Hancock: A Language for Extracting Signatures from Data Streams

Kathleen Fisher
Karin Högstedt
Anne Rogers
Fred Smith
AT&T Infolab

Networks:
- Long distance
- Frame Relay
- ATM
- IP

Applications:
- Manage Network
- Prevent/Detect Fraud
- Understand Customers

Challenge:
To convert this data into useful information.
Whole data analysis

Individualized analysis: Signatures
• Anomaly detection: fraud, access arbitrage, etc.
• Classification problems: target marketing, biz/res, etc.

Technical challenge:
• Massive data sets and real-time queries ⇒
  Hard I/O and storage requirements ⇒
  Complex programs (hard to read, write, and maintain).

Solution:
• A system that reduces the complexity of signature programs.
Processing transactions

Transaction Data

Data Warehouse

Hancock

Signature Database
Evolution of fraud detection

Country-based thresholds:
  • Aggregate calls in 1/4/24 hour windows.
  • Compare aggregates to fixed thresholds.
  • Exclude common false positives.

International signatures:
  • Signature is an evolving profile.
  • Match calls against the customer’s and known fraud signatures.

Domestic signatures?
  • Much larger scale…
Problem scale

![Graph showing the problem scale with two data points: one at 10 bytes and 100 M signatures labeled 'Small Domestic' and another at 1000 bytes and 10B signatures labeled 'International'.]
Computational issues

Efficiently managing communications-scale data requires substantial programming expertise.

Locality, locality, locality!
Hancock

- Identified abstractions for computing with large data streams.
- Embedded these abstractions in Hancock, a C-based domain-specific programming language.
- Built experimental and production signatures using a number of different data streams.
- Intended as an experiment in practical language design.
Concrete results
Outline

• Introduction
• Language overview
• Implementation overview
• Conclusions
**Abstraction overview**

**Streams.** The transactional data to be consumed “daily.”

**Iterate statement.** How to combine today’s data with historic signatures and other data.

**Maps.** The collection of customer signatures.

**Views.** The information to store for each “customer.”
Hancock maps

- Persistently associate data with keys.
- Support direct addressing, programmable defaults, and a customized, compressed format.

```c
map sig_m {
    key 1999999999LL .. 9999999999LL;
    split (10000, 100);
    value sig_t;
    default SIG_DEFAULT;
    compress sig_compress;
    uncompress sig_uncompress;
};
```
Map operations

- **Supported**: read, write, test, remove, iteration, and copy.

  ```
  sig_m myMap = "data/myMap";
  long long id1, id2;
  sig_t oldS, newS;

  oldS = myMap<:id1:>;       /* read id1’s old data   */
  ...
  myMap<:id1:> = newS;       /* write id1’s new data   */

  if (myMap@$<:id2:>)        /* test if id2 is in myMap */
    myMap\<:id2:>;          /* remove id2’s data      */
  ```

- **Unsupported**: atomic transactions, locking, secondary indices, declarative queries.
Computation model

- Detect “events of interest” in transactional stream; respond to those events.

```
begin_line(A)
call(C1)
call(C2)
begin_line(B)
call(C4)
end_line(B)
```

- Hancock’s iterate statement
  - prepares stream for computation,
  - separates event detection from event response, and
  - generates scaffolding code.
Iterate statement

```
Iterate

over calls

filteredby only International

sortedby origin

with events detectCalls

{
    event line_begin (pn_t pn) { ... }

    event call (callRec_t c)

    { ... }

    event line_end (pn_t pn) { ... }
}
```
Iterate statement

Iterate over calls filtered by only International sorted by origin with events detectCalls

{  
    event line_begin (pn_t pn) { ... }  
    event call (callRec_t c)  
    { ... }  
    event line_end (pn_t pn) { ... }  
}
Iterate statement

```
Iterate
    over    calls
    filteredby onlyInternational
    sortedby origin
with events detectCalls
{
    event line_begin (pn_t pn) { ... }
    event call (callRec_t c)
    { ... }
    event line_end (pn_t pn) { ... }
}
```
Iterate statement

Iterate over calls
filtered by only International
sorted by origin
with events detectCalls

{ 
    event line_begin (pn_t pn) { ... } 
    event call (callRec_t c) 
    { ... } 
    event line_end (pn_t pn) { ... } 
}
Iterate statement

Iterate over calls
   filteredby onlyInternational
   sortedby origin
   with events detectCalls
{
    event line_begin (pn_t pn) { ... }
    event call (callRec_t c)
      { ... }
    event line_end (pn_t pn) { ... }
}
Iterate statement

Iterate

over calls
filtered by only International
sorted by origin
with events detect Calls

{ 
  event line_begin (pn_t pn) { ... } 
  event call (callRec_t c) 
    { ... } 
  event line_end (pn_t pn) { ... } 
}
Iterate

over calls

filtered by only International

sorted by origin

with events detectCalls

```c

<table>
<thead>
<tr>
<th>C2</th>
<th>intl 3608077</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>intl 3608675</td>
</tr>
<tr>
<td>C3</td>
<td>intl 3608675</td>
</tr>
</tbody>
</table>

line_begin(...) call(C2) line_end(...)

line_begin(...) call(C1)

call(C3) line_end(...)
```

```c

event line_begin (pn_t pn) { ... }

event call (callRec_t c)

{ ... }

event line_end (pn_t pn) { ... }
```
Iterate

over calls
filtered by only International
sorted by origin
with the events detect Calls

{ }

event line_begin (pn_t pn) { ... } 

event call (callRec_t c) 
{ ... }

event line_end (pn_t pn) { ... }
Iterate statement

```
{  
event line_begin(pn_t pn)   {  numToday = 0;  }
event call(callRec_t c)         {  numToday++;  }
event line_end(pn_t pn)       {  numCalls<:pn:> =  
                                    0.8 * numCalls<:pn:> + 0.2 * numToday; 
}
}
```
Result: Cleaner code

- Hand-coding results in complex event detection code that obscures simple event response code:

```plaintext
end_line(?)
begin_line(A)
call(C1)

call(C2)

call(C3)

end_line(A)
begin_line(B)
call(C4)
```
Representing events

- Hancock’s multi-union (*munion*): A set of labels and associated values.

```
munion line_e {:
    pn_t begin_line,
    callRec_t call,
    pn_t end_line,
};
```

- **Supported operations**: value construction, right-dominant union, test for label, access value, difference, and remove.
Detecting events

- An event detection function:
  - takes a window onto a stream
  - returns a munion that describes the detected events.
Event detection example

```c
line_e originDetect(callRec_t *w[3:1]){
    line_e b, e;
    callRec_t *prev, *current, *next;
    prev = w[0];
    current = w[1];
    next = w[2];
    b = beginLineDetect(prev, current);
    e = endLineDetect(current, next);
    return b :+: { : call = *current :} :+: e;
}
```
**Code sizes**

**Streams**: 110 to 250 lines.

**Iterate statement**: 70 to 330 lines.

**Views**: 5 to 30 lines.

**Maps**: 5 to 200 lines.
Implementation

• **Compiler:**
  – Based on CKIT C-to-C translator (SML/NJ).

• **Runtime system:**
  – Written in C.
  – Map representation is essentially a stripped-down database.
  – Goal: balance space limits against access-time requirements.

• Available for non-commercial use.
Maps: On-disk representation

Multi-level table. Key split into three pieces: block, stripe, entry.

(973360 86 75)
Maps: In-memory representation

Map index and compressed stripe cache kept in memory (973360 86 75).
Performance requirements

Process transactions: < ½ batch window

Select single key: web time (1 second)
Select worklist: coffee break time (5 minutes)
Touch all values: lunch time (1 hour)
Experimental setup

- **Platform:**
  
  R12000 processor (SGI Origin 2000).
  
  32-KB primary cache/8-MB secondary cache
  
  6GB main memory

- **Activity:** 1.27GB on disk
  
  - maps phone numbers (464M) to 3-byte signatures

- **Features:** 1.10 GB on disk
  
  - maps phone numbers (163M) to 124-byte signatures
# Experimental results

<table>
<thead>
<tr>
<th>Action</th>
<th>Requirement</th>
<th>Activity</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process calls (275M calls)</td>
<td>½ batch time</td>
<td>19m</td>
<td>30m</td>
</tr>
<tr>
<td>Select single value</td>
<td>web time</td>
<td>1.1s</td>
<td>1.1s</td>
</tr>
<tr>
<td>Select worklist (160,000 keys)</td>
<td>coffee time</td>
<td>3m30s</td>
<td>4m3s</td>
</tr>
<tr>
<td>Touch all values</td>
<td>lunch time</td>
<td>17m3s</td>
<td>12m18s</td>
</tr>
</tbody>
</table>
Concrete results

![Graph showing the relationship between the size of signatures and the number of signatures. The graph has two axes: Size of Signature on the y-axis, ranging from 10 bytes to 1,000 bytes, and Number of Signatures on the x-axis, ranging from 10M to 1B. There are three data points: International, Medium Domestic Hancock, and Small Domestic Hancock.]
Communities of Interest

Used to detect 800 subscription fraud.

- Known fraudster
- Inbound calls
- Outbound calls
Performance: Outgoing COI

- Signature size: 120 bytes
- Active keys: 228M
- Signature collection size: 7GB
- Daily update time: 2 hours
- Neighborhood computation: 1 second to compute a neighborhood of size 2 from a seed phone number.
Why a language?

- **Disadvantages:**
  - Limited scope
  - New language (albeit one based on C).
  - Lack of tools, *e.g.* source-level debuggers, profilers, *etc.*

- **Advantages:**
  - Static type-checking
    - protects data integrity and promotes clarity of use
  - High-level and tailored abstractions
    - reduce code size, hide issues of scale, and provide a framework for structuring applications.
Further work

• Compare with database implementations (DBPL 2001)
• Allow users to specify streams declaratively (see PADS).
• Add support for variable-width data (urls).
• Improve compression mechanisms.
Try it!

Hancock is available for non-commercial use:


Inquiries to hancock@research.att.com.

References:

• Domain Specific Languages Conference, 1999
• Knowledge Discovery and Data Mining Conference, 2000
• Databases and Programming Languages workshop, 2001
• Transactions on Programming Languages and Systems, 2004