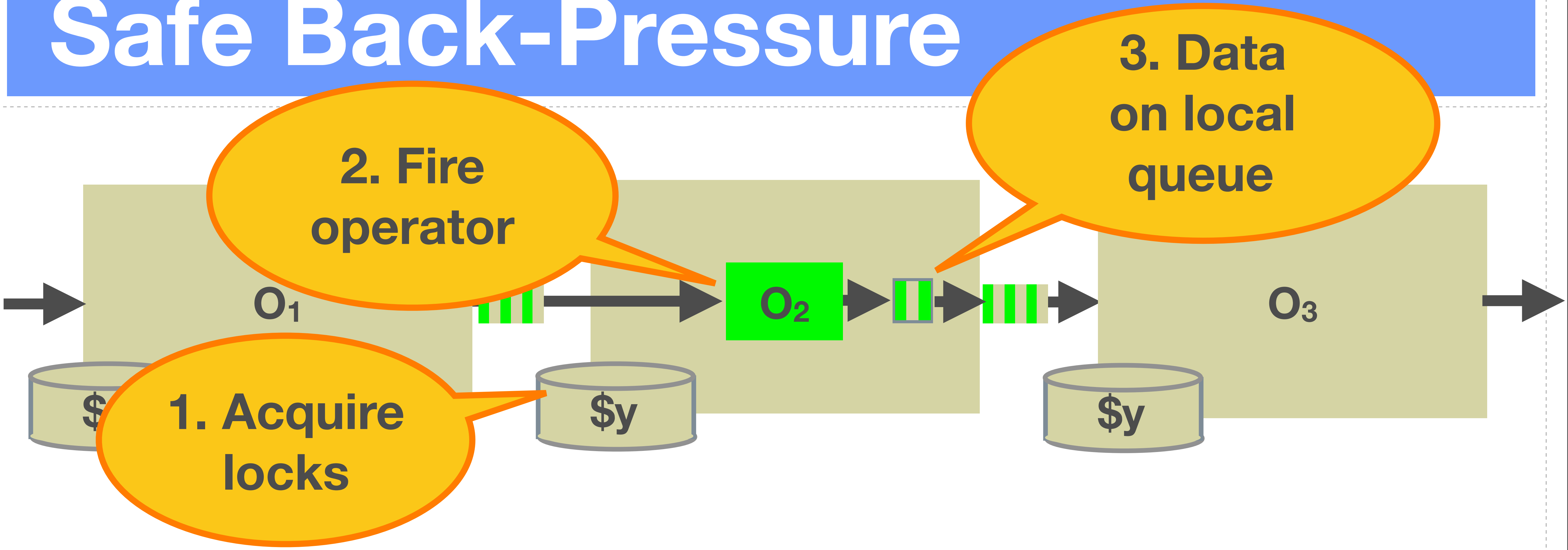
Restricted Execution: Safe Back-Pressure



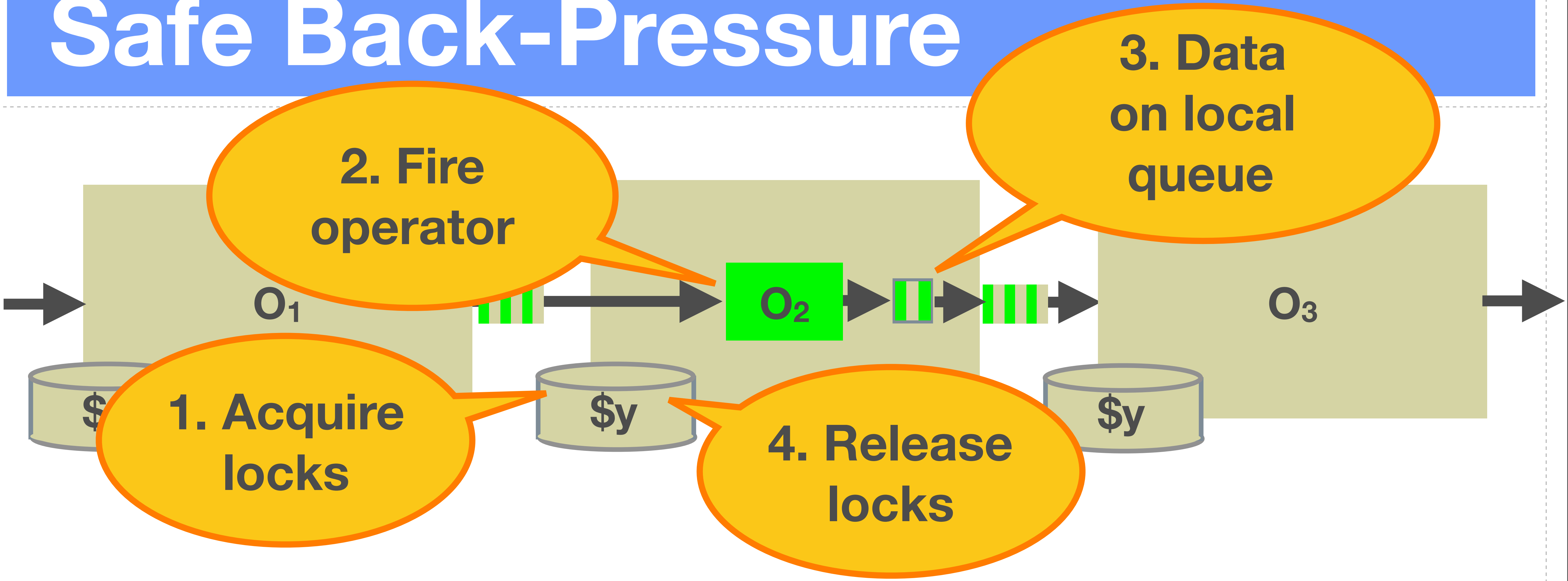
Restricted Execution: Safe Back-Pressure



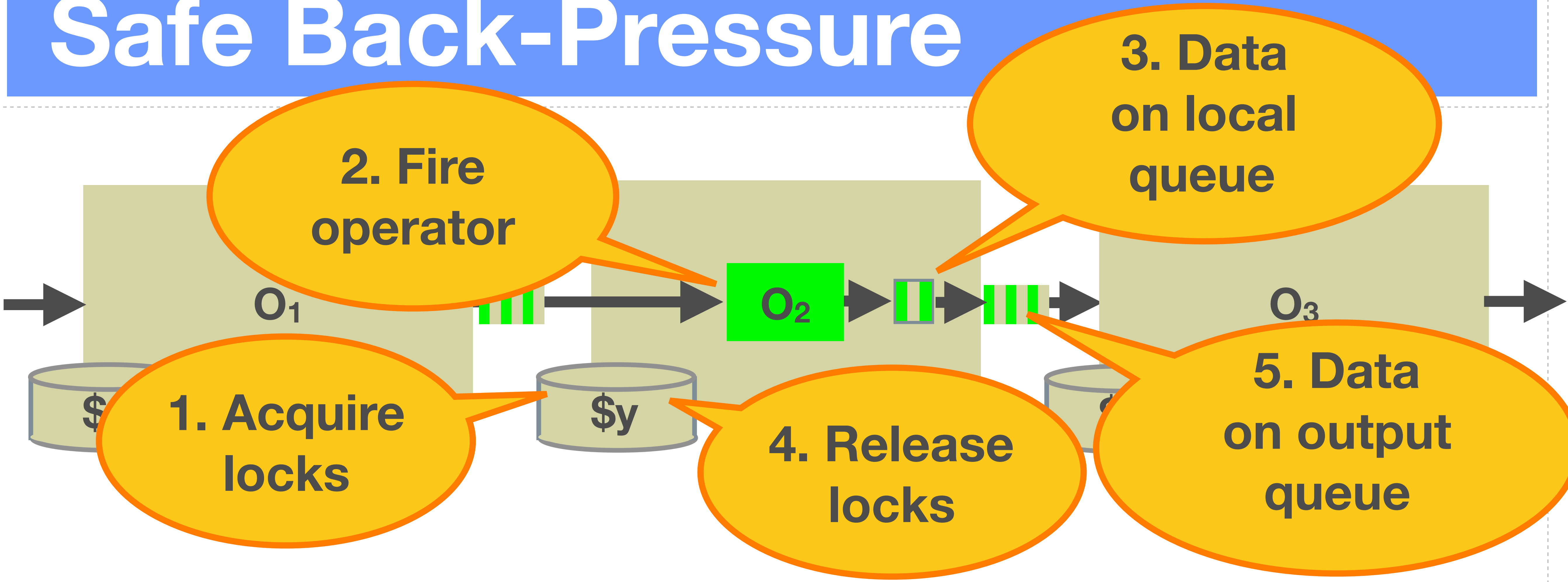
Restricted Execution: Safe Back-Pressure



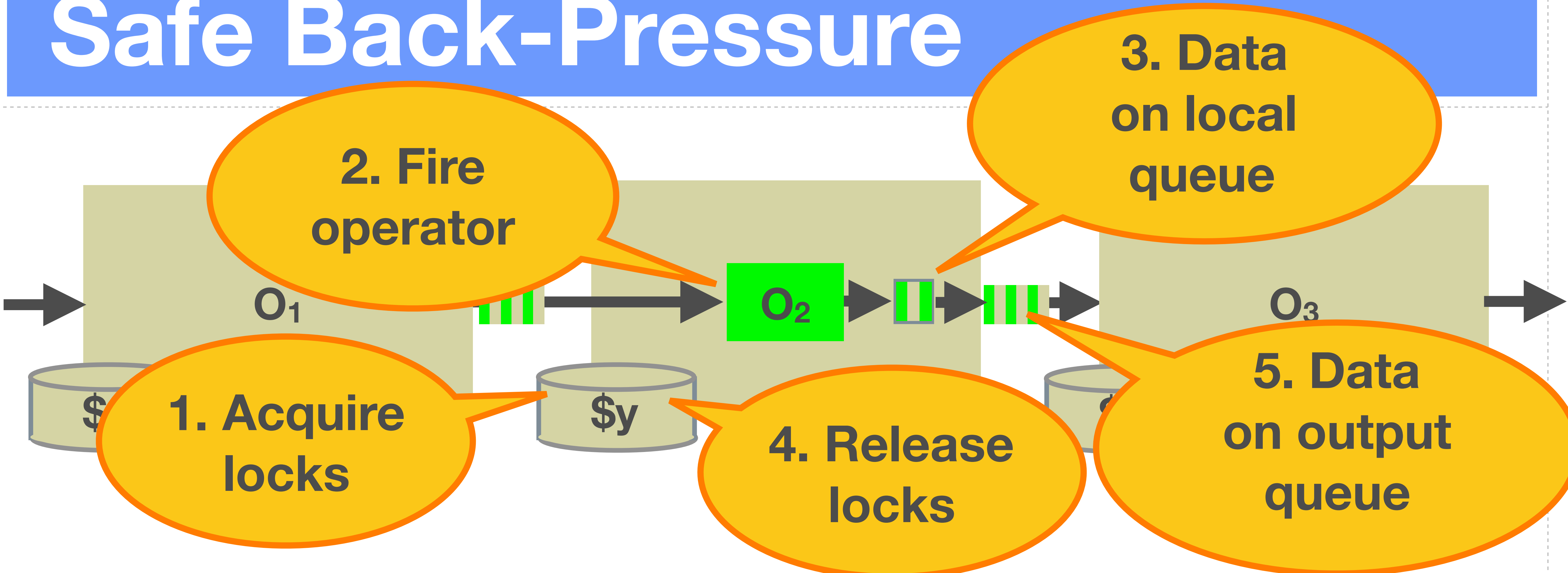
Restricted Execution: Safe Back-Pressure



Restricted Execution: Safe Back-Pressure



Restricted Execution: Safe Back-Pressure



- Only step 5 can block
- Locks have already been released, so O_3 can execute
- Even if downstream is full, there is no deadlock

Applications of an Intermediate Language

- Must make language development economic
  - Implementation language, language modules, operator templates
- Must support a broad range of optimizations
  - Annotations provide additional information between source and IL

Function Implementations and Translations

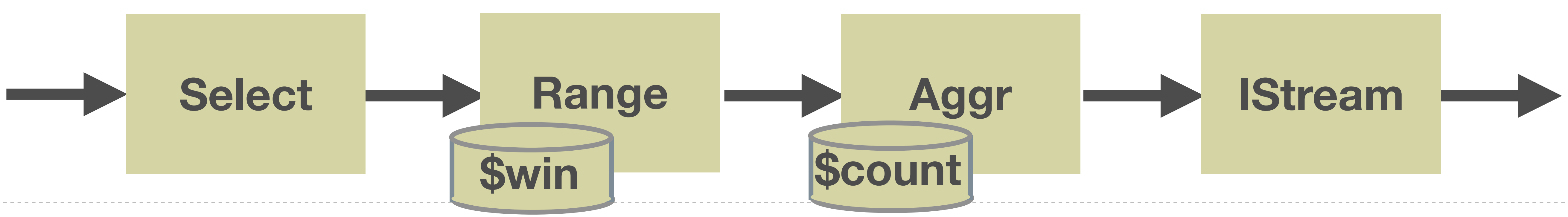
```
logs : {origin : string; target : string} stream;
hits : {origin : string; count : int} stream =
  select istream(origin, count(origin))
    from logs [range 300]
    where origin != target
```

Pre-existing operator templates

Bag.filter (fun x -> #expr)

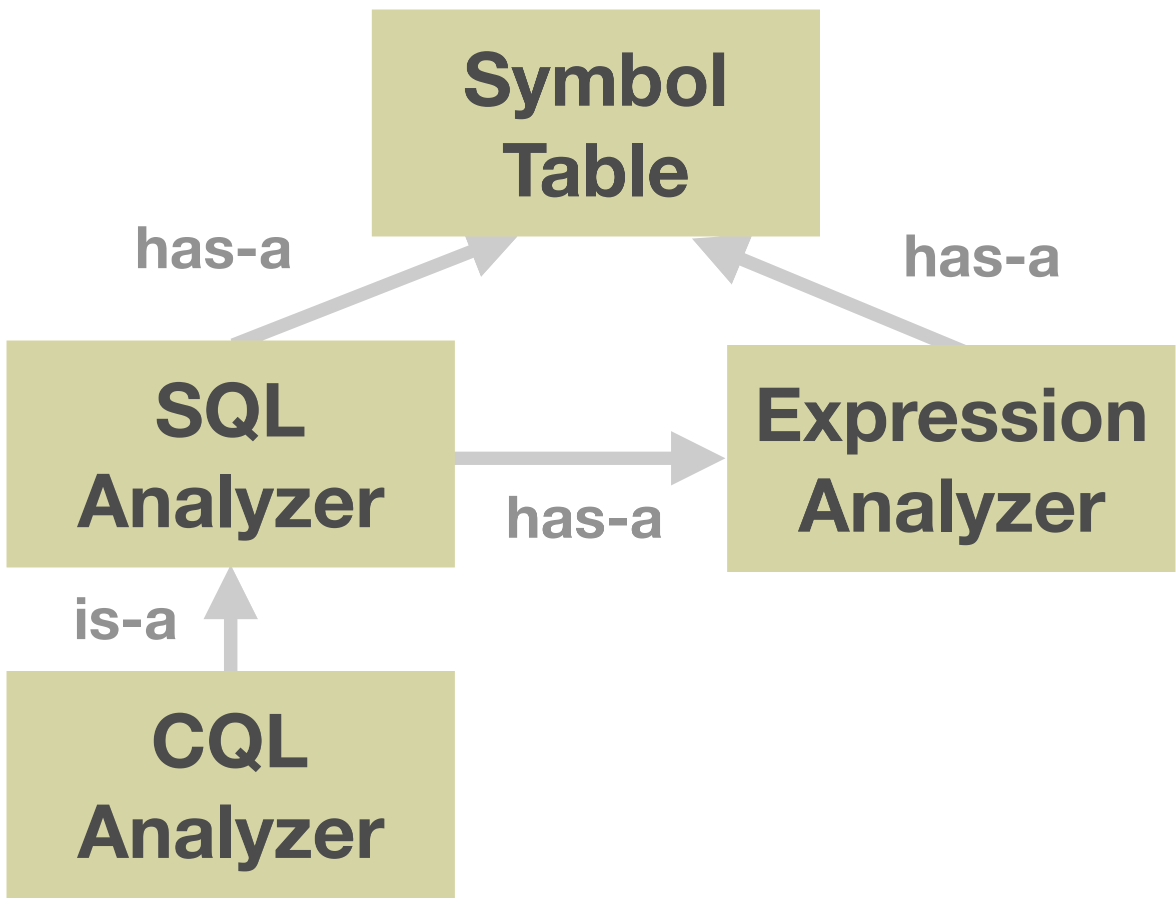
Expose operators, communication, and state

Bag.filter (fun x -> origin != target)



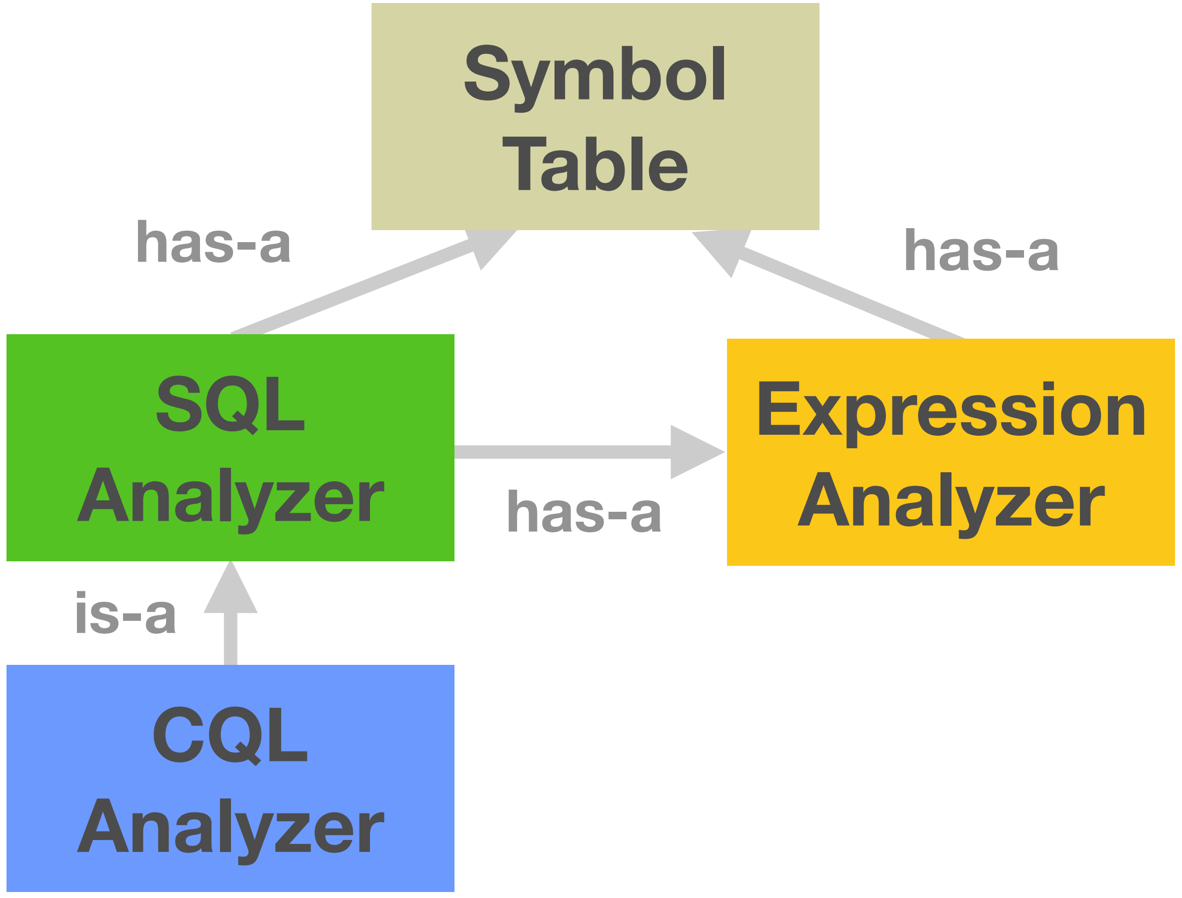
Translations with Modules

select istream(*)
from quotes[now], history
where quotes.ask <= history.low
and quotes.ticker = history.ticker



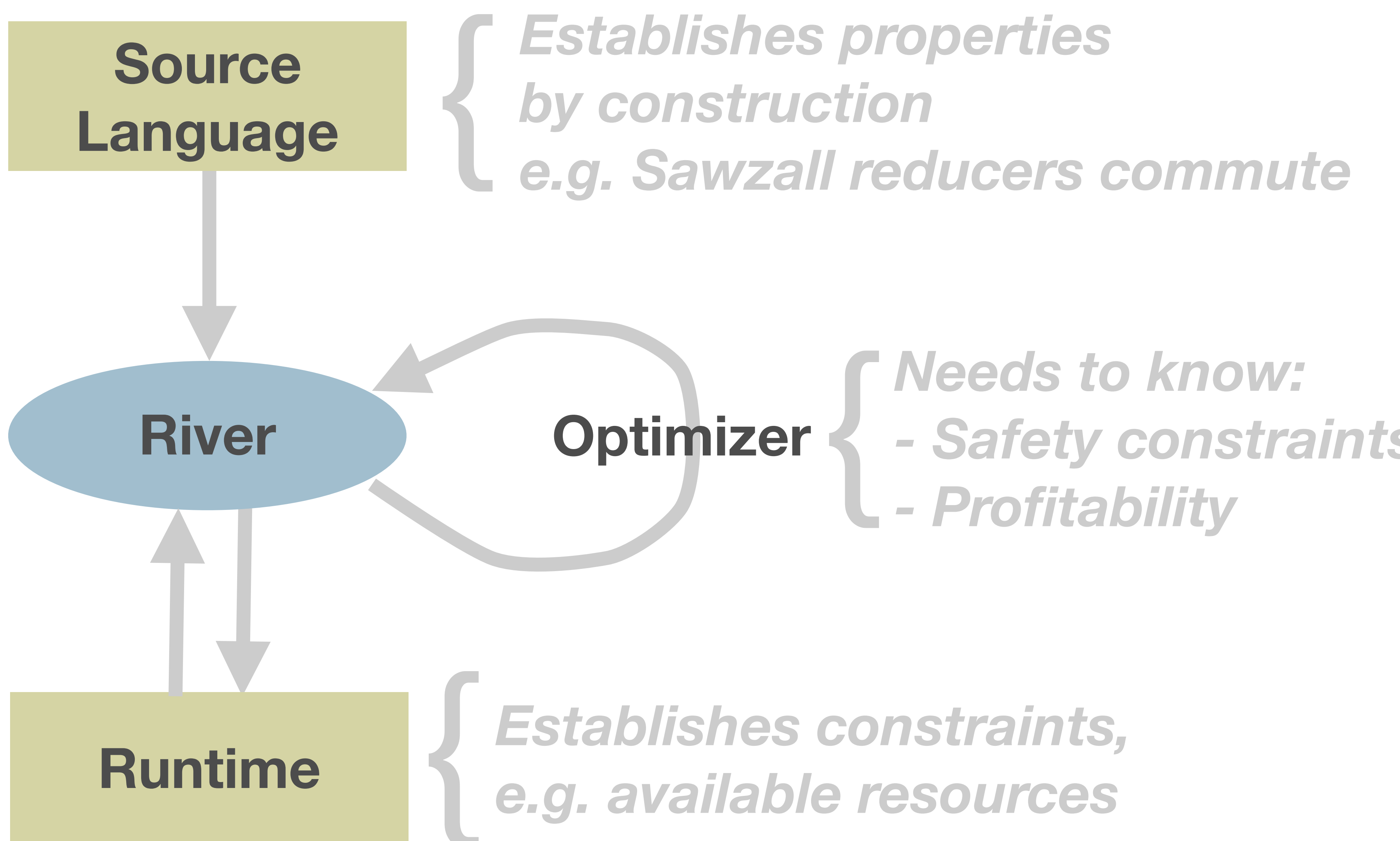
Translations with Modules

select istream(*)
from quotes[now], history
where quotes.ask <= history.low
and quotes.ticker = history.ticker

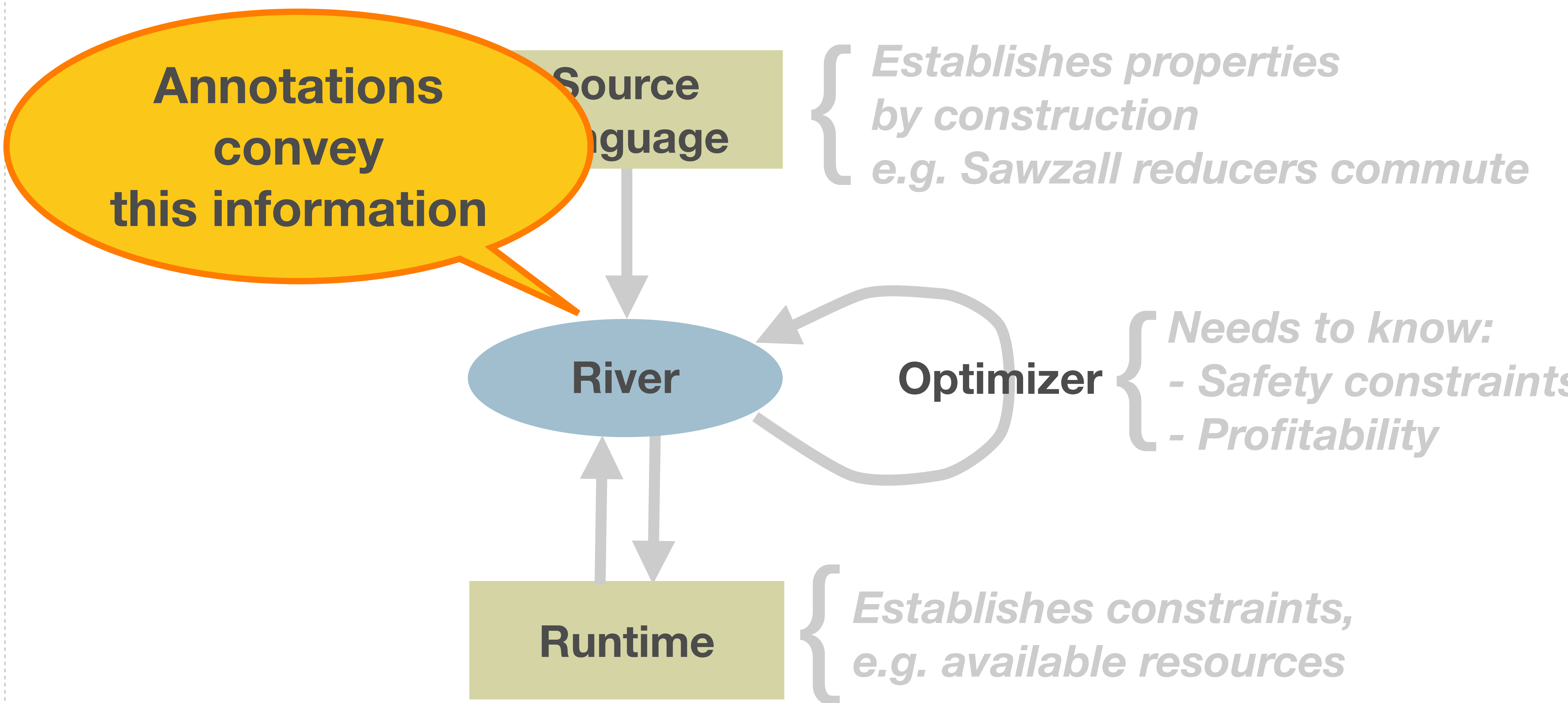


CQL = SQL + Streaming + Expressions

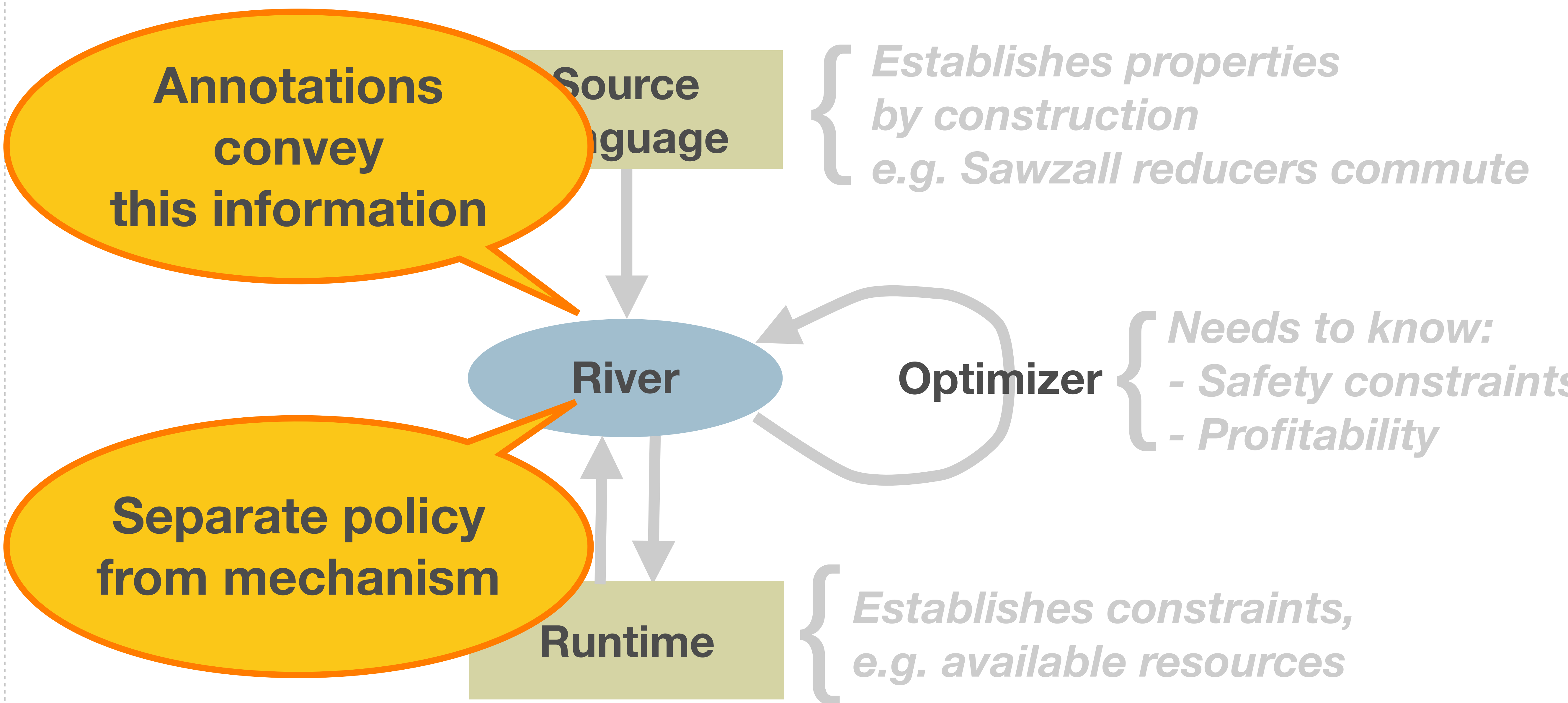
Optimization Support: Extensible Annotations



Optimization Support: Extensible Annotations



Optimization Support: Extensible Annotations



Optimization Support: Current Annotations

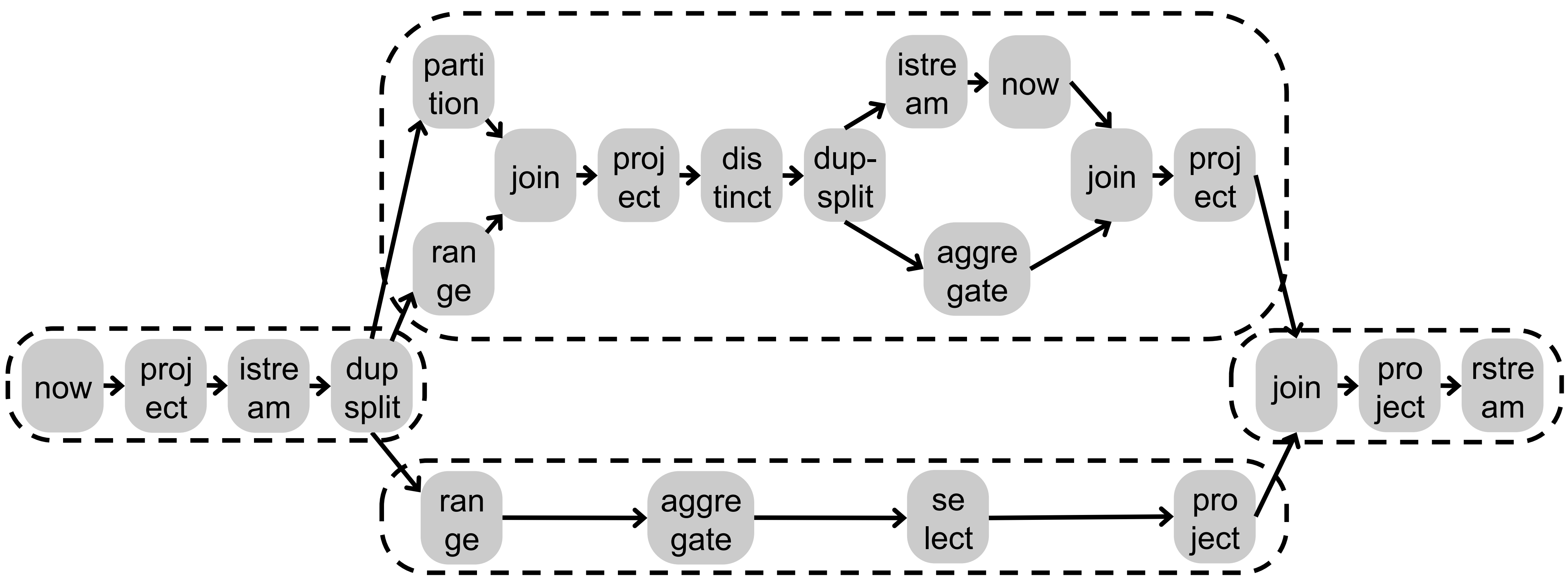
Annotation	Description	Optimization
@Fuse(ID)	Fuse operators with same ID in the same process	Fusion
@Parallel()	Perform fission on an operator	Fission
@Commutative()	An operator's function is commutative	Fission
@Keys($k_1,\ldots,k_n$)	An operator's state is partitionable by the key fields $k_1,\ldots,k_n$	Fission
@Group(ID)	Place operators with same ID on the same machine	Placement

Evaluation

- Four benchmark applications
  - CQL Linear Road
  - StreamIt FM Radio
  - Sawzall Batch Web Log Analyzer
  - CQL Continuous Web Log Analyzer

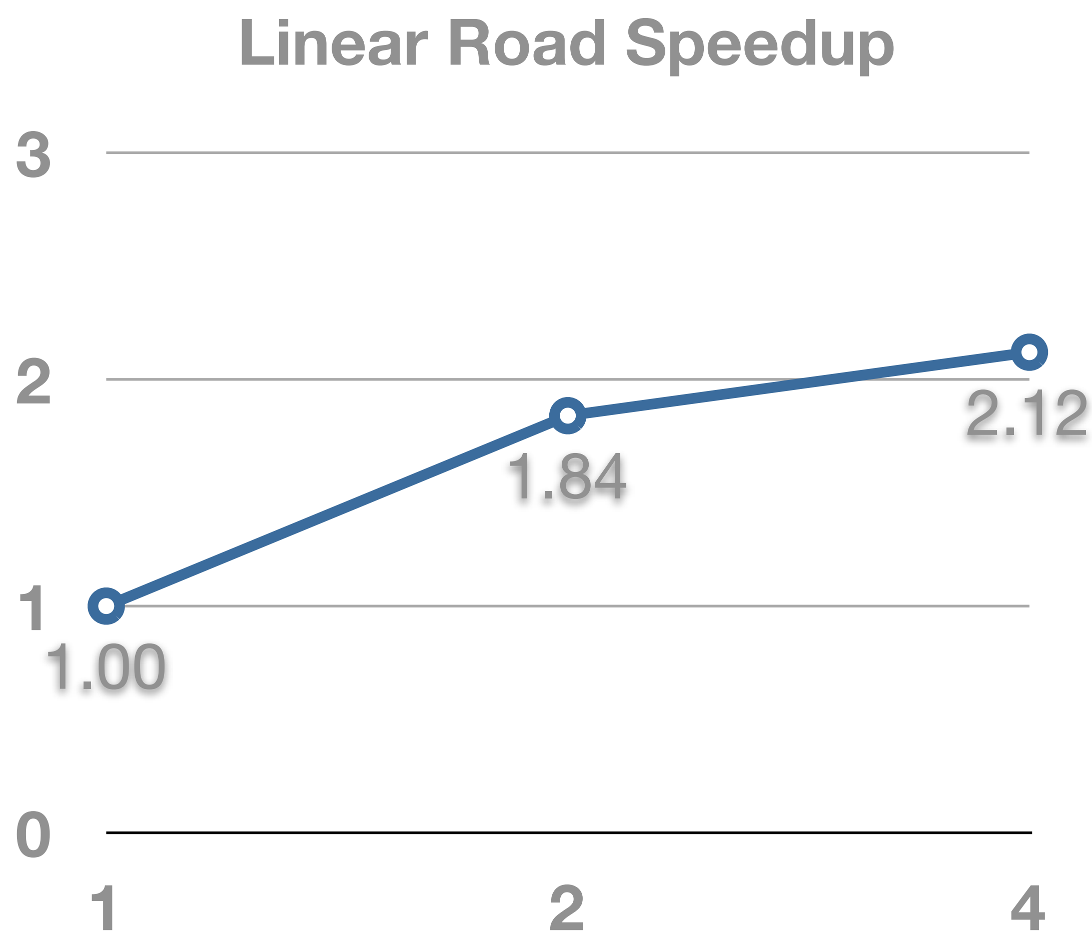
- Three optimizations
  - Placement
  - Fission
  - Fusion

Distributed Linear Road

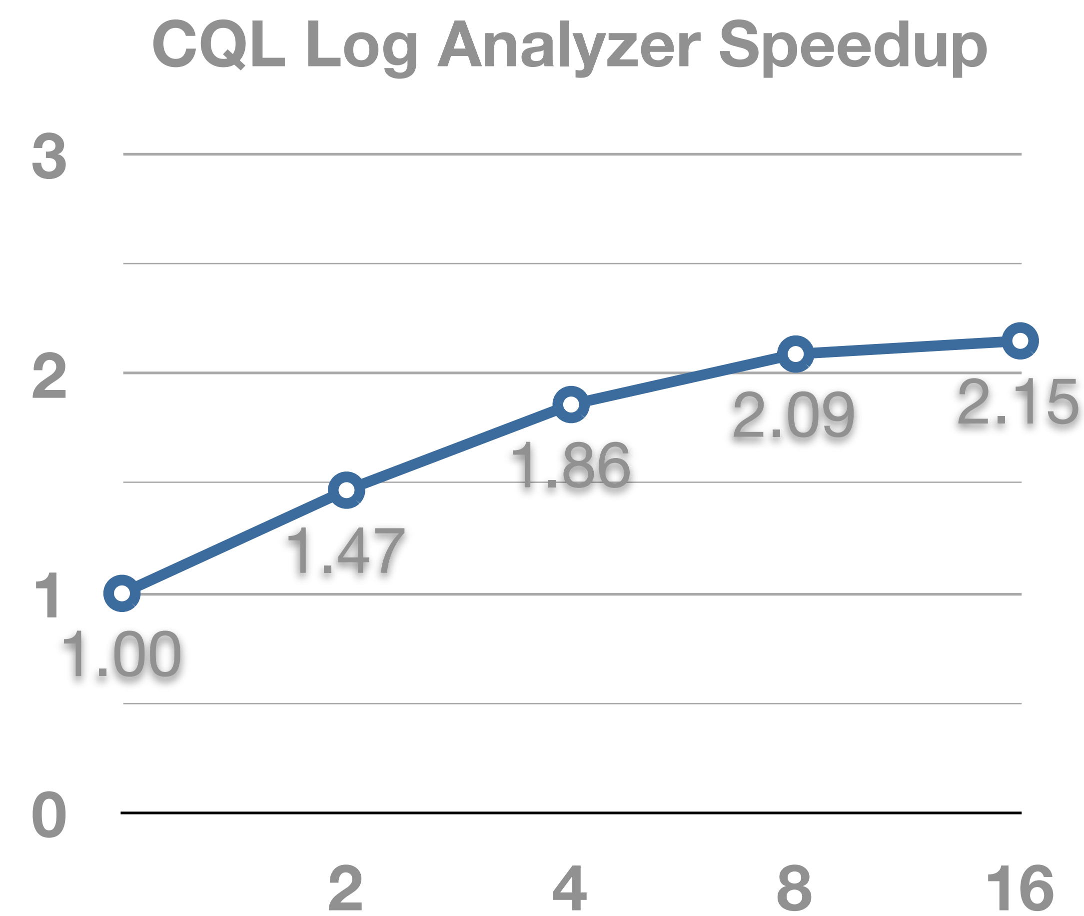


First distributed CQL implementation

CQL Parallelization Has Limited Effect

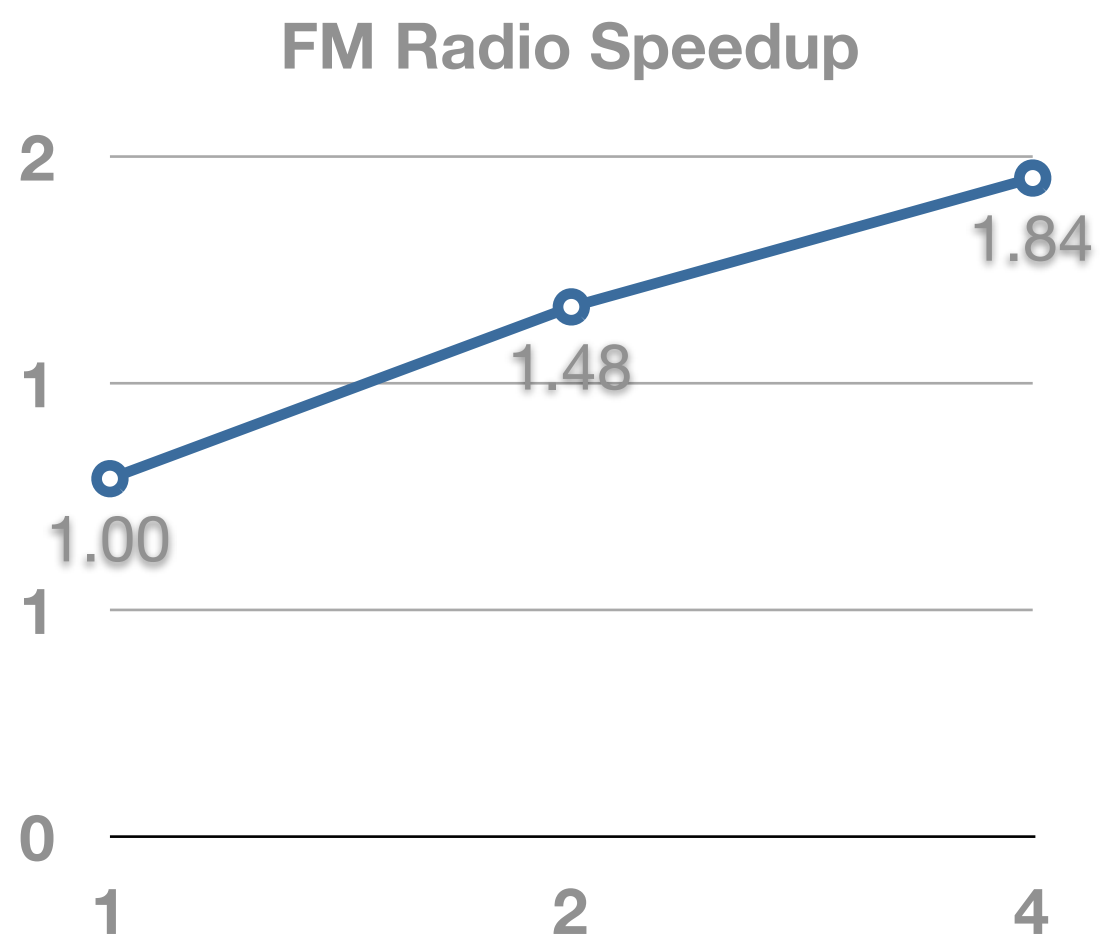


- 2.12x speedup on 4 machines
- Limited task and pipeline parallelism



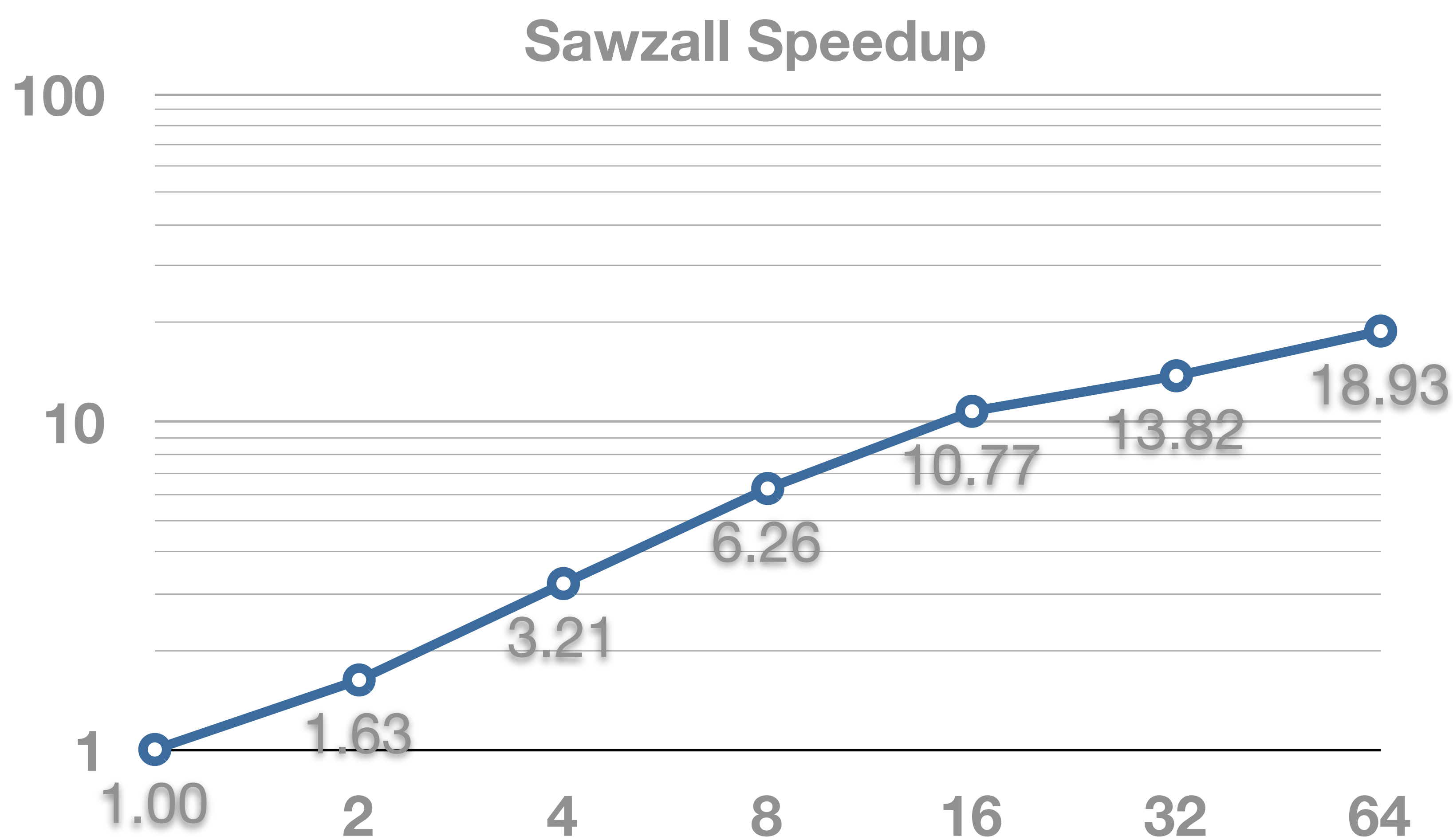
- 2.15x speedup on 16 machines
- Synchronization is bottleneck

Reusable Optimizations



- StreamIt FM Radio can re-use the placement optimization
- 1.84x speedup on 4 machines

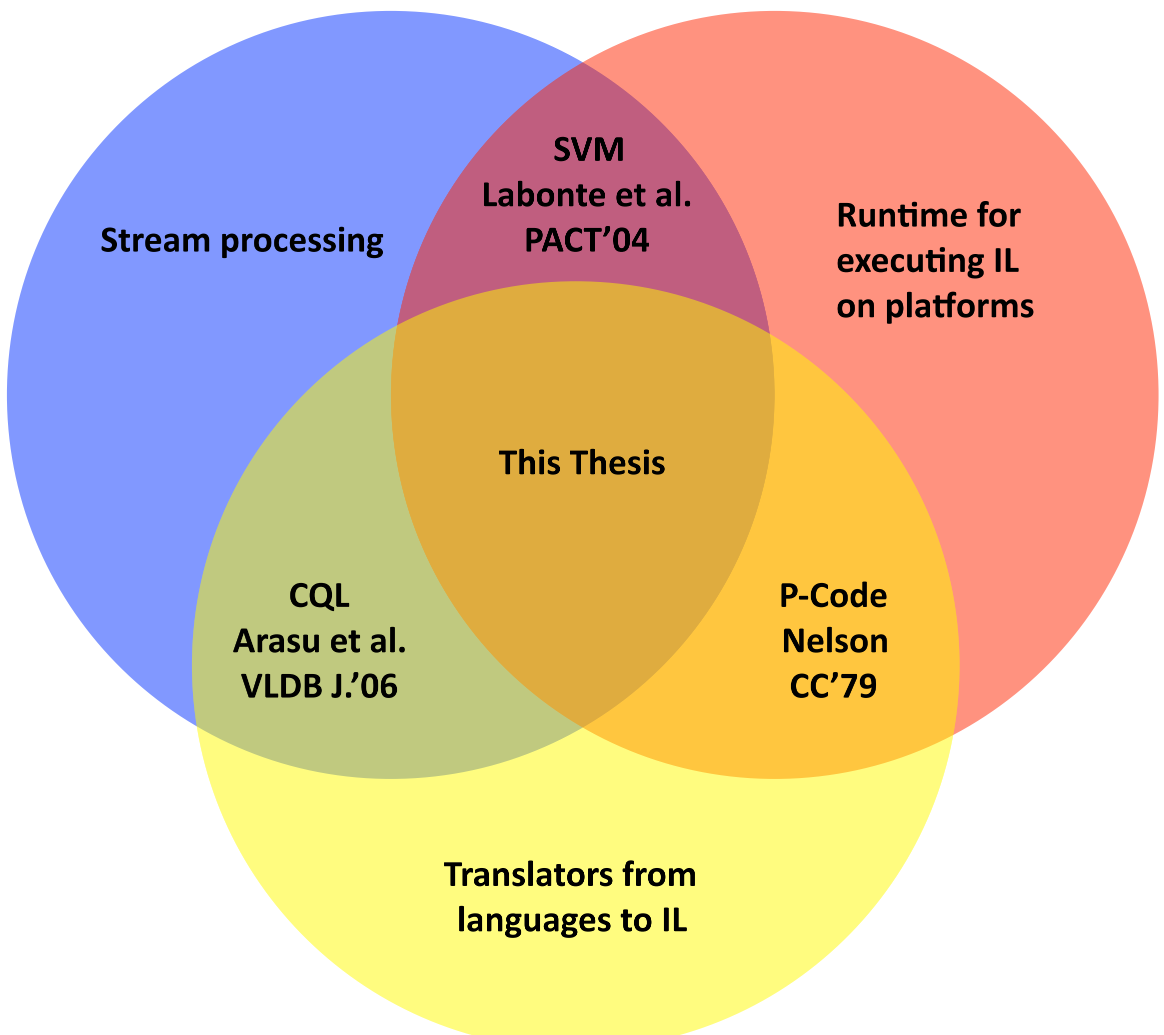
MapReduce on River Scales (Almost) Linearly



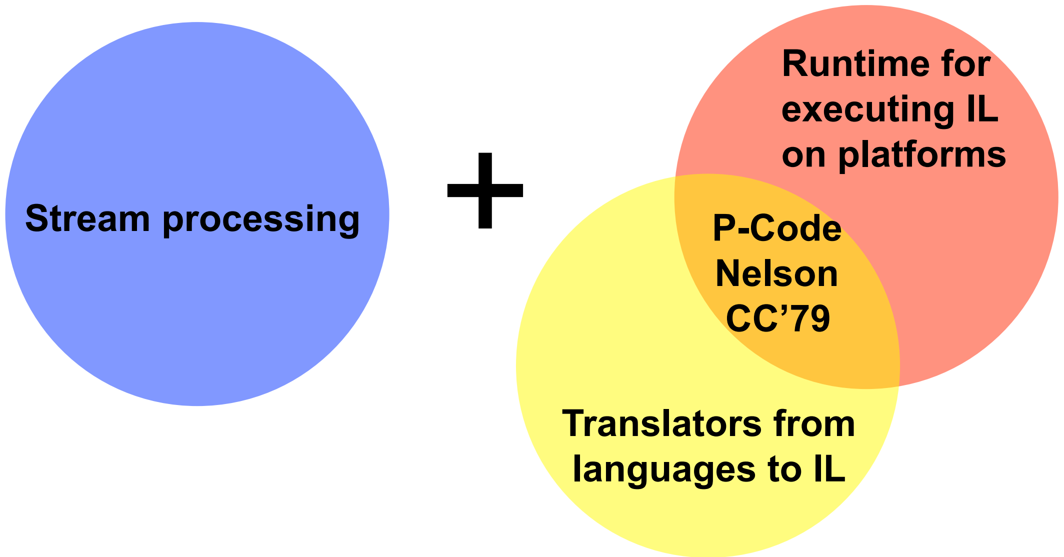
- Our Sawzall uses the same data-parallelism optimizer as CQL
- 10.77x speedup on 16 machines, 18.93x speedup on 64 cores

Related Work

Related Work

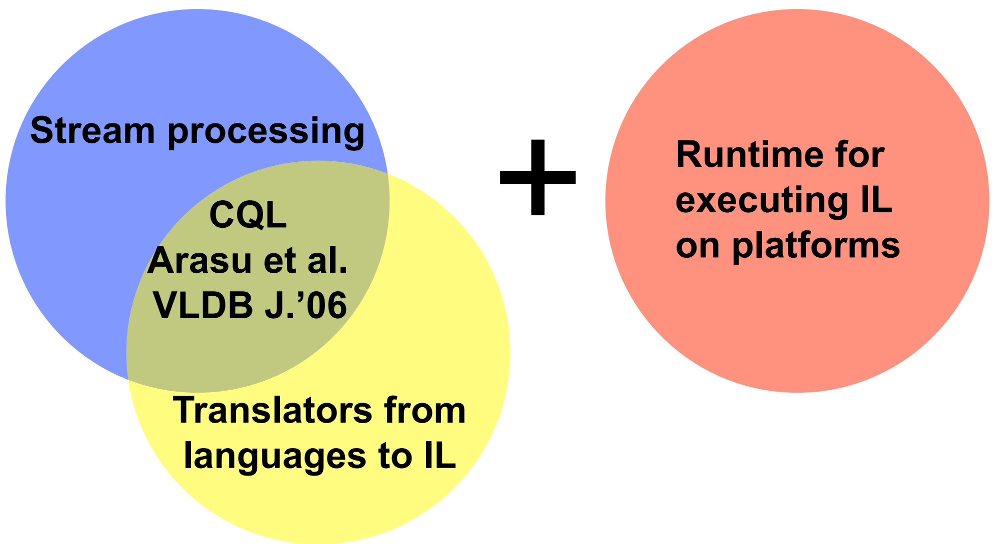


Comparison to Traditional ILs



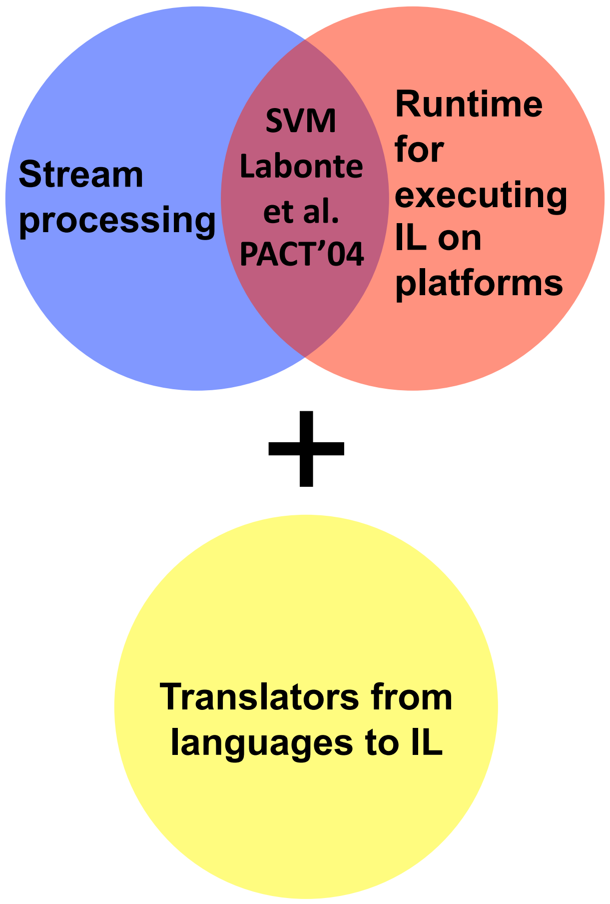
Traditional IL	River IL
For Pascal, Java, C#	For StreamSQL, Sawzall, StreamIt
IL is lower-level	IL for explicit streaming topology
Data at rest (registers)	Data in motion (queues)
Instructions that run in a sequence, one after the other	Functions that run in parallel, continuously

Comparison to CQL



CQL	River IL
Described in terms of SRA (stream-relational algebra)	Uses more general streaming IL (not restricted to relational)
Inter-dependent with a single runtime	Virtual, independent of any particular runtime

Comparison to SVM



SVN	River IL
Missing translators from any language	Translation by recursion over syntax, making state explicit, encapsulating computation in functions
Synchronous, assumes centralized controller	Asynchronous, no centralized controller
Assumes machine model with shared memory and CPUs	Abstracts away streaming runtime (may even be a distributed cluster)

Conclusions

Limitations

Component	Limitations or Future Work
Optimizations Catalog	Interaction of optimizations, compiler analysis, standard benchmarks
Brooklet	Relationship to other calculi, time constraints, more optimizations, dynamism
River	Support for dynamism, performance, design of new languages

Conclusion

- Stream processing is crucial, and needs software infrastructure
  - Identify requirements with a catalog of optimizations
  - Provide a formal foundation with a calculus
  - Design a practical IL with a rigorous semantics
- Overall this work:
  - Enables further advances in language and optimizations design
  - Encourages innovation in stream processing

CQL Translation Rules

CQL program translation: $[\![F_c, P_c]\!]^p_c = \langle F_b, P_b \rangle$

$$\frac{}{[\![F_c, SName]\!]^p_c = \emptyset, \mathtt{output}\, SName\mathtt{;input}\, SName\mathtt{;}\bullet} \quad (\mathrm{T}^p_c\text{-SNAME})$$

$$\frac{}{[\![F_c, RName]\!]^p_c = \emptyset, \mathtt{output}\, RName\mathtt{;input}\, RName\mathtt{;}\bullet} \quad (\mathrm{T}^p_c\text{-RNAME})$$

$$\frac{\begin{array}{c} F_b, \mathtt{output}\ q_o\mathtt{;}\ \mathtt{input}\ \overline{q}\mathtt{;}\ \overline{op} = [\![F_c, P_{cs}]\!]^p_c \\ q'_o = freshId() \qquad v = freshId() \\ F'_b = [S2R \mapsto wrapS2R(F_c(S2R))]F_b \\ \overline{op}' = \overline{op}, (q'_o, v) \leftarrow S2R(q_o, v)\mathtt{;} \end{array}}{[\![F_c, S2R(P_{cs})]\!]^p_c = F'_b, \mathtt{output}\ q'_o\mathtt{;}\ \mathtt{input}\ \overline{q}\mathtt{;}\ \overline{op}'} \quad (\mathrm{T}^p_c\text{-S2R})$$

$$\frac{\begin{array}{c} F_b, \mathtt{output}\ q_o\mathtt{;}\ \mathtt{input}\ \overline{q}\mathtt{;}\ \overline{op} = [\![F_c, P_{cr}]\!]^p_c \\ q'_o = freshId() \qquad v = freshId() \\ F'_b = [R2S \mapsto wrapR2S(F_c(R2S))]F_b \\ \overline{op}' = \overline{op}, (q'_o, v) \leftarrow R2S(q_o, v)\mathtt{;} \end{array}}{[\![F_c, R2S(P_{cr})]\!]^p_c = F'_b, \mathtt{output}\ q'_o\mathtt{;}\ \mathtt{input}\ \overline{q}\mathtt{;}\ \overline{op}'} \quad (\mathrm{T}^p_c\text{-R2S})$$

$$\frac{\begin{array}{c} \overline{F_b, \mathtt{output}\ q_o\mathtt{;}\ \mathtt{input}\ \overline{q}\mathtt{;}\ \overline{op}} = \overline{[\![F_c, P_{cr}]\!]^p_c} \\ n = |\overline{P_{cr}}| \qquad q'_o = freshId() \qquad \overline{q}' = \overline{q}_1, \ldots, \overline{q}_n \\ \forall i \in 1 \ldots n : v_i = freshId() \qquad \overline{op}' = \overline{op}_1, \ldots, \overline{op}_n \\ F'_b = [R2R \mapsto wrapR2R(F_c(R2R))](\cup \overline{F_b}) \\ \overline{op}'' = \overline{op}', (q'_o, \overline{v}) \leftarrow R2R(\overline{q_o}, \overline{v})\mathtt{;} \end{array}}{[\![F_c, R2R(\overline{P_{cr}})]\!]^p_c = F'_b, \mathtt{output}\ q'_o\mathtt{;input}\ \overline{q}'\mathtt{;}\overline{op}''} \quad (\mathrm{T}^p_c\text{-R2R})$$

CQL operator wrappers:

$$\frac{\begin{array}{c} \sigma, \tau = d_q \qquad s = d_v \\ s' = s \cup \{\langle e, \tau \rangle : e \in \sigma\} \qquad \sigma' = f(s', \tau) \end{array}}{wrapS2R(f)(d_q, \_, d_v) = \langle \sigma', \tau \rangle, s'} \quad (\mathrm{W}_c\text{-S2R})$$

$$\frac{\sigma, \tau = d_q \qquad \sigma' = d_v \qquad \sigma'' = f(\sigma, \sigma')}{wrapR2S(f)(d_q, \_, d_v) = \langle \sigma'', \tau \rangle, \sigma} \quad (\mathrm{W}_c\text{-R2S})$$

$$\frac{\begin{array}{c} \sigma, \tau = d_q \qquad d'_i = d_i \cup \{\langle \sigma, \tau \rangle\} \\ \forall j \neq i \in 1 \ldots n : d'_j = d_j \\ \exists j \in 1 \ldots n : \nexists \sigma : \langle \sigma, \tau \rangle \in d_j \end{array}}{wrapR2R(f)(d_q, i, \overline{d}) = \bullet, \overline{d}'} \quad (\mathrm{W}_c\text{-R2R-WAIT})$$

$$\frac{\begin{array}{c} \sigma, \tau = d_q \qquad d'_i = d_i \cup \{\langle \sigma, \tau \rangle\} \\ \forall j \neq i \in 1 \ldots n : d'_j = d_j \\ \forall j \in 1 \ldots n : \sigma_j = aux(d_j, \tau) \end{array}}{wrapR2R(f)(d_q, i, \overline{d}) = \langle f(\overline{\sigma}), \tau \rangle, \overline{d}'} \quad (\mathrm{W}_c\text{-R2R-READY})$$

$$\frac{\langle \sigma, \tau \rangle \in d}{aux(d, \tau) = \sigma} \quad (\mathrm{W}_c\text{-R2R-AUX})$$

Operator Fission

$$\frac{\begin{array}{c} op = (q_{out}) \leftarrow f(q_{in}); \\ \forall i \in 1 \ldots n : q_i = \mathit{freshId}() \qquad \forall i \in 1 \ldots n : q'_i = \mathit{freshId}() \\ F'_b, op_s = [\![\, \emptyset, \texttt{split roundrobin}, \overline{q}, q_{in} \,]\!]^p_s \\ \forall i \in 1 \ldots n : op_i = (q'_i) \leftarrow f(q_i); \\ F''_b, op_j = [\![\, \emptyset, \texttt{join roundrobin}, q_{out}, \overline{q}' \,]\!]^p_s \end{array}}{\langle F_b, op \rangle \longrightarrow^N_{split} \langle F_b \cup F'_b \cup F''_b, op_s\ \overline{op}\ op_j \rangle}$$

Dynamism

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