CSE503: SOFTWARE ENGINEERING

PROJECT PROPOSALS

David Notkin
Spring 2011

Lazy Evaluation for MapReduce Workflows

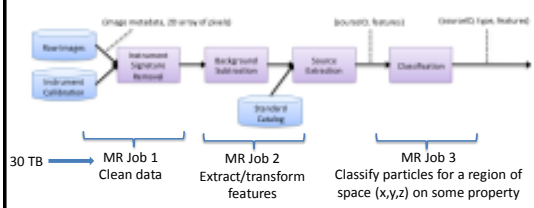
Kristi Morton, Magdalena Balazinska, Dan Grossman, and Christopher Olston

Motivating Scenario

- Data deluge in sciences
  - LSST workflow process 30TB of new data every day
    - Only a subset of data needed
- High-latency analysis tasks in workflows
  - On MapReduce (MR) can take hours to run
  - Limits scientific discovery

Goal:
Make workflows efficient by being lazy: only run on region of interest.

LSST Workflow



Lazy processing: observe usage in workflow and only processes data of interest.
Any additional data is computed on demand, per user's request.

Lazy Evaluation for MapReduce Workflows

Context for Project # 2:

- Continuation from Project 1
- Workflows expressed as PigLatin scripts
- Use User Defined Function (UDF) for lazy evaluation framework

Lazy Evaluation for MapReduce Workflows

- To produce the lazy materialized view, UDF needs:
  1. PigLatin workflow script

```
REGISTER udfs.jar;
gas43 = LOAD 'gas43full' USING BinStorage() AS
        (pid:long, mass:double, px:double, py:double,
        pz:double,temp:double, ...);

gas   = FILTER gas43 BY pid is not null AND mass is not null
        AND px is not null AND py is not null
        AND pz is not null AND temp is not null...;
regionA = FILTER gas BY temp > udfs.VirialTemp(Mvir,Mvir,pid)
        AND px >= -0.5 AND px < -0.25
        AND py >= 0 AND py < 0.5
        AND pz >= -0.3 AND pz < 0;

STORE regionA INTO 'result' USING BinStorage()
```

  2. Fields needed by user: e.g. gas, temp
  3. Delimiter

Lazy Evaluation for MapReduce Workflows

We generate the following materialized view + metadata:

Gas	Temp
a	200
b	150
c	1000
d	999
…	…

All other fields with pointers to PigLatin scripts to generate them on the fly (i.e. the "closures" part

Project 2 work to do

- Previous project generated simple metadata for on demand fields
  - Naïve approach: each field pointed to full workload script (i.e. computed all fields)
  - Need to have fields point to appropriate subset to compute only the data for the field not all fields
- Need to generate subset scripts to compute individual fields
  - Involves looking at level of execution plan tree and pruning off subtrees
  - Will need to hack Pig query compiler, which is a nontrivial task ☺

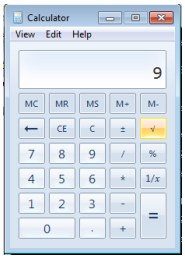
Testing (and Scripting) GUIs

Daniel Leventhal – cse 503

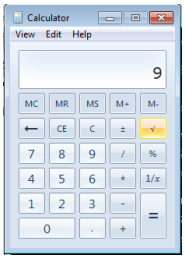
We know how to test code

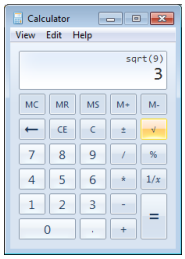
```
bool testSqrt(int input, double out)
{
  return out == sqrt(input);
}


testSqrt(9, 3);
testSqrt(0, 0);
testSqrt(100, 10);
```

How do we test GUIs?



How do we test GUIs?





- Test what the user actually sees

GUI testing is miserable

- Support is often worse than "normal" tests
- Often hard to write the test
- Can't run in parallel
- They take longer to run
- Harder to debug
- Harder to change
- Harder to figure out what to change

- They are testing what the user actually sees!
  - (or should be)

Scripting Tests

- How do you specify a test?
  - List screen coordinates
  - Mark the UI widget in some way
  - Specify what the widget look like [Sikuli]

- How do you adapt to change?
  - UIs often undergo visual polish close to shipping
  - Specify widgets based on their content or their position in the tree

Accessibility API

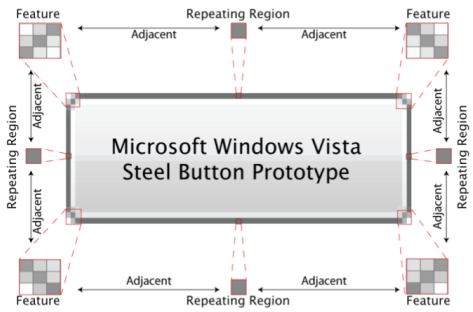
- Doesn't the accessibility API provide everything you need?
  - Only covers 80% of the widgets out there [Hurst 2010]
  - If the API doesn't work you are toast.

Prefab to the Rescue!

- Prefab identifies occurrences of a widget from its pixels
  - Text is recovered
  - Hierarchy of widgets is identified

- Use Prefab to build tools for scripting and testing GUIs
  - Tests/scripts can be less brittle
  - Specify the widget at an appropriate level of abstraction.

Prefab basics

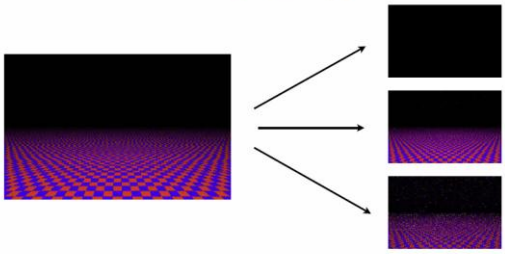


Testing/Scripting Framework with Prefab

- Develop API for testing/scripting applications
- Leverage Prefab's identification of widgets, content, and hierarchy
- Author tests that are resilient to changes in the UI.

Adrian S.

Detailed Quality-of-Service Profiling



```
t=-1;
xd=(x-w1)/w1;
yd=(h1-y)/h1;
zd=-1;
l=xd*xd+yd*yd+zd*zd;
xd/=l;
yd/=l;
zd/=l;

if((k-ye)*yd<=0) {
  t=-1;
} else {
  t=(k-ye)/yd;
}
```

super-extra-important!

not very important.

- Where do I need to be super-extra-careful about error resilience?
- Where am I perhaps doing too much work where a simpler, less precise computation would save time?
- *Eventually:* Here's a modified program that runs faster and works almost as well!

Monday, May 9, 2011

Design Patterns for Security & Privacy

Franzi Roesner
CSE 503 Project Proposal
May 10, 2011

Security Design Pattern Sampler

- **Single Access Point**: Providing a security module and a way to log into the system
- **Roles**: Organizing users with similar security privileges
- **Session**: Localizing global information in a multi-user environment
- **Limited View**: Allowing users to only see what they have access to
- **Secure Access Layer**: Integrating app security with low level security
- **Partitioned Application**: Splits a large, complex application into simpler components; any dangerous privilege is restricted to a single, small component.
- **Input Validator**: Validate input from the client to the server

Privacy Design Pattern Sampler

- **Informed Consent for Web-Based Transactions**: Describes how websites can inform users whenever they intend to collect and use an individual's personal information
- **Masking Your Online Traffic**: Decreasing the flow of information from the data owner to the data collector
- **Minimal Information Asymmetry**: Increasing the flow of information from data collectors to data owners.
- **Protection Against Cookies**: Provides countermeasures against the misuse of cookies in the WWW.
- **Pseudonymous Email**: Describes the mechanism of a pseudonymous email delivery system

Adding Security to Patterns

- **Secure Blackboard Pattern**: Decouples interacting agents from each other with an intermediary agent.
- **Secure Broker Pattern**: Architectural pattern can be used to structure distributing software systems with decoupled components that interact by remote service invocations.

Proposal: Patterns for Web Tracking

- Web tracking:
  - Analytics on a page
  - Third-party tracking across sites
- Tracking methods
  - cookies, LocalStorage, Flash LSOs, cache data, images, web history, window.name, …

Proposal: Patterns for Web Tracking

- Properties we might want:
  - Trackers can't track me across sites unless I consent.
  - Seamlessly associate different browsing profiles with different roles.
  - Retroactively remove visited sites from tracking list.
  - Robustness against future developments in tracking methods.
  - Functionality (e.g. Facebook) while opting out of tracking.
  - Tracking history while logged out (of e.g. Facebook) can't be linked to my identity when I log back in.
  - Same guarantees on mobile browser as on desktop browser.
- Today's tools are insufficient…
- Can design patterns help?

Incidental Research Questions

- Are any of these security and privacy design patterns actually in use as such?
- If not, why haven't they been useful?
- Do there need to be so many? Can they be reduced to a canonical set?

Ben P.

Answering Tomorrow's Problems

Current and Coming Tech

- Tablet PCs
- Kinect, PS Move
- International high-speed networks

Current and Coming Needs

- Better domestic education system
- Better trained and connected workforce
- Better education and stability in developing countries

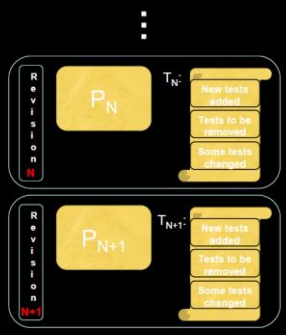
How will software answer these questions with these technologies?

Finding present and future solutions

Researching design and development challenges

Test Prioritization

Investigating the parallelism between the software evolution and test evolution
by
Kıvanç MUŞLU & Bilge SORAN



Software changes

- *New versions are released*
- *Open source contributions*
- *Bug fixes, and workarounds are added*

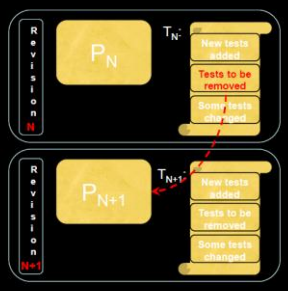
Test suites also changes

- *New tests are added*
- *Some tests are updated*
- *Regression tests are added*

Q1: Is there a correlation between these changes?

Q2: Can we use this correlation for prioritization?

Removed Tests from T_N to T_{N+1}



T compiles & fails: ***Interesting*** since you expect the old tests to pass when program evolves

T compiles & passes:

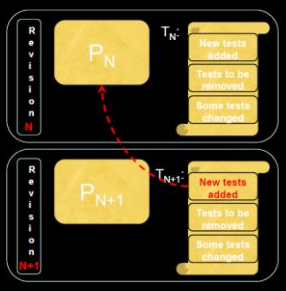
Let (1) Coverage(T_{N+1}, P_{N+1}) = C_1,

(2) Coverage(T_{N+1} + {T}, P_{N+1}) = C_2

If $C_2 > C_1$ => ***Interesting***

T does not compile: ***Interesting*** since it might correlate to a change in the software

New Tests added to T_{N+1}



T compiles & fails: ***Interesting*** since it represents a change from P_N to P_{N+1}

T compiles & passes:

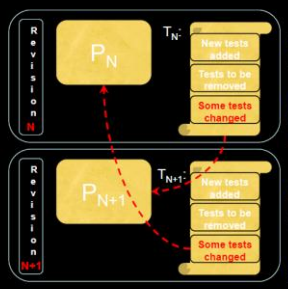
Let (1) Coverage(T_N, P_N) = C_1,

(2) Coverage(T_N + {T}, P_N) = C_2

If $C_2 > C_1$ => ***Interesting***

T does not compile: Not interesting since it might correlate to a new functionality in the software rather than a change between versions

Both Versions Have The Same Test (wrt its name)



Content of T is the same: Not interesting

(1) Content added on T_{N+1}: Evaluate the test as if it is added in version N+1

Assume that it didn't exist in T_N

(2) Content removed on T_{N+1}: Evaluate the test as if it is removed in version N+1

Assume that it didn't exist in T_{N+1}

Both (1) & (2): Evaluate the test as if it is both removed and added in version N+1

Test Prioritization I

- Example program history: Apache Commons CLI
  - 152 revision histories
  - Analyzed the difference between consecutive test revisions: 85 revisions have changes
  - 222 tests added
  - 97 tests removed
  - 67 tests changed

Test Prioritization II

- Compile and run all tests wrt their programs (within revisions). Remove revisions that have failing tests.
- For the remaining revisions, do the analysis for half of them
- Investigate the results and try to understand what do these tests (selected set) really represent?
- Any ideas on evaluation?

Yanping H.

GUI for Daikon

What is Daikon

- Dynamic Analysis for Inferring Likely Invariants
- Guessing invariants with static analysis is hard
- Solution: guessing invariants by watching actual program behavior is easy

Daikon in action

- Generate lots and lots of potential invariants
- The initial set can be infinite, provided there is a way to prune to a finite set with only a few observations
- Let the tests weed out most of the candidates

- Daikon checks invariants over variables at the entrance and exist of programs.

```
void sum(int *b,int n) {
        pre:
          n  ≥ 0   i, s := 0, 0;
        do i  ≠ n
                i, s := i+1, s+b[i]
        post: s=sum(b[j],  0 ≤j<n)
}
```

- No all detected invariants/variables are interesting.

For example 0 ≤ i≤n

Goals

- Develop a GUI for daikon that allows users to select variants/functions/variable they are interested in
- Show resulting invariant in a GUI that are easy for user to comprehend.

Not all interesting invariants are included

- Need to modify and recompile Daikon to include new invariants.
- Use Java reflection to add new-defined invariants
- How to check legal invariant class?

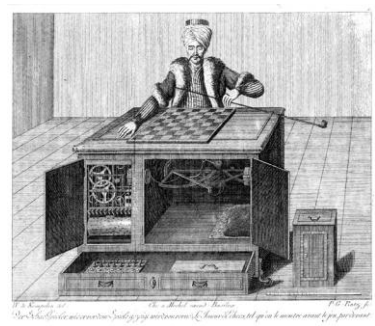
If time permits

- Include unities to highlight invariant changes between a pair of program versions.

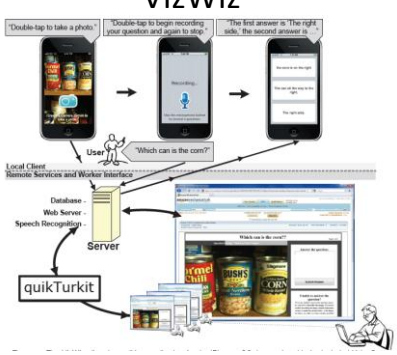
Engineering Crowdsourcing Software

Hao Lu and Joseph Xu

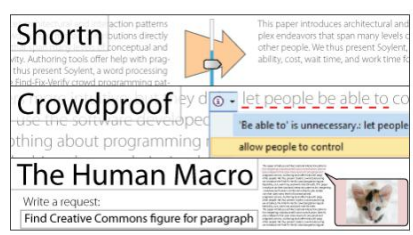
The Turk



VizWiz



Soylent



Platform

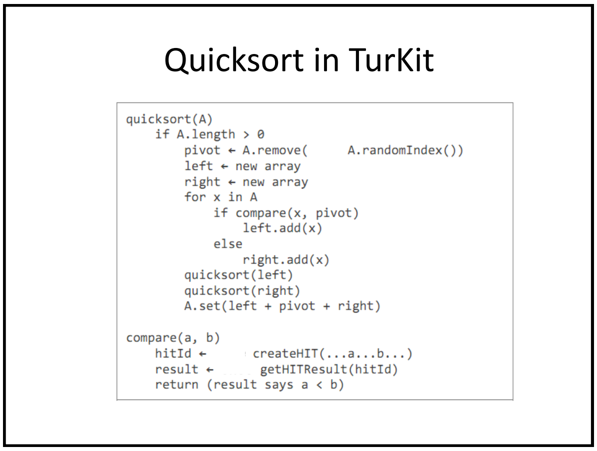
- Mechanic Turk
- …

Existing Toolkit

- TurKit
- CrowdForge

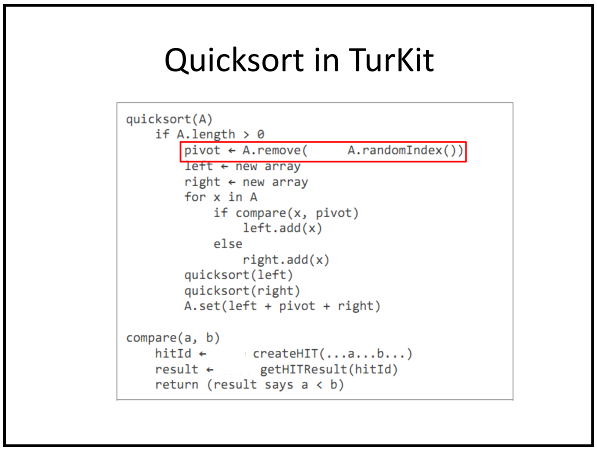
Is it hard?

Use the crowd just as function call?



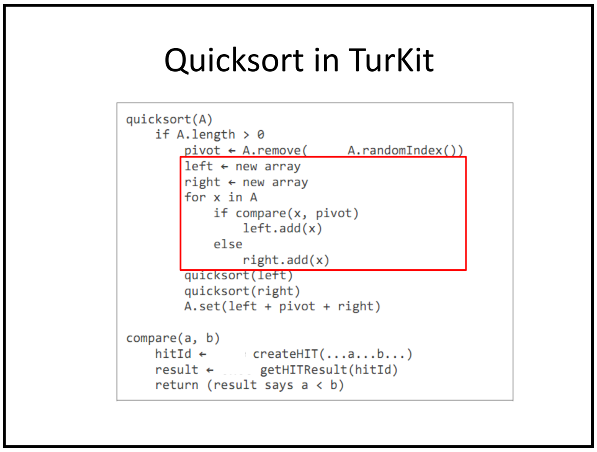
Quicksort in TurKit

```
quicksort(A)
    if A.length > 0
        pivot ← A.remove(     A.randomIndex())
        left ← new array
        right ← new array
        for x in A
            if compare(x, pivot)
                left.add(x)
            else
                right.add(x)
        quicksort(left)
        quicksort(right)
        A.set(left + pivot + right)

compare(a, b)
    hitId ←      createHIT(...a...b...)
    result ←      getHITResult(hitId)
    return (result says a < b)
```



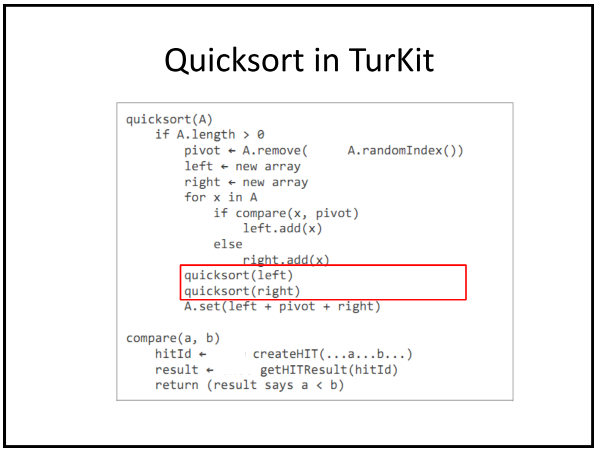
Quicksort in TurKit

```
quicksort(A)
    if A.length > 0
        pivot ← A.remove(     A.randomIndex())
        left ← new array
        right ← new array
        for x in A
            if compare(x, pivot)
                left.add(x)
            else
                right.add(x)
        quicksort(left)
        quicksort(right)
        A.set(left + pivot + right)

compare(a, b)
    hitId ←      createHIT(...a...b...)
    result ←      getHITResult(hitId)
    return (result says a < b)
```



Quicksort in TurKit

```
quicksort(A)
    if A.length > 0
        pivot ← A.remove(     A.randomIndex())
        left ← new array
        right ← new array
        for x in A
            if compare(x, pivot)
                left.add(x)
            else
                right.add(x)
        quicksort(left)
        quicksort(right)
        A.set(left + pivot + right)

compare(a, b)
    hitId ←      createHIT(...a...b...)
    result ←      getHITResult(hitId)
    return (result says a < b)
```



Quicksort in TurKit

```
quicksort(A)
    if A.length > 0
        pivot ← A.remove(     A.randomIndex())
        left ← new array
        right ← new array
        for x in A
            if compare(x, pivot)
                left.add(x)
            else
                right.add(x)
        quicksort(left)
        quicksort(right)
        A.set(left + pivot + right)

compare(a, b)
    hitId ←      createHIT(...a...b...)
    result ←      getHITResult(hitId)
    return (result says a < b)
```

Quicksort in TurKit

```
quicksort(A)
    if A.length > 0
        pivot ← A.remove(      A.randomIndex())
        left ← new array
        right ← new array
        for x in A
            if compare(x, pivot)
                left.add(x)
            else
                right.add(x)
        quicksort(left)
        quicksort(right)
        A.set(left + pivot + right)

compare(a, b)
    hitId ←      createHIT(...a...b...)
    result ←      getHITResult(hitId)
    return (result says a < b)
```

Quicksort in TurKit

```
quicksort(A)
    if A.length > 0
        pivot ← A.remove(      A.randomIndex())
        left ← new array
        right ← new array
        for x in A
            if compare(x, pivot)
                left.add(x)
            else
                right.add(x)
        quicksort(left)
        quicksort(right)
        A.set(left + pivot + right)

compare(a, b)
    hitId ←      createHIT(...a...b...)
    result ←      getHITResult(hitId)
    return (result says a < b)
```

Crash-and-rerun programming

```
quicksort(A)
    if A.length > 0
        pivot ← A.remove(once A.randomIndex())
        left ← new array
        right ← new array
        for x in A
            if compare(x, pivot)
                left.add(x)
            else
                right.add(x)
        quicksort(left)
        quicksort(right)
        A.set(left + pivot + right)

compare(a, b)
    hitId ← once createHIT(...a...b...)
    result ← once getHITResult(hitId)
    return (result says a < b)
```

Issues

- “Crash-and-rerun” is unnatural

Parallelism

```
quicksort(A)
    if A.length > 0
        pivot ← A.remove(once A.randomIndex())
        left ← new array
        right ← new array
        for x in A
            if compare(x, pivot)
                left.add(x)
            else
                right.add(x)
        quicksort(left)
        quicksort(right)
        A.set(left + pivot + right)

compare(a, b)
    hitId ← once createHIT(...a...b...)
    result ← once getHITResult(hitId)
    return (result says a < b)
```

Parallelism

```
quicksort(A)
    if A.length > 0
        pivot ← A.remove(once A.randomIndex())
        left ← new array
        right ← new array
        for x in A
fork        if compare(x, pivot)
                left.add(x)
            else
join            right.add(x)
fork    quicksort(left)
fork    quicksort(right)
join    A.set(left + pivot + right)

compare(a, b)
    hitId ← once createHIT(...a...b...)
    result ← once getHITResult(hitId)
    return (result says a < b)
```

Issues

- “Crash-and-rerun” is unnatural
- Fork and Join adds complexity

Existing knowledge

```
quicksort(A)
    if A.length > 0
        pivot ← A.remove(once A.randomIndex())
        left ← new array
        right ← new array
        for x in A
            if compare(x, pivot)
                left.add(x)
            else
                right.add(x)
        quicksort(left)
        quicksort(right)
        A.set(left + pivot + right)

compare(a, b)
    hitId ← once createHIT(...a...b...)
    result ← once getHITResult(hitId)
    return (result says a < b)
```

Existing knowledge

```
quicksort(A)
    if A.length > 0
        pivot ← A.remove(once A.randomIndex())
        left ← new array
        right ← new array
        for x in A
            if compare(x, pivot)
                left.add(x)
            else
                right.add(x)
        quicksort(left)
        quicksort(right)
        A.set(left + pivot + right)

compare(a, b)
    hitId ← once createHIT(...a...b...)
    result ← once getHITResult(hitId)
    return (result says a < b)
```

Issues

- "Crash-and-rerun" is unnatural
- Fork and Join adds complexity
- Existing knowledge adds more complexity

Questions to answer in our project

- Are these issues general?

Questions to answer in our project

- How can we hide these complexity?

Question?