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

Data-centric applications

Stream processing

Multicores and clusters

Languages and optimizations need to adapt
## Streaming Languages and Optimizations

<table>
<thead>
<tr>
<th>Streaming Languages</th>
<th>Streaming Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>CQL, StreamIt, Sawzall, Hancock, Gigascope, Lime, etc.</td>
<td>Fusion, fission, placement, reordering, etc.</td>
</tr>
<tr>
<td>Represent an application as a graph of streams and operators</td>
<td>Maximize utilization of available resources</td>
</tr>
<tr>
<td>Tailored to the needs of a particular application domain</td>
<td>Often re-write the data-flow graph</td>
</tr>
</tbody>
</table>
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
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.
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 forest)
- 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

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

Safety
- Preconditions for correctness

Profitability
- Throughput (higher is better)
- Micro-benchmark
- Runs on System S
- Relative numbers

Variations
- Most influential published papers

Central trade-off factor
- Dynamism
- How to optimize at runtime

Items highlighted in red will be addressed in this talk.
List of Optimizations

**Graph changed**
- Operator reordering
- Redundancy elimination
- Operator separation
- Fusion
- Fission

**Graph unchanged**
- Load balancing
- Placement
- State sharing
- Batching
- Algorithm Selection
- Load shedding

Semantics unchanged

Semantics changed

Saturday, May 19, 12
Operator Reordering

Move more selective operators upstream to filter data early.

Safety
- Commutative
- Attributes available

Profitability

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

Dynamism
- Eddy
Redundancy Elimination

Combine or remove redundant operators.

Safety
- Same algorithm
- Data available

Profitability

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

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
- Separating Aggregation
  - Not separated
  - Separated

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

Variations

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

Profitability

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

Dynamism

Throughput vs. Number of Cores

- p/s/o = 1/1/0
- p/s/o = 1/0/1
- p/s/o = 1/0/0
Assignment operators to hosts and cores.

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

**Profitability**
- Submission-time
- Online, via operator migration

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

Profitability

Variations

- Sharing queues
- Sharing windows
- Sharing operator state

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

- $A_\alpha(s) \equiv A_\beta(s)$
- 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

- ![Graph showing load shedding impact on throughput and accuracy](image)

Dynamism

- **Always dynamic**
Optimizations Enable Optimizations

Traditional → Stream
Stream → Traditional
Stream → Stream
Stream → Traditional

Traditional compiler analyses
Operator reordering
Fission
Load balancing
Redundancy elimination
Batching
State sharing
Traditional compiler optimizations
Placement
Fusion
Operator separation
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

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

```
Queue --> Operator --> Queue

State
```

Saturday, May 19, 12
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

Make explicit
Requirements for Calculus

Make explicit

Make explicit and 1-to-1
Requirements for Calculus

Make trigger non-deterministic

Make explicit

Make explicit and 1-to-1

Saturday, May 19, 12
Requirements for Calculus

- Make trigger non-deterministic
- Make explicit
- Make explicit and 1-to-1
- Treat functions as opaque
Brooklet Syntax

\[
\text{(volume, $\text{total}$)} \leftarrow \text{Sum(trades, $\text{total}$)}
\]
Function Environment

F: The function implementations

trades → volume

$\text{total}$
Queue Store

Q: The contents of the queues

trades $\rightarrow$ Sum $\rightarrow$ volume

$\text{Sum} \leftarrow \text{trades} \rightarrow \text{volume}$

Saturday, May 19, 12
Variable Store

$\text{total}$

trades

Sum

volume

V: The contents of the variables
Brooklet
Operational Semantics

\[ F \vdash <Q, V> \rightarrow <Q', V'> \]
Brooklet syntax:

*Brooklet program*

\[ P_b ::= \text{out in } \overline{op} \]

*Output declaration*

\[ \text{out ::= output } \overline{q} ; \]

*Input declaration*

\[ \text{in ::= input } \overline{q} ; \]

*Operator*

\[ \text{op ::= ( } \overline{q}, \overline{v} ) \leftarrow f( \overline{q}, \overline{v} ) \]

*Queue identifier*

\[ q ::= \text{id} \]

*Variable identifier*

\[ v ::= \$ \text{id} \]

*Function identifier*

\[ f ::= \text{id} \]

Brooklet example: IBM market maker.

output result;
input bids, asks;
(\text{ibmBids}) \leftarrow \text{SelectIBM(bids)};
(\text{ibmAsks}) \leftarrow \text{SelectIBM(asks)};
(\$\text{lastAsk}) \leftarrow \text{Window(\text{ibmAsks})};
(\text{ibmSales}) \leftarrow \text{SaleJoin(\text{ibmBids},\$\text{lastAsk})};
(\text{result},\$\text{cnt}) \leftarrow \text{Count(\text{ibmSales},\$\text{cnt})};

Brooklet semantics:

\[ F_b \vdash \langle V, Q \rangle \longrightarrow \langle V', Q' \rangle \]

\[ d, b = Q(q_i) \]

\[ op = (\_, \_) \leftarrow f(\overline{q}, \overline{v}) ; \]

\[ (\overline{b'}, \overline{d'}) = F_b(f)(d, i, V(\overline{v})) \]

\[ V' = \text{updateV}(op, V, \overline{d'}) \]

\[ Q' = \text{updateQ}(op, Q, q_i, \overline{b'}) \]

\[ F_b \vdash \langle V, Q \rangle \longrightarrow \langle V', Q' \rangle \]

(E-FIREQUEUE)

\[ \text{updateV}(op, V, \overline{d}) = [\overline{v} \mapsto \overline{d}] V \]

(E-UPDATEV)

\[ op = (\overline{q}, \_) \leftarrow f(\_, \_); \]

\[ d_f, b_f = Q(q_f) \]

\[ Q' = [q_f \mapsto b_f] Q \]

\[ Q'' = [\forall q_i \in \overline{q} : q_i \mapsto Q(q_i), b_i] Q' \]

(E-UPDATEQ)

\[ \text{updateQ}(op, Q, q_f, \overline{b}) = Q'' \]
Example: A Fannie Mae Bid/Ask Join

- $\text{lastAsk}$
- $\text{lastBid}$
- $\text{asks}$
- $\text{bids}$
- $\text{trades}$
- $\text{total}$
- $\text{volume}$
Example: A Fannie Mae Bid/Ask Join

$\text{lastAsk} = <\text{FNM}, 0, \infty>$

$\text{lastAsk} = <\text{FNM}, 1, \€2>$

$\text{lastBid} = <\text{FNM}, 0, \€1>$

$\text{lastBid} = <\text{FNM}, 1, \€2>$

$\text{lastBid} = <\text{FNM}, 0, 0>$

$\text{total} = 0$

$\text{total} = 0$

$\text{volume}$
Example: A Fannie Mae Bid/Ask Join

$lastAsk = <FNM,0,\infty>

$lastAsk

<FNM,1,\€2>

asks

bids

$lastBid = <FNM,0,0>

$lastBid

SaleJoin

$total = 0

$total

trades

volume

Total volume:

$lastAsk = <FNM,0,\infty>

$lastBid = <FNM,0,0>

Saturday, May 19, 12
Example: A Fannie Mae Bid/Ask Join

$lastAsk = \langle FNM, 0, \infty \rangle$

$lastAsk = \langle FNM, 0, 0 \rangle$

$lastBid = \langle FNM, 0, 0 \rangle$

$lastBid = \langle FNM, 1, \epsilon 1 \rangle$

$lastBid = \langle FNM, 1, \epsilon 2 \rangle$

$lastAsk = \langle FNM, 1, \epsilon 2 \rangle$

$total = 0$

$total = 0$

Saturday, May 19, 12
Example: A Fannie Mae Bid/Ask Join

$\text{lastAsk} = \langle \text{FNM}, 0, \infty \rangle$

$\langle \text{FNM}, 1, \€2 \rangle$

$\langle \text{FNM}, 1, \€2 \rangle$

$\text{lastBid} = \langle \text{FNM}, 0, 0 \rangle$

$\text{lastBid} = \langle \text{FNM}, 1, \€1 \rangle$

$\text{total} = 0$

Saturday, May 19, 12
Example:
A Fannie Mae Bid/Ask Join

\( \text{total} = 0 \)

\( \text{lastBid} = \langle \text{FNM}, 1, \text{€1} \rangle \)

\( \text{lastAsk} = \langle \text{FNM}, 0, \infty \rangle \)
Example: A Fannie Mae Bid/Ask Join

$\text{lastAsk} = \langle \text{FNM}, 0, \infty \rangle$

$\langle \text{FNM}, 1, \€2 \rangle$

$\text{asks}$

$\text{lastAsk}$

$\langle \text{FNM}, 1, \€2 \rangle$

$\text{bids}$

$\text{SaleJoin}$

$\text{trades}$

$\text{lastBid} = \langle \text{FNM}, 1, \€1 \rangle$

$\text{total} = 0$

$\text{total} = 0$

$\text{volume}$

$\text{Sum}$
Example: A Fannie Mae Bid/Ask Join

$lastAsk = <FNM,0,\infty>$

$lastAsk = <FNM,1,\€2>$

$lastBid = <FNM,1,\€1>$

$total = 0$

Saturday, May 19, 12
Example: A Fannie Mae Bid/Ask Join

$lastAsk = <FNM,0,\infty>

$lastAsk

$lastBid

$lastBid = <FNM,1,\euro1>

$total = 0

$total

tradess

volume
Example:
A Fannie Mae Bid/Ask Join

$lastAsk = <FNM,0,\infty>
$lastAsk = <FNM,1,\€2>

$lastBid = <FNM,1,\€1>

$total = 0
$total = 0

Saturday, May 19, 12
Example:
A Fannie Mae Bid/Ask Join

\$lastAsk = \langle FNM, 1, \text{€2} \rangle

\$lastBid = \langle FNM, 1, \text{€1} \rangle

\$total = 0
Example: A Fannie Mae Bid/Ask Join

$\text{lastAsk} = \langle \text{FNM}, 1, \€2 \rangle$

$\text{lastAsk}$

$\text{lastBid} = \langle \text{FNM}, 1, \€1 \rangle$

$\text{lastBid}$

$\text{total} = 0$

$\text{total}$

Saturday, May 19, 12
Example: A Fannie Mae Bid/Ask Join

\[ \text{lastAsk} = \langle \text{FNM}, 1, 2\rangle \]

\[ \text{total} = 0 \]

\[ \text{lastBid} = \langle \text{FNM}, 1, 1\rangle \]
Example: A Fannie Mae Bid/Ask Join

$lastAsk = \langle FNM, 1, €2 \rangle$

$lastAsk$

$lastBid$

$lastBid = \langle FNM, 1, €1 \rangle$

$lastAsk$

$lastBid$

$total = 0$

$total$

$lastAsk$

$lastBid$

$lastAsk$

$lastBid$

trades

volume

Sat, May 19, 12
Example: A Fannie Mae Bid/Ask Join

\[
\text{\$lastAsk} = \langle \text{FNM}, 1, \varepsilon 2 \rangle \\
\text{\$lastBid} = \langle \text{FNM}, 1, \varepsilon 1 \rangle \\
\text{\$total} = 0
\]
Example: A Fannie Mae Bid/Ask Join

-$lastAsk = \langle FNM, 1, \text{€2} \rangle$

-$lastAsk$

-asks

-$lastBid$

-bids

-$lastBid = \langle FNM, 1, \text{€1} \rangle$

-$total = 0$

-$total$

-trades

-SaleJoin $\langle FNM, 1, \text{€2} \rangle$

-Sum

-volume

Saturday, May 19, 12
Example: A Fannie Mae Bid/Ask Join

$lastAsk = <FNM,1,€2>

$lastBid = <FNM,1,€2>

$total = 0
Example: A Fannie Mae Bid/Ask Join

- $\text{lastAsk} = \langle \text{FNM,1,€2} \rangle$
- $\text{lastBid} = \langle \text{FNM,1,€2} \rangle$
- $\text{total} = 0$

**Diagram:**
- SaleJoin
  - $\text{Bids}$
  - $\text{Asks}$
- $\text{trades}$
- $\text{Sum}$
- $\text{volume}$
- $\text{total} = 0$
Example:  
A Fannie Mae Bid/Ask Join  

$lastAsk = <FNM,1,€2>$  

$lastBid = <FNM,1,€2>$  

$total = 0$  

Saturday, May 19, 12
Example: A Fannie Mae Bid/Ask Join

$lastAsk = <FNM,1,€2>

$lastBid = <FNM,1,€2>

$total = 0

Saturday, May 19, 12
Example:
A Fannie Mae Bid/Ask Join
Example:
A Fannie Mae Bid/Ask Join

$lastAsk = <FNM,1,€2>

SaleJoin

$lastAsk

trades

$lastBid

$lastBid = <FNM,1,€2>

$total = 0

<FN,1>

$total

volume

<FN,1>

Sum

Saturday, May 19, 12
Example: A Fannie Mae Bid/Ask Join

$\text{lastAsk} = \langle \text{FNM,1,€2} \rangle$

$\text{lastBid} = \langle \text{FNM,1,€2} \rangle$

$\text{total} = 1$

Saturday, May 19, 12
Example: A Fannie Mae Bid/Ask Join

$lastAsk = <FNM,1,\€2>

$lastBid = <FNM,1,\€2>

$total = 1

SaleJoin

trades

volume
Example: A Fannie Mae Bid/Ask Join

\[
\text{\$lastAsk} = \langle \text{FNM,1,}\$2\rangle
\]

\[
\text{\$lastBid} = \langle \text{FNM,1,}\$2\rangle
\]

\[
\text{\$total} = 1
\]
Translations

Demonstrating Brooklet’s generality by translating three rather diverse streaming languages
CQL, StreamIt, Sawzall: One Translation Approach

- Expose graph topology
- Expose implicit and explicit state
- Wrap original operators in higher-order functions

Functions \(<\) Queues , Variables \(>\)
CQL, StreamIt, Sawzall: One Translation Approach

- Expose graph topology
- Expose implicit and explicit state
- Wrap original operators in higher-order functions

Make queues explicit and 1-to-1
CQL, StreamIt, Sawzall: One Translation Approach

- Expose graph topology
- Expose implicit and explicit state
- Wrap original operators in higher-order functions
  - Make queues explicit and 1-to-1
  - Make state explicit

Functions $\leftarrow$ Queues, Variables $\rightarrow$
CQL, StreamIt, Sawzall: One Translation Approach

Do not model local computations

Wrap original operators in higher-order functions

Expose graph topology

Expose implicit and explicit state

Make queues explicit and 1-to-1

Make state explicit

Functions → < Queues , Variables >
Example: CQL to Brooklet

\[ \text{select } \text{Sum}(\text{shares}) \text{ from } \text{trades} \text{ where } \text{trades.ticker} = \text{“FNM”} \]
Example: CQL to Brooklet

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

CQL

Brooklet

Make queues explicit and 1-to-1
Example: CQL to Brooklet

select \text{Sum}(\text{shares}) \text{ from } \text{trades}
\text{ where } \text{trades.ticker} = "\text{FNM}"

\textit{CQL}

\textit{Brooklet}

\textbf{Make queues explicit and 1-to-1}

\textbf{Make state explicit}
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"
```
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

<table>
<thead>
<tr>
<th>Brooklet</th>
<th>River</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence of atomic steps</td>
<td>Operators execute concurrently</td>
</tr>
<tr>
<td>Pure functions, state threaded through invocations</td>
<td>Stateful functions, protected with automatic locking</td>
</tr>
<tr>
<td>Non-deterministic execution</td>
<td>Restricted execution with bounded queues, and back-pressure</td>
</tr>
<tr>
<td>Opaque functions</td>
<td>Function implementations</td>
</tr>
<tr>
<td>No physical platform, independent from runtime</td>
<td>Abstract representation of runtime e.g. placement</td>
</tr>
<tr>
<td>Finite execution</td>
<td>Indefinite execution</td>
</tr>
</tbody>
</table>
Concurrent Execution: No Shared State
Concurrent Execution: No Shared State

Single threaded

O₁ → O₂ → O₃

$x$ → $y$ → $z$
Concurrent Execution: No Shared State

Single threaded

Atomic queue operations

$O_1$ $O_2$ $O_3$

$x$ $y$ $z$
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 the 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

Minimal locking
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

1. Acquire locks
Restricted Execution: Safe Back-Pressure

1. Acquire locks
2. Fire operator
Restricted Execution: Safe Back-Pressure

1. Acquire locks
2. Fire operator
3. Data on local queue
1. Acquire locks

2. Fire operator

3. Data on local queue

4. Release locks
Restricted Execution: Safe Back-Pressure

1. Acquire locks
2. Fire operator
3. Data on local queue
4. Release locks
5. Data on output queue
Restricted Execution: Safe Back-Pressure

- **1. Acquire locks**
- **2. Fire operator**
- **3. Data on local queue**
- **4. Release locks**
- **5. Data on output queue**

- Only step 5 can block
- Locks have already been released, so O₃ 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 -> origin != target)

Expose operators, communication, and state
Translations with Modules

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

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

\[\text{CQL} = \text{SQL} + \text{Streaming} + \text{Expressions}\]
Optimization Support: Extensible Annotations

Source Language

River

Optimizer

Runtime

Establishes properties by construction, e.g. Sawzall reducers commute

Needs to know:
- Safety constraints
- Profitability

Establishes constraints, e.g. available resources

Needs to know:
- Safety constraints
- Profitability

Saturday, May 19, 12
Optimization Support: Extensible Annotations

Source Language

River

Optimizer

Runtime

Annotations convey this information

Establishes properties by construction
- e.g. Sawzall reducers commute

Establishes constraints
- e.g. available resources

Needs to know:
- Safety constraints
- Profitability

Saturday, May 19, 12
Optimization Support: Extensible Annotations

Annotations convey this information

Separate policy from mechanism

Source Language

River

Optimizer

Runtime

Establishes properties by construction, e.g. Sawzall reducers commute

Needs to know:
- Safety constraints
- Profitability

Establishes constraints, e.g. available resources
## Optimization Support: Current Annotations

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
<th>Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Fuse(ID)</td>
<td>Fuse operators with same ID in the same process</td>
<td>Fusion</td>
</tr>
<tr>
<td>@Parallel()</td>
<td>Perform fission on an operator</td>
<td>Fission</td>
</tr>
<tr>
<td>@Commutative()</td>
<td>An operator’s function is commutative</td>
<td>Fission</td>
</tr>
<tr>
<td>@Keys(k₁,...,kₙ)</td>
<td>An operator’s state is partitionable by the key fields k₁,...,kₙ</td>
<td>Fission</td>
</tr>
<tr>
<td>@Group(ID)</td>
<td>Place operators with same ID on the same machine</td>
<td>Placement</td>
</tr>
</tbody>
</table>
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

- **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

Saturday, May 19, 12
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

Stream processing

SVM
Labonte et al.
PACT’04

Runtime for executing IL on platforms

This Thesis

CQL
Arasu et al.
VLDB J.’06

Translators from languages to IL

P-Code
Nelson
CC’79
## Comparison to Traditional ILs

### Traditional IL
- For Pascal, Java, C#
- IL is lower-level
- Data at rest (registers)
- Instructions that run in a sequence, one after the other

### River IL
- For StreamSQL, Sawzall, StreamIt
- IL for explicit streaming topology
- Data in motion (queues)
- Functions that run in parallel, continuously

### Stream processing
- Runtime for executing IL on platforms
- Translators from languages to IL

### Traditional ILs
- P-Code
- Nelson CC'79
Comparison to CQL

**CQL**
- Described in terms of SRA (stream-relational algebra)
- Inter-dependent with a single runtime

**River IL**
- Uses more general streaming IL (not restricted to relational)
- Virtual, independent of any particular runtime

Stream processing
- CQL
- Arasu et al.
- VLDB J.’06
- Translators from languages to IL

Runtime for executing IL on platforms

---

Saturday, May 19, 12
## Comparison to SVM

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

**Stream processing**

- **SVM Labonte et al. PACT’04**
- Runtime for executing IL on platforms

**Translators from languages to IL**

+ 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

<table>
<thead>
<tr>
<th>Component</th>
<th>Limitations or Future Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimizations Catalog</td>
<td>Interaction of optimizations, compiler analysis, standard benchmarks</td>
</tr>
<tr>
<td>Brooklet</td>
<td>Relationship to other calculi, time constraints, more optimizations, dynamism</td>
</tr>
<tr>
<td>River</td>
<td>Support for dynamism, performance, design of new languages</td>
</tr>
</tbody>
</table>
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 ]_c^p = \langle F_b, P_b \rangle \]
\([ F_c, SName ]_c^p = \emptyset, output SName; input SName; \bullet \quad \text{(T}_c^p \text{-SNAME)}

\([ F_c, RName ]_c^p = \emptyset, output RName; input RName; \bullet \quad \text{(T}_c^p \text{-RNAME)}

\[ F_b, output q_o; \text{input } \overline{q}; \overline{op} = [ F_c, Pcs ]_c^p \]
\[ q_o' = \text{freshId}() \quad v = \text{freshId}() \]
\[ F'_b = [ S2R \mapsto \text{wrapS2R}(F_c(S2R))]F_b \]
\[ \overline{op}' = \overline{op}, (q_o', v) \leftarrow S2R(q_o, v); \]
\[ [ F_c, \text{S2R}(Pcs) ]_c^p = F'_b, output q_o'; \text{input } \overline{q}; \overline{op}' \quad \text{(T}_c^p \text{-S2R)} \]

\[ F_b, output q_o; \text{input } \overline{q}; \overline{op} = [ F_c, Pcr ]_c^p \]
\[ q_o' = \text{freshId}() \quad v = \text{freshId}() \]
\[ F'_b = [ R2S \mapsto \text{wrapR2S}(F_c(R2S))]F_b \]
\[ \overline{op}' = \overline{op}, (q_o', v) \leftarrow R2S(q_o, v); \]
\[ [ F_c, \text{R2S}(Pcr) ]_c^p = F'_b, output q_o'; \text{input } \overline{q}; \overline{op}' \quad \text{(T}_c^p \text{-R2S)} \]

\[ F_b, output q_o; \text{input } \overline{q}; \overline{op} = [ F_c, Pcr ]_c^p \]
\[ n = | Pcr | \quad q_o' = \text{freshId}() \quad \overline{q}' = \overline{q}_1, \ldots, \overline{q}_n \]
\[ \forall i \in 1 \ldots n : v_i = \text{freshId}() \quad \overline{op}' = \overline{op}_1, \ldots, \overline{op}_n \]
\[ F'_b = [ R2R \mapsto \text{wrapR2R}(F_c(R2R))](\cup F_b) \]
\[ \overline{op}' = \overline{op}', (q_o', \overline{v}) \leftarrow R2R(q_o, \overline{v}); \]
\[ [ F_c, \text{R2R}(Pcr) ]_c^p = F'_b, output q_o'; \text{input } \overline{q'}; \overline{op}'' \quad \text{(T}_c^p \text{-R2R)} \]

**CQL operator wrappers:**
\[ \sigma, \tau = d_q \quad s = d_v \]
\[ s' = s \cup \{ \langle e, \tau \rangle : e \in \sigma \} \quad \sigma' = f(s', \tau) \]
\[ \text{wrapS2R}(f)(d_q, \_, d_v) = \langle \sigma', \tau \rangle, s' \quad \text{(W}_c\text{-S2R)} \]
\[ \sigma, \tau = d_q \quad \sigma' = d_v \quad \sigma'' = f(\sigma, \sigma') \]
\[ \text{wrapR2S}(f)(d_q, \_, d_v) = \langle \sigma'', \tau \rangle, \sigma \quad \text{(W}_c\text{-R2S)} \]
\[ \forall j \neq i \in 1 \ldots n : d_j' = d_j \]
\[ \exists j \in 1 \ldots n : \# \sigma : \langle \sigma, \tau \rangle \in d_j \]
\[ \text{wrapR2R}(f)(d_q, i, \overline{d}) = \bullet, \overline{d} \quad \text{(W}_c\text{-R2R-WAIT)} \]
\[ \forall j \neq i \in 1 \ldots n : d_j' = d_j \]
\[ \forall j \in 1 \ldots n : \sigma_j = \text{aux}(d_j, \tau) \]
\[ \text{wrapR2R}(f)(d_q, i, \overline{d}) = \langle f(\overline{\sigma}), \tau \rangle, \overline{d}' \quad \text{(W}_c\text{-R2R-READY)} \]
\[ \langle \sigma, \tau \rangle \in d \quad \text{aux}(d, \tau) = \sigma \quad \text{(W}_c\text{-R2R-AUX)} \]
Operator Fission

\[ \text{op} = (q_{out}) \leftarrow f(q_{in}); \]
\[ \forall i \in 1 \ldots n : q_i = \text{freshId}() \quad \forall i \in 1 \ldots n : q'_i = \text{freshId}() \]
\[ F'_b, op_s = [\emptyset, \text{split roundrobin}, \overline{q}, q_{in}]^p_s \]
\[ \forall i \in 1 \ldots n : op_i = (q'_i) \leftarrow f(q_i); \]
\[ F''_b, op_j = [\emptyset, \text{join roundrobin}, q_{out}, \overline{q'}]^p_s \]
\[ \langle F_b, \text{op} \rangle \rightarrow^N_{\text{split}} \langle F_b \cup F'_b \cup F''_b, op_s \overline{\text{op}} \text{ op}_j \rangle \]
Dynamism

- Compile time
- Submission time
- Runtime disruptive
- Runtime nimble

- Operator separation
- Redundancy elimination
- Load balancing
- Operator reordering

- Fusion
- Fission

- State sharing
- Placement

- Algorithm selection
- Batching
- Load shedding