Towards a Universal Stream Processing Platform

Robert Soulé, Martin Hirzel, Robert Grimm, Buğra Gedik

New York University and IBM Research
Streaming Is Everywhere
Developers are already familiar with a particular syntax

Wall Street IT already knows SQL, so CQL is natural

Compilers can statically enforce safety properties

Video playback needs fixed data transfer rate, StreamIt ensures it

Languages hide system complexity

Sawzall expresses business logic, MapReduce handles distribution
For Every Language, Repeat

CQL

Front End
Optimizer
Runtime

StreamIt

Front End
Optimizer
Runtime

Sawzall

Front End
Optimizer
Runtime

Thursday, October 7, 2010
For Every Language, Repeat

CQL
- Front End
- Optimizer
- Runtime

StreamIt
- Front End
- Optimizer
- Runtime

Design for an application domain
For Every Language, Repeat

CQL
Front End
Apply optimizations
Runtime

StreamIt
Front End
Optimizer
Runtime

Design for an application domain
Optimizer
Runtime
For Every Language, Repeat

CQL

Front End

Apply optimizations

Runtime

StreamIt

Front End

Optimizer

Runtime

Design for an application domain

Optimizer

Build custom runtime

Thursday, October 7, 2010
For Every Language, Repeat

CQL
Front End
Apply optimizations
Runtime

StreamIt
Front End
Optimizer
Runtime

Design for an application domain

Build custom runtime
Simplifying the Process

- CQL
  - Front End
- StreamIt
  - Front End
- Sawzall
  - Front End
- Optimizer
- Runtime
Simplifying the Process

Reduce front end development effort
Simplifying the Process

Reduce front end development effort

Implement optimizations in common IL
Simplifying the Process

- Reduce front end development effort
- Build universal runtime
- Implement optimizations in common IL
Reduce Front End Development Effort
Goal of Language Composition

CQL
Front End

StreamIt
Front End
Optimizer
Runtime

Sawzall
Front End
Goal of Language Composition

CQL
Front End
Optimizer
Runtime

StreamIt
Front End

Sawzall
Front End

Language Modules
Goal of Language Composition

- Focus on innovative new language features

Diagram:
- CQL
- StreamIt
- Sawzall
- Front End
- Optimizer
- Runtime
- Language Modules
Goal of Language Composition

- **Focus on innovative new language features**
- **Re-use language modules to reduce implementation effort**
select istream(*)
from quotes[now], history
where quotes.ask <= history.low
and quotes.ticker = history.ticker
Language Composition: Syntactic Analysis

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

CQL = SQL + Streaming + Expressions
Language Composition: Semantic Analysis

Symbol Table

SQLAnalyzer

CQLAnalyzer

Expression Analyzer

has-a

has-a

has-a

is-a
Language Composition: Semantic Analysis

CQL = SQL + Streaming + Expressions
Other languages follow a similar pattern

- **StreamIt** = Streams + Java-like language
- **Sawzall** = Streams + Expressions + Foreign Function Interface

Composition of two languages used in several projects

- **Jinn** [Lee et al., PLDI ‘10], **Blink** [Lee et al., OOPSLA ‘09], **Jeannie** [Hirzel and Grimm, OOPSLA ‘07], **Rats!** [Grimm, PLDI ‘06]
Implement Optimizations in a Common IL
Goal of the Intermediate Language

CQL

StreamIt

Sawzall

Front End

Front End

Front End

Optimizer

Runtime
Goal of the Intermediate Language

- Decouple languages from runtimes

Diagram:

- CQL Front End
- StreamIt Front End
- Sawzall Front End
- Optimizer
- Runtime

Thursday, October 7, 2010
Goal of the Intermediate Language

- Decouple languages from runtimes
- Reduce implementation effort by re-using optimizations
Design of the Intermediate Language

- Start with a formal foundation: Brooklet [Soulé et al., ESOP ‘10]
  - Provably correct system behavior
  - Provably correct language translations
  - Provably correct optimizations
- See what additional features we will need
Elements of a Streaming App
Elements of a Streaming App
Start With a Formal Foundation: Brooklet
Start With a Formal Foundation: Brooklet

Make operators explicit
Start With a Formal Foundation: Brooklet

Make operators explicit

Make state explicit
Start With a Formal Foundation: Brooklet

Make state explicit

Make operators explicit

Make communication explicit
Start With a Formal Foundation: Brooklet

Captures program’s structural features

Make state explicit

Make operators explicit

Make communication explicit
Identify Additional Features
Identify Additional Features

Missing program’s representative features
Identify Additional Features

Missing program's representative features

Operator implementations are opaque

Data types are opaque
River Intermediate Language

Brooklet Calculus
Expressions
Types
Generics
Simple Templates

Structural Features
Representative Features
Meta-Programming
River Intermediate Language

OCaml

Brooklet Calculus
Expressions
Types
Generics
Simple Templates

River
Structural Features
Representative Features
Meta-Programming
The Trouble With State

- Current streaming systems restrict operators
  - MapReduce and DryadLINQ assume side effect free operators
  - Surprisingly little database research on optimizations with operators that update
- How to apply these optimizations in the presence of arbitrary operators?
- How to apply the optimizations automatically?
Build Universal Runtime
Bad Idea 1

Don’t keep re-implementing the same services
Bad Idea 2

Don’t become “the CORBA of streaming”
Goal of the Universal Runtime

CQL

Front End

StreamIt

Front End

Sawzall

Front End

Optimizer

Runtime
Goal of the Universal Runtime

Provide a *minimal* set of features that are *expressive* enough to support the desired applications.
Start With an Existing System

- CQL
  - Front End
- StreamIt
  - Front End
- Sawzall
  - Front End

- Optimizer
- System S

Thursday, October 7, 2010
Start With an Existing System

- System S provides data transport, process management, distribution, etc.

Diagram:
- CQL Front End
- StreamIt Front End
- Sawzall Front End
- Optimizer
- System S
Start With an Existing System

System S provides data transport, process management, distribution, etc.

Need to get our hands dirty: CQL and Sawzall running
Open Issue 1: Dynamic Topology

- Dynamic optimizations:
  - Can exploit the more complete knowledge of the system
  - Depend on changing the application topology at runtime

- Challenge is how to implement them effectively
  - *Dynamic reconfiguration*
  - *Dynamic rerouting*
Dynamic Reconfiguration

Filter
Dynamic Reconfiguration

1. Stop the data flow
Dynamic Reconfiguration

2. Let data drain out

Diagram:

2. Let data drain out

Diagram:

Dynamic Reconfiguration

2. Let data drain out

Diagram:
Dynamic Reconfiguration

3. Reconfigure the application

Filter
Dynamic Reconfiguration

3. Reconfigure the application

- Able to exploit runtime knowledge
Dynamic Reconfiguration

3. Reconfigure the application

- Able to exploit runtime knowledge
- Introduces a “hiccup” while the application is running
Dynamic Rerouting

Diagram:
- Eddy
  - Connects to Filter 1 and Filter 2
- Filter 1
- Filter 2

Thursday, October 7, 2010
Dynamic Rerouting

1. Statically insert router operator

Eddy

Filter 1

Filter 2
Dynamic Rerouting

1. Statically insert router operator*  

*[Avnur et al., SIDMOD ‘00]*
Dynamic Rerouting

2. Dynamically decide data path

- Eddy
- Filter 1
- Filter 2
Dynamic Rerouting

2. Dynamically decide data path

No “hiccup”
Dynamic Rerouting

2. Dynamically decide data path

- Eddy
- Filter 1
- Filter 2

- No “hiccup”
- Can be generalized to other optimizations

Thursday, October 7, 2010
Dynamic Rerouting

2. Dynamically decide data path

- No “hiccup”
- Can be generalized to other optimizations
- Introduces an extra processing step
Open Issue 2: Distributed State

- Distributed state is a necessary feature:
  - Distributed data in MapReduce applications
    Ex. PageRank’s scores
  - Classifier models in learning applications
    Ex. Weighted linear classifier when determining if message is SPAM

- Implementing distributed state is problematic:
  - Locality, partitioning, failure-recovery, consistency
Open Issue 3: Can We Close the Loop?

Can we use the system to design better languages?
Conclusion

- Streaming is everywhere and it needs language support
- Building new streaming languages should be easier
  - Composition reduces front end development effort
  - An intermediate language allows for optimization re-use
  - A universal runtime allows for re-use of common services
- Designing the common runtime is a work-in-progress
  - Investigating support for dynamic topology and shared state
http://cs.nyu.edu/brooklet