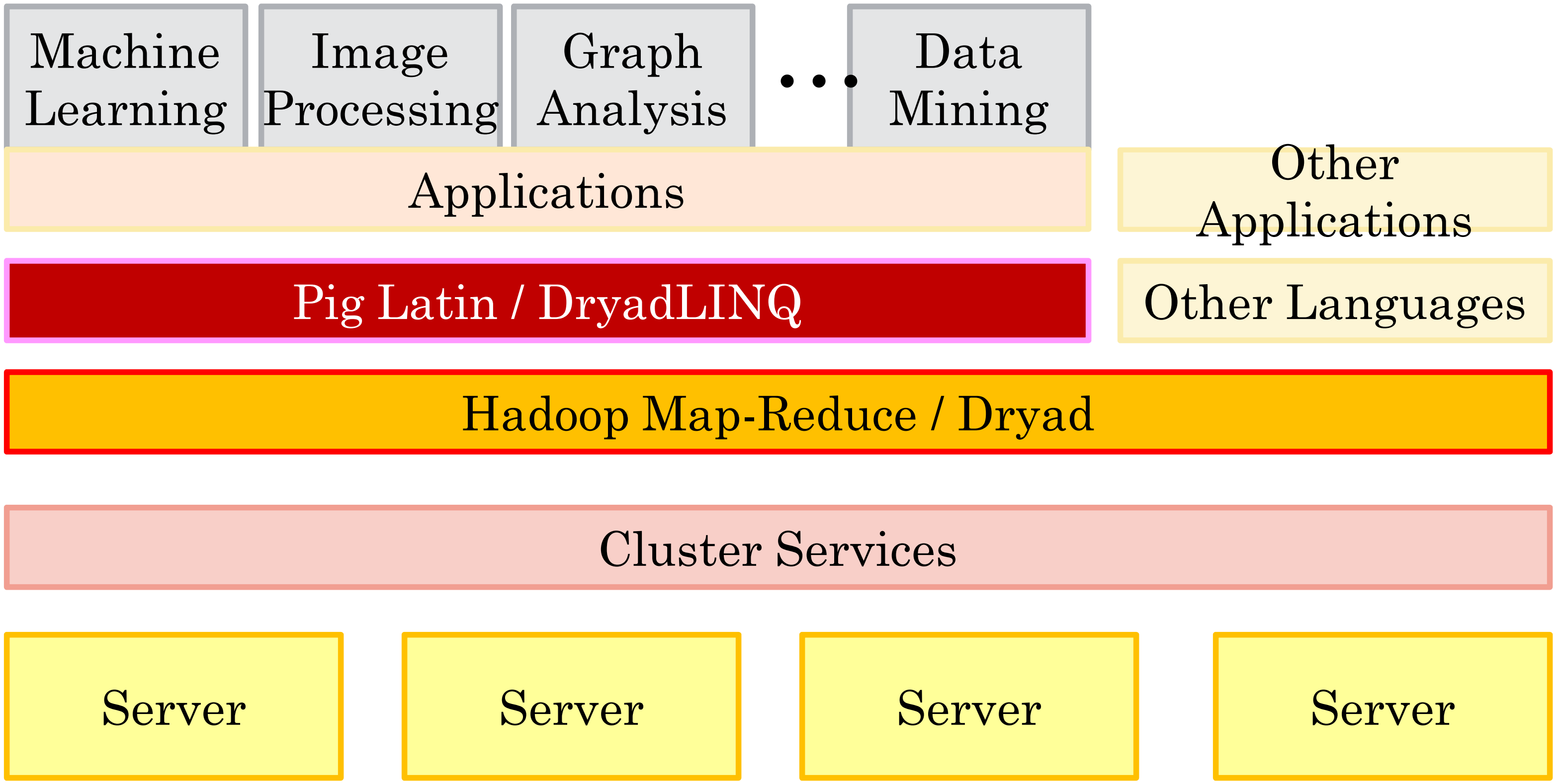
# CLOUD PROGRAMMING

**Andrew Harris & Long Kai**

# Motivation

- **Research problem**: How to write distributed data-parallel programs for a compute cluster?
- **Drawback of Parallel Databases (SQL)**: Too limited for many applications.
  - Very restrictive type system
  - The declarative query is unnatural.
- **Drawback of Map Reduce:** Too low-level and rigid, and leads to a great deal of custom user code that is hard to maintain, and reuse.

# LAYERS


Machine Learning
Image Processing
Graph Analysis
…
Data Mining
Applications
Other Applications
Pig Latin / DryadLINQ
Other Languages
Hadoop Map-Reduce / Dryad
Cluster Services
Server
Server
Server
Server

# PIG LATIN:

## A Not-So-Foreign Language for Data Processing

# DATAFLOW LANGUAGE

- User specifies a sequence of steps where each step specifies only a single, high level data transformation.  Similar to relational algebra and procedural – desirable for programmers.
- With SQL, the user specifies a set of declarative constraints.  Non-procedural and desirable for non-programmers.

# AN SAMPLE CODE OF PIG LATIN

## SQL

```
SELECT category, AVG(pagerank)
FROM urls WHERE pagerank > 0.2
GROUP BY category HAVING COUNT(*) > 10^6
```

## Pig Latin

*Pig Latin program is a sequence of steps, each of which carries out a single data transformation.*

```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY
    COUNT(good_urls)>10^6;
output = FOREACH big_groups GENERATE
    category, AVG(good_urls.pagerank);
```

# DATA MODEL

- *Atom*: Contains a simple atomic value such as a string or a number, e.g., ‘Joe’.
- *Tuple*: Sequence of fields, each of which might be any data type, e.g., (‘Joe’, ‘lakers’)
- *Bag*: A collection of tuples with possible duplicates. Schema of a bag is flexible.

$$\left\{ \begin{array}{c} (\texttt{'alice'},\ \texttt{'lakers'}) \\ \big(\texttt{'alice'},\ (\texttt{'iPod'},\ \texttt{'apple'})\big) \end{array} \right\}$$

- *Map*: A collection of data items, where each item has an associated key through which it can be looked up. Keys must be data atoms.

$$\left[ \begin{array}{c} \texttt{'fan of'} \to \left\{ \begin{array}{c} (\texttt{'lakers'}) \\ (\texttt{'iPod'}) \end{array} \right\} \\ \texttt{'age'} \to 20 \end{array} \right]$$

# A Comparison with Relational Algebra

### Pig Latin

- Everything is a bag.
- Dataflow language.
- FILTER is same as the Select operator.

### Relational Algebra

- Everything is a table.
- Dataflow language.
- Select operator is same as the FILTER cmd.

*Pig Latin has only included a small set of carefully chosen primitives that can be easily* ***parallelized****.*

# SPECIFYING INPUT DATA: LOAD

```
queries = LOAD `query_log.txt'
          USING myLoad()
          AS (userId, queryString, timestamp);
```

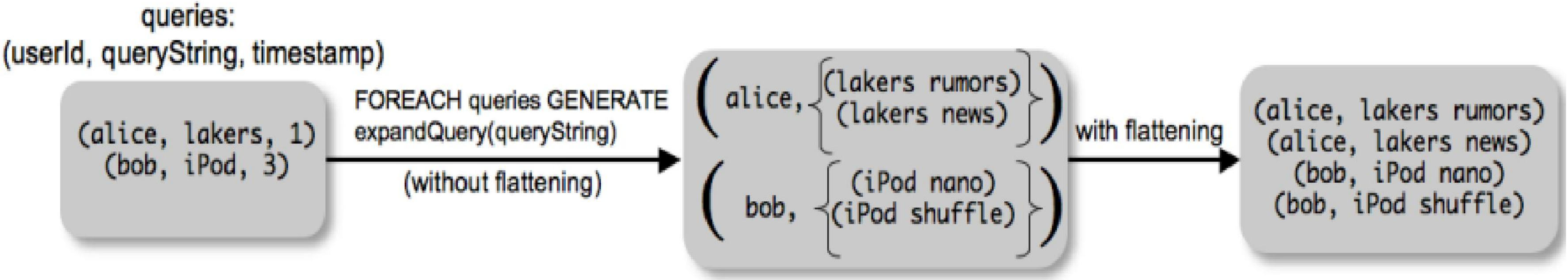
- The input file is “query_log.txt”.
- The input file should be converted into tuples by using the custom myLoad deserializer.
- The loaded tuples have three fields named userId, queryString, and timestamp.

*Note that the LOAD command does not imply database-style loading into tables. It’s only logical.*

# Per-tuple Processing: FOREACH

Expanded_queries = FOREACH queries GENERATE userId, expandQuery(queryString);

- *expandQuery* is a User Defined Function.
- Nesting can be eliminated by the use of the FLATTEN keyword in the GENERATE clause.
  - userId, FLETTEN(expandQuery(queryString));


queries:
(userId, queryString, timestamp)
(alice, lakers, 1)
(bob, iPod, 3)
FOREACH queries GENERATE expandQuery(queryString)
(without flattening)
( alice, {(lakers rumors) (lakers news)} )
( bob, {(iPod nano) (iPod shuffle)} )
with flattening
(alice, lakers rumors)
(alice, lakers news)
(bob, iPod nano)
(bob, iPod shuffle)

# Discarding Unwanted Data: FILTER

real_queries = FILTER queries BY userId neq `bot';

real_queries = FILTER queries BY NOT isBot(userId);

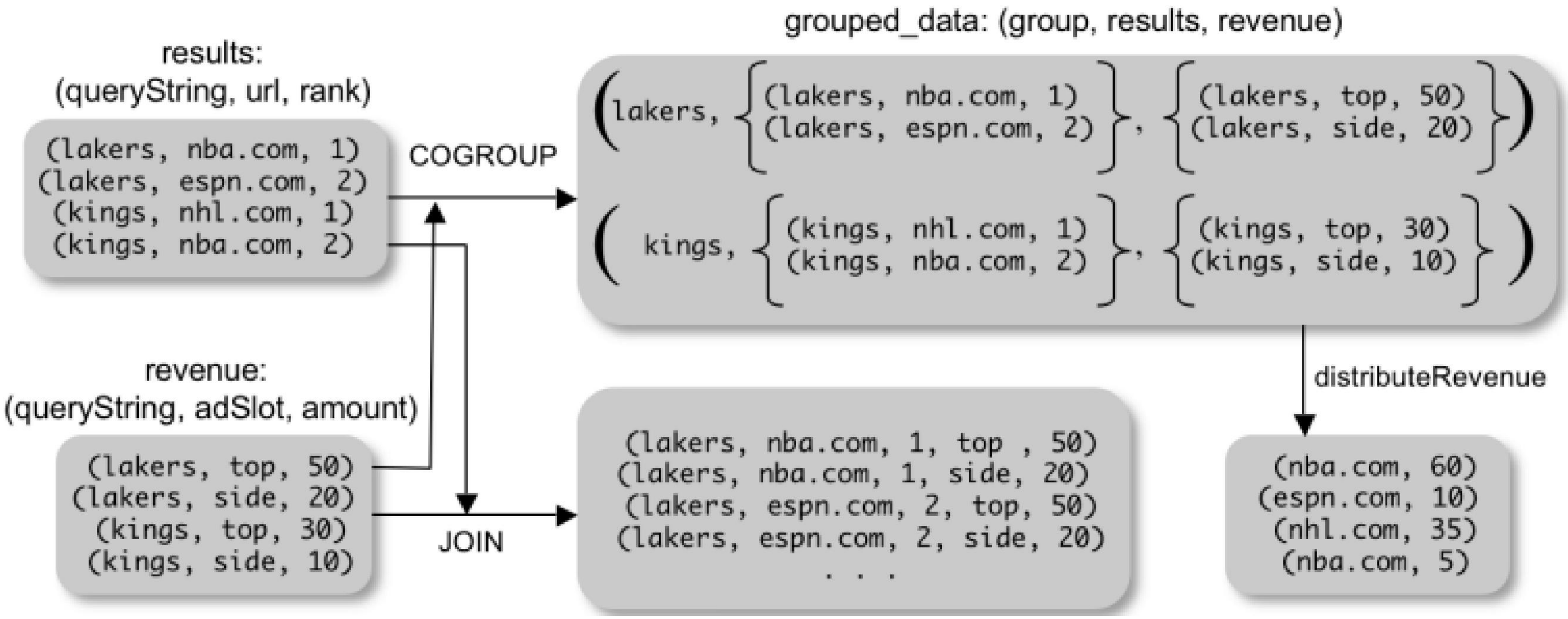
- Again, *isBot* is a User Defined Function
- Operations might be ==, eq, !=, neq, <, >, <=, >=
- A comparison operation may utilize Boolean operators (AND, OR, NOT) with several expressions

# Getting Related Data Together: COGROUP

```
grouped_data = COGROUP results BY queryString,
                       revenue BY queryString;
```

- group together tuples from one or more data sets, that are related in some way, so that they can subsequently be processed together.
- In general, the output of a COGROUP contains one tuple for each group.
- The first field of the tuple (named group) is the group identifier. Each of the next fields is a bag, one for each input being cogrouped.

# MORE ABOUT COGROUP



results:
(queryString, url, rank)

```
(lakers, nba.com, 1)
(lakers, espn.com, 2)
(kings, nhl.com, 1)
(kings, nba.com, 2)
```

revenue:
(queryString, adSlot, amount)

```
(lakers, top, 50)
(lakers, side, 20)
(kings, top, 30)
(kings, side, 10)
```

grouped_data: (group, results, revenue)

```
(lakers, {(lakers, nba.com, 1)
          (lakers, espn.com, 2)}, {(lakers, top, 50)
                                   (lakers, side, 20)})
( kings, {(kings, nhl.com, 1)
          (kings, nba.com, 2)}, {(kings, top, 30)
                                 (kings, side, 10)})
```

```
(lakers, nba.com, 1, top , 50)
(lakers, nba.com, 1, side, 20)
(lakers, espn.com, 2, top, 50)
(lakers, espn.com, 2, side, 20)
. . .
```

```
(nba.com, 60)
(espn.com, 10)
(nhl.com, 35)
(nba.com, 5)
```

*COGROUP + FLATTEN = JOIN*

# EXAMPLE: MAP-REDUCE IN PIG LATIN

```
map_result = FOREACH input GENERATE
   FLATTEN(map(*));
key_groups = GROUP map_result BY $0;
output = FOREACH key_groups GENERATE reduce(*);
```

- A map function operates on one input tuple at a time, and outputs a bag of key-value pairs.
- The reduce function operates on all values for a key at a time to produce the final results.

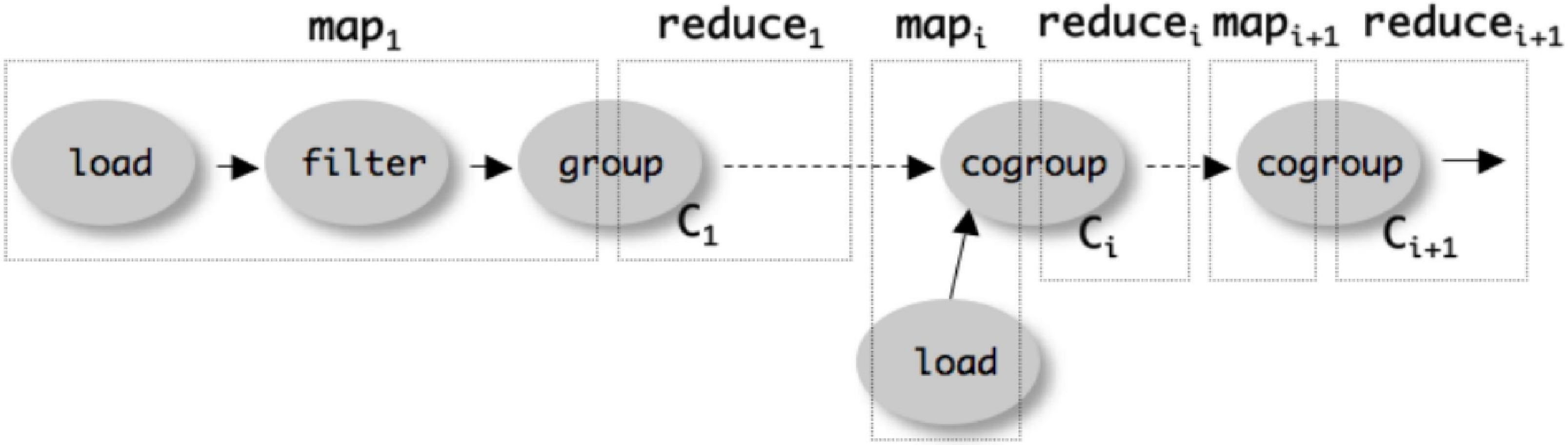
# IMPLEMENTATION

- Building a ***logical plan***:
  - Pig builds a logical plan for every bag that the user defines.
  - No processing is carried out when the logical plans are constructed. Processing is triggered only when the user invokes a STORE command on a bag.

- Compilation of the logical plan into a ***physical plan***.

# Map-Reduce Plan Compilation

- The map-reduce primitive essentially provides the ability to do a large-scale group by, where the map tasks assign keys for grouping, and the reduce tasks process a group at a time.
- Converting each (CO)GROUP command in the logical plan into a distinct map-reduce job with its own map and reduce functions.



# OTHER FEATURES

- Fully nested data model.
- Extensive support for user-defined functions.
- Manages plain input files without any schema information.
- A novel debugging environment.

# DISCUSSION:
# PIG LATIN MEETS MAP-REDUCE

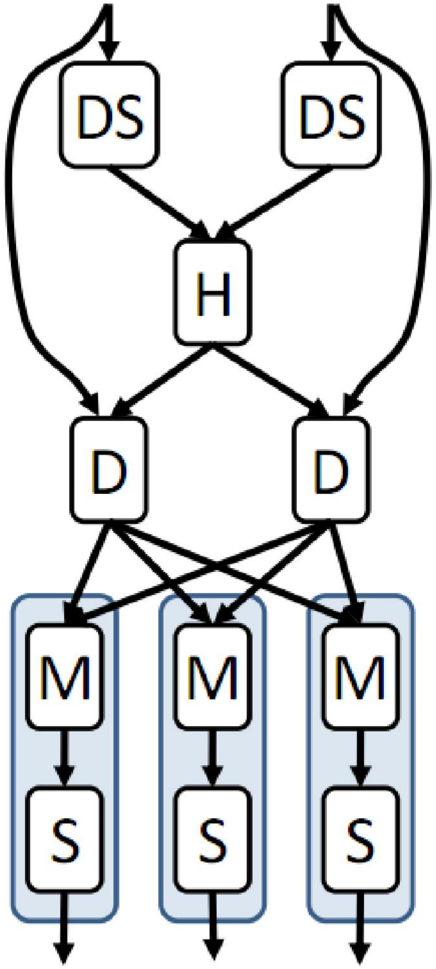
- Is it necessary to run Pig Latin on Map-Reduce platform?
- Is Map-Reduce a perfect platform for Pig Latin? Any drawbacks?
  - Data must be materialized and replicated on the distributed file system between successive map-reduce jobs.
  - Not flexible enough.
- Well, it does work fine. parallelism, load-balancing, and fault-tolerance……

# DryadLINQ

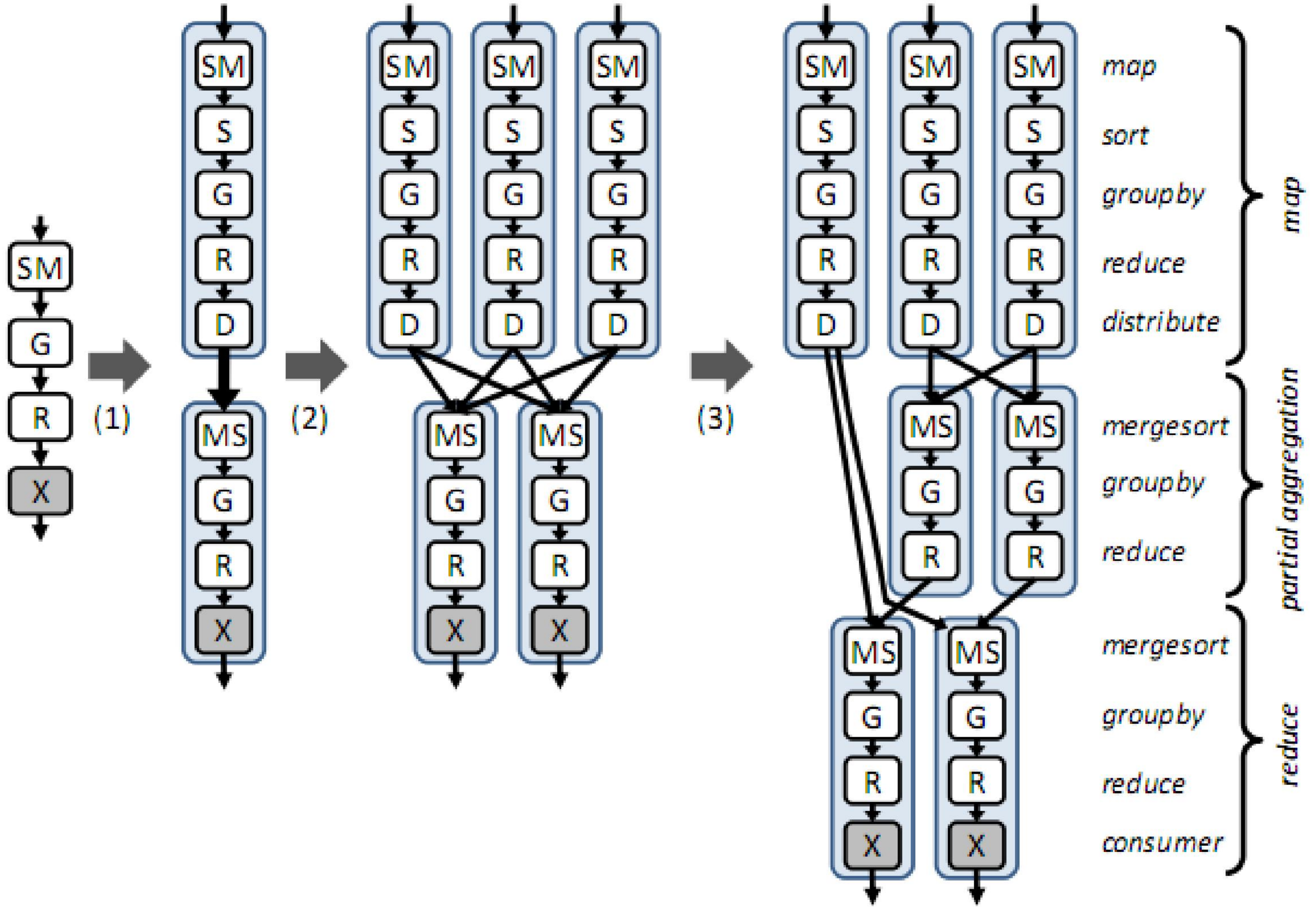
## A System for General-Purpose Distributed Data-Parallel Computing

# DRYAD EXECUTION PLATFORM

- Job execution plan is a dataflow graph.
- A Dryad application combines computational "vertices" with communication "channels" to form a dataflow graph.



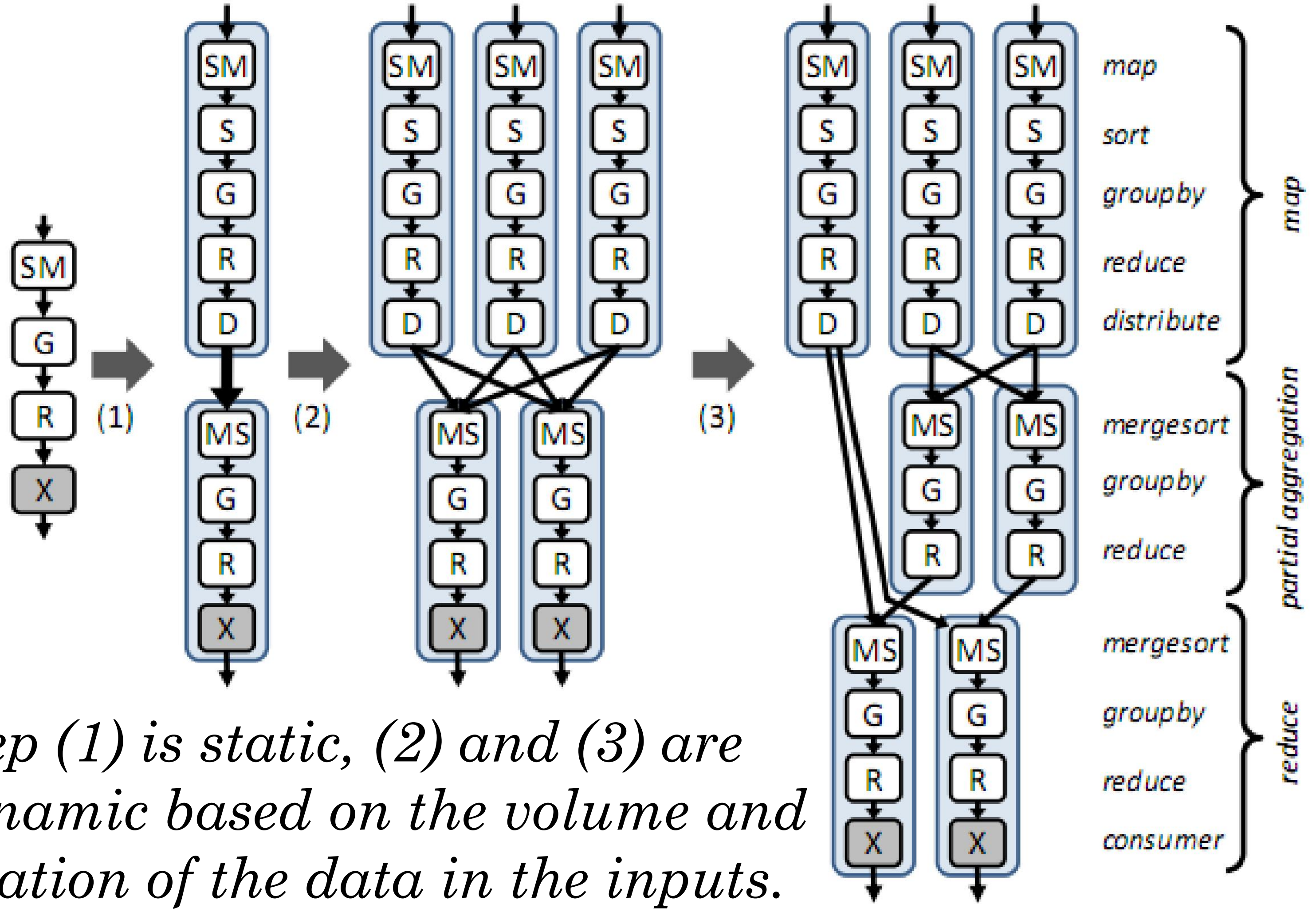
# Map-Reduce in DryadLINQ



# IMPLEMENTATION - OPTIMIZATIONS

- Static Optimizations
  - **Pipelining**: Multiple operators may be executed in a single process.
  - **Removing redundancy**: DryadLINQ removes unnecessary partitioning steps.
  - **Eager Aggregation**: Aggregations are moved in front of partitioning operators where possible.
  - **I/O reduction**: Where possible, uses TCP-pipe and in-memory FIFO channels instead of persisting temporary data to files.
- Dynamic Optimizations
  - Dynamically sets the number of vertices in each stage at run time based on the size of its input data.
  - Dynamically mutate the execution graph as information from the running job becomes available.

# Map-Reduce in DryadLINQ



*Step (1) is static, (2) and (3) are dynamic based on the volume and location of the data in the inputs.*

# Incremental Processing with Percolator

Long Kai and Andrew Harris

We optimized the flow of processing... Now what?

Make it update faster!

# Incremental Processing

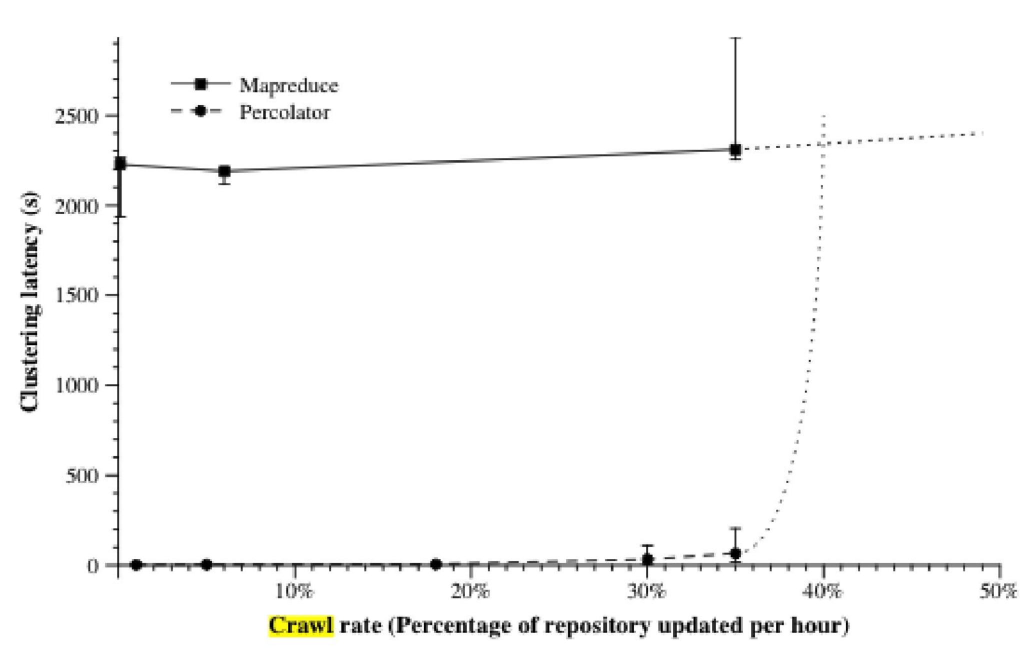
- Instead of processing the entire dataset, only process what needs to be updated
- Requires random read/write access to data
- Suitable for data that is independent (data pieces do not depend on other data pieces) or only marginally dependent
- Reduces seeking time, processing overhead, insertion/update costs

# Google Percolator

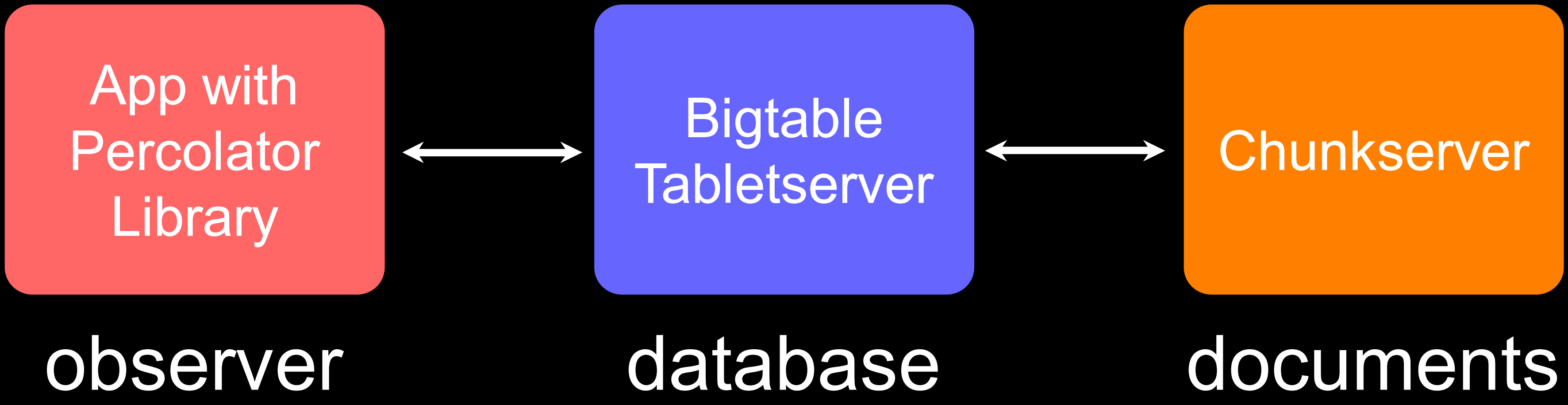
- Introduced at OSDI ’10
- Core tech behind Google Caffeine search platform - driving app: Google’s indexer
- Allows random access and incremental updates to petabyte-scale data sets
- Dramatically reduces cost of updates, allowing for “fresher” search results

# Previous Google System

- Same number of documents (billions per day)
- 100 MapReduces to compile web index for these documents
- Each document spent 2-3 days being indexed



# How It Works

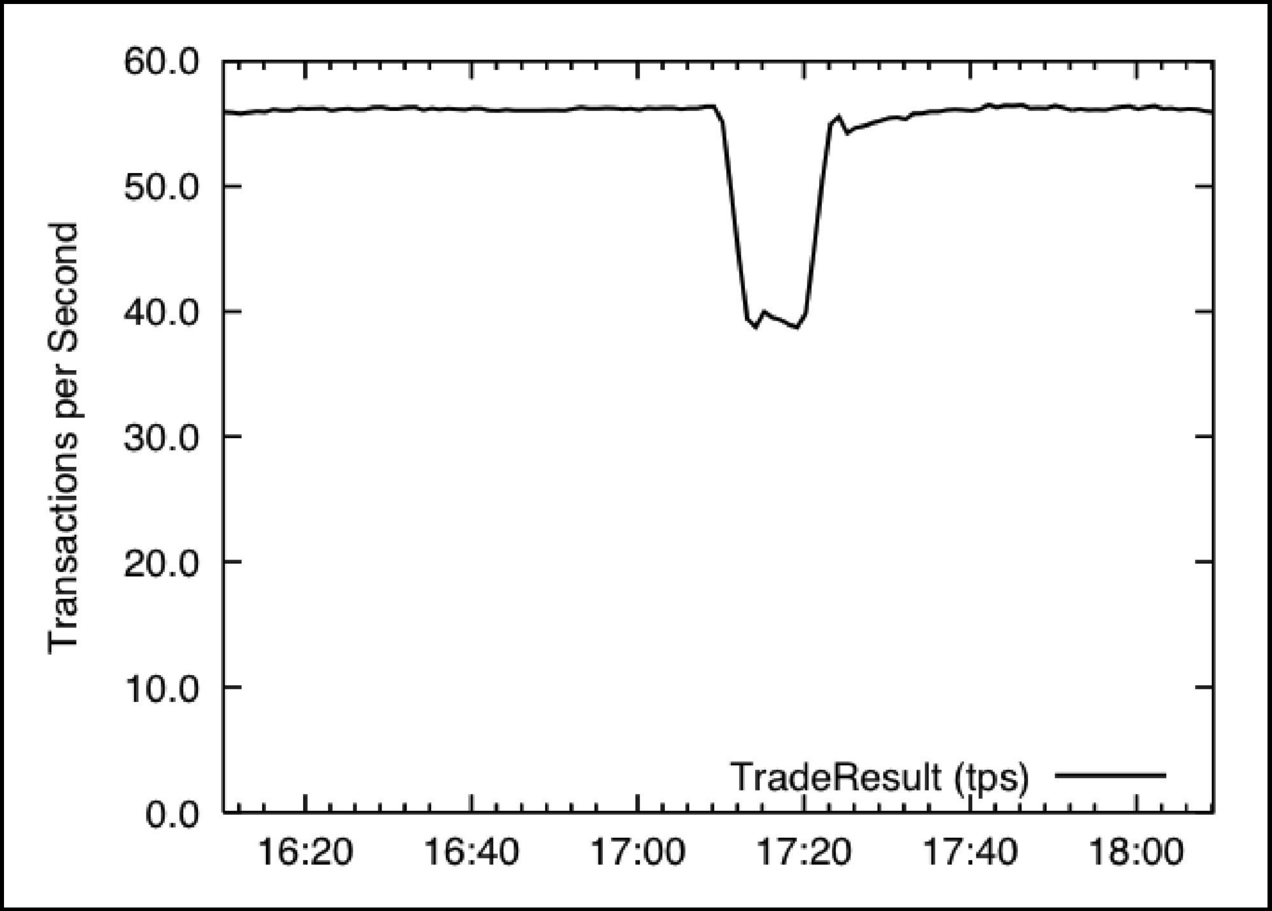


All communication handled via RPCs
Single lines of code in observer
Google indexing system uses ~10 observers

# Transactions

- Observer-Bigtable communication is handled as an ACID transaction
- Observer nodes themselves handle deadlock resolution
- Simple lock cleanup synchronization
- All writes are increasingly timestamped via coordinated timestamp oracle

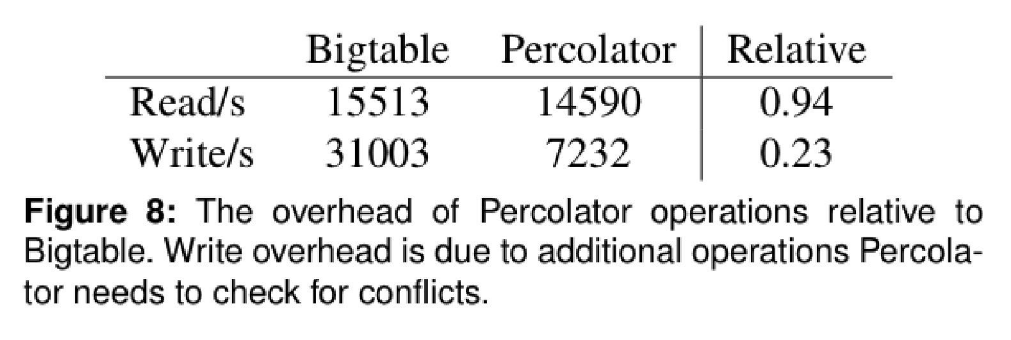
# Fault Tolerance



Result of dropping 33% of tablet servers in use

# Pushing Updates

- Percolator clients open a write-only connection with Bigtable
- Obtain write lock for specific table location
  - If locked, determine if lock is from a previously failed transaction
- Overhead:



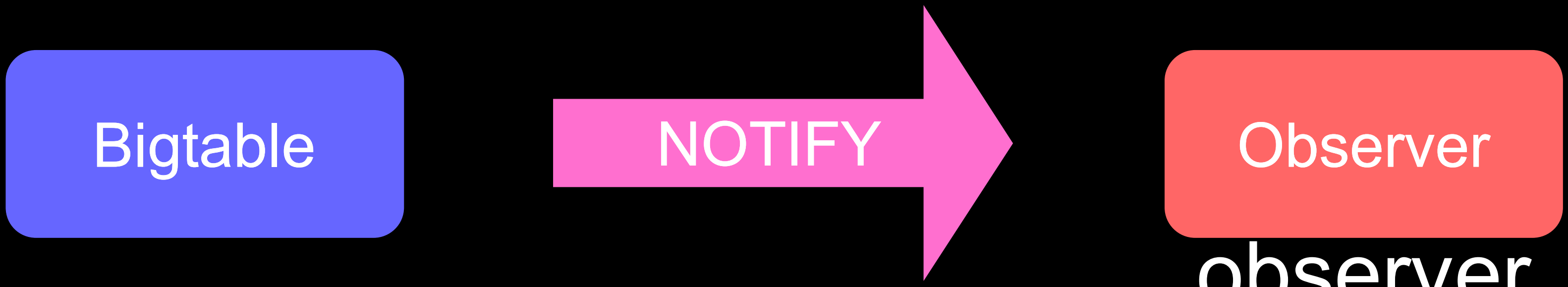
| | Bigtable | Percolator | Relative |
|---|---|---|---|
| Read/s | 15513 | 14590 | 0.94 |
| Write/s | 31003 | 7232 | 0.23 |

**Figure 8:** The overhead of Percolator operations relative to Bigtable. Write overhead is due to additional operations Percolator needs to check for conflicts.

# Notifying the Observers

- Handled separately from writes (data connections are unidirectional)
- Otherwise similar to database triggers
- Multiple Bigtable changes may produce only one notification

# Notifying the Observers



Bigtable

NOTIFY

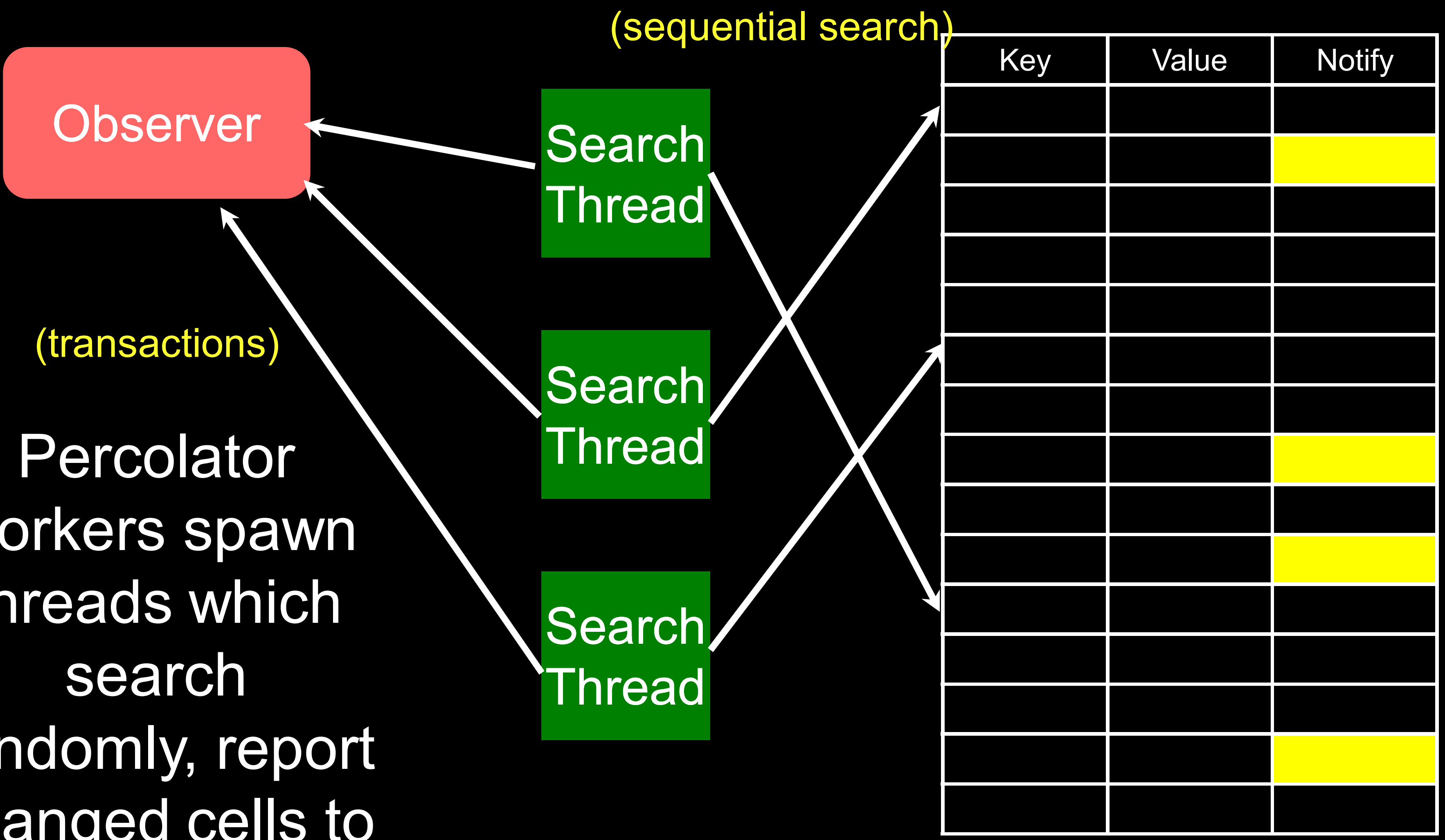
Observer

observed column is changed one or more times

new update transaction

observer receives **most recent** column data

# Keeping Clean



| Key | Value | Notify |
|---|---|---|
| | | |
| | | 🟨 |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | 🟨 |
| | | |
| | | 🟨 |
| | | |
| | | |
| | | |
| | | 🟨 |
| | | |

Percolator workers spawn threads which search randomly, report changed cells to observer

# Benefits!

- Closer to DBMS performance
  - “Only” 30x processing overhead against comparison DBMS (TPC-E, a stock market trading backend)
- Fresher data pushed for lower costs
  - 100x faster document movement
  - 1000x faster document processing
  - Data set is also 3x larger than previous!
  - Fixes stragglers - everything updates

# Discussion

- Transactions introduce read/write overhead relative to Bigtable size - when does scaling break down?
- Not suitable for updating heavily dependent or rapidly mutating data sets - how do you adapt for these?
- In lightly dependent data sets, causally linked children may report updates before their parents - implications?