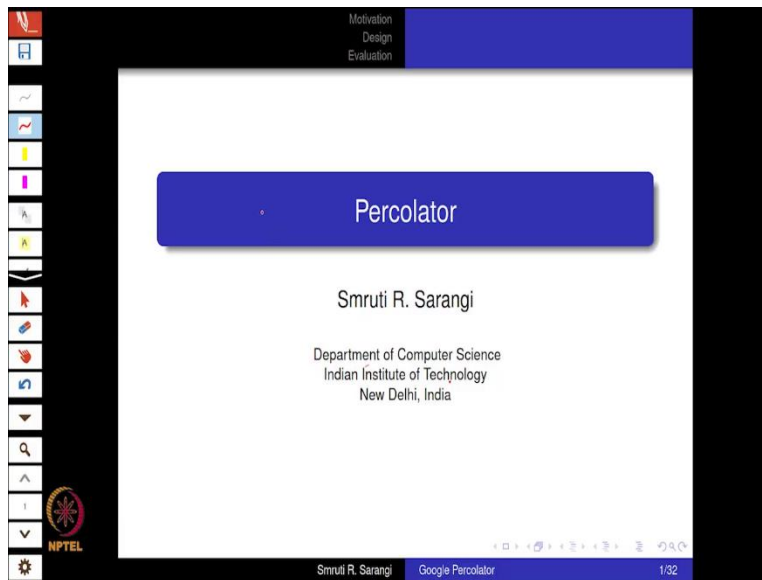


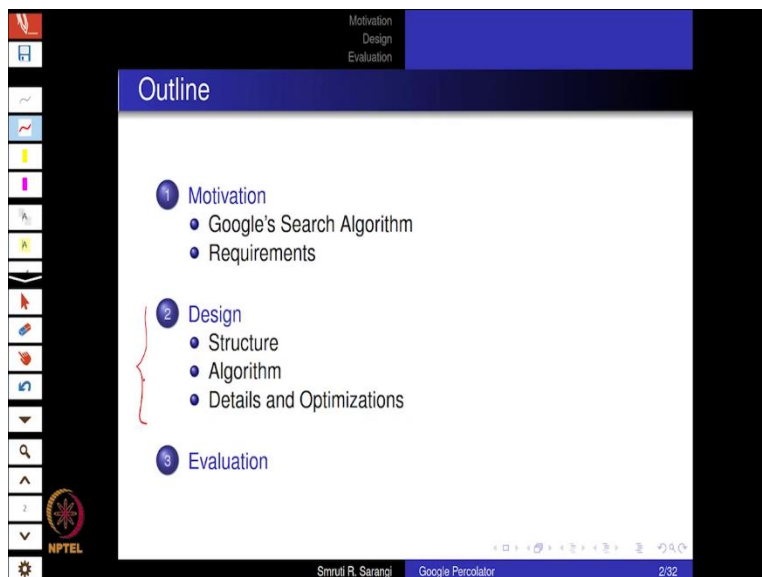
**Advanced Distributed Systems**  
**Professor Smruti R. Sarangi**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Delhi**  
**Lecture 19**  
**Google Percolator**

(Refer Slide Time: 00:17)



In this lecture, we will discuss the Google Percolator system. The percolator system is used to manage large part of Google's search infrastructure.

(Refer Slide Time: 00:27)



So, we will first discuss a little bit about Google's search algorithm, the requirements of such an algorithm, and then the design of the system, and finally, the evaluation.

(Refer Slide Time: 00:41)

This slide, titled "Outline", is the third slide in the presentation. It features a blue header with "Google's Search Algorithm" and "Requirements". The main content is a list of three main sections: 1. Motivation (with sub-points: Google's Search Algorithm, Requirements), 2. Design (with sub-points: Structure, Algorithm, Details and Optimizations), and 3. Evaluation. The slide is part of a presentation by Smruti R. Sarangi, titled "Google Percolator".

This slide, titled "Updating Google's web index continuously is a major challenge.", is the fourth slide in the presentation. It lists four challenges: Tens of petabytes of data, Billions of updates per day, Thousands of machines, and Cascading updates. The slide is part of a presentation by Smruti R. Sarangi, titled "Google Percolator".

So, the Google's web index is very, very large. So, in a certain sense, it indexes the entire web and the web is very large. In fact, we do not know how large it is, but at least it has millions of websites might be in the billions as well. So, tens of petabytes of data is generated every single day. So, Google has to go through all of this data. And it has to figure out which page points to which other page and also index all the search items, all the search queries, possible search queries, that can be given, all of those words need to be indexed.

There are billions of updates per day. So, also, Google has to keep track of the updates. And there are thousands of machines. And we will see that the main problem with the percolator based system is to deal with cascading updates. So, in the next few slides we will see what these are.

(Refer Slide Time: 01:45)

Motivation  
Design  
Evaluation

Google's Search Algorithm  
Requirements

### 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 **popular** link, then it has a high page rank. ☺
  - If my website points to some website, it will have a very low page rank. ☹

Smriti R. Sarangi Google Percolator 5/32

So, the core of Google's search algorithm is the page rank algorithm. So, every page has a page rank. So, the page rank is a measure of how popular the page is. So, let us say a given page is very, very popular, it has a high page rank, and if a page has low popularity, then it means that it has a low page rank.

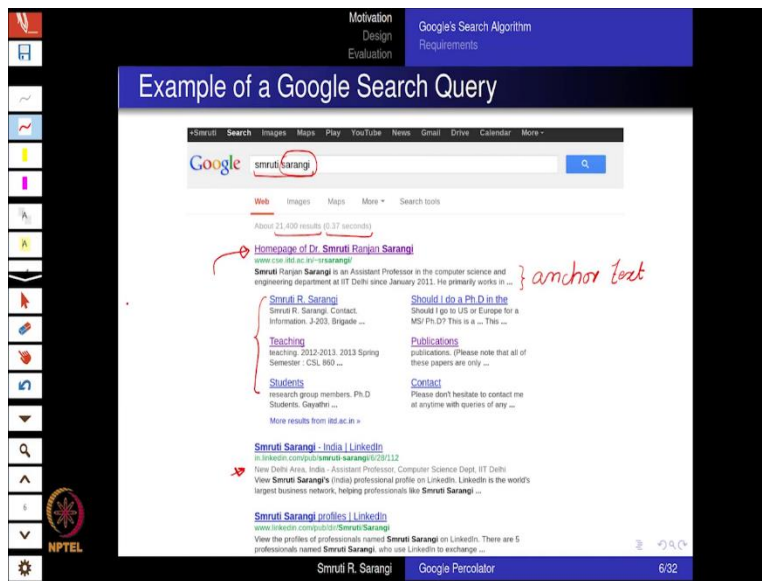
So, the page rank of a page is also determined by the page rank of all the pages that link to it. So, let us say if a lot of popular pages point to one page, then it will automatically mean that this page's page rank should also become high. The reason is that, if extremely popular pages are pointing to a page, so this page also becomes popular kind of by default. So, the page rank of this page increases.

Consider an example. So, assume that a very popular newspaper like the New York Times, it points to some link, then automatically that link is presumed to be important or is presumed to be popular. So, the page rank for this link will also be high. Now, this video is being recorded from IIT Delhi, so, and I am professor Sarangi here. So, if my website points to some other websites, then, which is my official IIT Delhi website, if it points to some other websites since I am not a very important person, my page rank is low. So, I will contribute very little to the page rank of the other page.

And of course, if there are many people like me who point to that other page, then that other page will also have a very, very low page rank.

So, essentially, page rank is like a self serving metric, where popular pages point to a page, then the page rank of that page is high, and so on and so forth. And if less popular pages even if they point to other pages, they do not contribute much when it comes to page rank.

(Refer Slide Time: 04:13)



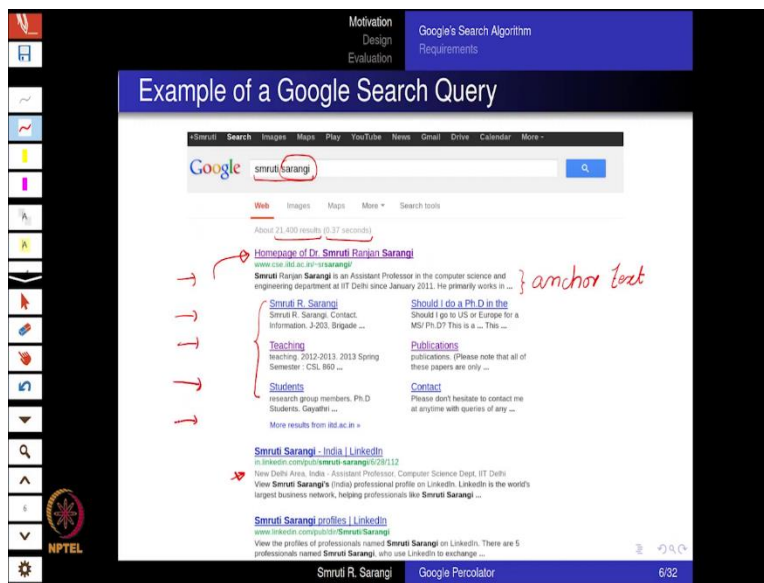
So, if I would, so let us say that I take a Google search page, and I just type in my name. So, what are the things that Google shows? Well, Google shows that there are 21,400 results. So, again, that does not make me very popular. So, actually, Sarangi is the name of an Indian musical instrument. And this is probably majority alliance share of the results more than 99% . And the number of results with my name is very few. Nevertheless, Google took 0.37 seconds.

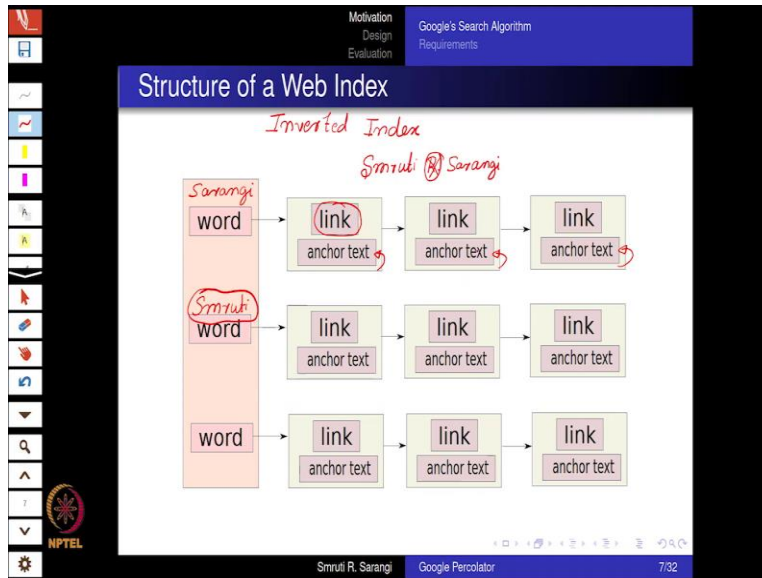
And so let us take a look at what are the results that Google actually got. So, in terms of relevance, this is probably the most relevant, which is my official IIT Delhi homepage. And with every link, Google attaches a piece of text. So, this piece of text over here is known as the anchor text. So, the anchor text helps kind of describe what the link is. Additionally, Google went through my page, and actually went through all the information, all of the links, it went through all the links, and essentially, it is listing the six most popular links along with the anchor text.

After that, after my homepage, Google was able to find my LinkedIn profile. So, it turns out there are many people with my name. So, I think one of them, the first one is me. So, this it has correctly found out. And here also there is a little bit of anchor text. So, these links over here, the search results, are arranged by the page rank, in a descending order of page rank, with the most popular pages coming first and the less popular, less popular, again, determined by the page rank coming later.

So, with this, this is actually a very good idea. It is a very revolutionary idea. And here the main aim is that on the first page itself, the first few links are the most relevant ones. And as I keep going, as I keep scrolling down, the relevance of the links reduces, it decreases.

(Refer Slide Time: 06:39)





So, now how does Google exactly do this? Well, so Google creates a web index. So, this is also called an inverted index. Inverted index, so we have a word, so let us see, consider my last name. So, with the last name, what it does is it maintains a linked list, where of course a linked list is ordered in a descending order of page rank. So, we have the links over here. So, we have multiple links. And along with each link, we have the anchor text.

So, along with each link over here, we have the anchor text as we see. So, if I were to consider my first name, again, we would have a set of links. So, now, if the query is my full name, so actually, my full name, I have a middle initial, which I do not use. So, Google will probably make no sense out of my middle initial. But from my first name, it would find all of these links. From my last name, it will find all of these links. It will take an intersection of these sets. So, again, the intersection will be ordered in the descending order of page rank. And this is how it will generate the search results, which appear over here.

So, this is only a small part of the story. So, this is the story after the inverted index has been created. So, after that searching is actually easy. So, what Google actually does is that Google stores parts of this index in a large number of servers, of course, with redundancy. So, we have been studying many ways of doing that. So, I will not go over it. And whenever I do a search, it quickly locates where my first name would be, it goes to that server, and fetches a list of links. It does the same for my last name, computes an intersection and shows me the results.

(Refer Slide Time: 08:56)

The Problem of Updates

*Crawl the entire web*

- 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**.

*1, 2, 3*

*percolate*

So, the main problem that percolator actually tries to solve, which is by far the most computationally heavy part within Google system, is that if a page rank of a website changes or if the website itself changes in terms of content, then it is necessary to update Google's internal data structures. So, this process is the most computationally intensive. So, what Google used to do prior to percolator is that it used to crawl the entire web.

Crawling the entire web essentially means, so let me write that. Crawling the entire web basically means that going to the most popular websites, looking at their links, visiting those links, so on and so forth such that an entire web is covered, every single website on the web Google goes through it, finds all the other websites it points to, captures all the anchor text and creates the inverted list. So, this needs to be done periodically. This is a pretty heavy operation. And this entire process is known as crawling.

So, what can happen is that if a page rank of a certain website changes, we need to update the inverted list to reflect this change. The page rank of sites that this site points to need to change. So, this is a cascading update. So, what is it? So, let us say this page suddenly becomes more popular, its page rank increases, and this page points to two other pages, even their page rank increases. Then this page again points to, let us say, three other pages. For each of them their page rank increases, so on and so forth. So, this problem is known as a cascading update.

And this is, it is by far a problem because we have millions of web pages changing their page ranks every day. So, Google has to actually work a lot to ensure that its database is in sync with the web. So, prior to percolator what Google used to do is that periodically, let us say, every week or every two weeks, it used to crawl the entire web and create the inverted list. But if I want to make the system more responsive, and let us say within a span of one or two or three days, and maybe for news even less than that, if I want to keep an up to date database, so I cannot crawl the entire web.

What I have to do is that I have to increase the page rank of one page, go to all the pages at points to increase their page rank, so on and so forth. So, of course there, so let us say the page rank increases by 5 percent in arbitrary units, the increase in the page rank for the other pages will be far lower. So, this might be 2 percent. So, gradually, the effect will die down.

But essentially it is important to percolate this change. So, this is where the name percolator comes from, that whenever there is a change, it needs to be percolated to at least some depth. And this used to be expensive. The cost of this was reduced by the percolator project, which essentially percolate such kind of updates.

(Refer Slide Time: 12:37)

The image shows a presentation slide titled "Outline" with a navigation menu. The menu is organized into three main sections:

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

The slide also features a top navigation bar with "Motivation", "Design", and "Evaluation" tabs, and a bottom status bar with the text "Smruti R. Sarangi", "Google Percolator", and "9/32".



Motivation  
Design  
Evaluation

Google's Search Algorithm  
Requirements

### 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 → Need new technology
- Random access to data such that changes can percolate
- Consistency Model: Snapshot Isolation

pre-req

Serializability

NPTEL

Smrutii R. Sarangi | Google Percolator | 10/32

So, let us look at the requirements. So, the requirement that Google had is that it wanted to have a strict ACID transaction semantics. So, here, of course, what is ACID, this is a prerequisite for this lecture. So, we will not go over it. So, any database book or any, so we have also covered it in the previous lecture, any book on distributed systems would have the ACID semantics. So, I would request the viewers to go over that or maybe read the Wikipedia article on ACID semantics before continuing.

So, the reason for the ACID semantics over here is that Google did not want its database to be in an inconsistent state, because poor search results is something that Google was not comfortable with. So, it should have high throughput, needless to say, and an acceptable latency which means that either a search query or an update should not take a very long time. And of course the throughput should be high, because you are talking of a large number of changes should be able to handle petabytes of data. And for this, you clearly cannot use traditional database systems, because they are too slow. They are inefficient.

So, we need random access to data such that the changes can percolate. So, this is also required, because it will not be a sequential scan of the repository. So, it will be one page pointing to a few other pages, which are again pointing to a few other pages. So, this will be random access to data as required. And furthermore, the consistency models, so there are a set of consistency models, serializability, strict serializability, conflict serializable, view serializable.

So, I would suggest that the readers before proceeding forward get an idea of what is serializability which essentially means that all the transactions can be organized in the serial schedule. So, this is serializability. But of course Google did not go for that. So, we will see in a second why. They went for another consistency model which is quite popular for large datasets. It is called snapshot isolation.

(Refer Slide Time: 15:10)

CAP

- 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.

Diagram illustrating Snapshot Isolation in the context of a linked list:

Linked list structure:  $\square \rightarrow a \rightarrow b \rightarrow c \rightarrow \square \rightarrow d$

Transactions and their operations:

- $T_1: a \rightarrow w$  (writes to node a)
- $T_2: a \rightarrow r \rightarrow b \rightarrow c \rightarrow w$  (reads a, reads b, writes to c)

Ordering and conflict analysis:

- $T_1 \rightarrow T_2$
- $T_2 \rightarrow T_1$
- $T_1 \times T_2$  are conflicting

Snapshot Isolation steps:

- Take a snapshot of the data
- Operate on the snapshot

So, snapshot isolation can be explained very well in the context of linked lists. So, consider traditional link list, so let us select this element a, b, c. Now, consider two transactions T1 and T2. So, assume T1 wants to modify the contents of element a, and T2 wants to modify the contents of

element c. So, what would happen in this case is that, so what T1 would actually do, so assume that, in this case, let us say, T2 goes first.

So, if T2 goes first then T2 would essentially traverse the linked list, then it will first traverse a cell. It might read its contents as well, assume it does. Then from a it will go to b, then from b it will go to c. And on c it will do the write. So, what T1 would do after that is after T2 has gone, T1 would basically, from the first cell, it would finally reach a, and on a it would do the write. So, if you think about it, there is a read write conflict over here. Because we are assuming that while traversing the linked list, we are reading each and every element.

And so even if you are not reading the elements, we are at least reading the next pointer. And if this a was to be deleted, the next pointer of this would change. But I mean, let us assume for the sake of simplicity, that the contents of each of these linked list elements are being read. So, one thing is clear that T2 has read a, it has done a read access on a and T1 is doing write access on a. So, between T1 and T2, there is a conflict.

So, T1 and T2 both are conflicting, which means in a traditional transaction system, whenever there is a read write conflict of the same data, then clearly one of them needs to abort. But if we actually think about it, then in this case none of the transactions actually have to abort. Why is this the case?

Well, the case is like this that even though T2 is just scanning through the linked list, it is not really interested in the contents of a, b or anything else, it is only interested in the contents of c where it just has to cross a, b to get to c. Once it gets to c it performs the write and it writes a new value. And a is, and T1 is interested in a and it is changing the contents of a.

So, we can allow T1 and T2 to proceed in parallel. So, they would technically not cause an issue. And the reason is that T2 just wants to cross a and go to the other side. It is not really interested what exactly you do with a and that is exactly why even if T1 e comes before T2 or T1 comes after T2, the results are the same. So, why do we typically have a conflict in a distributed system?

Well, we have a conflict because the order of the transactions matters if T1 comes first and T2 comes later, or if T2 comes first and T1 comes later, then the final outcomes are different. But in this case while updating a linked list as long as the elements are physically different, which in this

case they are, a and c are different, what is happening is that essentially the order of transactions T1 and T2 does not matter. Hence, they can run in parallel.

So, this behavior will typically not be allowed in many database consistency systems. But snapshot isolation will allow. So, if you think about it, what T2 is essentially doing is that before it starts is kind of taking a photograph of the system. It is going to see and making a change over there.

What T1 is doing is it is taking a photograph of the system, it is going to a, and writing its updates over there. And for any transaction that is coming after T1 and T2, so let us say after T2 we have T3, and similarly after T1, we have T3, even if T1 and T2 might not see each other's updates or might see, well, we do not know. So, it does not matter. But T3, if it is coming after both of them, it will see the updates are both T1 and T2. So, this is called snapshot isolation.

So, snapshot isolation would essentially allow this behavior. So, it is also called snapshot isolation semantics. So, what this means is that before execution, before starting, so you take a snapshot, which means you create a full copy, so it is a logical copy, it is not a physical copy. But logically it appears as if you have created a copy, then you operate on the snapshot and you create a new version of it.

Now, for any transaction that is, so if let us say there are multiple concurrent transactions, so let us say there was another T1', and let us say that was changing some other element over here d. So, as far as T1 dash is concerned, it does not care about the updates of T1 and T2, because they are happening to different elements. So, whenever it starts, it will, that semantics is as if it takes a photograph of the entire system, then it traverses to d and makes its changes over there.

And of course, if T3 is coming after T1', it will see the updates of T1' as well. So, this is known as a snapshot isolation semantics, where essentially if we start at some point, just before the transaction starting it appears as if we are taking a photograph of the entire system, then we do the perform the reads and writes on our snapshot. And as long as parallel transactions do not actually modify the same exact piece of data, we are fine. And then once all that parallel transactions end, every subsequent transaction sees all the updates. So, this is snapshot isolation. And this is exactly what Google went for.

Which, of course, this is weaker than many of the known consistency models like serializability, for example. However, this is pretty much what we can afford in a large distributed system, because recall the CAP theorem, there is a trade-off between consistency, availability and tolerance to partitions. So, if we want a higher availability, we need to reduce our consistency requirements, which in a certain sense has been done here.

So, the example that I gave about linked lists is summarized over here, that if the same node or its parent is not being accessed, we can have parallel accesses. And when a transaction starts, it takes a consistent snapshot of the entire database. Then the updates are as if, as if means appear to be updating the private copy of the database and the values are committed if they have not been changed by another transaction since the snapshot. So, it does, snapshot isolation still does not admit concurrent updates to the same item.

(Refer Slide Time: 24:20)

The image shows a presentation slide titled "Outline" with a navigation menu. The menu is organized into three main sections:


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

The slide also features a top navigation bar with the following items: Motivation, Design, Evaluation, Structure, Algorithm, and Details and Optimizations. The bottom of the slide includes the NPTEL logo, the name "Smruti R. Sarangi", the title "Google Percolator", and the slide number "12/32".

Motivation Design Evaluation Structure Algorithm Details and Optimizations

## 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.

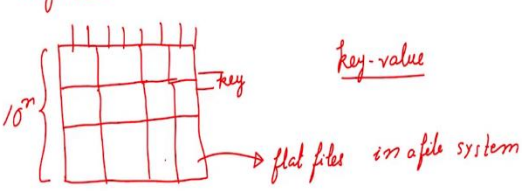


Smruti R. Sarangi Google Percolator 13/32

Windows Journal

File Edit View Insert Actions Tools Help

Big Table



key-value

flat files in a file system

\* Multi-row transactions  
atomicity

\* Observer framework  
→ later

Percolator

Big Table

GFS

reliability

availability

NPTEL

Type here to search

2:08 PM 5/5/2008

Now, the structure, so the structure of percolator is like this that percolator is built on other Google technologies. So, the technology that it is actually built on is called Bigtable and Google File System. So, let us discuss Google Bigtable. So, Google Bigtable is essentially a large multi-dimensional database. So, you can think of as a large 2D or 3D or 4D database. But as far as we are concerned, we will only look at a 2D database.

So, consider a Bigtable to be a huge table, which has maybe like a million entries and each row over here, so it is a row column format, so maybe each row has like 1,000 entries. So, this is huge. So, Bigtable can be treated as a distributed key value store, where a primary key of each row is

essentially the key for that row, a unique identifier, and the value would be the entire contents of the row.

So, of course, here we save the rows, the contents of the row, the contents of the column and a timestamp. So, what Bigtable gives us is that in this large setup, it provides us atomic read modify, write operations for each row. And of course, we can have each row has a large number of columns. It allows us to update all of the columns simultaneously, atomically. And there are separate columns for regular data and metadata. So, this is the basic framework that Google percolator is built on top of.

So, if I were to explain this slightly better, this is how it would look. So, let me go back to our notes over here. So, let us go to a new page. So, essentially, the core technologies that percolator uses are Google Bigtable and Google File System. So, Google File System is a distributed file system on the lines of the coda file system that we have studied. So, I am not discussing that.

So, what we essentially do is that, so the Bigtable, as I said, is a huge table with a very, very large number of rows  $(10)^n$ , where n is a large number. And for each of them also we have a lot of columns. So, Bigtable, so every row is uniquely identified by a key. And value is of course the contents of the row. So, we can think of Bigtable as essentially a key value store. And we can use a regular DHT based mechanism to store the entries.

So, what Bigtable does is that it partitions this table into a set of tablets. So, the tablets can either be row wise or they can be rectangular grids like this, so where you would have multiple rows and columns. So, the Bigtable is pretty much broken down into a set of tablets. And we have dedicated tablet servers whose job is to supply the values of these tablets. Now, the question is, how are these tablets stored?

Well, the tablets are stored as regular flat files in a file system. So, essentially, percolator is built on Bigtable. So, if I were to draw the diagram, Google percolator is built on Bigtable, which is also a Google system. And there is a beautiful paper that describes it. So, what is Bigtable? Well, it is a large multi-dimensional database, you can think of it as a key value store, and large sections of it, which can, so we can divide this in many ways. It can be multiple rows, or it can be the way we have divided it into grids. So, each such grid is known as a tablet.

So, we have dedicated tablet servers that provide these tablets. And each of these tablets is stored as a flat file. And this is stored in the Google File System or the GFS stores. So, it actually stores something called chunks, but each chunk in this case can be thought of as a signal tablet. And this is stored in the Google File System.

And Google File System provides all kinds of guarantees. For example, it provides reliability guarantees, that you will always get it. It provides guarantees of availability that certain set of files which in this case certain tablets are always available. And it has many other useful properties, which is high throughput and low latency and so on.

So, GFS is used to implement Bigtable which is used to implement percolator. And in percolator is, you can think of it as a sophisticated library on top of Bigtable. We will see how. And so before a reader asks, why not just use Bigtable. Well, the reason is that Bigtable is essentially a very flat DHT like system. So, it does not have some of the additional features that percolator requires. So, what are these features that percolator requires?

So, what it requires is it needs multi-row transactions. Why is this? Well, if we update the page rank of one page and then we want to update the page rank of all the pages it points to, they might be on other rows of the Bigtable. So, we would like to update all of them together in one go. There is a multi-row transaction.

Second, so basically this would of course have atomicity as one of its requirements. So, this is what something that comes along with multi-row transactions. The second, we need an observer framework which essentially means that if the page rank of one page changes, automatically we go to all the pages it points to and then we update their pages. So, the question is do we do it immediate or do we do it later?

So, in general, we defer this. Whenever we have the computational bandwidth we do it, which means that there has to be a dedicated process called an observer to look for changes to pages. So, it stores that information somewhere and then it comes back and then does the updates, essentially does the job of percolation. So, let us go back.



So, as we just discussed, any row, any entry has a set of observers, and whenever the data changes, which means the change in content, the change in anchor text, the change in the page rank, specialized functions are run to propagate or percolate these changes.

(Refer Slide Time: 32:08)

Motivation Design Evaluation Structure Algorithm Details and Optimizations

### 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 a 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.

SI -> locks  
↓ timestamps

NPTEL Smruti R. Sarangi Google Percolator 14/32

So, of course, ACID transactions are hard to do in such a large database. So, we do not want to leave the database inconsistent. So, that is the reason we use a timestamp for each data item. And at the beginning of a transaction, the set of timestamps where all the data items that are being modified or accessed, we can think of that as the snapshot of the transaction.

So, in this case, transactions involve multiple rows as we have just discussed. So, there is a need to implement the snapshot isolation. So, that would require both timestamps as well as logs. So, snapshot isolation semantics, to implement it we would need regular logs, along with that we would need timestamps.

(Refer Slide Time: 33:04)

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

So, let us now look at, so what is percolator again. Well, it is an instantiation of Bigtable and in that we add a few extra additional columns. So, we add a log column that contains a pointer to the log, we will see how, the timestamp of committed data, write, the data value, the list of observers and the last time at which a certain observer O ran. So, at this point it is not necessary to memorize the contents of this table. We will be coming back to this table over and over again.

(Refer Slide Time: 33:47)

A transfers B 7₹

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

*committing*

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

So, let us look at a simple example. I was able to get the rupee symbol. And so the Indian rupee symbol has, is there in letech right now. So, it can be use, which I have done. So, now, let us take

a look at a sample transaction where I just transfer money. So, of course, this is not the same as percolator, but this gives you an idea of percolator's logic. So, percolator uses exactly the same logic.

So, at the beginning, so let us say a timestamp 5 or entry B, we have 2 rupees in B's account. And at timestamp 6 we have said that look the final data resides at timestamp 5. So, this 5 points over here. Similarly, A has 10 rupees and we also record the fact that the final data is at timestamp 5 which means A has 10 rupees. So, we are using the key column which is the ID of the agent A or B, the data column that stores the actual contents.

So, instead of rupees over here in the case of percolator it will actually be the contents of the links and anchor texts and so on. And there are log column which we are not using right now, but we will. And a write column that says where does the final value lie, so this is used for committing a transaction. So, this is a very useful gadget which we will use to commit a transaction. So, this is what we also used to differentiate between a temporary write and a permanent write.

So, now what happens is let us assume A would like to transfer 7 rupees to B, so in this case what we do is we create a new timestamp 7, where we decrement 10 to 3. Furthermore, we acquire lock on this row. So, we say that this is a primary lock, because this is, this A is starting the transaction. So, it is the primary. So, we acquire a lock on 7 and we say that 7 is a primary.

But mind you, here is the important point. We do not commit the value. So, we still say that the final data item is at 5, which is 10. 3 is just a temporary write. It is not committed. And this is where this column is useful.

(Refer Slide Time: 36:21)

Example - II

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

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

2-phase commit

Example - III

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

Then what we do it, we acquire a lock again on B by a new timestamp and we also acquire a lock on B's row, but we say that lock the primary owner of the lock is A and because this is a secondary transaction. In this case, we credit the 7 rupees to B, so from 2 it becomes 9. Again, we do not commit the transaction.

Now, what we do is that, we are in a position to commit the transaction because we have acquired the logs. So, this is very similar to 2-phase commit, where we first acquire kind of an initial commitment which is similar to a lock and then we do the final commit. So, this is very, very similar to 2-phase commit.

So, in this case what we do is for the final write, we let go of the primary lock at A and we say that the data is at 7, which means that the final data of A is 3 rupees. And then we come to B and B still has the lock, so then we come at B and we say, so we get rid of B's log. So, if you can see, there is no lock with B again and we say that the final data is at timestamp 7 which is 9 rupees.

So, this completes the process of a transaction. So, what we will do in updating web index is very similar. Instead of money when we store web indices we also acquire lock to multiples rows, we make the changes and then we do the commit, which is a 2-phase commit mechanism which works in this particular manner.

(Refer Slide Time: 38:08)

The image shows a presentation slide with a dark blue header and a white main content area. The header contains the text 'Outline' in white. The main content area is a list of sections with sub-sections:

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

The slide also features a vertical toolbar on the left with various icons, an NPTEL logo at the bottom left, and a footer at the bottom right with the text 'Srnul R. Sarangi', 'Google Percolator', and '19/32'.

Motivation Design Evaluation Structure Algorithm Details and Optimizations

### Algorithm: Begin Transaction

```

Algorithm 1: Begin Transaction
1 startTs ← oracle.getTimeStamp()
2 Set(W):
   writes.push(W)

```

*temporary buffer.*

Smruti R. Sarangi Google Percolator 20/32

Now, the exact algorithm, so the exact algorithm works like this that when we begin a transaction, we assume that there is one Oracle machine in the system. So, Oracle is a Chinese concept. It is an entity that knows everything. Well, in this case, it does not. But what it does is, it provides a timestamp which monotonically increases. So, it is a global timestamp that increases monotonically. So, before I start a transaction I acquire a start transaction timestamp. And then for all my writes, I just push my writes into a temporary buffer. So, there is the writes is a temporary buffer.

(Refer Slide Time: 39:03)

Motivation Design Evaluation Structure Algorithm Details and Optimizations

### Get Method

```

1 Get(row, column, value):
  while True do
2   T ← startTrans(row)
3   if T.hasLock(0, startTs) then
     backOffAndMaybeRemoveLock(row,col)
     continue
4  end
5  latestWrite ← T.read(row, [0, startTs])
6  if !latestWrite then
     return
7  end
8  dataTs ← latestWrite.timeStamp
   return (T.read(row, "data", dataTs)
9 end

```

Smruti R. Sarangi Google Percolator 21/32

So, then what I do is that I, for my get operation, which is getting a row and a column, so for getting the contents of a row and a column what do I do. Well, I start a while loop. And so for the first row I start a transaction. If this transaction is between 0 and startTs which is when I started my transaction, somebody has acquired a lock, so which means there is an outstanding lock, then of course there is a problem and it means this row cannot be locked, because the row is currently locked. So, I back off and maybe remove the lock.

And so in this case so we will discuss why maybe remove the lock, because it is possible that the process that I locked this row as died, so then of course I wait for some time. And still if I get a feeling that maybe the process that should have remove this lock has died, I try to forcibly remove the lock. We will see that this does not cause problems and so I would refer the readers to the paper, the viewers to the paper, where this issue is discussed.

But what is the main idea? The main idea is that nobody should have locked this row. If somebody has, I kind of back off and I try again. And if I get a feeling that this process has died, I try to forcibly remove the lock. Otherwise, I do not. Then I find the latest write, which is I read it. I do a T. read, and I find the latest write, which is for this row between 0 and start Ts what is the latest write.

So, if there is no latest write, I return null, which means that this has not been returned to, otherwise I find the timestamp of the latest write and what do I return. Well, I read the row. So, I return of course the data and so the data that is read along with the timestamp of the data, so both of these I return.

(Refer Slide Time: 41:30)

```
1 PreWrite(Write w, Write primary)
  Column col ← w.col
  T ← startTransaction(w.row)
2 if T.read(w.row, "write", [startTs, ∞]) then
3   return false
4 end
5 if T.read(w.row, "lock", [0, ∞]) then
6   return false
7 end
8 T.write(w.row, "data", startTs, w.value)
   T.write(w.row, "lock", startTs, {primary.row, primary.col})
   return T.commit()
```

Handwritten annotations: Red arrows point from 'parallel write' to the first if-statement. A checkmark is next to 'return false' in the second if-statement. Another checkmark is next to 'return T.commit()' at the end.

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

Handwritten annotations: A red arrow points from the '8: x' entry in the lock column of transaction B to the '8: data @ 7' entry in the write column of transaction B.

So, while writing I have two stages, one is called pre-writing and writing similar to 2-phase commit. So, in this case what I do is I first find the column number that I want to write to. I start a transaction on the row, then for the given row, I see if anybody has already written between startTs and infinity, which means that after I started if there is a parallel write by some other transaction, if there is a parallel write, then I return false.

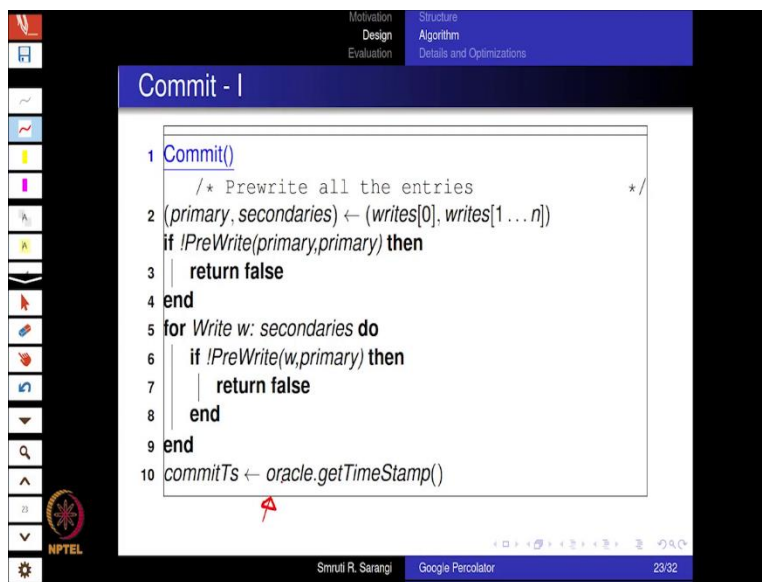
So, recall, so for understanding this, just go back to this slide. So, recall that for every row, we just have a set of data with different timestamps. So, this is exactly what I am doing over here, that after I started my transaction basically within my start and infinity, if there is a parallel write, then



clearly my pre-write will not be successful. So, we return a false. I also see that if anybody has acquired a lock, if there is any outstanding lock, then also I return false, because I will not be able to do the write.

If both of these conditions are false, I exit this function. Otherwise, I remain in this function. And then the data what do I write. Well, for the data field I write the startTs which is my transactions starting timestamp and the value. In addition, I write the lock and the starting timestamp, and also the row and column of the primary, which is the initiating, so one of the rows is the initiator of the transaction. So, I write its ID as the primary lock owner. And then I call the commit function, which is the second phase of 2-phase commit.

(Refer Slide Time: 43:33)



```
1 Commit()
   /* Prewrite all the entries */
2 (primary, secondaries) ← (writes[0], writes[1...n])
3 if !Prewrite(primary, primary) then
4   return false
5 end
6 for Write w: secondaries do
7   if !Prewrite(w, primary) then
8     return false
9   end
10 end
11 commitTs ← oracle.getTimeStamp()
```

So, in commit what I do is I first pre-write all the entries. So, for the both the primary and the secondary, I pre-write all the entries which means I take all the writes and I try to pre-write them. If I cannot write the primary then of course I return false. If I cannot write this any of the secondaries then also I return false. But if all of them work out, then what I do is I get commit timestamp from the Oracle and try to start the commit process.

(Refer Slide Time: 44:13)

Motivation Design Evaluation Structure Algorithm Details and Optimizations

### Commit - II

```
→ /* Commit the primary */
11 T ← startTransaction(primary.row)
    /* Test to see if aborted by somebody else */
12 if !T.read(primary.row, "lock", startTs) then
13   return false
14 end
    /* Write the primary and erase the lock */
15 T.write(primary.row, "write", commitTs, "data@" + startTs)
16 T.erase(primary.row, "lock", commitTs)
    /* Point of commit */
17 if (T.Commit()) then
18   return false
19 end
```

Handwritten annotations: A red line with '54' and '12' is drawn above line 11. A red arrow points to 'return false' in line 13. A red arrow points to 'data@" + startTs' in line 15. A red arrow points to 'T.erase' in line 16. A red circle is drawn around 'T.Commit()' in line 17.

NPTEL

Shruti R. Sarangi Google Percolator 24/32

Motivation Design Evaluation Structure Algorithm Details and Optimizations

### PreWrite

```
1 PreWrite(Write w, Write primary)
  Column col ← w.col
  T ← startTransaction(w.row)
    if T.read(w.row, "write", [startTs, ∞]) then
    2   return false
    3   end
    4   if T.read(w.row, "lock", [0, ∞]) then
    5     return false
    6   end
    7   T.write (w.row, "data", startTs, w.value)
    8   T.write (w.row, "lock", startTs, {primary.row, primary.col})
    return T.commit()
```

Handwritten annotations: 'Big Table' is written in red next to 'startTransaction(w.row)'. A red arrow points to the 'if T.read(w.row, "lock", [0, ∞])' condition in line 5. A red circle is drawn around 'T.commit()' in line 8.

NPTEL

Shruti R. Sarangi Google Percolator 22/32

```
1 Commit()
   /* Prewrite all the entries */
2 (primary, secondaries) ← (writes[0], writes[1...n])
   if !Prewrite(primary, primary) then
3     return false
4 end
5 for Write w: secondaries do
6     if !Prewrite(w, primary) then
7         return false
8     end
9 end
10 commitTs ← oracle.getTimeStamp()
```

Let us now look at the second phase of the commit process which is after all the pre-writes have been performed. So, in this case, we commit the primary first. So, to commit the primary we start a transaction on the row of the primary. So, kindly understand that it is possible that between the start of a transaction and this point, so if I were to draw a line, if this is the start of a transaction and this is the current point which is line number 12, so in between this it is possible that some other transaction might have aborted this transaction. Why would this be the case?

Well, the reason that this would be the case is basically on this slide, where we actually take a lock for a row. So, let us say we take a lock for a row and the current transaction actually holds it for a long time, other transactions might perceive that maybe this certain row, which is the row of the primary, will never get unlocked, because the transaction that was supposed to unlock it has died. So, in this case, we need to check if the transaction is still alive, which means if the transaction has been aborted by somebody else or not.

So, the way we do it is that in the primary row, we check if at the startTs time instant, the lock is still there or not. If it is there, if we can read a lock, then well everything is good. If we cannot read a lock, then it means that the transaction has been aborted. So, we return false. So, we do not proceed with the commit. Otherwise, if the lock is there, this means that the transaction has not been aborted, so we can proceed and commit.

So, what we do is that we write the primary, and erase the lock. So, we write in the primary's row we write and we set the commit timestamp to basically say that the data is at the startTs. So if you

would recall, the way that we were actually committing data is essentially making a record in the commit column, in the final data column, that the data is, the data exists at a given timestamp, and this was the process of committing, if you would go back to the table that we were showing where A transfers money to B. So, the moment we make this entry that the final data resides at timestamp startTs, this is akin to committing, so this is what we do.

So, in a transaction, we commit the data as well as we erase the lock. So, we removed the lock on the primary row and this process is the commit process. So, then what we do is, we do, so we would see many of these expressions at different points. So, I should have maybe explained it. Let me do it right now. So, we are seeing T dot commit over here and we are also seeing another T dot commit over here. So, note that these are Bigtable transactions, these are not percolator transactions.

So, the percolator transaction is for these high level commit and pre-write messages. But these transactions that you see which is start transaction over here and this commit, these are Bigtable transactions. So, essentially, so this the Bigtable takes care of, we do not. So, in this case, if we have written everything that we need to write to percolator, then the Bigtable transaction has to be committed features akin to doing both of these things, setting the write as well as the lock for the same row.

So, recall that Bigtable provides transactional access to single rows. So, in this case, what we are doing is that we are committing the Bigtable transaction T.commit(). And we cannot do it, we return false, else we continue.

(Refer Slide Time: 48:32)

```
19 for Write w: secondaries do
20   write(w.row, "write", commitTs, "data@" + startTs)
21   xerase (w.row, "lock", commitTs)
22 end
23 return true
```

*BigTable → does not support multi-row transactions*

*lock → lock @ primary*

key	data	lock	write
A	7:3	7: primary	7:
	6:	6:	6:data@5
	5:10	5:	5:
B	7:9	7: <u>primary@A</u>	7:
	6:	6:	6:data@5
	5:2	5:	5:

key	data	lock	write
A	8:	8:	8: <u>data @ 7</u>
	7:3	7:	7:
	6:	6:	6:data@5
	5:10	5:	5:
B	7:9	7: <u>primary@A</u>	7:
	6:	6:	6:data@5
	5:2	5:	5:

So, we do the same. So, once the primary has been committed, so there is no question of not finishing the commit. So, for all the secondary rows, so recall that Bigtable, so let me maybe write it. So, recall that Bigtable does not support multi-row transactions, so percolate does. So, what we will do is we will use Bigtable's basic transactional infrastructure for single row transactions, which we saw over here, starting a Bigtable transaction and committing it and also doing the same for secondaries.

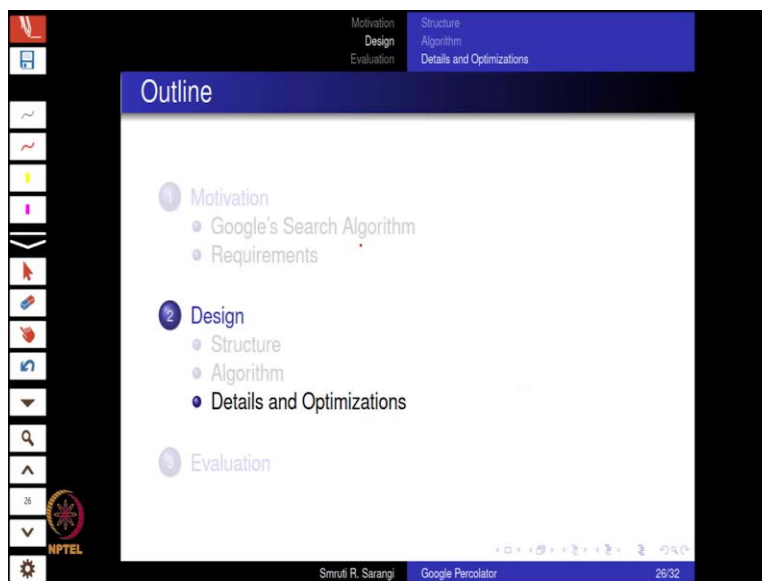
So, in this case, what we do is that for each of the secondaries we write the row, we write the write, and we perform the commit, which says the data is at this point. And furthermore, we erase the

lock that the secondaries would have acquired. So, what do we do? We first look at the primary and then we commit the data of the primary and we then erase the lock. And then we come to the secondaries. And for each of the secondaries, we erase the lock.

So, let us assume that the thread that is doing this fails at some point over here. Well, that is not an issue. The reason that that is not an issue is basically because if we take a look at the lock that a secondary holds, so essentially the lock is of this form, it points to the primary. So, I can give an example with the code we had here. So, if you can see the lock is basically saying that the primary is at A.

So, just in case if any of these threads actually fail in the middle, we will at least get to know that who holds the parent lock. We can go to the primary. If that still holds the lock, we will know that the primary is not done yet. If that does not hold the lock, we will know that the primary is done. From that we can get the details of the transaction and the same can be implemented at the secondary just in case some work is remaining. And after that, we can finish the job of the secondary.

(Refer Slide Time: 51:32)



The screenshot shows a presentation slide titled "Timestamps". At the top, there is a navigation bar with "Motivation", "Design", and "Evaluation" on the left, and "Structure", "Algorithm", and "Details and Optimizations" on the right. The slide content includes a diagram of three boxes connected to a central circle labeled "Oracle" with a red arrow pointing to it. To the right of the diagram, the text "last timestamp" is written in red. Below the diagram is a bulleted list of requirements for the timestamp oracle. The slide also features a vertical toolbar on the left with various navigation icons, the NPTEL logo at the bottom left, and the text "Sriruti R. Sarangi" and "Google Percolator" at the bottom center, along with the slide number "27/32" at the bottom right.

### Timestamps

- The timestamp oracle needs to be able to sustain a 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.

So, a few details and optimizations, well, so the central point for us was the timestamp Oracle, which needs to sustain a very high throughput, because all the nodes essentially ask the Oracle for a timestamp which is a global value, all of them ask it for a timestamp. So, this is of course slow. But one important point that needs to be kept in mind is that latency is not really a big concern for us in the design of percolator, but throughput definitely is.

So, one thing we can do is from the same node, we can batch several RPC calls, not one, but we can create a batch of several and we can send it to the Oracle to reduce the overall network load. And if the Oracle let us say fails, then it needs to recover, but at least in some non-volatile storage, it needs to store the last timestamp that it had issued. So, any subsequent timestamp needs to be more than the timestamp that has been issued in the past.

(Refer Slide Time: 52:45)

**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. *ack-0 > commit-Ts*
  - If the latest timestamp of an observer run (**ack-0**) is less than the commit timestamp of the update, then run the observer.
  - Worker threads avoid **clumping** by scanning random parts of the database.

The diagram shows a vertical rectangle representing an observer. On the left side, there are several horizontal arrows pointing into the rectangle, representing columns being monitored. Inside the rectangle, there are some scribbles and a small circle at the bottom, possibly representing a function or state.

The diagram in the Notepad window shows a horizontal row of five columns. The first column is labeled 'row id'. The second column contains the value 'x0'. Above the second and third columns, the word 'notify' is written. Above the fourth and fifth columns, 'ack-0' is written. A curved arrow labeled 'Observer' points from the 'notify' area to the 'ack-0' area. Below the row, there are four horizontal lines representing a thread. An arrow labeled 'make changes' points from these lines up to the 'x0' value in the second column.

So, the main aim of having a percolator kind of system was to have observers in the sense that if one entry changes, then other entries will also change. So, we want changes to percolate. So, what happens is that each observer registers a set of columns and a function. So maybe this can be shown better by drawing a more elaborate diagram. So, if we take a look at a row in percolator. So, essentially, for every row ID we have a set of columns.

So, it is possible that the data in any of these columns changes. So, if the data changes then it means that the changes have to percolate. So, we can define an observer thread, so an observer thread or an observer process what that does is that that essentially observes a few of these columns.



So, this is an asynchronous process. So, what we do is that whenever we change a row, we have a certain notify column.

So, in notify column we just set a bit from 0 to 1. So, once we set a bit from 0 to 1 in the notify column, what the observer process is going to do is that it will scan the entire percolator database till it finds a 1 in the notify column. So, then based on the columns that have changed, it will go and access a few more rows and make changes over there. It will go and access maybe this row over here, maybe this row over here and it will make changes.

So, just to explain this process once again, whenever there is some change in the database, whenever page rank changes, so the process that changes the page rank accesses a row of percolator, and it need not, so since it is crawling the entire web repository, it need not make changes to other websites immediately. So, what it can do is it can just set 1 in the notify column such that an observer thread later picks it up and makes the changes. Once it has atomically made the changes, it can set the notify bit to 0. And furthermore, it can set up an acknowledgment field that with the timestamp at which the observer ran.

So, the reason we need the timestamp is that we will never have two concurrent observers because it is possible that two observer threads can see the notify bit to be 1 and both can be running concurrently. So, of course, concurrent modifications our system will not allow because of snapshot isolation, one of them will fail. And another quick mechanism of ensuring that we do not have this last work is actually via the acknowledgment field. So, via this field, we will get to know that an observer has already run and it has done its job. So, we will not try to run once again.

So, this is pretty much what is being shown on the slide. Each observer registers a set of columns and a function. The function gets invoked, if any of the columns get updated. So, here what we can do is we can collapse messages. So, it is just to reduce the amount of work. We need not do it for a single change. But for a set of changes we can run the observer once. And for each column, for each commit per column, we will just run one observer transaction.

So, the way that this works is that the worker thread ultimately picks up this information, runs the observer. If the latest timestamp of an observer  $ack\_O > commit\_Ts$  of the update, then it means that we need to run the observer because it would not have seen the update. But if the observer's

timestamp which is ack\_O is greater than the commit timestamp, then it means that it has seen the update and it has done the job.

So, what we actually, what the authors actually found is a moment we have many concurrent observer threads, most of them start clumping in similar areas, which means that there is a lot of contention at different parts of the database, a large parts of it are empty. So, that is the reason what they actually suggested is that different threads can start at random parts of the database and simply start scanning it. Whenever they see a notify bit to be 1, they will start propagating the update, percolating the update. And so this will ensure that a large part of the database gets covered and we do not have a lot of queuing at the same place.

(Refer Slide Time: 58:11)

Motivation Structure  
Design Algorithm  
Evaluation Details and Optimizations

### 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.

Shruti R. Sarangi Google Percolator 29/32

Performance improvement, so basically we need support for read modify write RPCs in Bigtable. So, this is already, this is something that percolator introduced. RPC calls were batched, which means that calls to Bigtable. If there are multiple calls to the same region, then a single RPC call was made. So, recall that an RPC is a remote procedure calls, I will not be discussing this here, because this was discussed in the past, to reduce reads, particularly reads of large parts of the Bigtable tablets, some pre-fetching is done.

So, this reduces the read time and that to vary significantly. So, there are numbers in the paper of how much pre-fetching actually reduces. And also the programming model is simple with a blocking API and a large number of threads. So, the percolator as such is a very simple system to

use, which Google has built primarily for its own applications, where they make some simple modifications to Bigtable and ensure that as compared to their previous system, which we will discuss very shortly, percolator provides a big advantage in terms of gradually evolving the database as the web changes.

So, what are again the main, some of the main Bigtable specific modifications, support for a read modify write RPC, where a single RPC can atomically change the contents of a Bigtable row, create batches of RPC calls, which means that there are multiple writes, instead of sending multiple messages we send a single message, and we employ pretty fetching for reads. So, this improves the performance significantly.

(Refer Slide Time: 60:05)

Motivation  
Design  
Evaluation

### Setup

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

14 days  
↳ map-reduce

Map-reduce

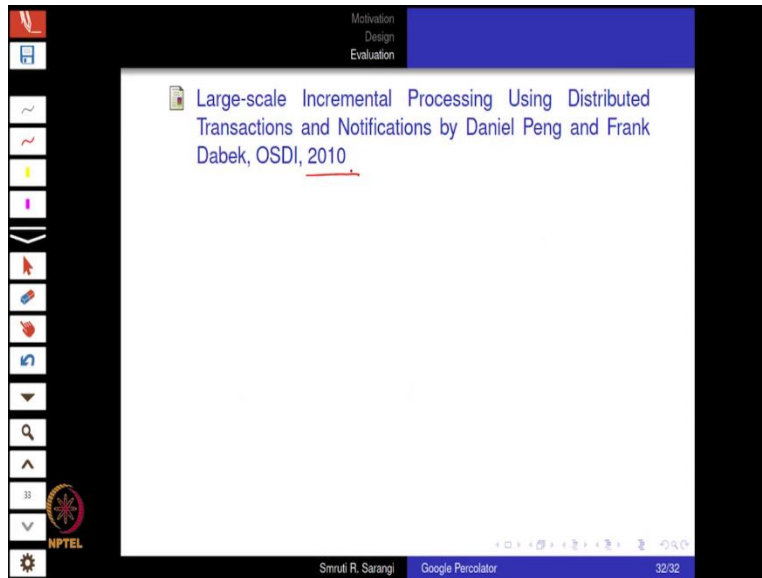
map

reduce

• Do not have to process the entire repo.  
• Process the affected nodes

NPTEL

Smriti R. Sarangi Google Percolator 30/32



So, the setup, well, the setup was that we crawl billions of documents. If you would go to the percolator paper, it was published in 2010, so most of the results that are shown over here are roughly predate 2010. So, I would say maybe 2009. And so in 2009, also, the web was massive, we had billions of documents. So, the default baseline system that actually Google was using was map reduce. So, map reduce is a paradigm that has been there for quite some time. So, I would not be covering the details of it, but readers, viewers are requested to take a look at map reduce elsewhere.

So, there, the basic idea is that we take a large computation, we break it into several small parts, which is the map part, then each of the processing elements does the computation, and finally, all of them are collated into a single point. So, all of them, so we essentially take the different computations and emerge their results and this is called the reduction step.

So, assume that we want to add a million numbers. So, the initial data set is the million numbers. Then we break it down. If let us say we have 100 processes, then we break it down into 100 parts. Each processor gets 10,000 numbers. Then what we do is that we simply add them together. So, this is the reduce step.

So, the map reduce system was fine and it typically required many rounds of map reduce to process the entire web repository, so typically 50 to 100 rounds and it took a long time for a document to finally get indexed. So, what Google used to do those days was that every 14 days, it used to re-crawl the web. And then it used to run a full map reduce job to create an index for the web. So,

this, and creating the index took two or three days. So, I recall way back in 2006-2007, once a website was created, it used to take more than 15 days for it to actually appear in the search results.

So, this period got reduced substantially by the percolator-based system per caffeine which is 100 times faster. And so in the percolator system the main advantage is that you do not have to basically process the entire, we do not have to process the entire repository. This is something map reduce had to do. So, we basically just process the affected nodes, all the nodes that are affected. So, let us say if a website changes its page rank, then all the other sites which are affected by this only those sites we need to change their data.

So, essentially, the time it takes is proportional to the number of websites whose page rank or content or some attribute changes. So, that is why it is much faster and also the average age of documents in the entire search results that get reduced by 50 percent. So, we see far newer documents.

(Refer Slide Time: 63:55)

Motivation  
Design  
Evaluation

### Performance vs Crawl Rate

- Crawl rate → Percentage of repository that is updated per hour.
- Let us plot the clustering latency (y axis) vs the crawl rate (x axis)
- For Map-reduce it starts at 2200s and rises to infinity when the crawl rate exceeds 33%.
- For Percolator it remains below 200s till about 37%. Then it continues to rise.

*Handwritten notes:*  
A red scribble is above the second bullet point.  
An arrow points from the text "3 minutes : 20 secs" to the "200s" in the fourth bullet point.

NPTEL  
Smruti R. Sarangi Google Percolator 31/32

Let us know evaluate the performance. So, the crawl rate is defined as the percentage of the repository that is updated per hour. So, let us plot look at the clustering latency. So, the time it takes to cluster similar documents, duplicates of documents, or the time it takes to cluster different documents which have the same content. So, I said, given the content, we can then take all the documents in the cluster and arrange them according to their page ranks. So, this clustering latency,

if that is the y-axis and the crawl rate is x-axis, we will find that for map reduce it starts at 2200 seconds and rises to infinity when the crawl rate exceeds 33 %.

So, even for a baseline map-reduce with a very low crawl rate percentage it kind of starts at 2200 seconds and it gradually rises to infinity, which means that after 33 % for map-reduce the graph would kind of go like that. But for percolator as long as the crawl rate is limited to 37 %, it actually remains very close to 0. It remains roughly around 200 seconds, which is only 3 minutes. So, 3 minutes is kind of negligible in the web scale. This is 3 minutes and 20 seconds. So, this is genuinely very small. So, till around 37 % it remains the same and after that percolator is not able to manage.

So, similar to map-reduce, the time that it also takes kind of explodes. So, this is a very important and useful statistics. So, we will typically never go beyond 30 %. So, we will never update an entire repository, we will never update more than 30 % of the repository per hour. So, this is like saying that 30 % of the entire worldwide web is changing per hour, which is not the case, so which means that most of the time will remain in the safe zone. And in the safe zone, as we can see, percolator is roughly 10 times faster, if not more than a comparable map-reduce job.

And of course, map-reduce has many, many more overheads in terms of multiple iterations and so on and percolator does not have that, mainly because it only tracks those entries, those rows that are affected.

(Refer Slide Time: 66:35)

Motivation  
Design  
Evaluation

### Scalability for TPC/E benchmarks

- The transactions per second (TPS) varies linearly as we scale the number of cores.
- 4000 TPS is achieved with 5,000 cores.
- It increases to 12,000 TPS for 15,000 cores.

Close to Linear Scaling

*sub-linear*  
*→ queuing, blocking*

NPTEL  
Sriruti R. Sarangi Google Percolator 32/32

So, similar experiments were done with the popular TPC benchmarks for measuring the transactions per second. So, this varies linearly, of course, as they scale the number of cores, which is always a good thing. Linear scaling is always very, very, a piece of very good news, which means that we do not have any bottlenecks. So, this was the case. So, 4000 transactions per second were achieved at 5000 cores, and this increase to roughly 12,000 for 15,000 cores.

So, we saw we see a linear scaling, which means that, so when do we not see linear scaling like when do we see sub-linear scaling. So, note we will never see super-linear, but we will see sub-linear particularly when there is some amount of queuing, delay or there is some kind of a blocking delay.

So, there is essentially some kind of a place where there is some kind of a packet jam. A lot of network packets are not able to flow through a given router. If that is the case, we will find that the network is not scaling. So, the moment it scales linearly with the number of cores, so we are certain that there are no bottlenecks in the system.

(Refer Slide Time: 67:53)

Motivation  
Design  
Evaluation

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

DHT } DBMS  
Percolator

NPTEL

Sriruti R. Sarangi Google Percolator 32/32

So, this was the description of our percolator system, the Google percolator system, which as we can see differs, so it is kind of somewhere between a DHT, of course, DHTs have very little structure so to speak, other than the fact that they are just raw key value stores. So, it lies somewhere between a DHT and it lies somewhere between a traditional DBMS. So, in the traditional DBMS, there is a massive amount of structure, you have tables, you have relations between tables, you have primary keys, you have foreign keys, you have a lot of relations.

But in this case, we have a single Bigtable. We just have relations between rows. And the Bigtable itself is kind of structured as a DHT, but percolator does not get to see most of that. So, it was, it is somewhere in the middle of the spectrum, which is kind of a good thing, because we get the scalability of DHTs and we get the ACID semantics of traditional DBMS systems.