A real trip on the wild side

- So far, we've been studying what business wants clouds to do.
- Now, for a brief time, let's study what computer scientists want the cloud to do.

Several key ideas:

- OOM more work with OOM less code.
- The CALM principle.
- An implementation: "Bloom".
- First paper: 2011. This is **hot**; really **hot**!

OOM

Wednesday, April 27, 2011
10:28 AM

OOM more work with OOM less code

- OOM=orders of magnitude
- OOM more work: more useful computation
- OOM less code: less drudge work.
- Source: Berkeley's BOOM project.

CALM

Wednesday, April 27, 2011
10:28 AM

The CALM principle:

- CALM: Consistency As Logical Mononicity
- A process is **logically monotonic** if additions to its input do not change its present output.
    - Example: determining the keywords mentioned today on twitter is **logically monotonic**, because if we discover more keywords, that does not make us take anything out of the current solution set of keywords that appear.
    - Example: determining the keywords that were not mentioned yesterday is **not logically monotonic**, because if we discover a new one that was mentioned yesterday, we have to **take it out** of the solution set.
- In other words, something is logically monotonic if once something is in the solution set, it always remains there.

Points of order

Wednesday, April 27, 2011
10:36 AM

The point of CALM

- Logically monotonic code is **eventually consistent without locking.**
- We can make non-monotonic code logically monotonic by use of locking, at what CALM calls **points of order.**
- Example: to make the non-monotonic example on the previous page monotonic, we change the problem:
  - Compute the list of keywords that appeared yesterday.
  - Let them become (eventually) consistent.
  - **Then** form the negation.
  - (A naïve view… I will make this more precise in a second)

Bloom

Wednesday, April 27, 2011
10:41 AM

Bloom

A language framework for **disorderly programming** in clouds. Based upon

- **temporal logic**,
- **eventual consistency**, and
- **logical monotonicity**.

Bloom Under Development: BUD

Wednesday, April 27, 2011
10:51 AM

A bloom programming language: **bud**

- based on ruby
- runs in a bud sandbox (like the hadoop sandbox)
- supports hadoop
- base data representation: key-value stores

Basics of BUD

Wednesday, April 27, 2011
11:21 AM

Basics of BUD

- A dialect of **ruby**
- with distributed objects and operators.

Basic data structure: a table

- table :clouds, [:key] =>[:value]

  Pasted from <https://github.com/bloom-lang/bud/blob/master/docs/getstarted.md>
- defines a table, which is a set of key-value pairs
  - a distributed object
  - logically, a set
  - :key, :value are field names (ruby atoms).
  - keys are unique
  - values need not be unique
  - can have keys or values that are themselves tuples

Collections, Sets, and Facts

Wednesday, April 27, 2011

2:43 PM

Collections, Sets, and Facts

- Bud has its roots in Prolog: logic programming
- It is perhaps best to think of a bud table as a **set of facts.**
- A fact, once known, is not forgotten.
- But a fact, once known, remains known.
- This is the crux of logical monotonicity.

You can do only two things with collections:

- Create new collections from them.
- Merge collections into them.
- You cannot delete from collections,
- but you can bind collections together via difference operations.

Four merge operators

Wednesday, April 27, 2011
11:46 AM

Four merge operators

```
clouds <= [[1, "Cirrus"], [2, "Cumulus"]]
```

Instantly merge the key/value pairs into the clouds array. In other words, invoke strong consistency!

```
clouds <~ [[1, "Cirrus"], [2, "Cumulus"]]
```

Eventually merge the key/value pairs into the clouds array. Asynchronous. Might not complete immediately.

```
clouds <+ small_clouds
```

Deferred merge: merge the key/value pairs after the RHS has strong consistency!

```
clouds <- small_clouds
```

Deferred delete: remove the keys listed after consistency of the right-hand side!

Because we aim for logical monotonicity,
merges are cheap.
deletes are expensive.

A simple quandary solved

Wednesday, April 27, 2011
12:25 PM

A simple quandary solved

- Keywords that are mentioned today but not yesterday

```
today <~ some_keyword_search
yesterday <~ another_keyword_search
today <- yesterday
# deferred omission, not assignment!
```

But how does this work?

- We start off two asynchronous search processes to determine "`today`" and "`yesterday`".
- When today and yesterday merges finish, we present today without yesterday's keys.

Continuous queries

Wednesday, April 27, 2011
4:04 PM

Continuous queries

Nothing we currently know about is a good analogue of a bud table.

It is best to think of a bud table as being like a **continuous query** in SQL.

It doesn't define "what to do with data you **have.**"

It defines "what to do with data you **get**."

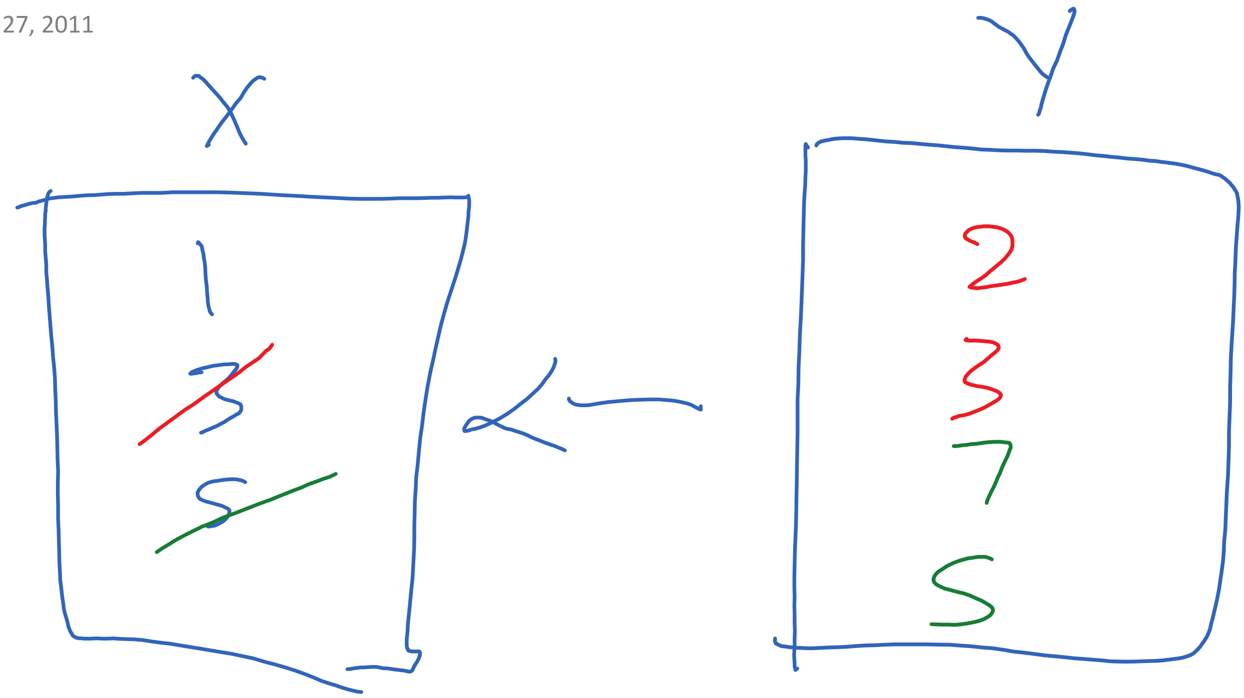
Understanding deferred operations

Wednesday, April 27, 2011
4:06 PM

`x<=y` means put data into x exactly when it arrives in y.
`x<~y` means put data into x some time after it arrives in y.
`x<-y` means subtract y values from x values regardless of when data arrives in x or y.
`x<+y` means put data into x from y even if actions on x try to delete it!

A better model of deferral

Wednesday, April 27, 2011
4:09 PM



A deferred subtraction is not an action, but rather, a binding.

It says: from now on, return for the value of x,
the things that are in x but not in y.
No matter how y changes.

You can bind multiple sets to x.

Note that:

you cannot subtract an element from a set.
You can bind sets together, so that the binding implements a subtraction,
while both sets are computed via monotonic logic!

Schemas

Wednesday, April 27, 2011
2:37 PM

Schemas

The schema for a table determines what parts constitute the key and what parts constitute the value.

The default schema for a table is

[:key]=>[:value],

which means that an arbitrary element e has parts e.key and e.value.

Can make any table have an arbitrary schema via, e.g.,

table :foo [:k1, :k2] => [:v1, :v2, :v3]

constructs a table foo that consists of quintuples

[e.k1, e.k2, e.v1, e.v2, e.v3] where e is an element

Queries

Wednesday, April 27, 2011
12:34 PM

The most basic query in Bud is the **implicit map** (in ruby). (Do something to every element of a table, return the result)

- t2 <~ t1 {|t| t if t.key>5}
  - |t| each table element in turn
    - t.key: its key
    - t.value: its value
  - t2 becomes the table with only keys > 5.
- In Pig, this is equivalent with FILTER -- BY
- t2 <~ t1 {|t| [t.key+1,t.value]}
  - literally, for every t in t1, produce [t.key+1,t.value]
  - t2 is the array with one greater index than t1.
- In Pig, this is equivalent with FOREACH -- GENERATE.

Some notes:

- Schemas are determined by the commands.
- Schema mismatches during merges are fatal.

Groupings

Wednesday, April 27, 2011
2:49 PM

Groupings

A grouping in Bud always has the outcome of creating aggregate data.

There is no such thing as a Pig grouping that creates hierarchy.

Example: if

```
t1 has schema [:game] => [:player, :score]
```

then

```
totals <= t1.groupby([:player],sum(:score))
```

has schema [:player]=>[:score]

(Implicitly, the grouped thing becomes a key!)

Builtin aggregate functions:

count, sum, choose, avg, min, max...

Products

Wednesday, April 27, 2011
12:34 PM

Products

As in Pig, one useful construction is the cross product.

For tables t1 and t2, (t1 * t2) is a new table with

keys that are pairs of keys from t1, t2,

values that are pairs of values.

I.e., if k1 => v1 is in t1 and k2 => v2 is in t2,

then [k1,k2] => [v1,v2] is in t1 * t2

In Pig, this is equivalent with CROSS

Joins

Wednesday, April 27, 2011
12:34 PM

A join is a filter for a product!

```
# simple join
out <= (r * s).pairs(:r.value => :s.key) do |t1, t2|
        [t1.key, t2.value]
end
```

Pasted from <https://github.com/bloom-lang/bud/blob/master/docs/cheat.md>

r   s
key, val => key, val
join cond.
[key, val]

(r*s).pairs(...condition...): do something for all pairs where the match condition is as specified.
do...end: ruby for multi-line implicit map {...}
(:r.value => :s.key): a match condition for the pairs
  (:r.value and :s.key are names).

```
# the above Bloom statements are equivalent to this SQL:
# SELECT r.key, s.value
#   FROM r, s
#  WHERE r.value = s.key;
```

A simple claim

Wednesday, April 27, 2011
3:01 PM

A simple claim

- Bud is at least as powerful as Pig
- Demonstration:
  - FILTER-BY
  - FOREACH-GENERATE
  - COGROUP-BY
  - etc…
- Are all implementable in Bud.

But wait, there's more!

Merges and ticks

Wednesday, April 27, 2011

11:21 AM

Merges and ticks

A bloom program executes in **ticks.**

Until a tick, all statements are considered to be declarations.

At a tick, statements are executed and the results are cached. (Compare with pig's "store" command).

A tick is logically equivalent to a **barrier synchronization**, i.e., it waits for (eventual) consistency of the affected tables.

Scratches

Wednesday, April 27, 2011
11:40 AM

Scratches

A regular table is persistent until it is deleted.
A scratch is a table that lasts only for one tick.
Purpose: to compute partial results or to serve as a temporary table.

```
scratch :passing_clouds
passing_clouds <= [[3, "Nimbus"], [2, "Cumulonimbus"]]
```

Pasted from <https://github.com/bloom-lang/bud/blob/master/docs/getstarted.md>

Note that while we write a pair as **[3,"Nimbus"]**, we think of that as a key/value relationship **3 => "Nimbus"**!

Channels

Wednesday, April 27, 2011
12:35 PM

Channels and Interfaces

A channel is a path of communication between two (distributed) entities
An interface is a path of communication between two (local) modules.
One can send a table through a channel or interface.
Channels are by nature **eventually consistent.**
Both channels and interfaces are scratches (i.e., they are temporary).

The stdio channel

Wednesday, April 27, 2011
12:38 PM

The most common channel is stdio

```
stdio <~ [['hello'], ['world']]
```

prints

```
hello
world
```

but not necessarily in that order!
Only <~ makes sense for channels.

Caveats: when one sends facts through a channel,

- order is not preserved!
- delivery is not even reliable!

Network service in a line

Wednesday, April 27, 2011
3:09 PM

A simple Bud hack: make a network chat server in a couple lines of code

```
nodelist <= connect.payloads
```

Instantaneously transfer a message coming in on connect to the nodelist channel.

```
mcast <~ (mcast * nodelist).pairs { |m,n| [n.key, m.val] }
```

Eventually respond by multi-casting the message sent to you to all subscribers.

Pasted from <https://github.com/bloom-lang/bud/blob/master/docs/getstarted.md>

Lessons learned

Wednesday, April 27, 2011
3:13 PM

Lessons learned

- Bud merges the offline power of Map/Reduce and the online power of Axis.
    - One can do M/R queries.
    - One can implement services.
- In the same language!
- There is great power in being able to specify consistency properties in the code itself.