Distributed CQL Made Easy

Robert Soulé, Martin Hirzel, Robert Grimm, Buğra Gedik

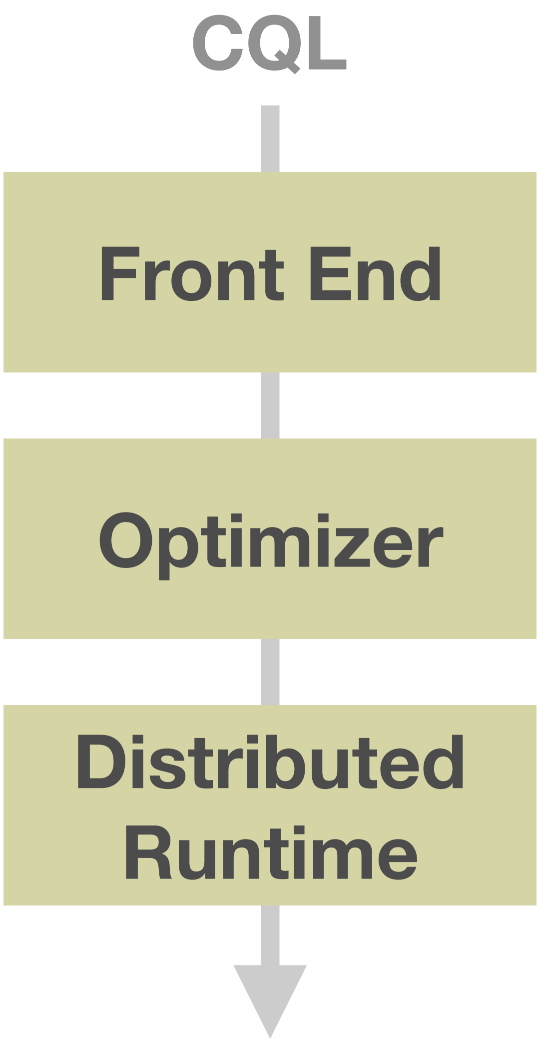
New York University and IBM Research

Streaming SQL Must Scale

- Stream computing is everywhere
    - Crucial to finance, government, science
- Streaming SQL is popular because it has a familiar syntax
    - CQL is a streaming SQL with a formally defined semantics
- More and more data means streaming SQL needs to scale
    - Either across large NUMA machines or clusters

Distributed CQL the Hard Way

- Build syntactic and semantic analyzers, code generator, etc.
- Implement core optimizations, such as re-ordering and parallelization
- Develop runtime for process management, data-transport, etc.
- This is *painful*!


CQL

Front End

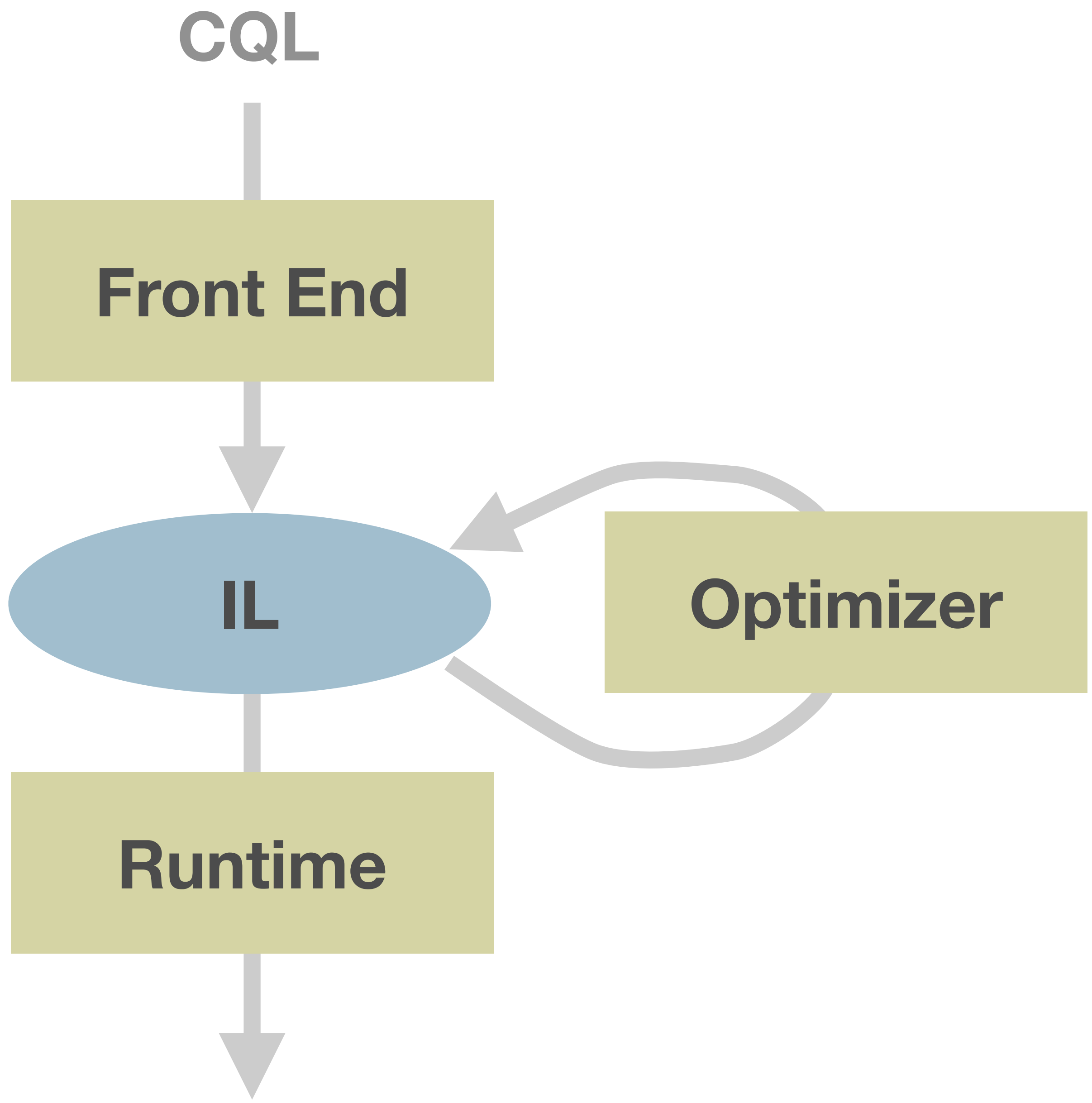
Optimizer

Distributed Runtime

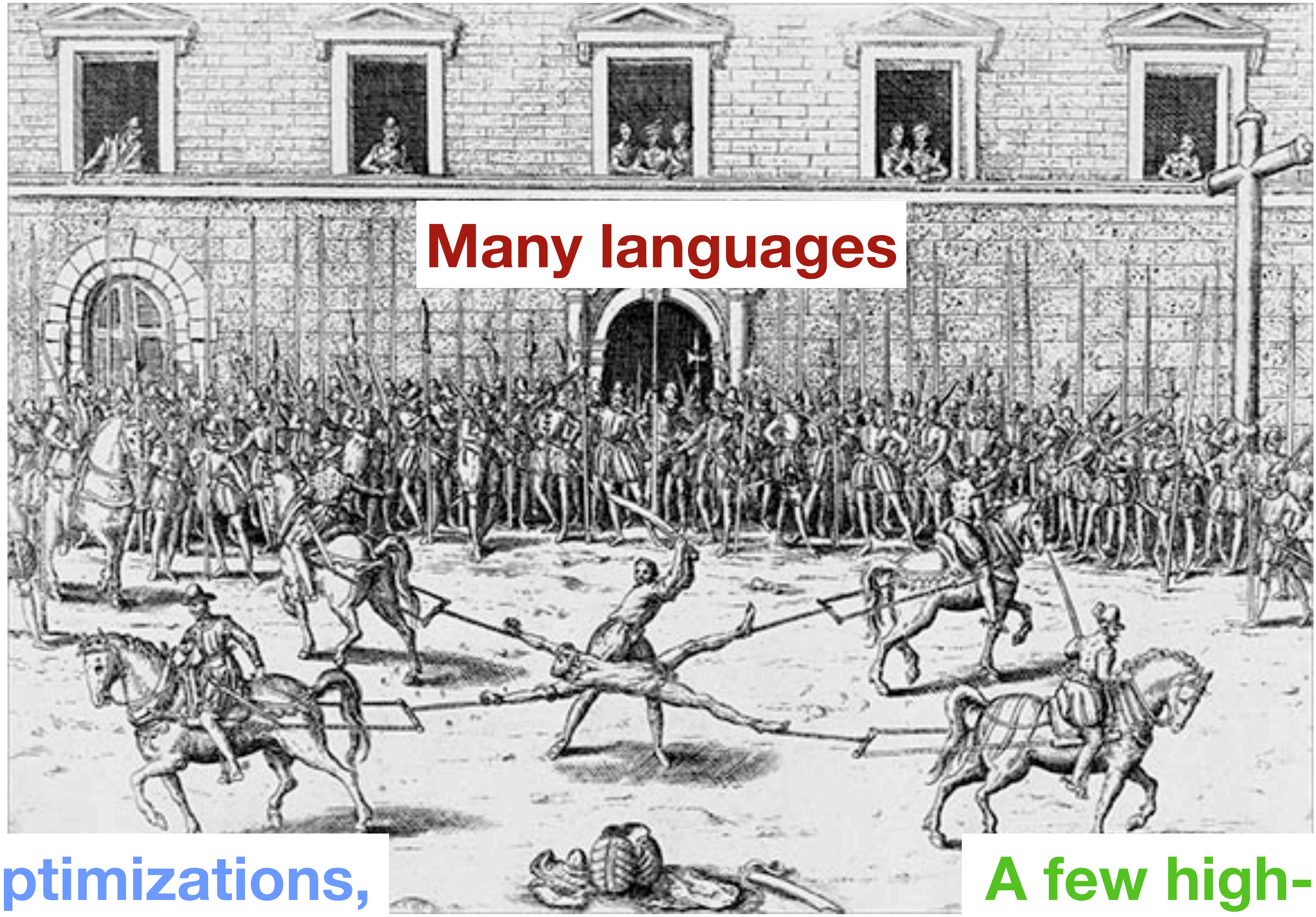
The Big Con:
A PL Talk at a DB Summit

Distributed CQL the Easy Way

- Translate source language to an *intermediate language* (IL)
- Optimize at the IL level directly
- Map IL to an existing distributed runtime



Design Tension For IL



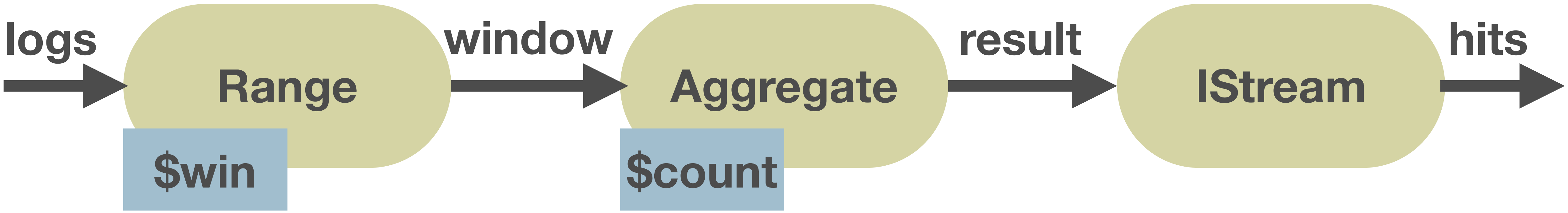
Many languages

Many optimizations, but a few core ones

A few high-quality runtimes

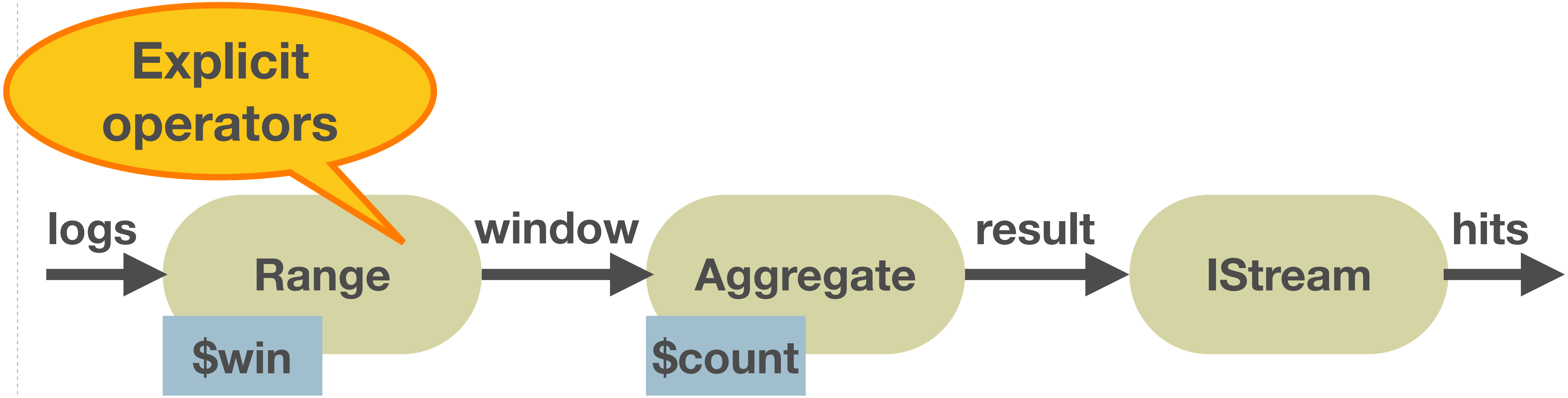
River, a Streaming IL: Make Everything Explicit

```
output hits;
input logs;
(window, $win) <-Range(logs, $win)                @{parallel, commutes, keys=[ ]};
(result,$count)  <-Aggregate(window, $count)@{parallel, commutes, keys=[origin]};
             (hits) <-IStream(result)               @{parallel};
```



River, a Streaming IL: Make Everything Explicit

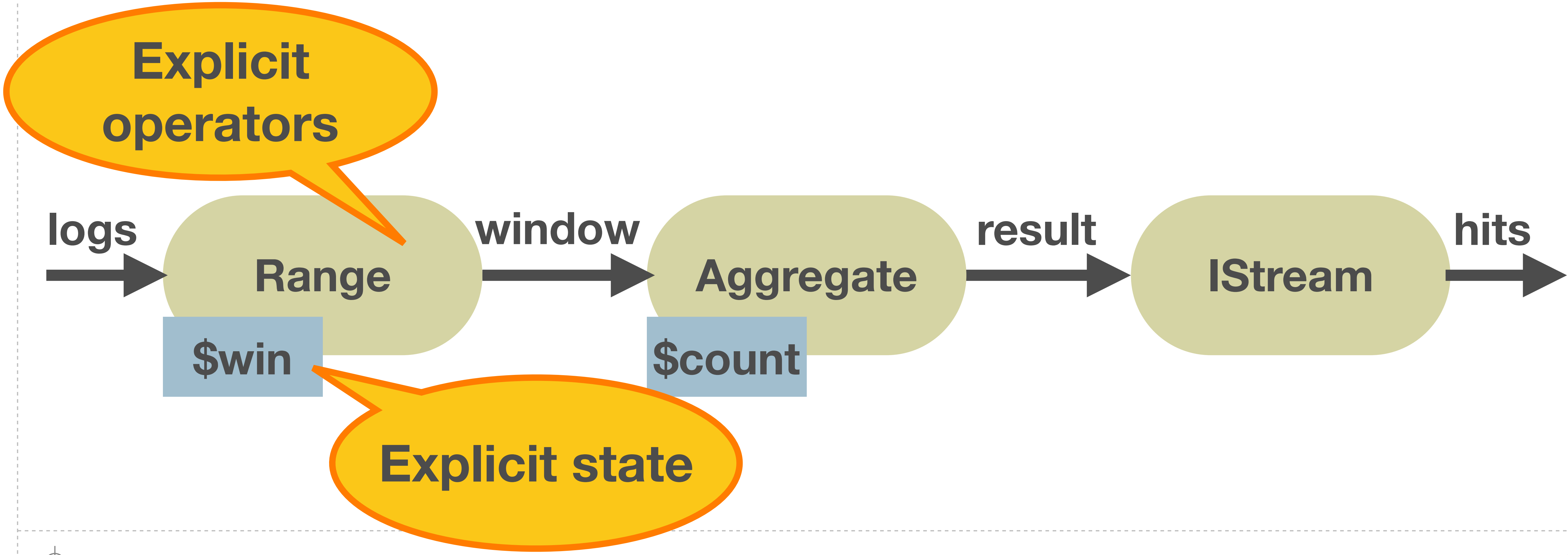
```
output hits;
input logs;
(window, $win) <-Range(logs, $win)              @{parallel, commutes, keys=[ ]};
(result,$count)  <-Aggregate(window, $count)@{parallel, commutes, keys=[origin]};
          (hits) <-IStream(result)              @{parallel};
```



Explicit operators

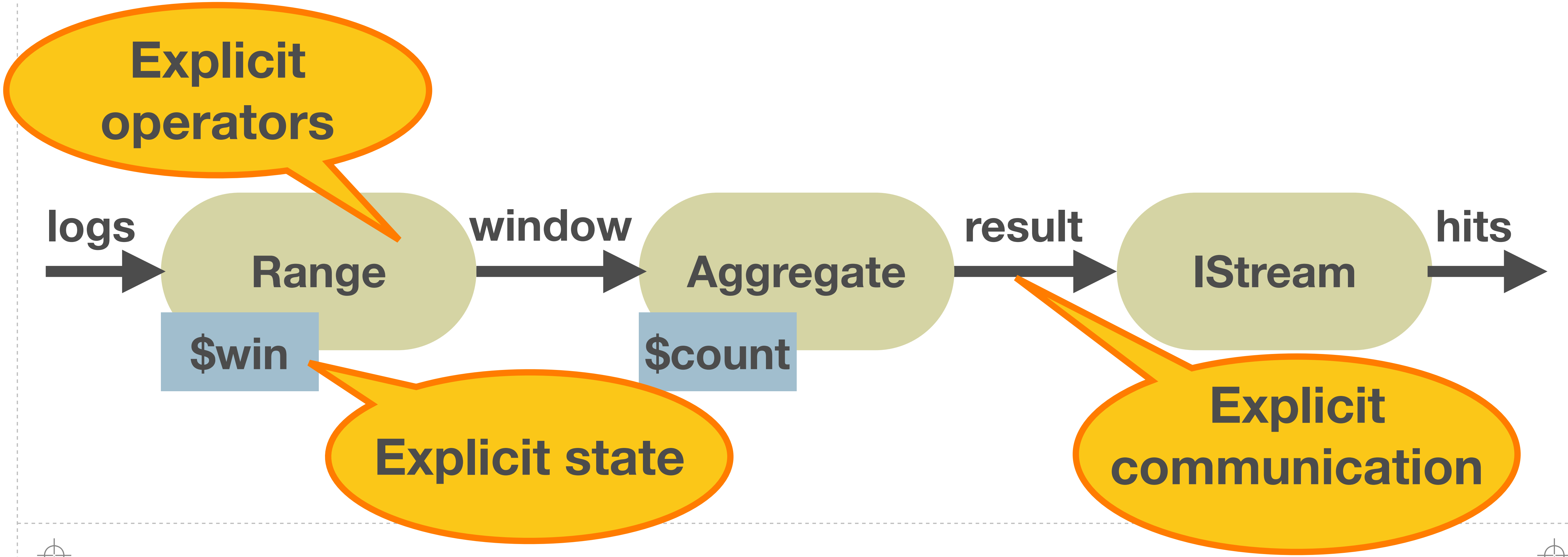
River, a Streaming IL: Make Everything Explicit

```
output hits;
input logs;
(window, $win) <-Range(logs, $win)              @{parallel, commutes, keys=[ ]};
(result,$count)  <-Aggregate(window, $count)@{parallel, commutes, keys=[origin]};
            (hits) <-IStream(result)             @{parallel};
```



River, a Streaming IL: Make Everything Explicit

```
output hits;
input logs;
(window, $win) <-Range(logs, $win)              @{parallel, commutes, keys=[ ]};
(result,$count)  <-Aggregate(window, $count)@{parallel, commutes, keys=[origin]};
          (hits) <-IStream(result)              @{parallel};
```



Explicit operators

logs

Range

$win

window

Aggregate

$count

result

IStream

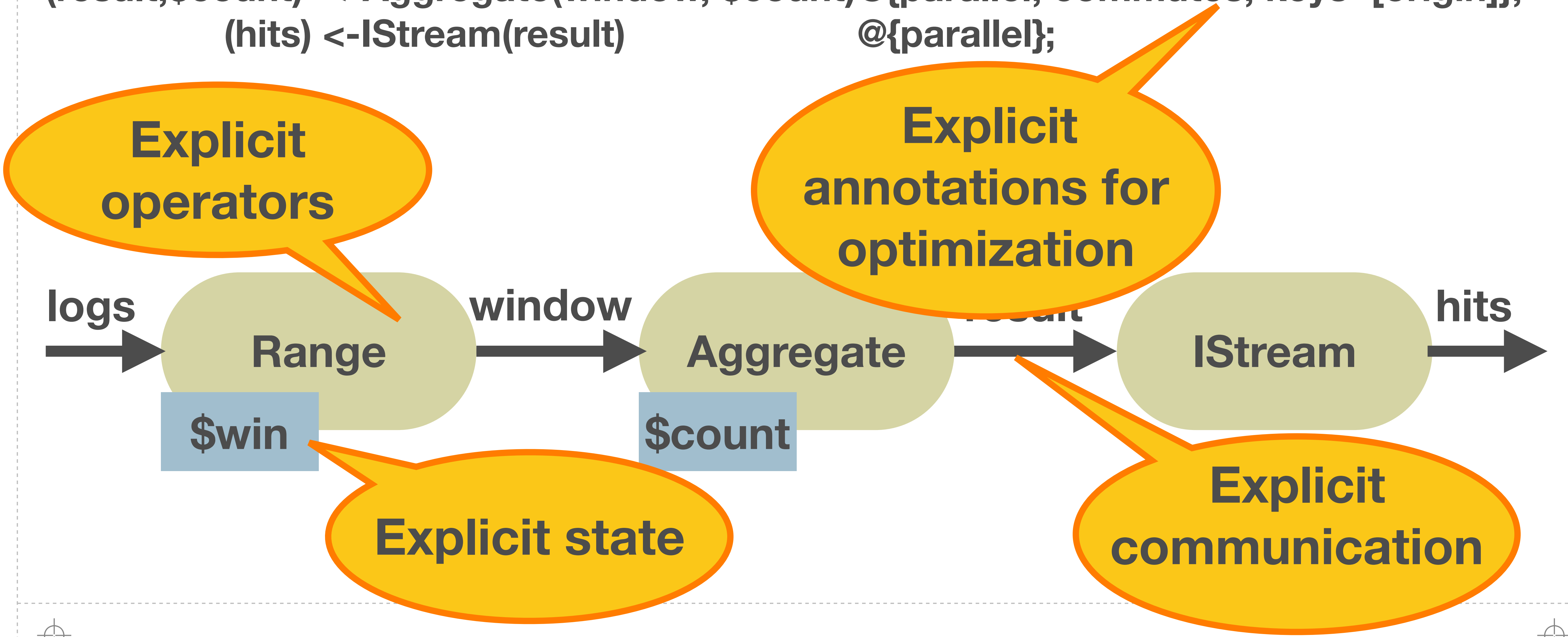
hits

Explicit state

Explicit communication

River, a Streaming IL: Make Everything Explicit

```
output hits;
input logs;
(window, $win) <-Range(logs, $win)              @{parallel, commutes, keys=[ ]};
(result,$count)  <-Aggregate(window, $count)@{parallel, commutes, keys=[origin]};
            (hits) <-IStream(result)              @{parallel};
```



Explicit operators

Explicit annotations for optimization

Explicit state

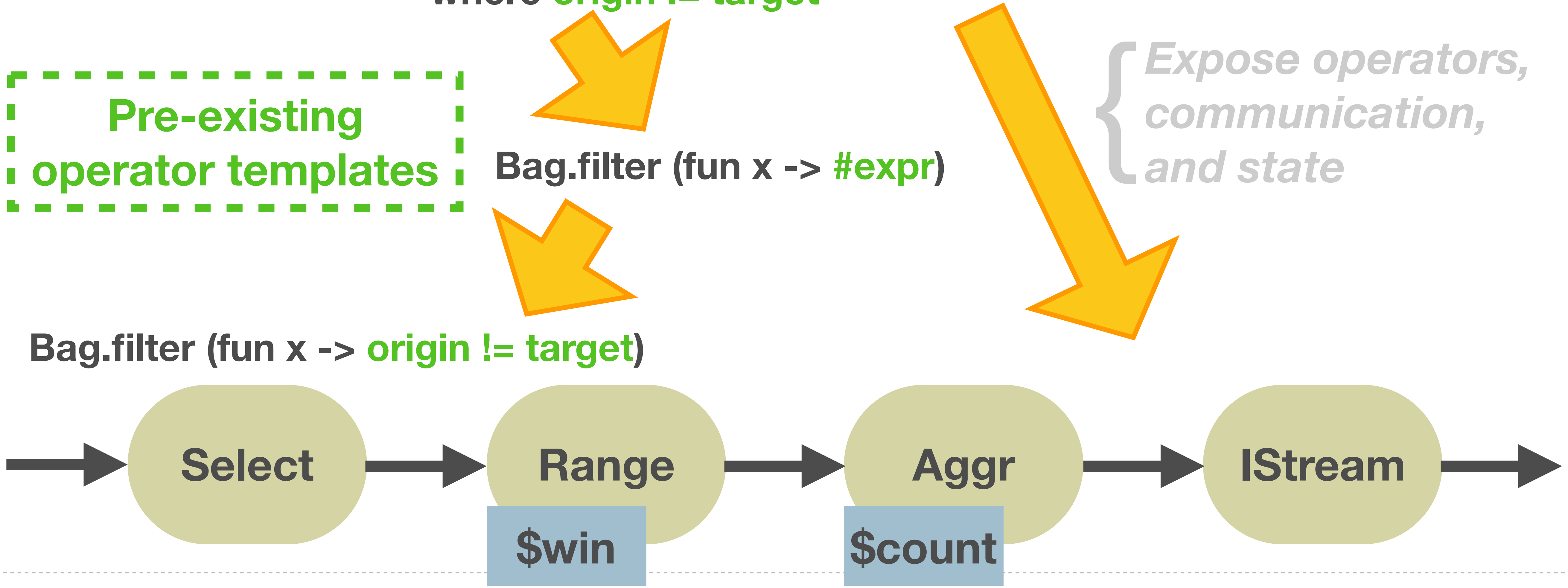
Explicit communication

An IL vs. a Query Plan

- Serves as a target for many languages
- Allows arbitrary operator graph, not restricted to a tree
- Allows arbitrary operators, not restricted to relational operators
- Makes all uses of state explicit
- Adds explicit properties for optimization

Translation

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



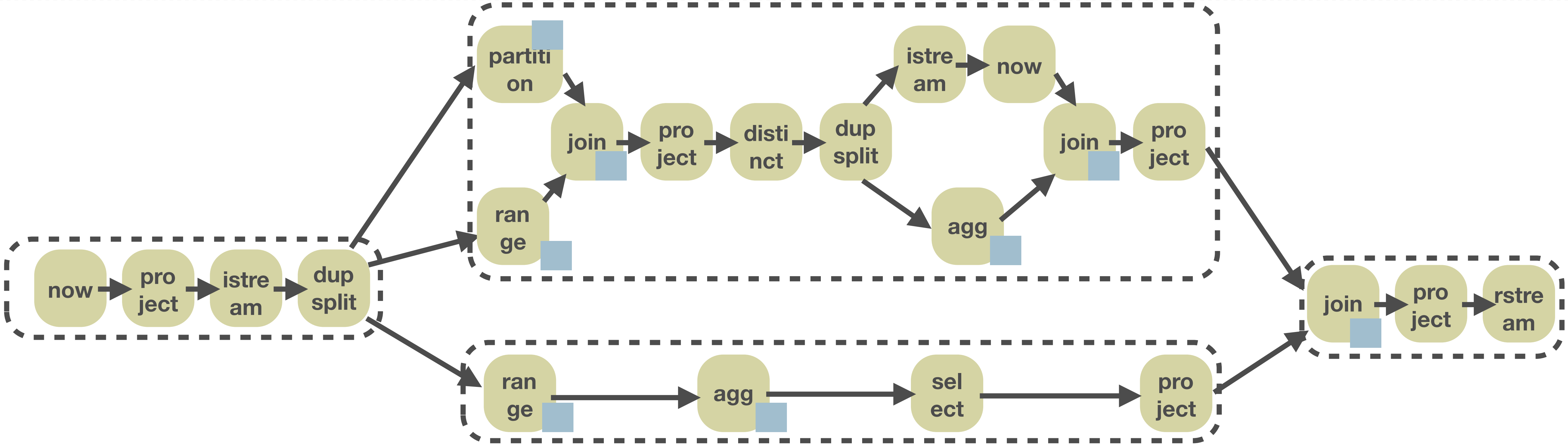
Pre-existing operator templates

Bag.filter (fun x -> #expr)

Expose operators, communication, and state

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

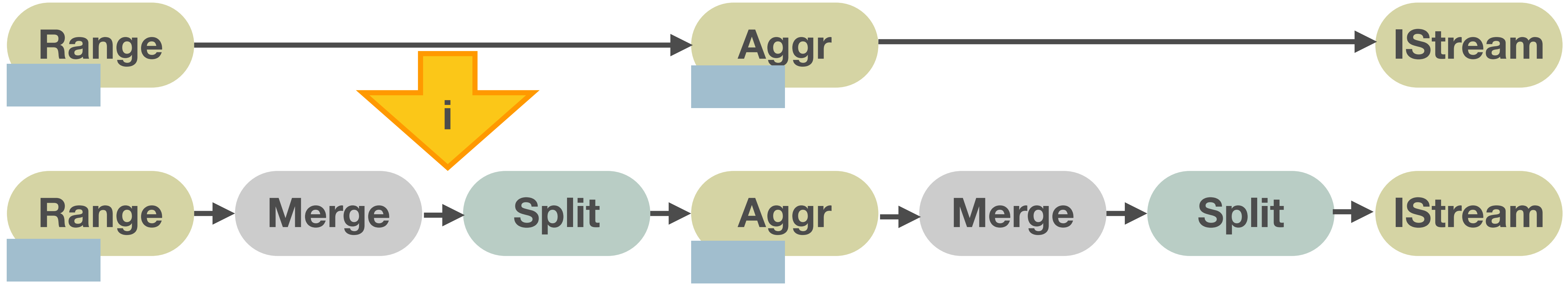
Changes for Distribution



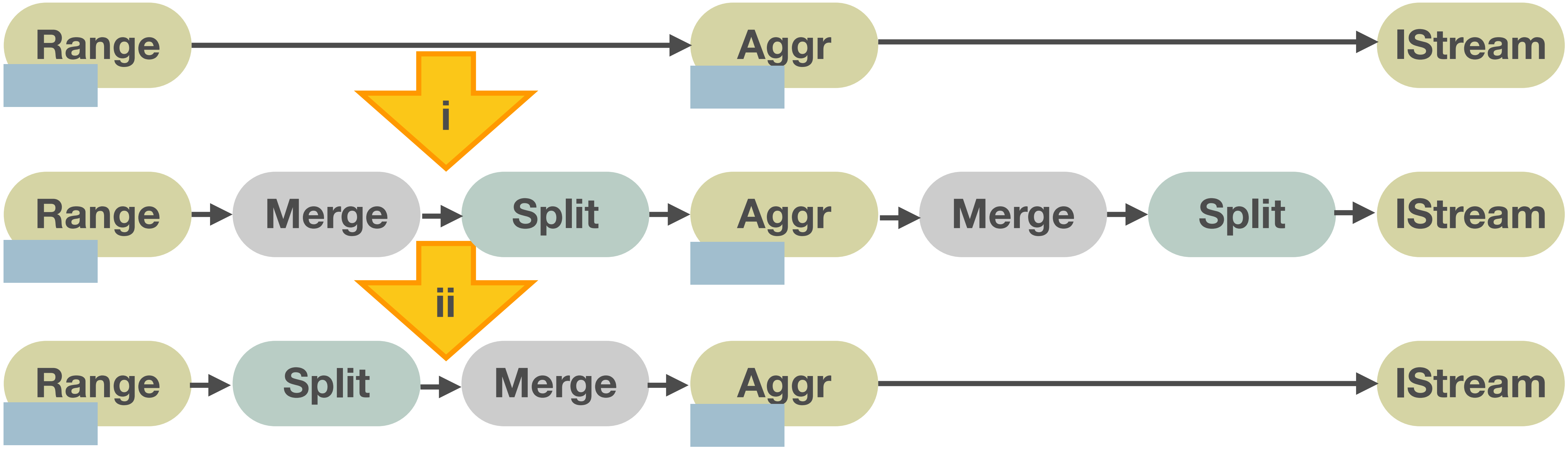
Original CQL	River CQL	Impact
Shared memory for operators and queues	Operator local memory	Don’t need distributed shared memory
Centralized scheduler	Each operator has its own thread and synchronization logic	Increased parallelism

Using Properties For Parallelization

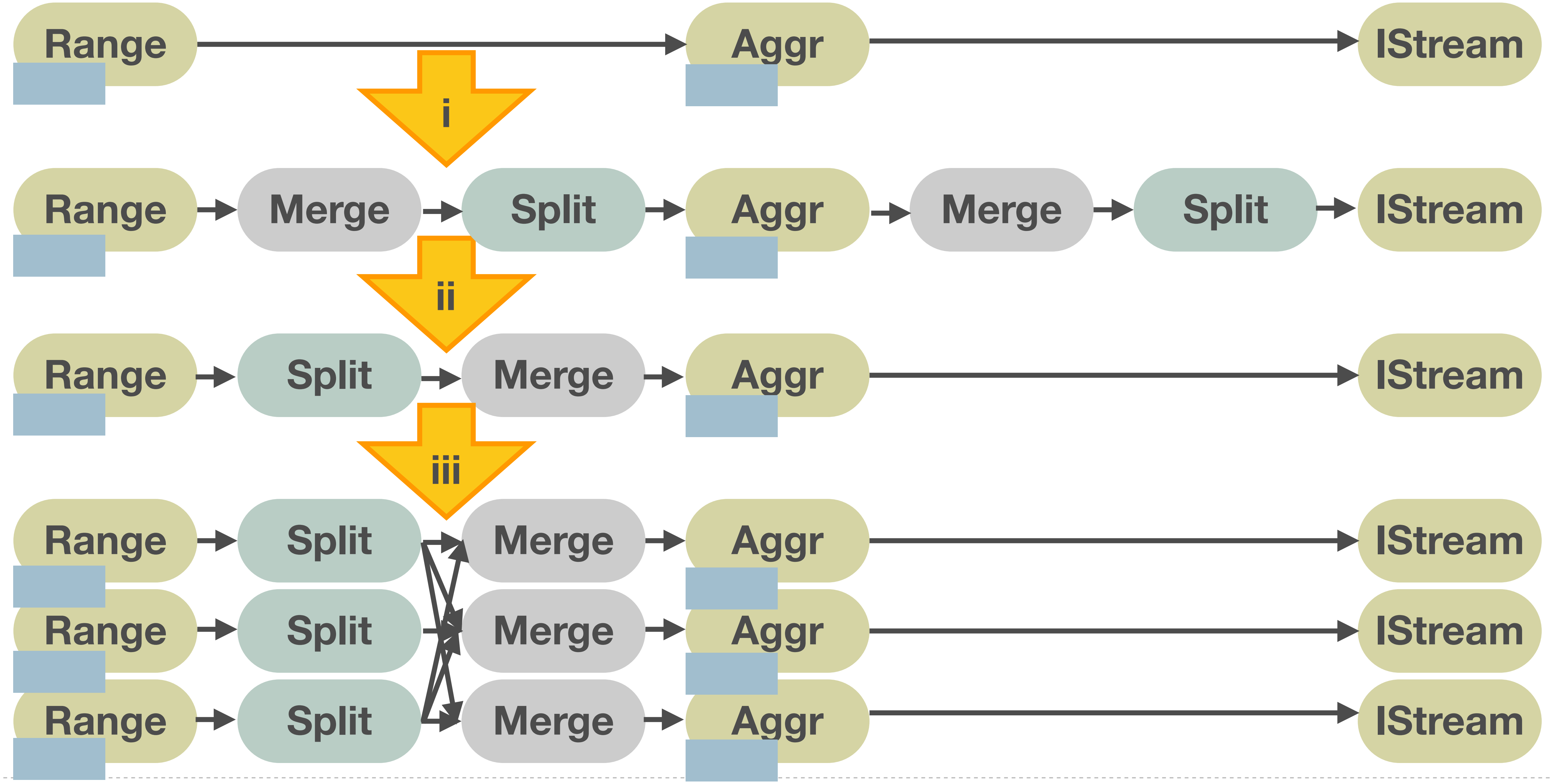
Using Properties For Parallelization



Using Properties For Parallelization



Using Properties For Parallelization



Start With an Existing Runtime

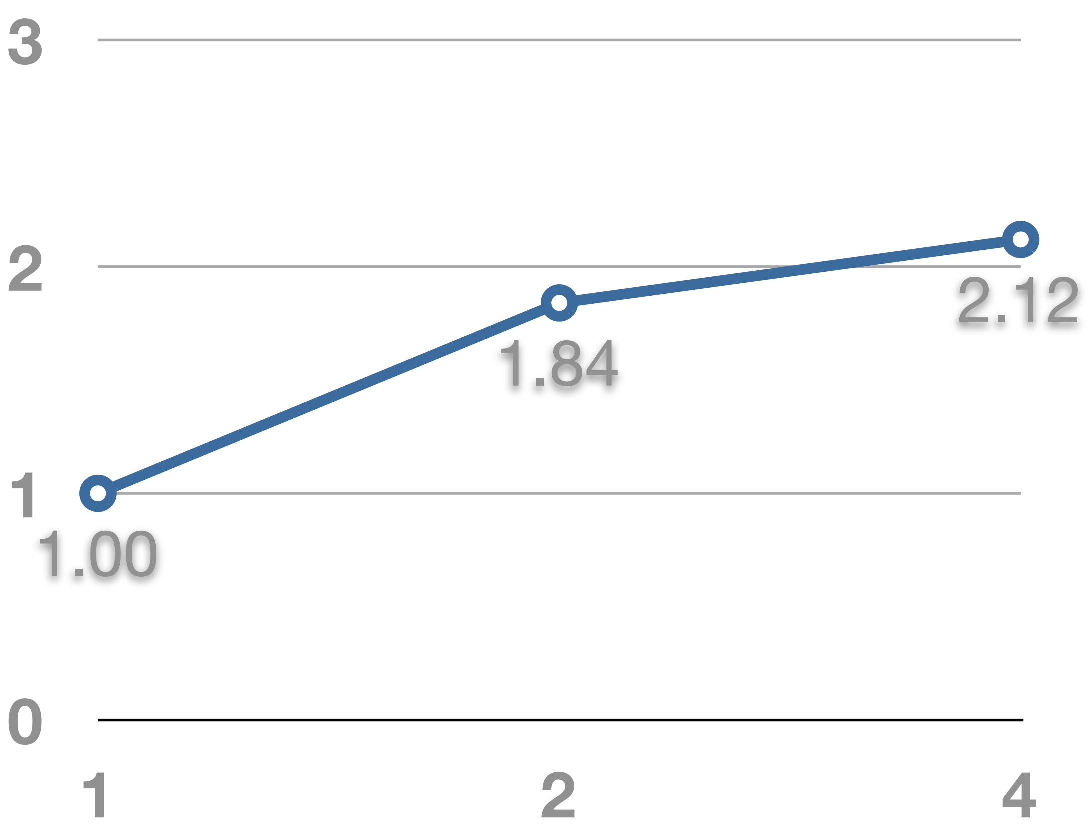
- Map from River to an existing streaming runtime
    - IBM’s streaming platform, System S
- Shared-nothing cluster of commodity machines
- Main abstractions: graph of streams and operators

It Works!

- Prototype runs on IBM’s System S
- Two benchmark applications
  - Linear Road on 1, 2, and 4 machines shows distribution
  - Web log query analyzer on 1-16 machines shows parallelism
- Results are promising, but our synchronization is a bottleneck

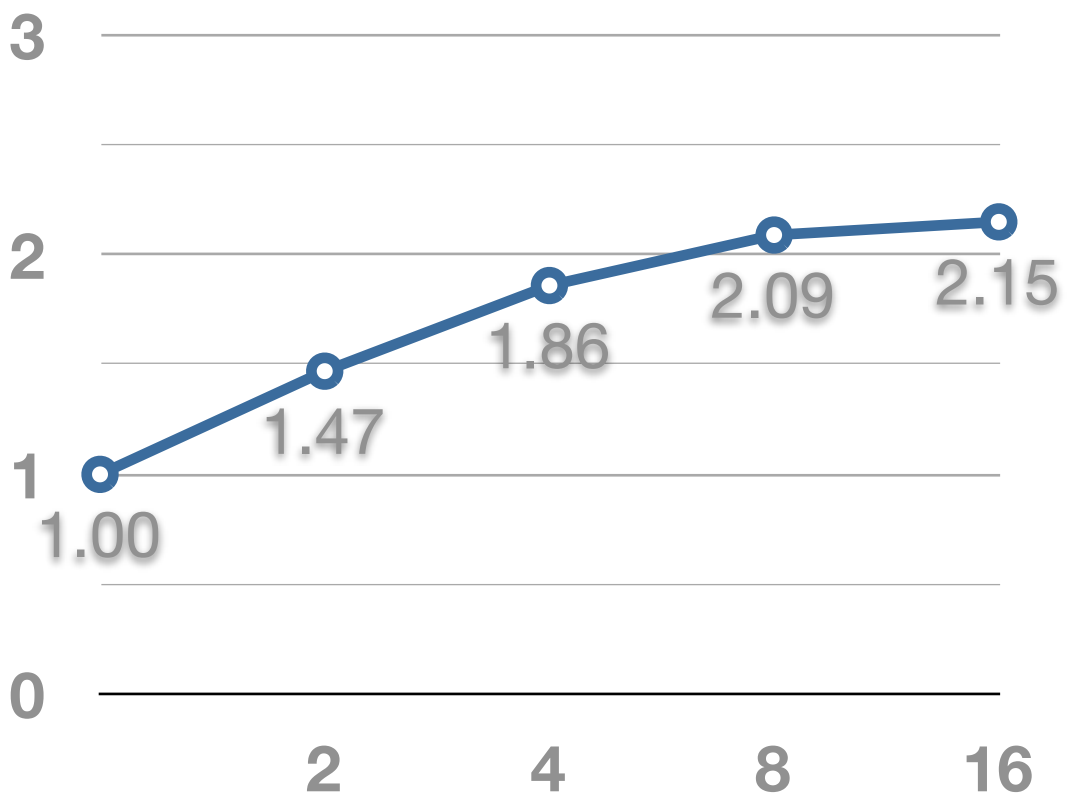
CQL Parallelization Has Limited Effect

Linear Road Speedup



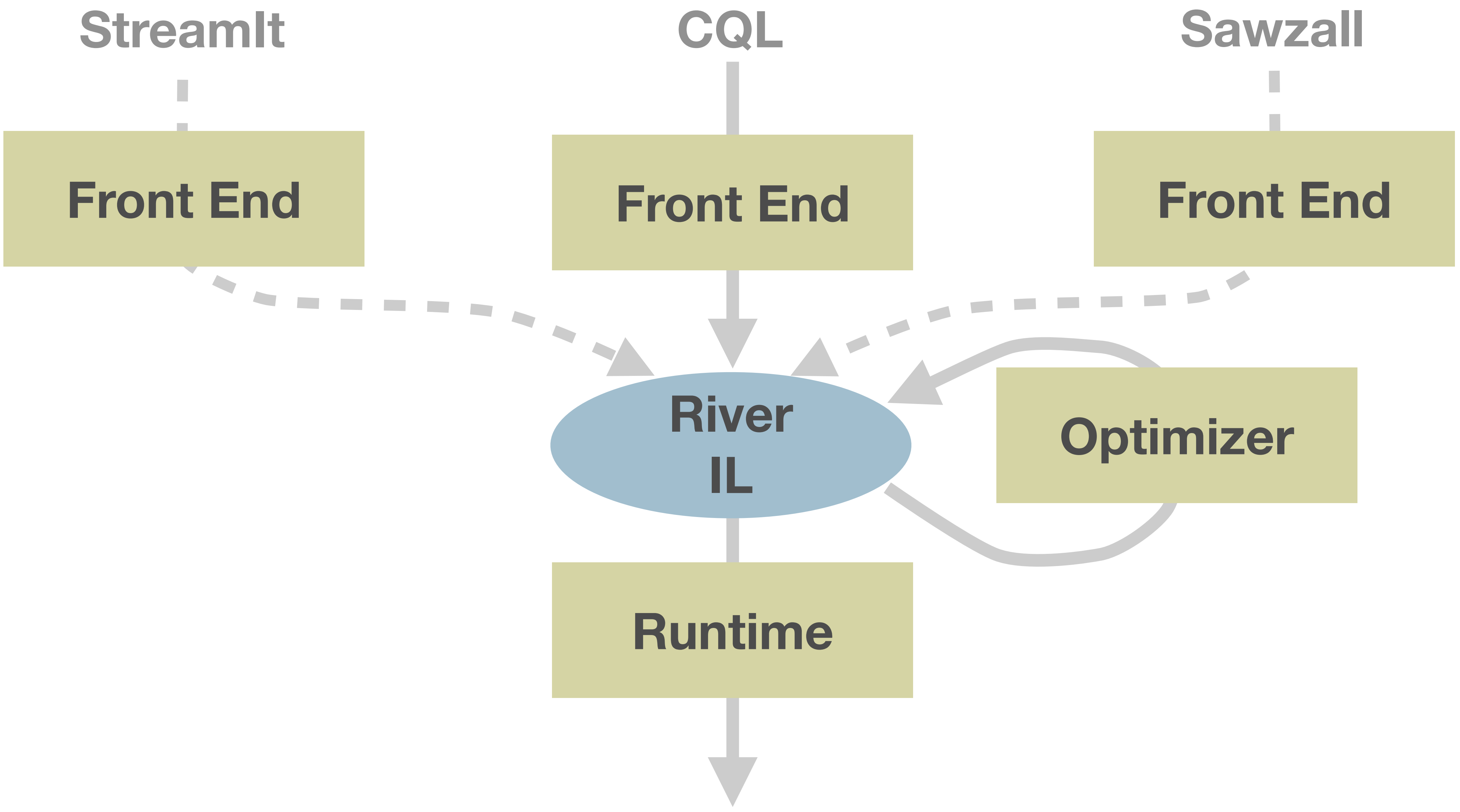
- 2.12x speedup on 4 machines
- Limited task and pipeline parallelism

CQL Log Analyzer Speedup

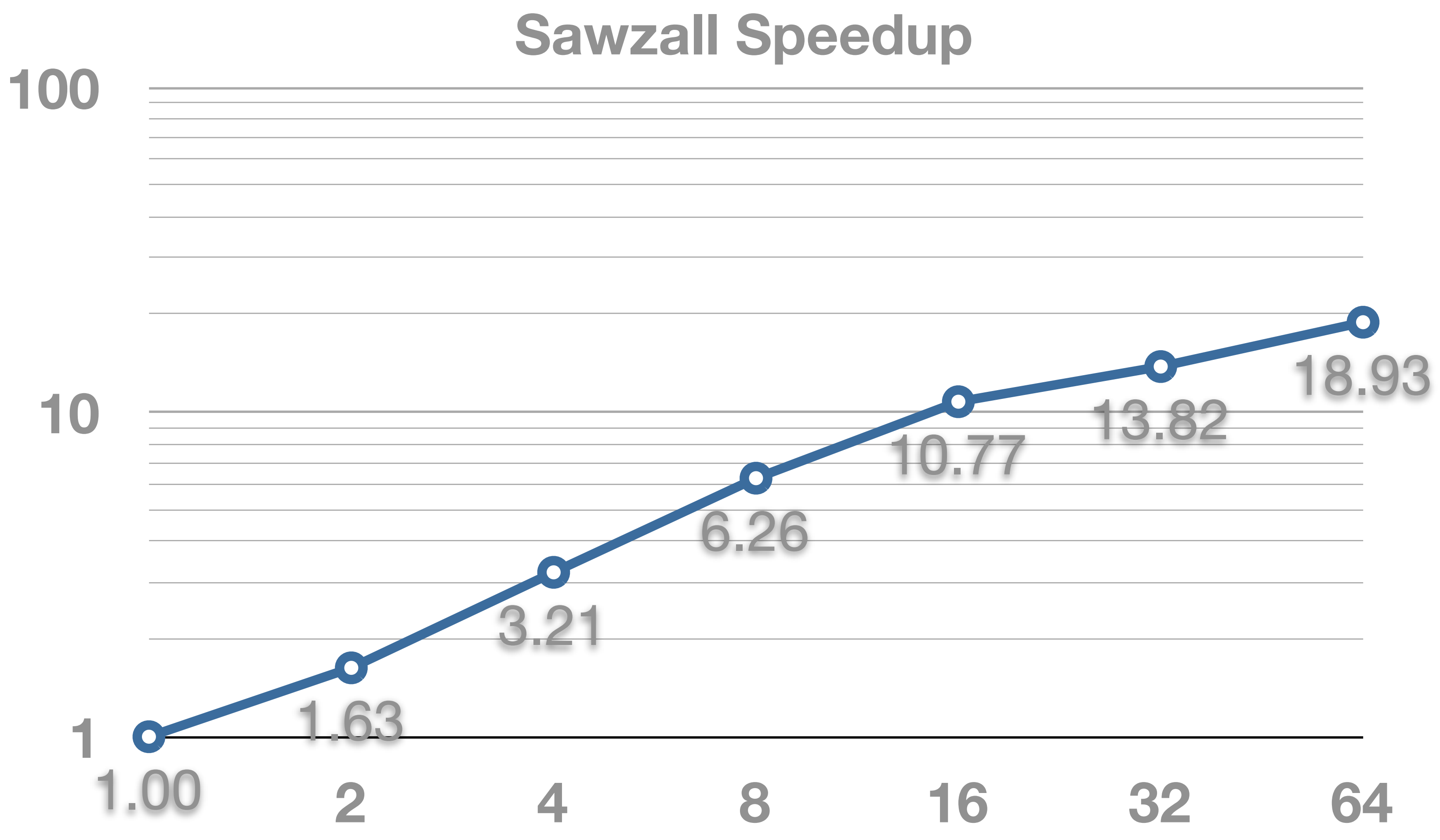


- 2.15x speedup on 16 machines
- Synchronization is bottleneck

It Works For Other Languages



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

Conclusion

- Streaming is everywhere and it needs language support
- A streaming IL makes it easier to implement a distributed CQL
    - Provides a lingua franca for mapping streaming languages to existing distributed runtimes
    - Provides a common substrate for optimizations

http://cs.nyu.edu/brooklet