Percolator

Smruti R. Sarangi

Department of Computer Science
Indian Institute of Technology
New Delhi, India

Outline

1. Motivation
   - Google's Search Algorithm
   - Requirements
2. Design
   - Structure
   - Algorithm
   - Details and Optimizations
3. Evaluation

Outline

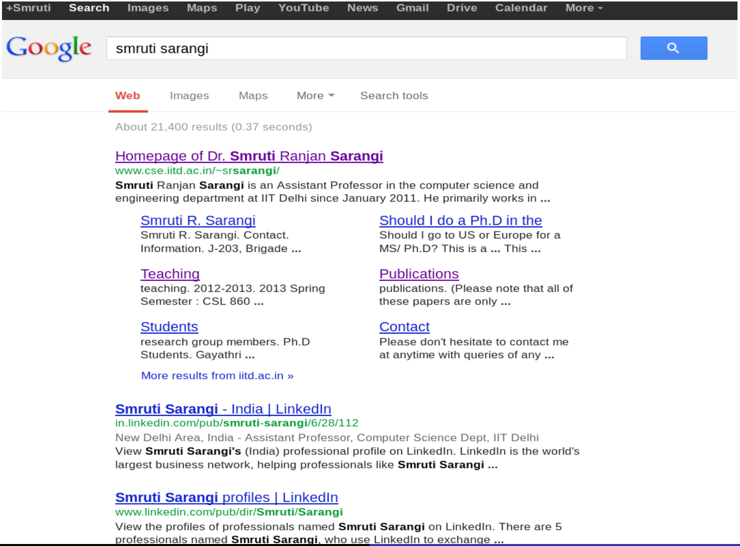
1. Motivation
   - Google's Search Algorithm
   - Requirements
2. Design
   - Structure
   - Algorithm
   - Details and Optimizations
3. Evaluation

- Updating Google's web index continually is a major challenge.
  - Tens of petabytes of data
  - Billions of updates per day
  - Thousands of machines.
  - Cascading updates.

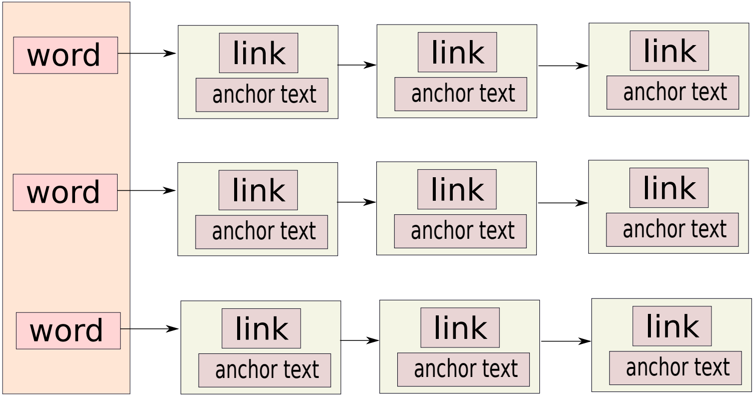
Google's Search Algorithm

- Every page has a "page rank".
- The page rank of a popular page is supposed to be high.
- The page rank of a page is determined by the page rank of all the pages that link to it.
- For example:
  - If the New York Times website points to some link, then it has a high page rank. ☺
  - If Hauz Khas Times points to some website, it will have a very low page rank. ☹

Example of a Google Search Query



Structure of a Web Index



The Problem of Updates

- The links in the inverted list are arranged according to their page rank.
- If the page rank of a website changes then:
  - We need to update the inverted list to reflect the change.
  - The page rank of sites that it points to need to change.
  - This problem is known as cascading update .

Outline

Requirements of a Solution

- Should provide ACID transaction semantics (do not want to corrupt database).
- Should have high throughput, and acceptable latency.
- Should be able to handle petabytes of data.
- Traditional DBMS systems are too slow $\rightarrow$ Need new technology
- Random access to data such that changes can percolate
- Consistency Model: Snapshot Isolation

Snapshot Isolation

- Assume two concurrent updates to a linked list.
  - If they do not access the same node or its parent, then they are disjoint.
  - Disjoint accesses can continue in parallel.
  - This is different from regular transaction semantics such as serializability.
- Definition :
  - When a transaction starts, it takes(appears to) a consistent snapshot of the entire database.
  - It then proceeds to update its private copy of the database.
  - The values are committed if they have not been changed by another transaction since the snapshot.

Outline

Design of Percolator

- Built on top of Bigtable – Google's distributed storage engine
- Bigtable is a multidimensional database
  - Distributed key-value store
  - We save – row, column, timestamp
  - Atomic read-modify-write operations for each row
  - Meta data is stored in separate columns
- Observer framework
  - Any row has a set of observers.
  - They run specialized functions when data in the row changes.

Model of Transactions

- Provides support for ACID transactions
  - Hard to do in such a large database
  - Required: do not want to have Google's database in an inconsistent state
  - Uses timestamp for each data item
  - The set of timestamps at the beginning of a transaction is its snapshot.
- Transactions can include multiple rows across multiple BigTable tables
- Percolator implements its own lock service
- Percolator adds a special column to save locks.

Columns in BigTable

Column	Use
lock	contains a pointer to the lock
write	timestamp of committed data
data	data value
notify	list of observers
ack_O	last timestamp at which observer *O* ran

Example

A transfers B 7$

key	data	lock	write
A	6: 5:10$	6: 5:	6:data@5 5:
B	6: 5:2$	6: 5:	6:data@5 5:

key	data	lock	write
A	7:**3$** 6: 5:10$	7:**primary** 6: 5:	7: 6:data@5 5:
B	6: 5:2$	6: 5:	6:data@5 5:

Example - II

key	data	lock	write
A	7:3\$ 6: 5:10\$	7: primary 6: 5:	7: 6:data@5 5:
B	7:**9\$** 6: 5:2\$	7: **primary@A** 6: 5:	7: 6:data@5 5:

key	data	lock	write
A	8: 7:3\$ 6: 5:10\$	8: 7: 6: 5:	8: **data @ 7** 7: 6:data@5 5:
B	7:9\$ 6: 5:2\$	7: primary@A 6: 5:	7: 6:data@5 5:

Example - III

key	data	lock	write
A	8: 7:3$ 6: 5:10$	8: 7: 6: 5:	8: **data @ 7** 7: 6:data@5 5:
B	8: 7:9$ 6: 5:2$	8: 7: 6: 5:	8: **data @ 7** 7: 6:data@5 5:

Outline

1. Motivation
   - Google's Search Algorithm
   - Requirements
2. Design
   - Structure
   - Algorithm
   - Details and Optimizations
3. Evaluation

Algorithm: Begin Transaction

Algorithm 1: Begin Transaction

1 *startTs* $\leftarrow$ oracle.getTimeStamp()

2 Set(W):
writes.push(W)

Get Method

```
Get(row, column, value):
while True do
    T ← startTrans(row)
    if T.hasLock(0,startTs) then
        backOffAndMaybeRemoveLock(row,col)
        continue
    end
    latestWrite ← T.read(row, [0,startTs])
    if !latestWrite then
        return φ
    end
    dataTs ← latestWrite.timeStamp
    return (T.read(row, "data", dataTs)
end
```

PreWrite

```
PreWrite(Write w, Write primary)
Column col ← w.col
T ←startTransaction(w.row)

if T.read(w.row, "write", [startTs, ∞]) then
    return false
end
if T.read(w.row, "lock", [0, ∞]) then
    return false
end

T.write (w.row, "data", startTs, w.value)
T.write (w.row, "lock", startTs, {primary.row, primary.col})
return T.commit()
```

Commit - I

```
Commit()
    /* Prewrite all the entries */
(primary, secondaries) ← (writes[0], writes[1 ... n])
if !PreWrite(primary,primary) then
    return false
end
for Write w: secondaries do
    if !PreWrite(w,primary) then
        return false
    end
end
commitTs ← oracle.getTimeStamp()
```

Commit - II

```
/* Commit the primary */
T ← startTransaction(primary.row)

/* Test to see if aborted by somebody else */
if !T.read(primary.row, "lock", startTs) then
    return false
end
/* Write the primary and erase the lock */
T.write(primary.row, "write", commitTs, "data@"+startTs)
T.erase (primary.row, "lock", commitTs)

/* Point of commit */
if !T.Commit() then
    return false
end
```

Commit - III

```
for Write w: secondaries do
    write(w.row, "write", commitTs, "data@"+startTs)
    erase (w.row, "lock", commitTs)
end
return true
```

Outline

Timestamps

- The timestamp oracle needs to be able to sustain very high throughput.
- Possible to batch several RPC calls to the oracle to reduce network load.
- Needs to give out timestamps in increasing order.
- If it fails, then it needs to recover and issue timestamps that are greater than the ones it issued earlier.

Observers

- Each observer registers a set of columns, and a function.
- The function gets invoked, if any of the columns are updated.
- Possible to do message collapsing
- At most one observer's transaction will commit per column.
- Steps in running an observer
  - After an update to a column, Percolator sets the notify column.
  - A worker thread, ultimately picks up this information, and runs an observer.
  - If the latest timestamp of an observer run (**ack_O**) is less than the commit timestamp of the update, then run the observer.
  - Worker threads avoid clumping by scanning random parts of the database.

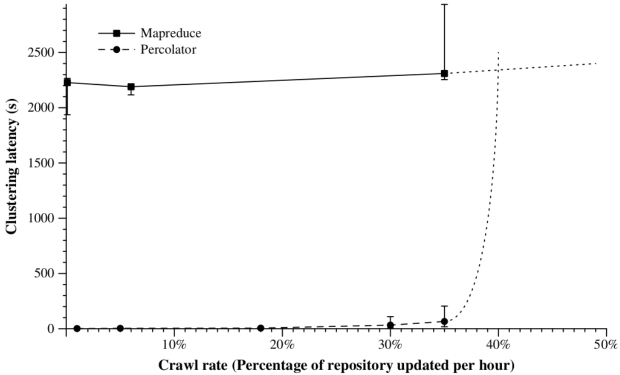
Performance Improvements

- Support for read-modify-write RPCs in BigTable.
- Create batches of RPC calls.
- Employ pre-fetching to reduce reads.
- Use blocking API calls, and a large number of threads to simplify the programming model.

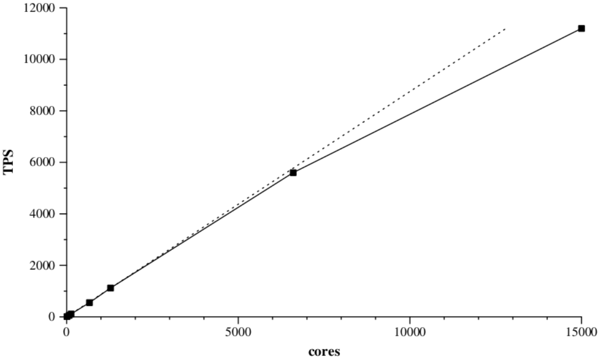
Setup

- Existing Setup:
  - Crawl billions of documents
  - Series of 100 map-reduces
  - document takes 2-3 days for getting indexed
- Percolator based indexing system – Caffeine
  - 100x faster
  - Average age of documents gets reduced to 50%

Performance vs Crawl Rate



Scalability for TPC/E benchmarks



Close to Linear Scaling

Large-scale Incremental Processing Using Distributed Transactions and Notifications by Daniel Peng and Frank Dabek, OSDI, 2010