PART 1. LARGE SCALE DATA ANALYSIS USING MAPREDUCE

Instructor: Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs435

Today’s topics
- FAQs
- Evaluation/Validation Techniques
- NoSQL Storage

FAQs
- PA2
  - Extended Deadline: March 25th (Friday) 5:00PM to March 27th (Sunday) 5:00PM via Canvas
- TP1
  - Extended Deadline: March 28th (Monday) to March 29th (Tuesday)
- Help session for PA2 has been posted
  - Please check the assignment page

Why validation? (1/2)
- Process for model selection and performance estimation
- Model selection (fitting the model)
  - Most of the models have one or more free parameters
  \[ h_\theta(x) = \theta_0 + \theta_1 x_1 \]
  You should find these values to generate (fit) your model
- How do we select the “optimal” parameter(s) or model for a given classification problem?

Why validation? (2/2)
- Performance estimation
  - Once we have chosen a model, how do we estimate its performance?
- Performance is typically measured by the true error rate
  - the classifier’s error rate on the entire population

Validation techniques
### Challenges (1/2)
- If we had access to an unlimited number of examples these questions have a straightforward answer
  - Choose the model that provides the lowest error rate on the entire population
  - Of course, that error rate is the true error rate
- In real applications we only have access to a subset of examples, usually smaller than we wanted
  - What if we use the entire available data to fit our model and estimate the error rate?
    - The final model will normally overfit the training data
    - We already used the test dataset to train the data

### Challenges (2/2)
- This problem is more pronounced with models that have a large number of parameters
  - The error rate estimate will be overly optimistic (lower than the true error rate)
  - In fact, it is not uncommon to have 100% correct classification on training data
- A much better approach is to split the training data into disjoint subsets: the holdout method

### The Holdout Method
- Split dataset into two groups
  - Training set
    - Used to train the model
    - Used to estimate the error rate of the trained model
  - Test set
    - Simplest cross validation method

### Drawbacks of the holdout method
- Drawbacks
  - For a sparse dataset, we may not be able to set aside a portion of the dataset for testing
  - Based on the where “split” happens, the estimate of error can be misleading
    - Sample might not be representative
- The limitations of the holdout can be overcome with a family of resampling methods
  - More computational expense
  - Stratified sampling
  - Cross Validation
    - Random subsampling
    - K-Fold cross validation
    - Leave-one-out Cross-Validation

### Validation techniques
1. The Holdout method

### Validation techniques
2. Random Subsampling
Random Subsampling
- $K$ data splits of the dataset
- Each split randomly selects a (fixed) number of examples without replacement
- For each data split, retain the classifier from scratch with the training examples and estimate $E_i$ with the test examples.

Experiment 1
Experiment 2
Experiment 3
Total number of examples

True Error Estimate
- The true error estimate is obtained as the average of the separate estimates $E_i$. 
  \[ E = \frac{1}{K} \sum_{i=1}^{K} E_i \]
- This estimate is significantly better than the holdout estimate.

$k$-Fold Cross-validation
- Create a $k$-fold partition of the dataset
- For each of the $k$ experiments, use $K-1$ folds for training and the remaining one for testing.

Experiment 1
Experiment 2
Experiment 3
Experiment 4
Total number of examples

True error estimate
- $k$-fold cross validation is similar to random subsampling
- The advantage of $k$-Fold Cross validation
  - All the examples in the dataset are eventually used for both training and testing.
  - The true error is estimated as the average error rate.
  \[ E = \frac{1}{K} \sum_{i=1}^{K} E_i \]
Leave-one-out Cross Validation

- Leave-one-out is the degenerate case of $k$-Fold Cross validation
- $k$ is chosen as the total number of examples
- For a dataset with $N$ examples, perform $N$ experiments
  - Use $N-1$ examples for training, the remaining example for testing

<table>
<thead>
<tr>
<th>Single Test example</th>
<th>Experiment 1</th>
<th>Experiment 2</th>
<th>\vdots</th>
<th>Experiment N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total number of examples

True error estimate

- The average error rate on test examples
  \[ E = \frac{1}{N} \sum_{i=1}^{N} E_i \]

- Is it suitable for large data samples?
  - No
  - Very computing intensive

How many folds are needed? (0/2)

- With a large number of folds
  - The bias of the true error rate estimator? (Large/Small)
  - The variance of the estimator? (Large/Small)
  - The computational time? (Large/Small)

- With a small number of folds
  - The number of experiments? (Large/Small)
  - The variance of the estimator? (Large/Small)
  - The bias of the estimator? (Large/Small)

How many folds are needed? (1/2)

- With a large number of folds
  - The bias of the true error rate estimator will be small
  - The estimate will be very accurate
  - The computational time will be very large
  - Many experiments

- With a small number of folds
  - The number of experiments are low
  - Computation time is reduced
  - The variance of the estimator will be small
  - The bias of the estimator will be large

How many folds are needed? (2/2)

- The choice of the number of folds depends on the size of the dataset
  - For large datasets, even 3-Fold Cross Validation will be quite accurate
  - For very sparse datasets, you may have to consider leave-one-out
    - To get maximum number of experiments

- A common choice for $k$-fold Cross Validation is $k=10$

Three-way data splits

- If model selection and true error estimates are computed simultaneously
  - The data needs to be divided into three disjoint sets
  - Training set
    - E.g. to find the optimal weights
    - Validation set
      - A set of examples used to tune the parameters of a model
      - To find the “optimal” number of hidden units or determine a stopping point for the backpropagation algorithm
    - Test set
      - Used only to assess the performance of a fully-trained model

- After assessing the final model with the test set, you must not further tune the model
Validating Classifiers

Classification

Plain Accuracy
- Classifier accuracy
  - General measure of classifier performance
  - Accuracy = (Number of correct decisions made) / (Total number of decision made)
  - 1 − error rate
- Pros
  - Very easy to measure
- Cons
  - Cannot consider realistic cases

The Confusion Matrix
- A type of contingency table
- A n x n matrix
  - The columns labeled with actual classes
  - The rows with predicted classes
- Separates out the decisions made by the classifier
  - How one class is being confused for another
  - Different sorts of errors may be dealt with separately

<table>
<thead>
<tr>
<th></th>
<th>p (positive)</th>
<th>n (negative)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y (predicted)</td>
<td>True positive</td>
<td>False positive</td>
</tr>
<tr>
<td>N (predicted)</td>
<td>False negative</td>
<td>True negative</td>
</tr>
</tbody>
</table>

Problems with Unbalanced Classes
- Consider a classification problem where one class is rare
  - Sifting through a large population of normal entities to find a relatively small
    number of unusual ones
  - Looking for defrauded customers, or defective parts
  - The class distribution is unbalanced or skewed

<table>
<thead>
<tr>
<th>Confusion Matrix of A</th>
<th>Confusion Matrix of B</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>n</td>
</tr>
<tr>
<td>Y</td>
<td>500</td>
</tr>
<tr>
<td>N</td>
<td>0</td>
</tr>
</tbody>
</table>

Which model is better?

Why accuracy is misleading
Which model is better?
**F-measure (F1 score)**

- Summarizes confusion matrix
  
<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>TP</td>
<td>FN</td>
</tr>
<tr>
<td>F</td>
<td>FP</td>
<td>TN</td>
</tr>
</tbody>
</table>

- True positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN)

- True positive rate (sensitivity) = \( \frac{TP}{TP+FN} \)
- False negative rate (miss rate) = \( \frac{FN}{TP+FN} \)

- F-measure = \( 2 \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \)
  - \( \text{precision} = \frac{TP}{TP+FP} \)
  - \( \text{recall} = \frac{TP}{TP+FN} \)

- Accuracy = \( \frac{TP + TN}{P + N} \)

---

**PART 2. DATA STORAGE AND FLOW MANAGEMENT**

This material is built based on,

- Pramod Sadalage, Martin Fowler, "NoSQL Distilled: A brief guide to the emerging world of polyglot persistent", Addison-Wesley, 2012


---

**Data Storage and Flow Management**

***NoSQL Storage***

---

**Before and After the analysis,**

- We need efficient data retrieval mechanism to store and retrieve the results of analysis

- What do we have?

---

**Why RDBMS? (1/2)**

- Manage large amounts of persistent data
  - Provides more flexible access to the data stored in the backing store

- Concurrency
  - Helps handle concurrency by controlling all access to data through transactions
  - Transactions is also important for the error handling

- Integration
  - Shared database integration
    - Multiple applications store their data in a single database
Why RDBMS? (2/2)
- A (Mostly) Standard Model
- Core mechanisms are accepted widely
- Query style (SQL)
- Transactional operations

Impedance Mismatch (1/2)
- The difference between the relational data model and the in-memory data structures
- The relational data model organizes data into a structure of tables and rows
  - Relations and tuples
  - Tuple
    - A set of name-value pairs
  - Relation
    - A set of tuples
- All operations in SQL consume and return “relations”

Impedance Mismatch (2/2)
- Relational algebra
  - Provides an elegant and simple way to interact with data
- Tuple cannot contain any structure
  - Nested record or list
- In-memory data structures
  - Richer structures than relations
  - Translation to a relational representation needed
  - To the different representations

Object-oriented DB
- 1990s
  - Some folks believed that it would replace RDBMS
- Not easy to integrate data

Application and integration with DB
- Integration mechanism between applications
  - Storing their data in a common database
  - Applications operate on a consistent set of persistent data
- Downsides
  - Structure becomes more and more complex
  - Hard to preserve database integrity
  - Database cannot trust applications
## Service-Oriented Architecture: Interoperability

- Requires better interaction protocols between applications and database
- Shift to web services as an integration mechanism
  - Requires more flexibility for the structure of the data being exchanged
  - XML (or JSON) provides a richer data structure than SQL

## Attack of the Clusters: early 2000s

- **Data Deluge**
  - Large sets of data appeared
    - Links, social networks, activity in logs, mapping data
  - We have two choices
    - **UP** or **OUT**

### Scale Up
- Bigger machines, more processors, disk storage, and memory

### Scale Out
- Lots of small machines in a cluster
  - Commodity hardware

## RDMBS for clusters?

- Relational databases are **not designed to be run on clusters**
- Clusters of RDMBS
  - e.g. Oracle RAC or MS SQL Server
  - Work on the concept of a shared disk subsystem
  - A cluster-aware file system
  - Cluster still has the disk subsystem as a single point of failure
    - Data Sharding
      - RDBMS run as separate servers for different sets of data
  - Running on a cluster
    - Raised prices

## Inspiring approaches

- Mismatch between relational database and clusters
- Challenges for Amazon and Google
  - Running large clusters
- Voluminous datasets

### BigTable from Google

### Dynamo from Amazon

## Emergence of NoSQL

- Hadoop summit, 2009, San Francisco
  - Johan Oskarsson
  - The name of meeting was NoSQL
  - Cassandra, Dynomite, Hbase, Hypertable, CouchDB, and MongoDB...

## NoSQL databases

- **Basic Idea**
  - Operates without a schema
  - Allows users to add fields without having to define any changes in structure first
  - Useful when dealing with nonuniform data and custom fields

- **Stands for “Not Only SQL”**
- Handles data access with **size and performance** that demand a cluster
- Improves the productivity of application development by using a more convenient data interaction style
Polyglot persistence

- Using different data stores in **different circumstances**
  - Without picking a particular database for all situations
- Most organizations have a mix of data storage technologies for different circumstances

Key-Value Store

- Simple **hash table**
  - All access to the storage is via primary key
    - Get the value for the key
    - Put a value for a key
    - Delete a key
    - Add a key
- "value" is stored as a **blob**
  - Without caring or knowing what’s inside
  - Application is responsible for understanding data

Suitable use cases

- Storing session information
- User profiles, preferences
- Shopping cart data

When Not to Use

- Relationships between data
- Multi-operation transactions
- Query by data
  - There is no way to inspect the value on the server side
  - Exceptions
    - Lucene, Solr, and Galileo
**Document Storage Model**

- Documents
- Self-describing
- Data structure
  - Maps, collections, tree, and scalar values
  - Stores documents in the value part of the key-value store
- MongoDB, CouchDB, OrientDB, RavenDB, etc.
- Users can query the data **inside the document**
  - without having to retrieve the whole document

**Suitable Use Cases**

- Event logging
- Content management system, blogging platforms
- Web analytics or real-time analytics

**When Not to Use**

- Complex transactions spanning different operations
- Queries against varying aggregate structure

**Column Family Stores**

- Cassandra, Hbase, Hypertable, and Amazon SimpleDB
- Stores data in column family as rows
  - Have many columns associated with a row key
- Column families
  - Groups of related data that is often accessed together
This material is built based on,


Column-family storage

- Optimized for the data
  - Sparse columns and no schema

- Aggregate-oriented storage
  - Most data interaction is done with the same aggregate
  - Aggregate
    - A collection of data that we interact with as a unit

- Stores groups of columns (column families) together

Storing data in a column-family store

- The stores organize their columns into column families
- Each column may be part of a single column family
- The column acts as unit for access
  - The assumption is that data for a particular column family will be usually accessed together

BigTable

- Google’s first answer to the question
  - “How do you store semi-structured data at scale?”

Scalability and latency

- Scale in capacity
  - E.g., webtable
    - 100,000,000,000 pages * 10 versions per page * 20KB/version
    - 20PB of data (200 million gigabytes)
  - E.g., google maps
    - 100TB of satellite image data

- Scale in throughput
  - Hundreds of millions of users
  - Tens of thousands to millions of queries per second

- Low latency
  - A few dozen milliseconds of total budget inside Google
  - Probably have to involve several dozen internal services per request
  - Few milliseconds for lookup
BigTable is used by,
- Web indexing
- Google Reader
- Google Maps
- Google Book Search
- Google Earth
- Blogger.com
- Google Code
- YouTube
- Gmail
- …

BigTable

- Provides a simple data model
- Dynamic control over the data layout and format
- Allows clients to reason about the locality properties of the data represented in the underlying storage
- Data is indexed using row and column names that can be arbitrary strings
- Data in BigTable
  - Uninterpreted strings
  - Clients often serialize various forms of structured and semi-structured data into these strings

BigTable 2/2

- Clients can control locality of their data
- Clients can control whether to serve data out of memory or from disk

Topics in BigTable

- Data model
- Locating tablet
- Data Compaction
- Data Compression
- Caching and prefetching

Data Model

- A BigTable is a sparse, distributed, persistent multi-dimensional sorted map
- The map is indexed by,
  - A row key
  - A column key
  - A timestamp
- Each value in the map is an uninterpreted array of bytes
  - (row:string, column:string, time:int64) → string

Example of data model with Webtable

- A large collection of web pages and related information
- URLs
- Contents
- Information

"contents:"

"com.cnn.www"
Rows
- Row keys
  - Arbitrary strings
  - Every read or write of data under a single row key is atomic
- BigTable maintains data in lexicographic order by row key
- Row range for a table
  - Dynamically partitioned

Tables (1/2)
- Large tables are broken into tablets at row boundaries
  - A tablet holds a contiguous range of rows
    - Clients can often choose row keys to achieve locality
    - Aim for ~100MB to 200MB of data per tablet
  - Serving machine responsible for ~100 tablets
    - Fast recovery
    - Allows a 100 machines to each pick up 1 tablet from the failed machine
    - Fine-grained load balancing
    - Migrate tablets away from the overloaded machine
    - Master makes load-balancing decisions

Tablets (2/2)
- Read of short row ranges are efficient
  - Require communication with only a small number of machines
  - Clients get good locality for their data access
  - maps.google.com/index.html is stored using the key com.google.maps/index.html
  - Storing pages under the same domain near each other makes some host and domain analysis more efficient

FAQs
- $\log_2(x)$?

Column Families (1/2)
- Column keys are grouped into sets called column families
  - Basic unit of access control
- All data stored in a column family is usually of the same type
  - BT compresses data in the same column family together
- A column family must be created before data can be stored under any column key in that family
  - After a family has been created, any column key within the family can be used

Column Families (2/2)
- Column key
  - Family and qualifier
  - Family name must be printable
  - Qualifier may be an arbitrary string
- Access control and disk/memory accounting
  - Performed at the column family level
Webtable with column-family

```
<html>
  <p>Content...</p>
</html>
```

```
<html>
  <p>Content...</p>
</html>
```

```
<html>
  <p>Content...</p>
</html>
```

```
<html>
  <p>Content...</p>
</html>
```

```
<html>
  <p>Content...</p>
</html>
```

```
<html>
  <p>Content...</p>
</html>
```

Webtable with multiple column-families

<table>
<thead>
<tr>
<th>Column Family</th>
<th>Column Family</th>
<th>Column Family</th>
</tr>
</thead>
<tbody>
<tr>
<td>language</td>
<td>contents</td>
<td>anchor:www.si.com</td>
</tr>
<tr>
<td>com.cnn.www</td>
<td>EN</td>
<td>anchor:www.si.com</td>
</tr>
<tr>
<td>com.cnn.weather</td>
<td>EN</td>
<td>anchor:www.si.com</td>
</tr>
<tr>
<td>com.cnn.weather/TECH</td>
<td>EN</td>
<td>anchor:www.si.com</td>
</tr>
<tr>
<td>com.weather</td>
<td>EN</td>
<td>anchor:www.si.com</td>
</tr>
</tbody>
</table>

Timestamps

- Each cell in Bigtable can contain multiple versions of the same data
- Indexed by timestamp

- BigTable timestamp
  - 64-bit integers
  - Assigned by BigTable
  - Real time in microseconds
  - Explicitly assigned by client application

- Application should generate unique timestamp to avoid collisions
- Different versions of a cell are stored in decreasing timestamp order
- The most recent versions can be read first

API

- Functions for creating and deleting tables and column families
- Changing cluster, table, and column-family metadata (access control rights)

```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

Garbage collection

- Two per-column-family settings
- Tell Bigtable to garbage-collect cell versions automatically
- The last n versions are kept
  - i.e. only recent versions are kept

System Structure

BigTable master
- Performs metadata ops + load balancing
- Handles failover, monitoring

BigTable tablet server
- Serves data

BigTable tablet server
- Serves data

BigTable tablet server
- Serves data

GFS
- Holds tablet data, logs

Lock service
- Holds metadata, handles master-collection
Building blocks
- The Google SSTable (Sorted String Table) file format
  - Internally used to store BigTable data
  - Persistently ordered immutable map from key to values
  - Keys and values are arbitrary byte strings

- SSTable contains a sequence of blocks
  - 64KB, configurable

- Block index
  - Stored at the end of SSTable
  - Index is loaded into memory when the SSTable is opened

SSTable: Sorted String Table
Reading and writing data can dominate running time
Random reads and writes are critical features

<table>
<thead>
<tr>
<th>Key</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>Offset</td>
</tr>
<tr>
<td>Key</td>
<td>Offset</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Access to the block
- Lookup can be performed with a single disk seek
  - Find the block by performing a binary search of the in-memory index
  - Read the block from disk

Locating tablets (1/2)
- Since tablets move around from server to server, given a row, how do clients find the right machine?
  - Need to find tablet whose row range covers the target row
  - Using the BigTable master
    - Central server almost certainly would be bottleneck in large system
    - Instead: store special tables containing tablet location info in BigTable cell itself

Locating tablets (2/2)
- 3-level hierarchical lookup scheme for tablets
  - Location is 3 levels of relevant server
  - 1st level: booted from Chubby (lock service), points to the root tablet
  - 2nd level: Uses root tablet data to find owner of appropriate metadata tablets
  - 3rd level: metadata table holds locations of tablets of all other tables
    - Metadata tablet itself can be split into multiple tablets
    - Root tablet is never split
    - To ensure that the tablet location hierarchy has no more than 3 levels
    - Metadata tablet
      - Stores the location of a tablet under a row key
      - Tablet's identifier and its end row
      - Each metadata row stores approximately 1KB of data in memory
      - Average limit of 128MB Metadata tablets
      - 2nd tablets are addressed
Caching the tablet locations (1/2)

- Client library caches tablet locations
- Traverses up the tablet location hierarchy
  - If the client does not know the location of a tablet
  - If it discovers that the cached location information is incorrect

Caching the tablet locations (2/2)

- If the client’s cache is empty?
  - One read from Chubby
  - One read from root tablet
  - One read from metadata tablet
  - Three network round-trips is required to locate the tablet
- If the client’s cache is stale? (BS quiz)
  - With given information, client could not find the data
  - Up to 6 round trips

Prefetching tablet locations

- Client library reads the metadata for more than one tablet
  - Whenever it reads the metadata table
- No GFS accesses are required
  - Table locations are stored in memory

Tablet Assignment (1/2)

- Each tablet is assigned to one tablet server at a time
  - The master keeps track of:
    - The set of live tablet servers
    - Which tablets are assigned
- New tablet assignment
  - The master assigns the tablet by sending a tablet load request to the tablet server

Tablet Assignment (2/2)

- A tablet server starts
  - Chubby creates a uniquely-named file in a specific Chubby directory
  - Exclusive lock
  - Master monitors this directory to discover tablet servers
- A tablet server terminates
  - Release its lock
  - Master will reassign its tablets more quickly

Table serving

- The persistent state of a tablet is stored in GFS
Tablet Representation

- Write buffer in memory (random-access) `MemTable`
- Append-only log on GFS

Read operation

- Tablet server checks
  - If the request is well-formed
  - If the user is authorized to read data
- Merged view of `MemTable` in memory and `SSTable` in disk
  - Read operation is performed

Data Compaction and Compression

- What is the difference between data compaction and data compression?

Minor Compactions

- As write operations executed
  - The size of the `memtable` increases

- Minor compaction
  - When the `memtable` size reaches a threshold
    - The `memtable` is frozen
    - A new `memtable` is created
    - A frozen `memtable` is converted to an `SSTable` (stored in GFS)
  - Shrinks the memory usage in the tablet server
  - Reduces the amount of data that has to be read from the commit log during recovery (if the server dies)
Merging Compaction
- New SSTable from the minor compaction will increase
  - Read operations need to merge updates from large number of SSTables
- Merging Compaction
  - Bounds the number of such files periodically
  - Reads the contents of a few SSTables and the memtable and writes out a
    new SSTable
  - Input SSTables and memtable can be discarded as soon as the merging
    compaction has finished

Major Compaction
- Rewrites multiple SSTables into exactly one SSTable
  - No deletion information or deleted data included

Background
- Sequential access to disk (magnetic or SSD) is at least three orders of magnitude faster than random IO
  - Journaling, logging or a heap file is fully sequential
  - 200-300 MB/s per drive
- But logs are only really applicable to “SIMPLE” workloads
  - Data is accessed entirely
  - Data is accessed by a known offset

Data Compaction: Log-Structured Merge (LSM) Trees

Sequential IO vs. Random IO

Existing approaches
- Search sorted file
- Hash
- B+ tree
- External file: create separate hash or tree index
Adding index structure improves read performance
• It will slow down write performance
• Log-structured merge trees
  - Fully disk-centric
  - Improved write performance
  - Read performance is still slightly poorer than B+ tree

LSM trees
• LSM trees manage batches of writes to be saved
• Each file contains a batch of changes covering a short period of time
  - Each file is sorted before it is written
  - Files are immutable
  - New updates will create new files
  - Reads inspect all files
  - Periodically files are merged

Basic idea of LSM trees
• LSM trees manage batches of writes to be saved
• Each file contains a batch of changes covering a short period of time
  - Each file is sorted before it is written
  - Files are immutable
  - New updates will create new files
  - Reads inspect all files
  - Periodically files are merged

In-memory buffer for LSM (MemTable)
• Data is stored as a tree (Red-Black etc) to preserve key-ordering
  - MemTable is replicated on disk as a write-ahead-log
  - When the MemTable fills the sorted data is flushed to a new file on disk
  - Only sequential I/O is performed
  - Each file represents a small, chronological subset of changes (sorted)
  - Periodically the system performs a compaction

Locality groups
• Clients can group multiple column families together into a locality group
  - Separate SSTable is generated for each locality group in each tablet
• Example
  - Locality group 1: Page metadata in Webtable
  - Locality group 2: Contents of the page
  - Application reading the metadata does not need to read through all of the page content

Compression
• Compression is required for the data stored in BigTable
  - Similar values in the same row/column
  - With different timestamps
  - Similar values in different columns
  - Similar values across adjacent rows
• Clients can control whether or not the SSTables for a locality group are compressed
  - User specifies the locality group to be compressed and the compression scheme
  - Keep blocks small for random access (~64KB compressed data)
  - Low CPU cost for encoding/decoding
  - Server does not need to encode/decode entire table to access a portion of it
Two-pass compression scheme

- Data to be compressed
  - Keys in BigTable (row, column and timestamp)
  - Sorted strings
  - Values in BigTable
  - BMDiff (Bentley and McIlroy’s Scheme) across all values in one family
  - BMDiff output for values 1..N is dictionary for value N+1
- Zippy is used for final pass over whole block
  - Localized repetitions
  - Cross-column-family repetition, compresses keys
- First pass: BMDiff
- Second pass: Zippy (now called as snappy)

BMDiff

- Adapted to VCDiff (RFC3284)
  - Shared Dictionary Compression over HTTP (SDCH)
  - Chrome browser

FAQs

- /tmp issue
  - Use /a/HOSTNAME/a/tmp
  - Please check http://www.cs.colostate.edu/~cs480/Assignments.html
- Deadline: Tomorrow (April, 04) 5:00PM
- Please sign up for the demo

Example of the Constitution of the US and the King James Bible

<table>
<thead>
<tr>
<th>File</th>
<th>Text</th>
<th>gzip</th>
<th>Relative compressed size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Const</td>
<td>Const</td>
<td>49523</td>
<td>1.0</td>
</tr>
<tr>
<td>Const</td>
<td>Const+Bible</td>
<td>99046</td>
<td>1.911</td>
</tr>
<tr>
<td>Bible</td>
<td>Bible+Bible</td>
<td>4460056</td>
<td>1.0</td>
</tr>
<tr>
<td>Bible+Bible</td>
<td>8920112</td>
<td>2642389</td>
<td>1.9995</td>
</tr>
</tbody>
</table>

The compression algorithm

- Representing the common string
  - start, length
  - start: initial position
  - length: size of the common sequence
- e.g. “the Constitution of the United States PREAMBLE We, the people of the United States, in order to form a more perfect Union, ...”
  - “the Constitution of the United States PREAMBLE We, the people <18,20>, in order to form a more perfect Union, ...”
Data compression

Hash table

Text

Calculate hash value

Lookup the hash table

If there is no match, this value is added

Rabin-Karp algorithm for text search (1/2)

- Brute-force substring search algorithm
  - The length of the string: \( n \)
  - The length of the substring: \( m \)
  - \( O(nm) \)

- Use of hashing for substring search algorithm
  - If two strings are equal, their hash values are also equal
  - Compare hash values of the portion of the string and substring
  - Still \( O(nm) \)

Rabin-Karp algorithm for text search (2/2)

- The efficient computation of hash values of the successive substrings of the text

- Using rolling hash
  - Current hashing value includes previous hashing value
  - \( H[s[i+1…i+m]]=H[s[i…i+m-1]] - H[s[i]] + s[m] \)

- \( O(n) \) for the most of the cases

- \( H=(c_1a^{k-1}+c_2a^{k-2}+c_3a^{k-3}+…+c_ka^0) \mod n \)
  - Where \( a \) is a constant and \( c_1, c_2, …, c_k \) are the input characters

Results

<table>
<thead>
<tr>
<th>Compression</th>
<th>Bible</th>
<th>Bible+Bible</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>4460056</td>
<td>8920112</td>
</tr>
<tr>
<td>gzip</td>
<td>1321495</td>
<td>2642389</td>
</tr>
<tr>
<td>com50</td>
<td>4384403</td>
<td>4384414</td>
</tr>
<tr>
<td>com20</td>
<td>3906771</td>
<td>3906782</td>
</tr>
<tr>
<td>com50gzip</td>
<td>1318587</td>
<td>1318599</td>
</tr>
<tr>
<td>com20gzip</td>
<td>1362413</td>
<td>1362422</td>
</tr>
</tbody>
</table>

Snappy

- Based on LZ77
  - Dictionary coders
  - Sliding window

- Very fast and stable but not high compression ratio
  - 20~100% lower compression ratio than gzip

BigTable and data compressions

- Large window data compression
  - BMDiff (~100MB/sec for write, ~1000MB/sec for read)
  - Identify large amounts of shared boilerplate in pages from same host

- Small window data compression
  - Looks for repetitions in 16KB window
  - Snappy

  - E.g. 45.1TB of crawled dataset (2.1B pages)
  - 4.2 TB compressed size
Caching for read performance

- Tablet servers use two levels of caching
  - Scan cache
    - Higher-level cache
      - Caches the key-value pairs returned by the SSTable interface in the table server
  - Block cache
    - Lower-level cache
      - Caches SSTables blocks that were read from GFS

Bloom filters

- Read operation has to read from all SSTables that make up the state of a tablet
  - SSTables in disk results many disk accesses
- Bloom filter
  - Detects if an SSTable might contain any data for a specified row/column pair
- Probabilistic data structure
  - Tests whether the element is a member of a set
  - The element either definitely is not in the set or may be in the set