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 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.
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.
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)
A language framework for **disorderly programming** in clouds. Based upon
- **temporal logic,**
- **eventual consistency,** and
- **logical monotonicity.**
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
A dialect of **ruby**
with distributed objects and operators.

Basic data structure: a table

```ruby
table :clouds, [:key] => [:value]
```

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

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

```plaintext
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.
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

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

\( x \leq y \) means put data into \( x \) exactly when it arrives in \( y \).
\( x \preceq y \) means put data into \( x \) some time after it arrives in \( y \).
\( x \prec y \) means subtract \( y \) values from \( x \) values regardless of when data arrives in \( x \) or \( y \).
\( x \rhd y \) means put data into \( x \) from \( y \) even if actions on \( x \) try to delete it!
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!
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.,

```ruby
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
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

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

\[
t_1 \text{ has schema } [:\text{game}] \Rightarrow [:\text{player}, :\text{score}]
\]

then

\[
totals \leq t_1.\text{groupby}([:\text{player}], \text{sum}(:\text{score}))
\]

has schema [:\text{player}]\Rightarrow[:\text{score}]

(Implicitly, the grouped thing becomes a key!)

Builtin aggregate functions:

count, sum, choose, avg, min, max...
As in Pig, one useful construction is the cross product. For tables \( t_1 \) and \( t_2 \), \( (t_1 \times t_2) \) is a new table with keys that are pairs of keys from \( t_1 \) and \( t_2 \), values that are pairs of values. I.e., if \( k_1 \to v_1 \) is in \( t_1 \) and \( k_2 \to v_2 \) is in \( t_2 \), then \( [k_1,k_2] \to [v_1,v_2] \) is in \( t_1 \times t_2 \). In Pig, this is equivalent with CROSS.
A join is a filter for a product!

```ruby
# 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).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

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

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

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"]]

Note that while we write a pair as [3,"Nimbus"], we think of that as a key/value relationship 3 => "Nimbus"!
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 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!
A simple Bud hack: make a network chat server in a couple lines of code

```ruby
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

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.