Interpreting the Data: Parallel Analysis with Sawzall

Rob Pike, Sean Dorward, Robert Griesemer, Sean Quinlan

Google, Inc.

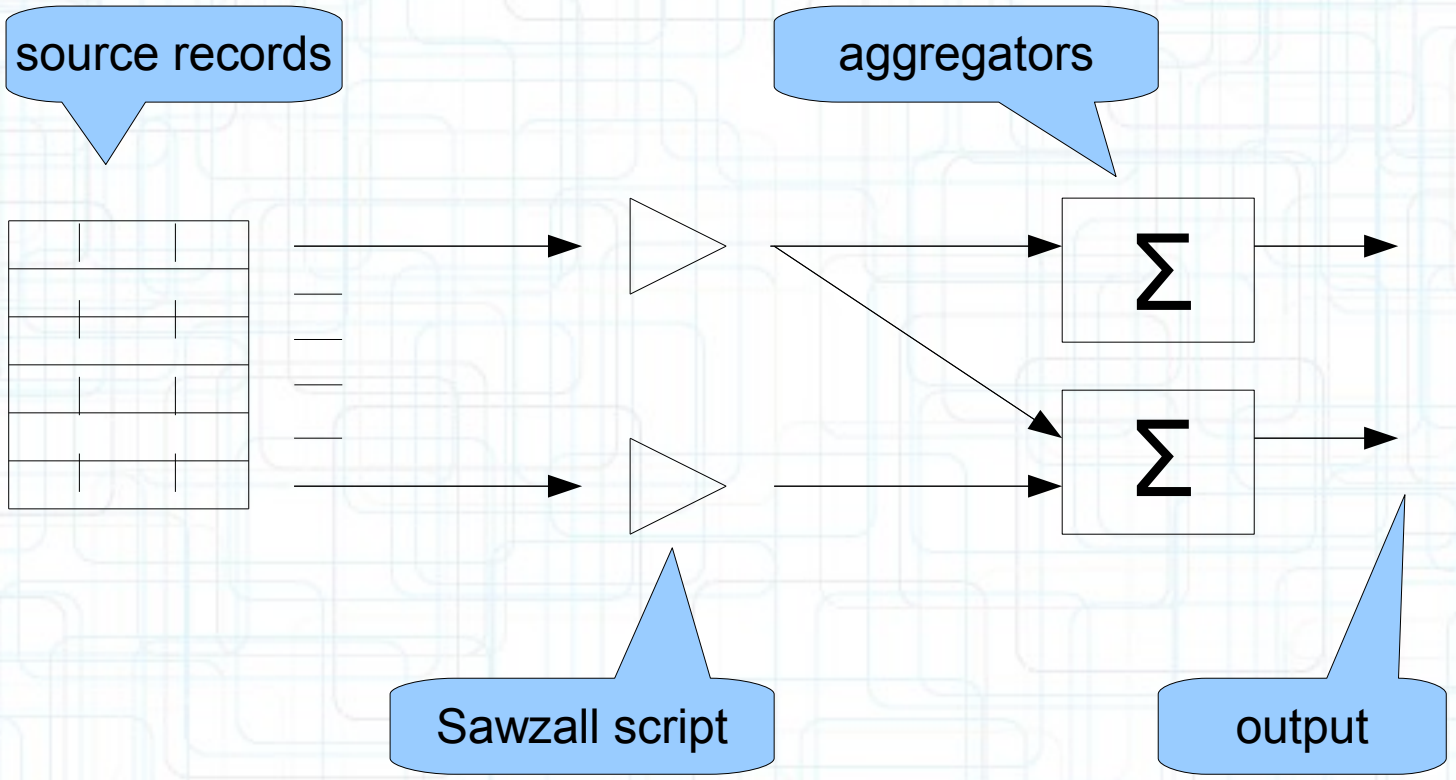
2005

Presented by Alexey Karyakin

Overview

- Data analysis on 1000's computers
- Isolating a programmer from details of distributed system
- Based on the observations of typical analysis tasks at Google

Model of computations

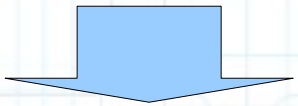


Model of computations

- Source data is a flat collection of records
- Two phases: record processing and aggregation
- Computation is commutative and associative with regard to each record

Model of computations

- Source data is a flat collection of records
- Two phases: record processing and aggregation
- Computation is commutative and associative with regard to each record



Easy to do in parallel!

Sawzall script

- Written by the user in a higher-level language
- Takes only one record at a time
- Emits output to one or more aggregators

Aggregators

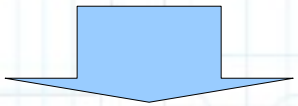
- Provided by Sawzall environment
- Used by Sawzall script

Model of execution

- Shared-nothing cluster
- Built on existing Google infrastructure
- Input files in GFS
- Uses MapReduce and Workqueue to schedule tasks

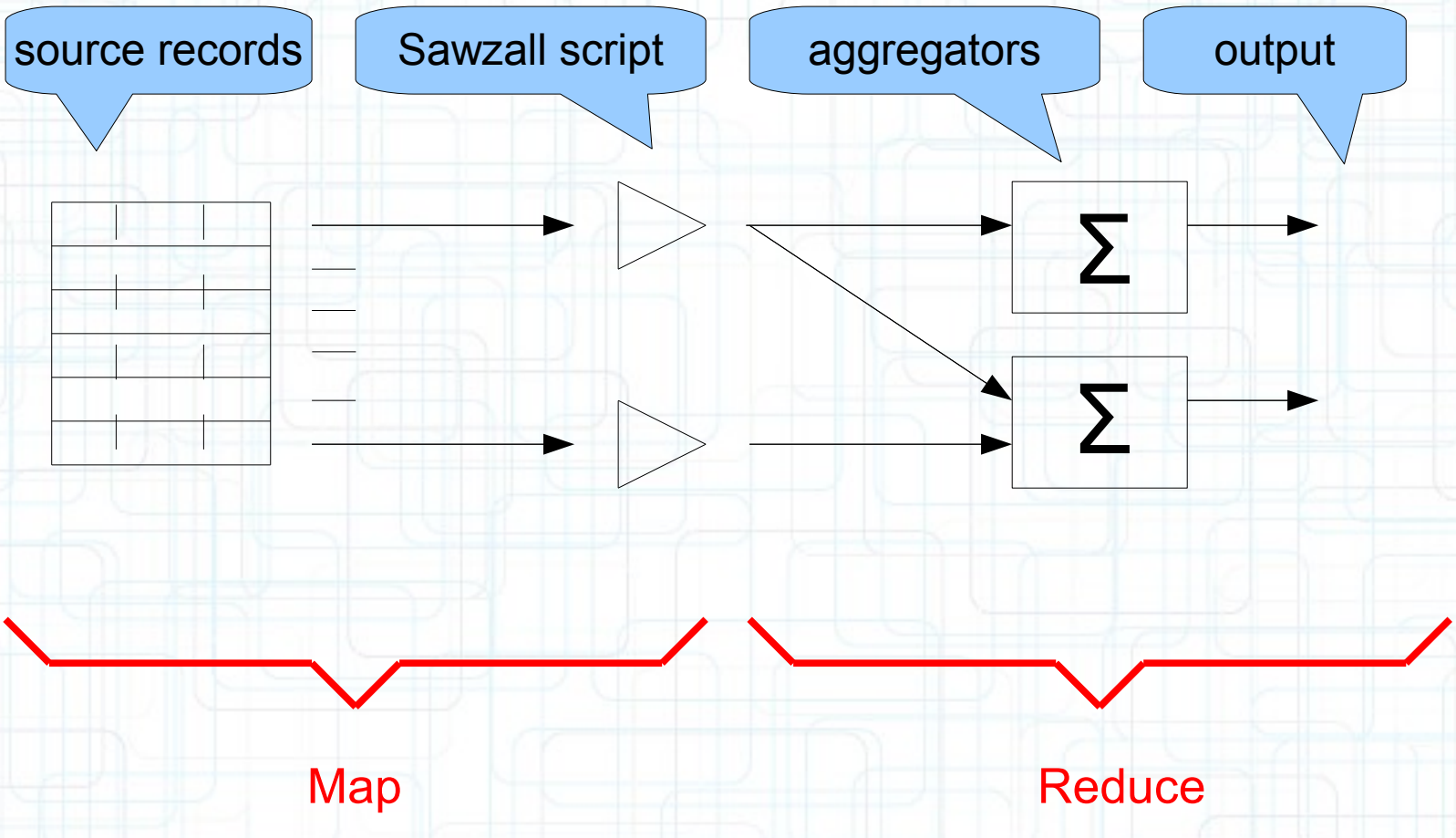
Model of execution

- Shared-nothing cluster
- Built on existing Google infrastructure
- Input files in GFS
- Uses MapReduce and Workqueue to schedule tasks



Fault-tolerance for free!

Model of execution



Sawzall tasks are executed under MapReduce

Sawzall language (basic)

- Procedural
- Influenced by C and Pascal
- Strongly typed
- Flow control: if, while, for, ...
- Compiled into byte-code

Sawzall language – data types

- Simple types (int, float, byte and Unicode strings)
- Collections: array, map, tuple
- Explicit type conversions
- Implicit type conversion on initialization

Sawzall language (special)

- Typed representation of input records using Google Protocol Buffers

```
parsed message Point {
    required int32 x = 1;
    required int32 y = 2;
    optional string label = 3;
};
```

```
08 64 10 c8 01 1a 06 63
65 6e 74 65 72
```

```
proto "point.proto";
p: Point = input;
```

```
p = { 100, 200, "center" };
```

Sawzall language (special)

- Quantifiers

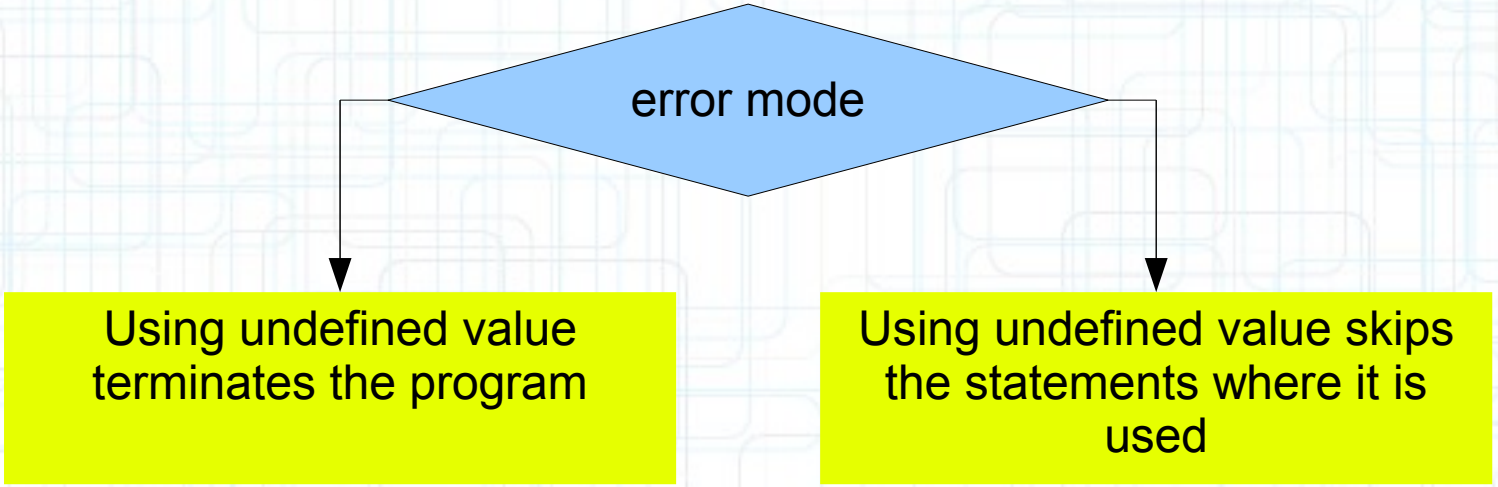
```
when (i: some int; a[i] == b)
   emit index <- i;

when (i: each int; a[i] > 0)
   emit sum <- a[i];

when (i: all int; a[i] > 0)
   emit positive_array <- 1;
```

Sawzall language (special)

- Error processing using undefined values



Aggregators

- Critical component, must be efficient, interacts with the system internals
- Run in Reduce and Combine phase
- Implemented by Sawzall run-time
- May be also created in C++ (min 200 l.o.c.)
- Not much details about aggregator interaction with MapReduce :-(

Standard aggregators

- `c: table collection of string;`
  `emit c <- "sample";`
- `s: table sample(100) of string;`
  `emit c <- "sample";`
- `s: table sum of int;`
  `emit s <- 1;`
- `t: top(10) of string;`
  `emit t <- "sample";`
- `m: table maximum(10) of string weight float;`
  `emit m <- "iron" weight 7.8;`
- `q: table quantile(101) of int;`
  `emit q <- 100;`

Indexed aggregators

- May be quite complex:

```
x: table top(1000) [country: string][hour: int]
of hits: int;

emit x[“France”][23] <- 165439976;
```

- Implemented using a map, using any data type as an index

Discussion

- Source data model looks like relational?
- Burden of implementing and supporting a procedural language with the library
- Too low level compared to ~~SQL,~~ Pig, Dryad
- Performance?
- One MapReduce step in each execution

Thank you!