Web: Challenges and Demands

- Emergence of web services
- Document searching
  - More similar to Information Retrieval than DB Querying
  - “Vertical” searching?
- Unprecedented scale
- “Deep Web”
- Data Integration
  - Schema discovery; schema mapping
  - Federated query processing
Web Services

- Tiered architecture
  - Web servers interact with clients – serve static pages
  - DBMS Backend
  - App servers between web servers and DBMS
    - Access the database, maintain the client state
    - Create dynamic webpages etc.

- Why?

- Much work on languages for specifying the functionalities, for putting them together etc.

- Aside: HTTP protocol
Web Searching

- Goal: Answering keyword queries over the WWW
- Main Components:
  - Crawling and Indexing
  - Ranking the documents
  - Data structures for efficient execution
  - Load balancing/dealing with scale
- Why not use DB for this?
- Eric Brewer: "Search engines should use DBMS primitives/abstractions, but not SQL/RDBMS"
  - Top-down Design
  - Data independence
  - Declarative languages
Many different algorithms for doing this

From the Brewer paper

Given a single document by itself, \( d \) and a query
\( Q = \{[w_1, \cdots, w_k]\} \):

\[
\text{score}(Q, d) = \sum \text{score}(w_i, d)
\]

\[
\text{score}(w_i, d) = \frac{1}{M^e} \left( 0.679 \times \log(\text{Freq}(w_i, d)) + 0.223 \times \log(\text{IDF}(w_i)) \right)
\]

- \( \text{freq}(w_i, d) \) = how many times \( w_i \) appear in \( d \)
- \( \text{IDF}(w_i) = 1/\text{fraction of documents in which the word appears} \)

Web search engines also use “relative positions”
Document Ranking

- Not good enough - must incorporate additional knowledge
  - \( \text{Quality}(d) = \) How important is the document?
  - Google uses “pagerank”
  - Kleinberg et al. proposed the HITS algorithm around the same time
  - Both use the “incoming links” to determine the quality
    - Intuitively, if a large number of high-quality sites link to \( x \), then \( x \) is high quality as well
  - Note: The algorithms are executed offline
  - Much more sophisticated now

- Total Score: Word score + Quality
Efficient Execution

- Originally built in an ad hoc manner
- Brewer: Should have been designed using DB principles

Schema:
- Document(DocID, URL, Date, Size, Abstract) (3B rows)
- WordTable(WordID, DocID, Score, Position Info) (1T)
- Property(WordID, DocID) (100 billion)
- Terms(String, WordID, Stats) (10 million)

A query contains:
- Properties that must be satisfied
- Keywords that should be used to rank the documents
- Arbitrary ANDs, ORs, or NOTs of these
Web Searching

Document table, $D$, about 3B rows

<table>
<thead>
<tr>
<th>DocId</th>
<th>URL</th>
<th>Date</th>
<th>Size</th>
<th>Abstract</th>
</tr>
</thead>
</table>

Word table, about 1T rows:

<table>
<thead>
<tr>
<th>WordID</th>
<th>DocId</th>
<th>Score</th>
<th>Position Info</th>
</tr>
</thead>
</table>

Property table, about 100B rows:

<table>
<thead>
<tr>
<th>WordID</th>
<th>DocId</th>
</tr>
</thead>
</table>

Term table, $T$, about 10M rows:

<table>
<thead>
<tr>
<th>String</th>
<th>WordID</th>
<th>Stats</th>
</tr>
</thead>
</table>

Figure 1: Basic Schema

Result Set = [DocId, Score, URL, Date, Size, Abstract]

$D$

Top($k$, Score)

$D$

score = Quality($d$)

$\sum_i \text{Score}(w_i, d)$

Matching documents:

score = $\sum_i \text{Score}(w_i, d)$

Figure 2: The General Query Plan

After finding the set of matching documents and their scores, the Top operator passes up the top $k$ results (in order) to an equijoin that adds in the document information.
Efficient Execution

- Key data structure
  - For each term or property, a list of document IDs that match
  - In sorted order (by Document ID)
  - Called an “inverted index”

- So, the key algorithm to use:
  - Sort-merge join

- Plan for an all-ANDs query:
  - Find the lists for each of the terms and properties
  - Sort-merge them together simultaneously

- Similarly, ORs and NOTs (treated as properties)
Efficient Execution

- Flattening the query
  - Shallow queries require fewer steps
- Caches
  - Critical, over-riding difference
  - Every result computed during the execution is cached
  - Analogous database question: How to use materialized views
  - Favors a top-down approach for planning
    - Find the largest parts of the query that are already computed
    - Several tricks for handling near-matches
Parallelization is a key
- Google has 100,000s of disks and machines

Simple approach
- Partition the large tables by DocIDs
- Small tables (like Terms) are replicated
- Execute the same query plan on all of them
- A master node chooses the query plan

Fault Tolerance?
- Replication-based
Aside: CAP

- Databases ensure ACID (atomicity, consistency, isolation, durability)
- Web search engines don’t care about most of these
- CAP Theorem:
  - Choose two of *consistency*, *availability*, and *tolerance to partitions*
- Databases choose C & P
- Web search engines choose A & P
- BASE? Basically Available, Soft-state, Eventually consistent
10x difference (probably much larger now)

Several reasons why custom-built search engine better

- No locking
- Single hand-optimized query plan
- Multi-way joins (instead of binary joins)
- Extensive compression
- Aggressive caching
- Careful data representation
- Hand-written access methods
- Single address space
- Not security or access control
Google’s solution to the same problem

Goal: efficient parallelization of various tasks across 1000’s of machines without the user having to worry about the details such as:

- How to parallelize
- How to distribute the data
- How to handle failures
Users needs to write two key functions:

- **Map**: generate a set of (key, value) pairs
- **Reduce**: group the pairs by key’s and combine them (GROUP BY)

**Borrowed from Lisp**

```java
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```
Mapreduce: Execution Overview

User Program

Master

(1) fork

worker

(1) fork

worker

(1) fork

(2)

assign

map

(2)

assign

reduce

split ...

are distributed across multiple machines by automatically partitioning the input data

To appear in OSDI 2004

3

Figure 1: Execution overview
A master for each task, assigns tasks to workers

- Data transfers using the file system (by passing file-names)
- Master pings the workers to make sure they are alive
  - If not, reassign the task to some other worker
- Work is divided into a large number of small chunks
  - Similar ideas used in parallel database for handling data skew
- Atomic commits using the file system
Google File System
- A distributed, fault-tolerant file system
- Data divided into blocks of 64MB
- Each block stored on several machines (typically 3)

Mapreduce uses the location information to assign work

Many other optimizations
- Backup tasks to handle “straggler”
- Control over partitioning functions
- Ability to skip “bad” records
Mapreduce

- Has been used within Google for:
  - Large-scale machine learning problems
  - Clustering problems for Google News etc..
  - Generating summary reports
  - Large-scale graph computations

- Also replaced the original tools for large-scale indexing
  - ie., generating the inverted indexes etc.
  - runs as a sequence of 5 to 10 Mapreduce operations
Mapreduce: Thoughts

- Abstract ideas have been known before
  - See [Mapreduce: A Major Step Backwards](#); DeWitt and Stonebraker
  - Can be implemented using user-defined aggregates in PostgreSQL quite easily
  - Top-down, declarative design
    - The user specifies what is to be done, not how many machines to use etc...

- The strength comes from simplicity and ease of use
  - No database system can come close to the performance of Mapreduce infrastructure
  - RDBMSs can’t scale to that degree, are not as fault-tolerant etc...
    - Again: this is mainly because of ACID
    - Databases were designed to support it
    - Most of the Google tasks don’t worry about that
Mapreduce: Thoughts

- Mapreduce is very good at what it was designed for
  - But may not be ideal for more complex tasks
    - E.g. no notion of “Query Optimization” (in particular, operator order optimization)
    - The sequence of Mapreduce tasks makes it procedural within a single machine
  - Joins are tricky to do
    - Mapreduce assumes a single input

- Trying to force use of Mapreduce may not be the best option

- However, much work in recent years on extending the functionality
  - See Pig project at Yahoo, Map-reduce-merge etc..
Pig Project @ Yahoo

- Generalization of Mapreduce
- An IDE for developing large-scale data analysis tasks
- Appears a simplification of SQL (at this point at least)

Figure 4: Pig Pen screenshot; displayed program finds users who tend to visit high-pagerank pages.

Nevertheless, there still remain cases where `COGROUP` is followed by something other than an algebraic UDF, e.g., the program in Example 3.5, where `distributeRevenue` is not algebraic. To cope with these cases, our implementation allows for nested bags to spill to disk. Our disk-resident bag implementation comes with database-style external sort algorithms to do operations such as sorting and duplicate elimination of the nested bags (recall Section 3.7).

5. DEBUGGING ENVIRONMENT

The process of constructing a Pig Latin program is typically an iterative one: The user makes an initial stab at writing a program, submits it to the system for execution, and inspects the output to determine whether the program had the intended effect. If not, the user revises the program and repeats this process. If programs take a long time to execute (e.g., because the data is large), this process can be inefficient.

To avoid this inefficiency, users often create a side data set consisting of a small sample of the original one, for experimentation. Unfortunately this method does not always work well. As a simple example, suppose the program performs an equijoin of tables `A(x,y)` and `B(x,z)` on attribute `x`. If the original data contains many distinct values for `x`, then it is unlikely that a small sample of `A` and a small sample of `B` will contain any matching `x` values [3]. Hence the join over the sample data set may well produce an empty result, even if the program is correct. Similarly, a program with a selective filter executed on a sample data set may produce an empty result. In general it can be difficult to test the semantics of a program over a sample data set.

Pig comes with a debugging environment called Pig Pen, which creates a side data set automatically, and in a manner that avoids the problems outlined in the previous paragraph. To avoid these problems successfully, the side data set must be tailored to the particular user program at hand. We refer to this dynamically-constructed side data set as a sandbox data set; we briefly describe how it is created in Section 5.1.

Pig Pen’s user interface consists of a two-panel window as shown in Figure 4. The left-hand panel is where the user enters her Pig Latin commands. The right-hand panel is populated automatically, and shows the effect of the user’s program on the sandbox data set. In particular, the intermediate bag produced by each Pig Latin command is displayed.

Suppose we have two data sets: a log of page visits, `visits: (user, url, time)` and a catalog of pages and their pageranks, `pages: (url, pagerank)`. The program shown in Figure 4 finds web surfers who tend to visit high-pagerank pages. The program joins the two data sets after first running the log entries through a UDF that converts urls to a canonical form. After the join, the program groups tuples by user, computes the average pagerank for each user, and then filters users by average pagerank.

The right-hand panel of Figure 4 shows a sandbox data set, and how it is transformed by each successive command. The main semantics of each command are illustrated via the sandbox data set: We see that the `JOIN` command matches `visits` tuples with `pages` tuples on `url`. We also see that grouping by `user` creates one tuple per group, possibly containing multiple nested tuples as in the case of Amy. Lastly we see that the `FOREACH` command eliminates the nesting via aggregation, and that the `FILTER` command eliminates Fred, whose average pagerank is too low.

If one or more commands had been written incorrectly, e.g., if the user had forgotten to include `group` following `FOREACH`, the problem would be apparent in the right-hand panel. Similarly, if the program contains UDFs (as is common among real Pig users), the right-hand panel indicates whether the correct UDF is being applied, and whether it...
Mapreduce: Thoughts

- **Hadoop**
  - Open-source implementation of Mapreduce
  - Has support for both the distributed file system and Mapreduce
  - University of Maryland is a major player in this
    - Jimmy Lin is running several projects related to NLP
    - If you want to play with this, let me know
  - IBM, Yahoo, other major players interested in this

- **Cloud Computing**
  - Somewhat vague term, but quite related
  - Talk tomorrow if you want to learn more
    - **Inside the Cloud**: Challenges and Solutions for Internet-Scale Computing (Eugene Hung, IBM): 4/30
      - 11am AVW 2120