



USC Viterbi
School of Engineering

Google Building Blocks:

MapReduce

Google File System

BigTable

These slides borrow material from Mining Massive Datasets by Rajaraman and Ullman and class notes by Rajaraman (Stanford) and Weld (U. Washington), D. Weld (Google) and S. Ghemawat (Google), Vera Goebel (Univ. of Oslo)



Google Specialized Software Systems

- **Google has built several major software systems for their internal processing**
 1. **MapReduce - an easy way to write and run large-scale jobs on clusters of machines**
 - generate production index data more quickly
 - perform ad-hoc experiments rapidly
 - Dean & Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*, OSDI, 2004
 2. **GFS (Google File System) a large-scale distributed file system**
 - Ghemawat, Gobioff, & Leung. *Google File System*, SOSP 2003
 3. **BigTable - a semi-structured storage system**
 - online, efficient access to per-document information at any time
 - multiple processes can update per-doc info asynchronously
 - critical for updating documents in minutes instead of hours
 - Chang, Dean, Ghemawat, Hsieh, Wallach, Burrows, Chandra, Fikes, & Gruber. *Bigtable: A Distributed Storage System for Structured Data*, OSDI 2006



USC Viterbi
School of Engineering

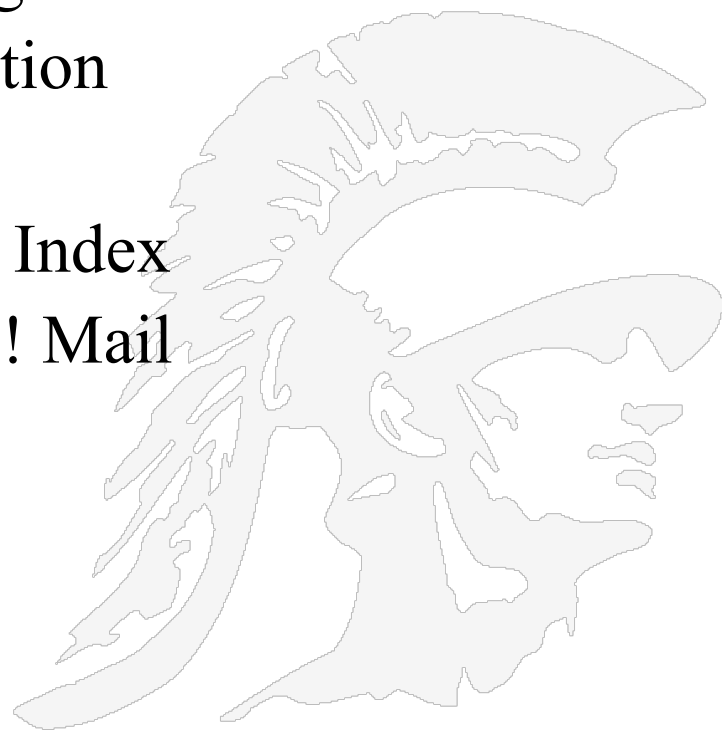
Introduction to MapReduce

- **MapReduce is a methodology for exploiting parallelism in computing clouds (racks of interconnected processors)**
- **It has become a common way to analyze very large amounts of data**
- **MapReduce was initially developed at Google**
- **In 2004 Google was using MapReduce to process 100TB/day of data**
- **Today there are MapReduce Users Groups around the world, see <https://www.meetup.com/topics/mapreduce/>**



How is MapReduce Used by Search Engines?

- **At Google:**
 - Building Google's Search Index
 - Article clustering for Google News
 - Statistical machine translation
- **At Yahoo!:**
 - Building Yahoo!'s Search Index
 - Spam detection for Yahoo! Mail
- **At Facebook:**
 - Data mining
 - Ad optimization
 - Spam detection



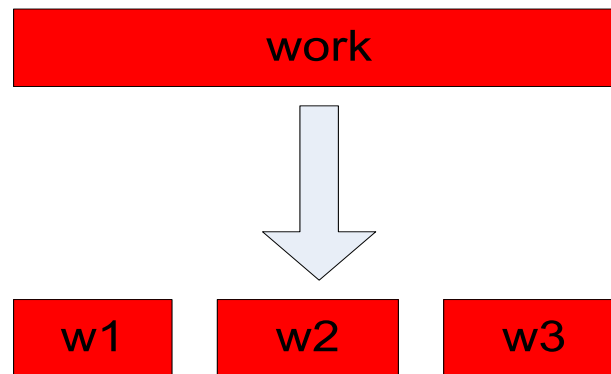


Motivation Beyond Search Engines

- **Modern Internet applications have created a need to manage immense amounts of data quickly.**
- **In many of these applications, the data is extremely regular, and there is ample opportunity to exploit parallelism.**
- **Some examples**
 - 1. Dish network collecting every click of the remote**
 - Dish network supplies TV reception via satellite; they collect data on their set top box and send it back to headquarters
 - 2. Tesla collecting every usage of the car**
 - Tesla's are connected to the cellular network; the car reports back all of its actions to Tesla

Why Parallelization is Hard

- Parallelization is “easy” if processing can be cleanly split into n units:



Partition
problem

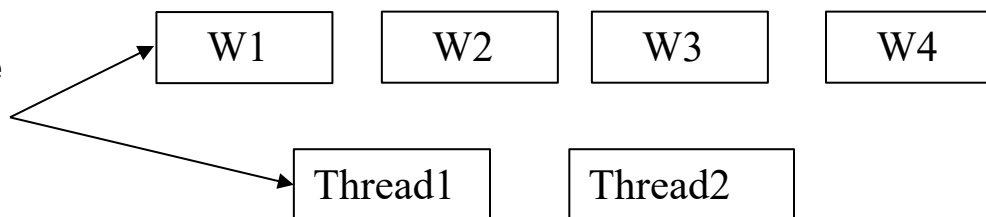
- Assigned to n workers
- And there is an easy way to combine the outputs



USC **Viterbi**
School of Engineering

Why Parallelization is Hard

we would like to have
as many threads as
there are work units,
but this may not be
the case

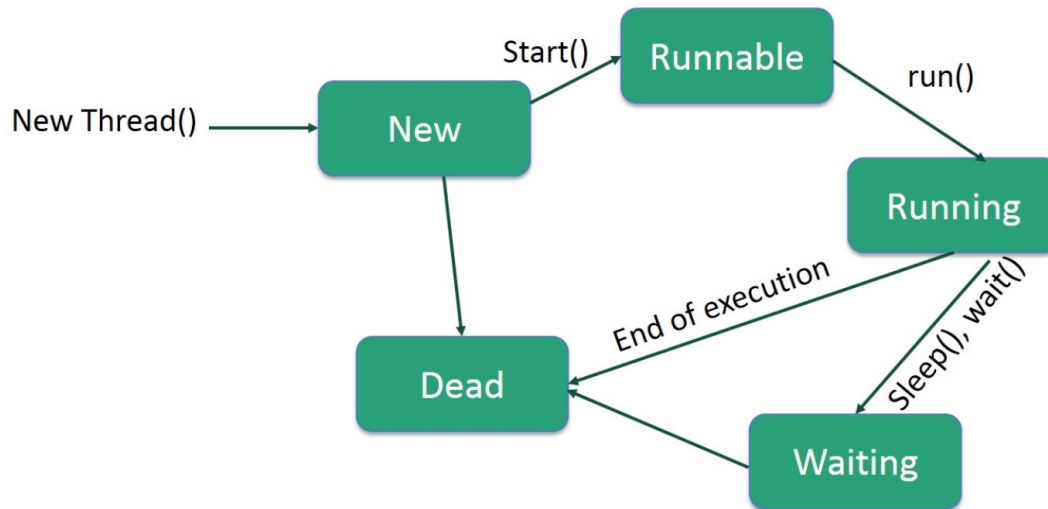


But there are complicated issues to deal with!

- **What if we have more work units than threads?**
- **How do we assign work units to worker threads?**
- **How do we aggregate/combine the results at the end?**
- **How do we know all the workers have finished?**
- **What if the work cannot be divided into completely separate tasks?**
- *MapReduce solves all of these problems so the programmer does not have to deal with them*



Programming with Multiple Threads Poses Challenges



Life Cycle of a Thread

Thread 1:	Thread 2:
void foo() {	void bar() {
x++;	y++;
y = x;	x++;
}	}

If the initial state is $x = 6$, $y = 0$, what are the final values of x and y after the threads finish running? Possible solutions include: (8,8) and (8,7)



Multithreaded = Unpredictability

- Many things that look like “one step” operations actually take several steps under the hood:

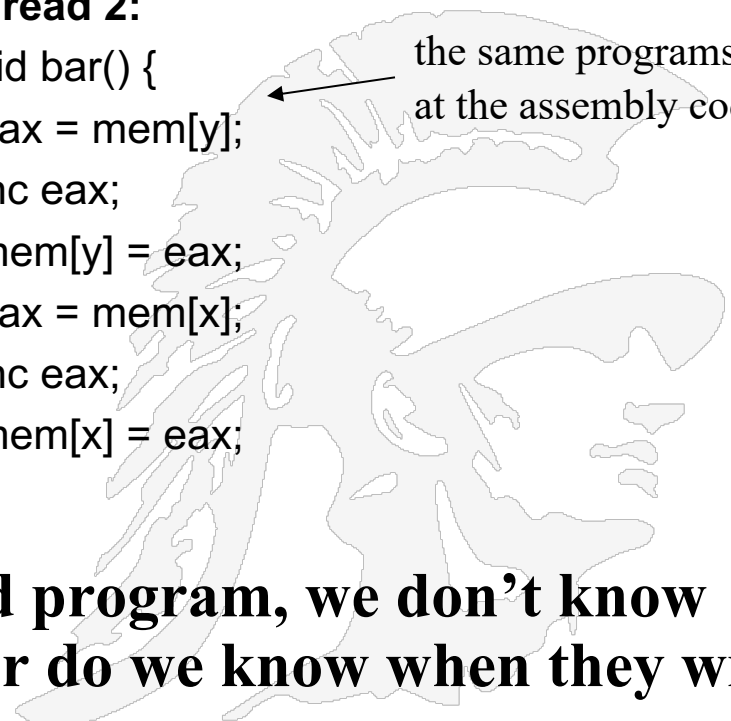
Thread 1:

```
void foo() {  
    eax = mem[x];  
    inc eax;  
    mem[x] = eax;  
    ebx = mem[x];  
    mem[y] = ebx;  
}
```

Thread 2:

```
void bar() {  
    eax = mem[y];  
    inc eax;  
    mem[y] = eax;  
    eax = mem[x];  
    inc eax;  
    mem[x] = eax;  
}
```

the same programs, but
at the assembly code level



- When we run a multithreaded program, we don't know what order threads run in, nor do we know when they will interrupt one another.



The “corrected” example

Thread 1:

```
void foo() {  
    sem.lock();  
    x++;  
    y = x;  
    sem.unlock();  
}
```

Thread 2:

```
void bar() {  
    sem.lock();  
    y++;  
    x++;  
    sem.unlock();  
}
```

The global variable `sem`, as defined by

```
Semaphore sem = new Semaphore();
```

guards access to `x` & `y`; semaphores are generally integer variables that are shared between threads; the variable protects the “critical section” from being simultaneously accessed



Processing Across a Machine Cluster Introduces Unpredictability on Many Levels

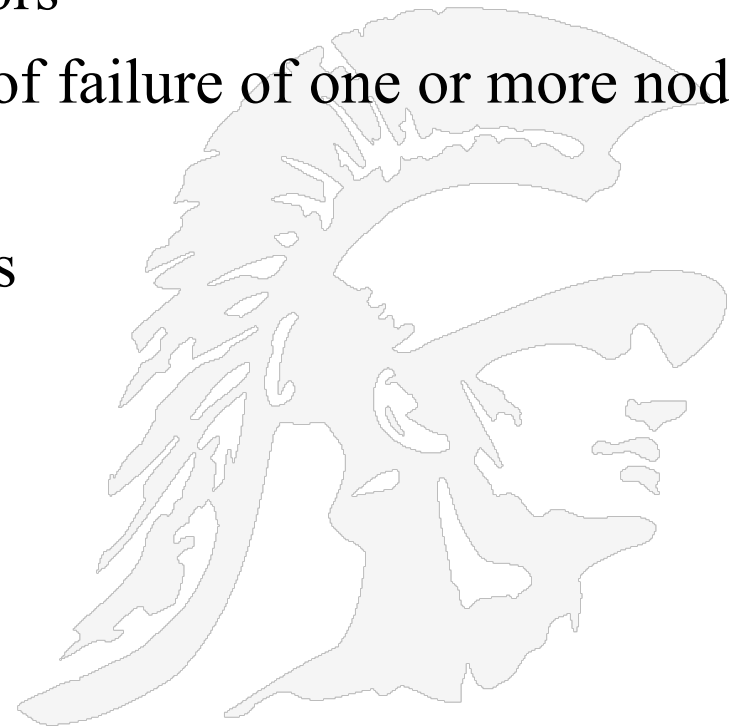
- **Synchronization problems apply to more than just low level operations within a critical section of code**
 - **Other issues include:**
 - **Pulling work units from a queue**
 - **Assigning work units to an available thread**
 - **Work units reporting back to the master unit**
 - **Telling another thread that it can begin the “next phase” of processing**

... All require synchronization!



How MapReduce Solves the Parallelization Problems

- **So MapReduce provides**
 - Automatic parallelization of code across multiple threads and across multiple processors
 - Fault tolerance in the event of failure of one or more nodes
 - I/O scheduling
 - Monitoring & Status updates





Map/Reduce - Beginnings

- **Map/Reduce**
 - Is a programming model borrowed from the programming language Lisp
 - (and other functional languages, e.g. ML*)
- **Many problems can be phrased using the MapReduce paradigm**
- **Easy to distribute computation across nodes**
- **Nice retry/failure semantics**

*ML programming language is a general purpose functional programming language, see [https://en.wikipedia.org/wiki/ML_\(programming_language\)](https://en.wikipedia.org/wiki/ML_(programming_language))



Map and Reduce Functions in LISP (Scheme is a dialect of Lisp)

- $(\text{map } f \text{ list } [list_2 \text{ list}_3 \dots])$ General formulation
- Specific example
 - $(\text{map square } '(1 \ 2 \ 3 \ 4))$
 - $(1 \ 4 \ 9 \ 16)$
- $(\text{reduce } f \text{ id list})$
- Specific example
 - $(\text{reduce } + \ 0 \ '(1 \ 4 \ 9 \ 16))$
 - $(+ \ 16 \ (+ \ 9 \ (+ \ 4 \ (+ \ 1 \ 0))))$
 - 30





What is MapReduce?

- MapReduce is a programming model that generically works this way:
 - A *map function* extracts some intelligence from raw data
 - A *shuffle step* organizes the resulting output
 - A *reduce function* aggregates the data output from the shuffle step
 - Users specify the computation in terms of a *map* and a *reduce* function,
 - Underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, and
 - Underlying system also handles machine failures, efficient communications, and performance issues.

-- Reference: Dean, J. and Ghemawat, S. 2008 “MapReduce: Simplified Data Processing on Large Clusters”, *Communication of ACM* 51, 1 (Jan. 2008), 107-113.



The Map/Reduce Paradigm

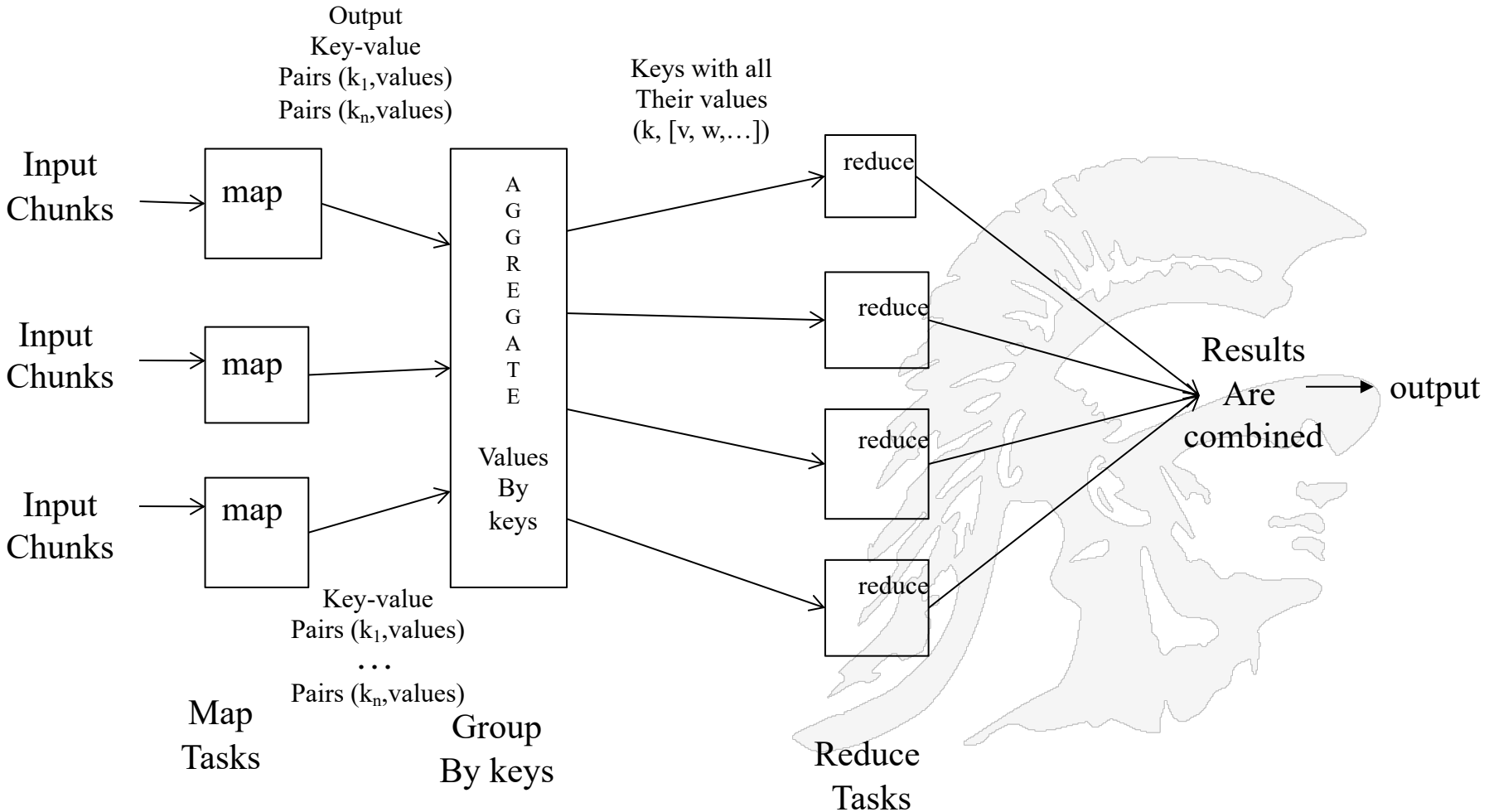
1. A large number of records are broken into segments
2. **Map**: extracts something of interest from each segment
3. **Group**: sorts the intermediate results from each segment (sometimes called **shuffle**)
4. **Reduce**: aggregates intermediate results
5. Generate final output

Key idea: to re-phrase problems in such a way that the input can be divided into parts and operated on in parallel and the results combined to produce a solution to the original problem

The Map&Reduce Routines

- Using map-reduce one must write 2 functions called *Map* and *Reduce*
- The system manages the parallel execution and coordination of tasks; it is all done automatically
- A map-reduce computation proceeds as follows:
 1. Some number of map tasks each are given one or more chunks to process
 2. These map tasks turn the chunk into a sequence of key-value pairs; the way the pairs are produced depends upon the code for the Map function
 3. Key-value pairs from each Map task are collected by a master controller and sorted by key; keys are divided among all the Reduce tasks, so all key-value pairs with the same key wind up at the same Reduce task
 4. Reduce tasks work on one key at a time, and combine all the values associated with that key in some way; the manner of combination depends upon the Reduce code

Schematic of a Map-Reduce Computation





A MapReduce Example – Counting Word Occurrences

- Counting the number of occurrences for each word in a collection of documents
- The input file is a repository of documents
- Each document is an element passed to a separate processor
- The Map function
 - Parses the document, extracts each word and uses each word as a key of type String (the words obtained by parsing), w_1, w_2, \dots
 - For each word it assigns an integer, 1;
 - Each processor outputs key-value pairs where the key is a word and the value is always 1, namely $(w_1, 1), (w_2, 1), \dots, (w_n, 1)$
- If a word w appears n times in a single document, then there will be n key-value pairs $(w, 1)$ in the output of the processor handling that document
- If a word w appears m times among all documents, then there will be m key-value pairs $(w, 1)$ in the output



Count Word Occurrences Pseudo-Code

- The code below is similar to what a programmer would write to process multiple documents on a cluster of machines using map/reduce

Map(String input_key, String input_value):

// input_key: document name

// input_value: document contents

for each word w in input_value:

EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values):

// output_key: a word

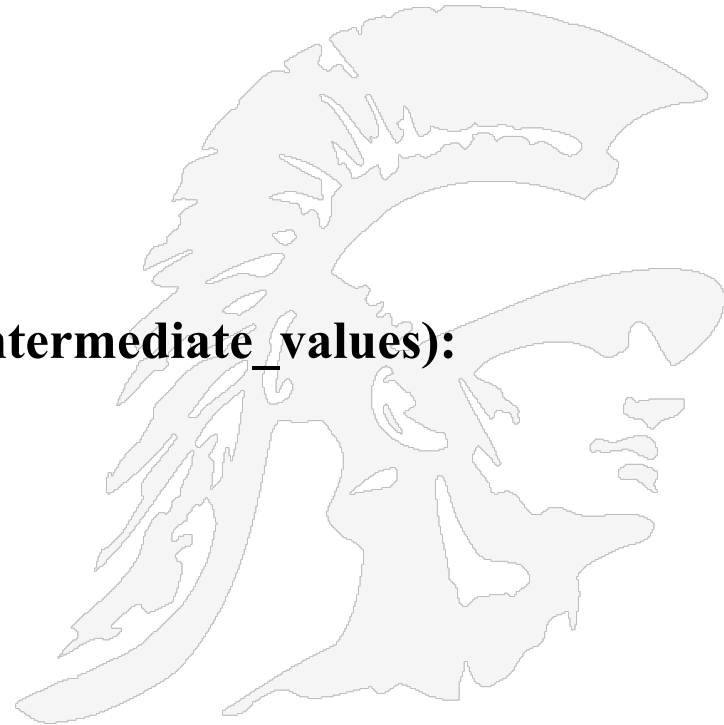
// output_values: a list of counts

int result = 0;

for each v in intermediate_values:

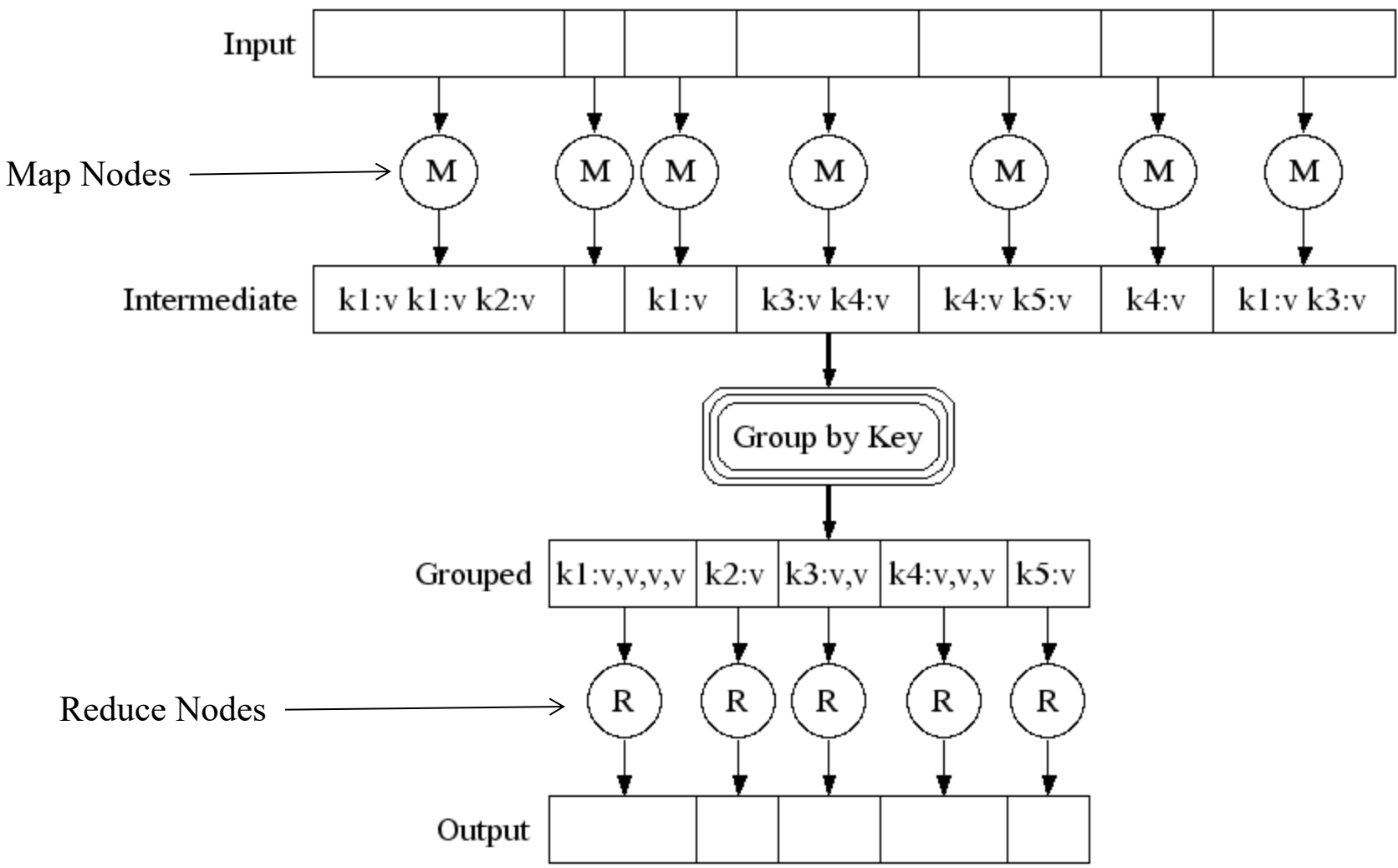
result += ParseInt(v);

Emit(AsString(result));

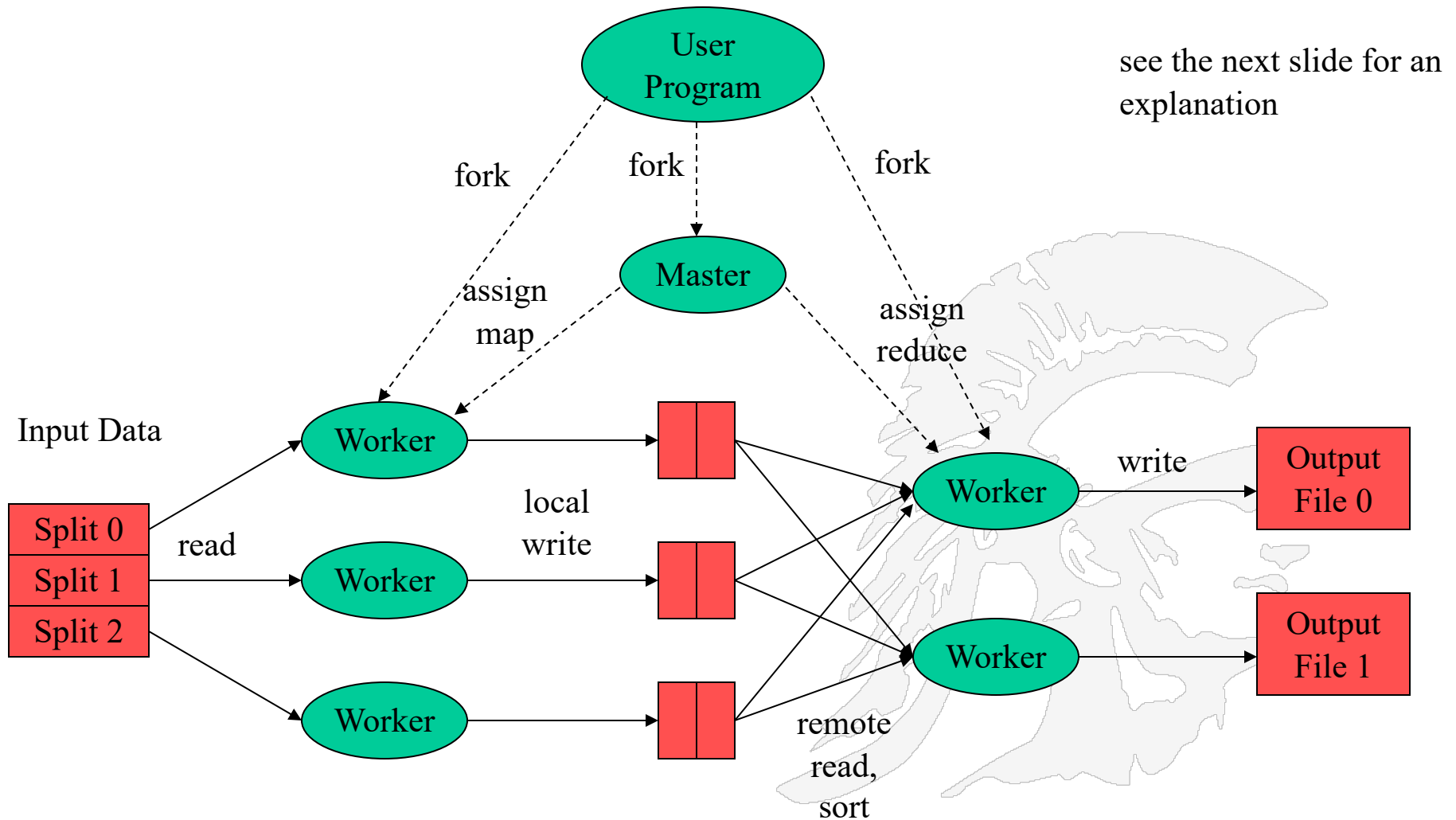


Word Count Execution

An Alternate Graphical View



Distributed Execution Overview





Looking Under the Hood at the Map Task

- The user program forks a *Master* controller process and some number of *Worker* processes at different compute nodes;
- A *Worker* handles either Map tasks or Reduce tasks, but not both
- The *Master* must
 - Create some number of Map and Reduce tasks
 - These tasks are assigned to *Worker* processes by the *Master*
 - Typically there is one Map task for every chunk of the input
 - Keeps track of the status of each Map and Reduce task (states are: idle, executing on a Worker, completed)
- Each Map task is assigned one or more chunks of the input file(s) and executes on it the code
- The Map task creates a file for each Reduce task on the local disk of the Worker that executes the Map task
- The *Master* is told of the location and sizes of each of these files and the Reduce task for which each is destined



Looking Under the Hood at the Reduce Task

- The *master controller* process knows how many Reduce tasks there will be, say r
- The user defines r
- The master controller picks a hash function that applies to keys and produces a bucket number from 0 to $r-1$
- Each key output by a Map task is hashed and its key-value pair is put in one of r local files
 - Each file will be processed by a Reduce task
- After all Map tasks have completed successfully, the master controller merges the file from each Map task that are destined for a particular Reduce task and feeds the merged file to that process
- For each key k , the input to the Reduce task that handles key k is a pair $(k, [v_1, \dots, v_n])$ where $(k, v_1), (k, v_2), \dots, (k, v_n)$ are all the key-value pairs with key k coming from all the Map tasks



Explanation of the Reduce Task

- The *Reduce function* is written to take pairs consisting of a key and a list of associated values, and combines them in some way
- The *Reduce function* output is a sequence of key-value pairs consisting of each input key k paired with the combined value
- Outputs from all Reduce tasks are merged into a single file
- *Reduce function* adds up all the values and outputs a sequence of (w, m) pairs where w is a word that appears at least once in the documents and m is the total number of occurrences
- The *Reduce function* is generally associative and commutative implying values can be combined in any order yielding the same result



Fault Tolerance in MapReduce

1. If a task crashes:

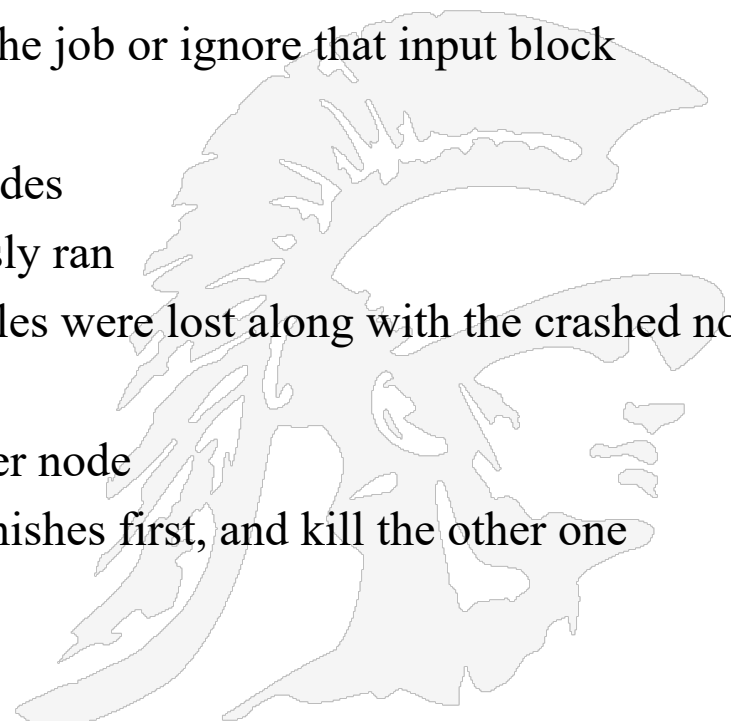
- Retry on another node
 - OK for a map because it had no dependencies
 - OK for reduce because map outputs are on disk
- If the same task repeatedly fails, fail the job or ignore that input block

2. If a node crashes:

- Relaunch its current tasks on other nodes
- Relaunch any maps the node previously ran
 - Necessary because their output files were lost along with the crashed node

3. If a task is going slowly (straggler):

- Launch second copy of task on another node
- Take the output of whichever copy finishes first, and kill the other one





Coping with Node Failure

- Worst case: **the compute node where the Master is executing fails**
 - **Result:** the entire map-reduce job must be restarted
- Other failures are less severe and are handled by the Master
- **The compute node of a Map worker fails**
 - This is detected by the Master and all Map tasks that were assigned are re-done
 - The Master sets the status of each Map task to idle and re-schedules them when a worker becomes available
 - The Master informs each Reduce task of the location of its new input
- **The compute node of a Reduce worker fails**
 - The Master sets the status of its currently executing Reduce tasks to idle and they will be re-scheduled on another reduce worker later

USC Viterbi
School of Engineering

Typical Data Center Cluster



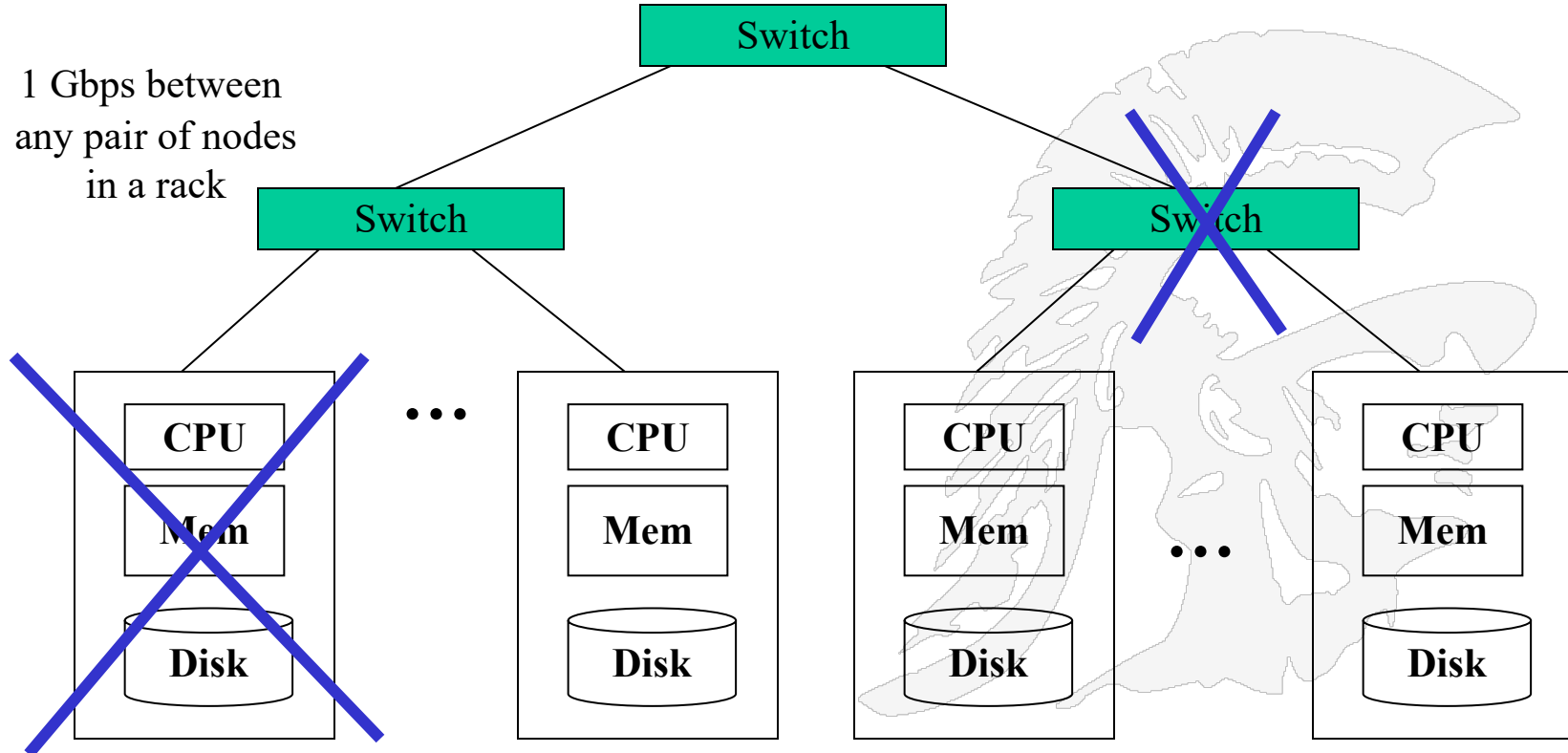


Characteristics of a Google Data Center

1. **Google data centers** (approx. two dozen): They come online and automatically, under the direction of the Google File System, start getting work from other data centers. These facilities, sometimes filled with 10,000 or more Google computers, find one another and configure themselves with minimal human intervention.
2. **Standard desktop PCs**: The hardware in a Google data center can be bought at a local computer store.
3. Each Google server comes in a standard case called a **pizza box** with one important change: the plugs and ports are at the front of the box to make access faster and easier.
4. **Google racks** are assembled for Google to hold servers on their front and back sides. This effectively allows a standard rack, normally holding 40 pizza box servers, to hold 80 servers.
5. A Google data center can go from a stack of parts to **online operation** in as little as **72 hours**, unlike more typical data centers that can require a week or even a month to get additional resources online.
6. Each server, rack and data center works in a way that is similar to what is called “**plug and play**.” Like a mouse plugged into the USB port on a laptop, Google’s network of data centers knows when more resources have been connected. These resources, for the most part, go into operation without human intervention.

Typical Cluster Architecture

- Each rack of cpu's contains between 16-64 nodes
- Nodes within a single rack are connected by gigabyte Ethernet
- Each rack is connected to another rack by a switch with speeds of 2-10 Gbps
- Individual cpu's can fail; switches between racks can fail
2-10 Gbps backbone between racks



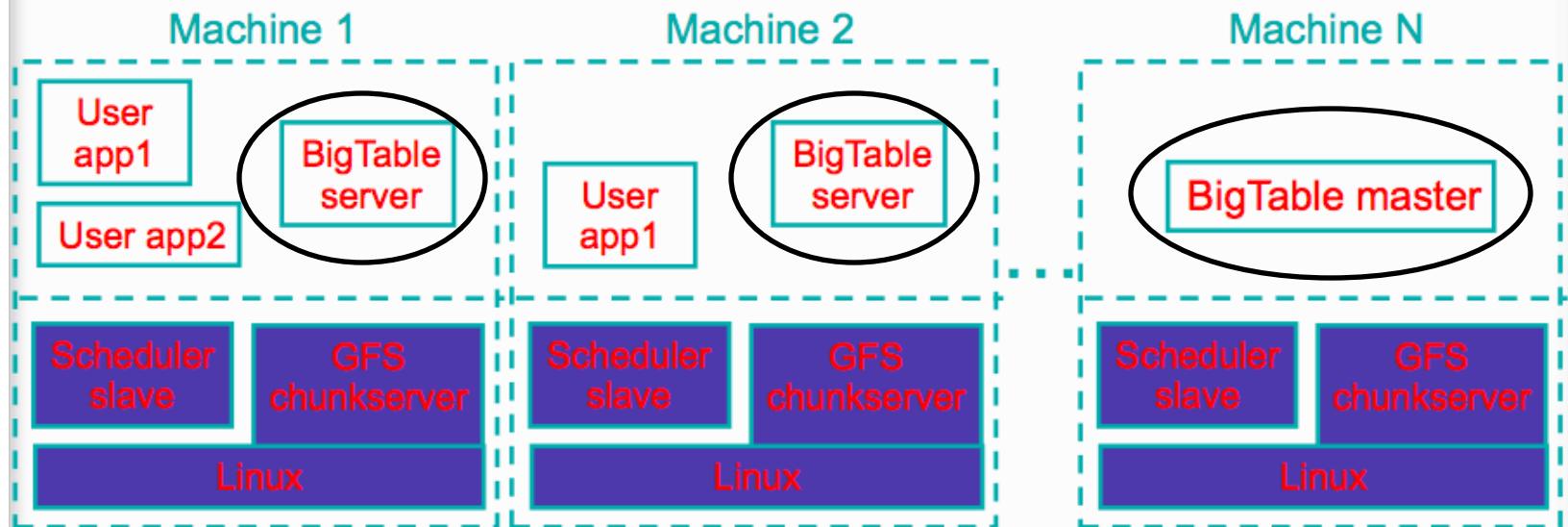
Typical Cluster Machine Configuration

Cluster scheduling master

Chubby Lock service

GFS master

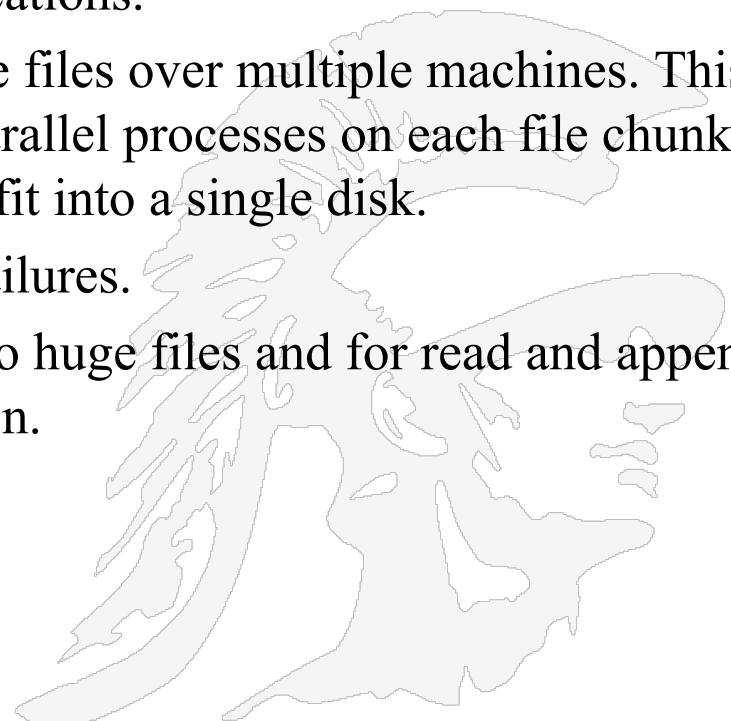
Shared pool of machines that also run other distributed applications





The Problems Google Tried to Solve with a New File System

- Google needed a large-scale and high-performance unified storage system that must:
 1. **Be global.** Any client can access (read/write) any file. This allows for sharing of data among different applications.
 2. **Support automatic sharding** of large files over multiple machines. This improves performance by allowing parallel processes on each file chunk and also deals with large files that cannot fit into a single disk.
 3. **Support automatic recovery** from failures.
 4. **Be optimized for sequential access** to huge files and for read and append operations which are the most common.





Google File System General Goals

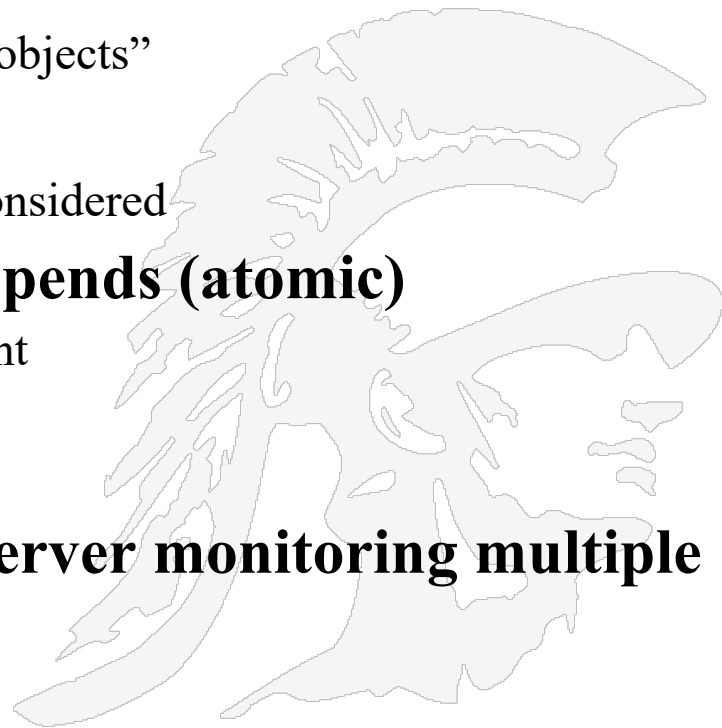
- **A scalable, distributed file system for large distributed data-intensive applications**
 - Provides fault tolerance
 - Runs on cheap, commodity hardware
 - Delivers high aggregate performance to large number of clients
- **GFS: not your typical file system**
 - Lacks typical per-directory data structure to list each file in the directory
 - Does not support aliases (i.e. hard or sym links)
 - Namespace: lookup table that maps full pathnames to metadata
 - Lookup table fits in memory (prefix compression)
 - Also known as incremental encoding is a type of delta encoding **compression** algorithm whereby common **prefixes** and their lengths are recorded so that they need not be duplicated
 - https://en.wikipedia.org/wiki/Incremental_encoding



The Google File System

Design Assumptions –

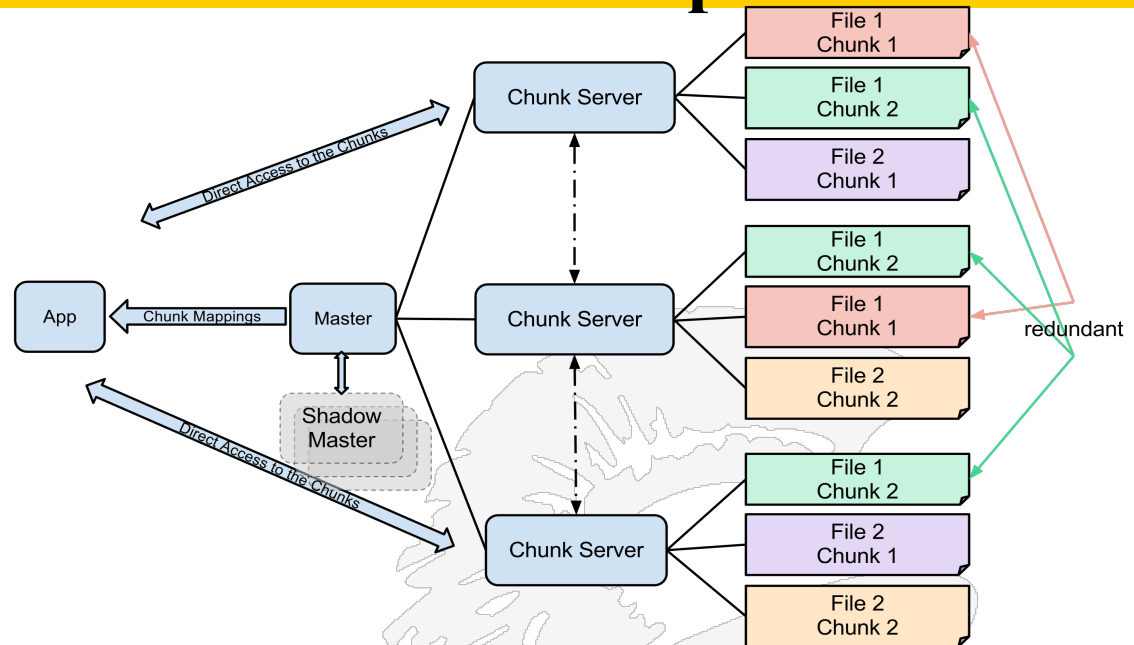
- **Files will be HUGE**
- **Multi-gigabyte files are common**
 - Not practical to have ~8 billion files
 - Each file contains many “application objects”
- **Multi-terabyte datasets**
 - I/O operations, block sizes must be considered
- **Most file modifications are appends (atomic)**
 - Random writes practically non-existent
 - Once written... sequential reads
 - Caching not terribly important
- **there will be a single master server monitoring multiple chunk servers**



Google File System

Top Level View

- **Google File System (GFS)**, is a proprietary distributed file system for efficient, reliable access to data using large clusters of commodity hardware
- Files are divided into fixed-size *chunks* of 64 megabytes, similar to clusters or sectors in regular file systems, which are only extremely rarely overwritten, or shrunk; files are usually appended to or read



GFS is designed for system-to-system interaction;
Chunk servers replicate the data automatically
See https://en.wikipedia.org/wiki/Google_File_System

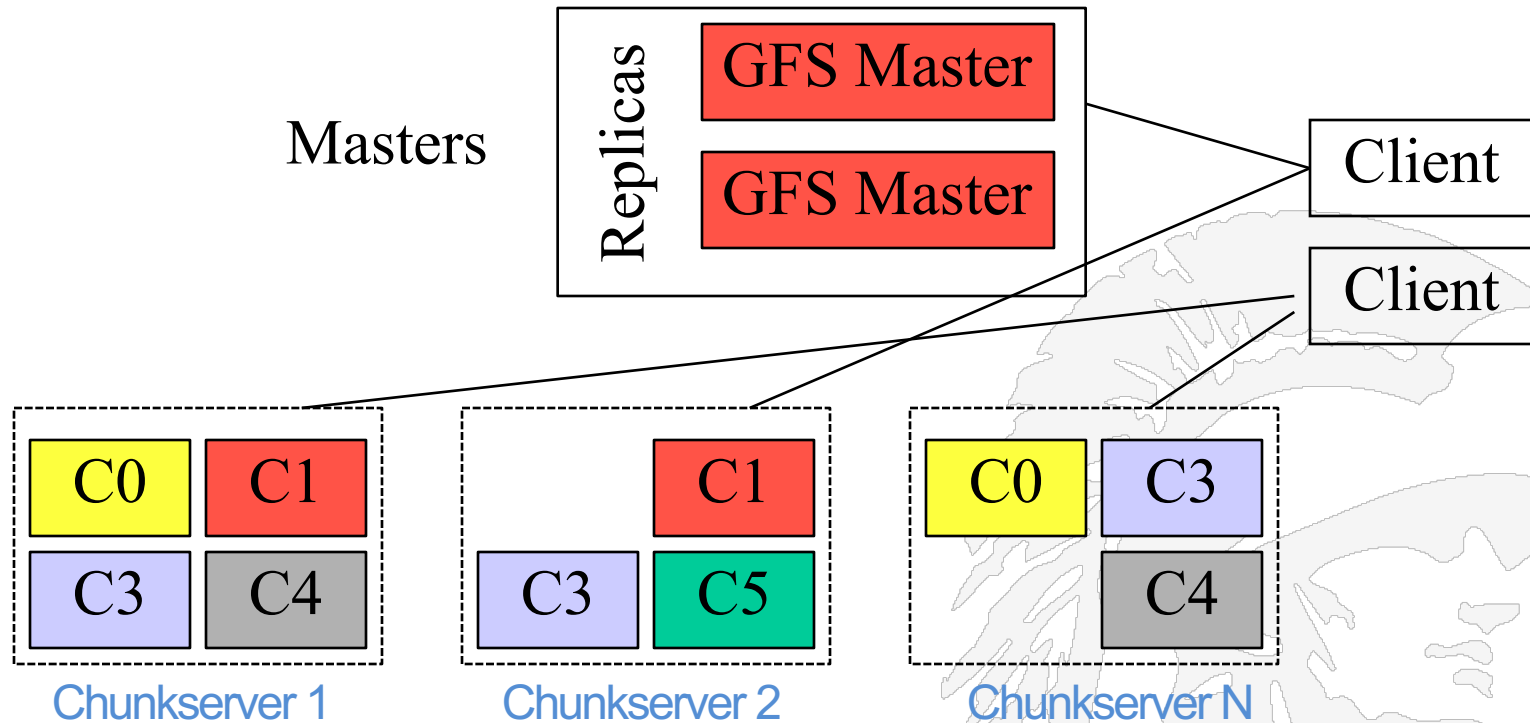


Master Server and Chunk Servers

- **Master Server** holds all metadata:
 - Namespace (directory hierarchy)
 - Access control information (per-file)
 - Mapping from files to chunks
 - Current locations of chunks (chunk servers)
- Delegates consistency management
- Garbage collects orphaned chunks
- Migrates chunks between chunk servers
- **Chunk Server**
 - Stores 64 MB file chunks on local disk using standard Linux filesystem, each with version number and checksum
 - Read/write requests specify chunk handle and byte range
 - Chunks replicated on configurable number of chunk servers (default: 3)
 - No caching of file data (beyond standard Linux buffer cache)



Google File System (GFS)



- Master manages metadata
- Data transfers happen directly between clients/chunkservers
- Files broken into chunks (typically 64 MB)
- Chunks triplicated across three machines for safety



GFS: Major Aspects

- **Append vs. Rewrite**

- GFS is optimized for appended files rather than rewrites. That's because clients within Google rarely need to overwrite files -- they add data onto the end of files instead. While it's still possible to overwrite data on a file in the GFS, the system doesn't handle those processes very efficiently

- **Which Replica Does GFS use?**

- The GFS separates replicas into two categories: **primary replicas** and **secondary replicas**. A primary replica is the chunk that a chunkserver sends to a client. Secondary replicas serve as backups on other chunkservers.
- The master server decides which chunks will act as primary or secondary. If the client makes changes to the data in the chunk, then the master server lets the chunkservers with secondary replicas know they have to copy the new chunk off the primary chunkserver to stay current.

- **What About Big Files?**

- If a client creates a write request that affects multiple chunks of a particularly large file, the GFS breaks the overall write request up into an individual request for each chunk. The rest of the process is the same as a normal write request.

- **Heartbeats and Handshakes**

- The GFS components give system updates through electronic messages called **heartbeats** and **handshakes**. These short messages allow the master server to stay current with each chunkserver's status.



Google File System vs. BigTable

- **GFS provides raw data storage**
- **But Google needs a system for handling:**
 - Trillions of URLs
 - Geographic locations such as physical entities, roads, satellite image data, etc
 - Per user data for billions of people including preference settings, recent queries and searches
 - And it must be capable of
 - storing semi-structured data
 - Reliable, scalable, etc
- **Bigtable is a compressed, high performance, proprietary data storage system built on top of the Google File System**
- **It is used by a number of Google applications, such as web indexing, MapReduce, Google Maps, YouTube and Gmail**



Big Table Data Model


- **Not a Full Relational Data Model**
- **Provides a simple data model**
 - Supports dynamic control over data layout
 - Allows clients to reason about the locality properties
- **A Table in Bigtable is a:**
 - Sparse
 - Distributed
 - Persistent
 - Multidimensional
 - Sorted map











Bigtable Storage Model

- **Data is indexed using row and column names**
- **Data is treated as uninterpreted strings**
 - (row:string, column:string, time:int64) → string
- **Rows**
 - Data maintained in lexicographic order by row key
 - Tablet: rows with consecutive keys
 - Units of distribution and load balancing
- **Columns**
 - Column families
- **Cells**
- **Timestamps**

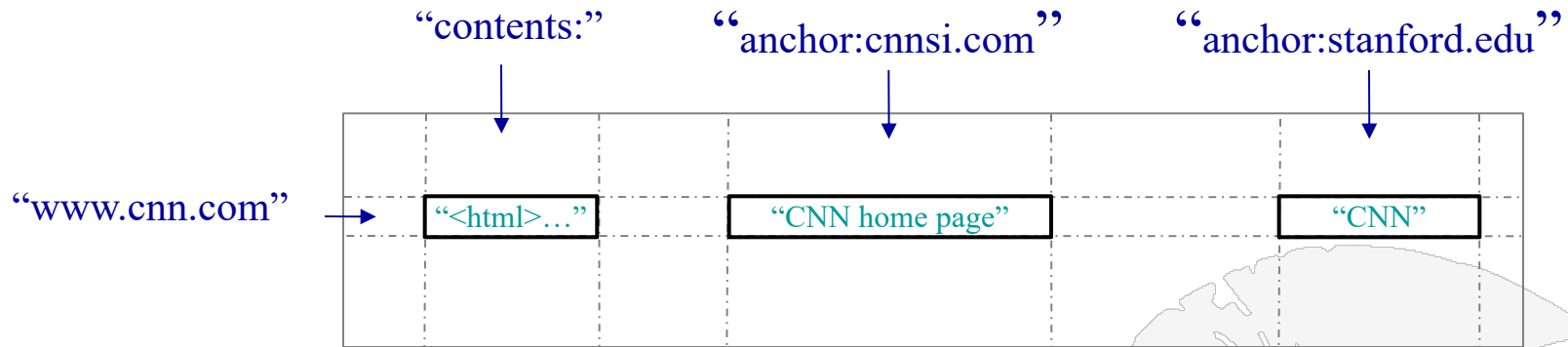


	Column family 1		Column family 2	
	<i>Column 1</i>	<i>Column 2</i>	<i>Column 1</i>	<i>Column 2</i>
Row key 1				
Row key 2				

t1
t2
t3



- **Name is an arbitrary string**
 - Access to data in a row is atomic
 - Row creation is implicit upon storing data
- **Rows ordered lexicographically**
 - Rows close together lexicographically usually reside on one or a small number of machines
- **Each row/column intersection can contain multiple cells**
 - Each cell contains a unique timestamped version of the data for that row and column
- **Storing multiple cells in a column provides a record of how the stored data has changed over time**



- **Columns have two-level name structure:**
 - **family:optional_qualifier**
- **Column family**
 - Unit of access control
 - Has associated type information
- **Qualifier gives unbounded columns**
 - Additional level of indexing, if desired



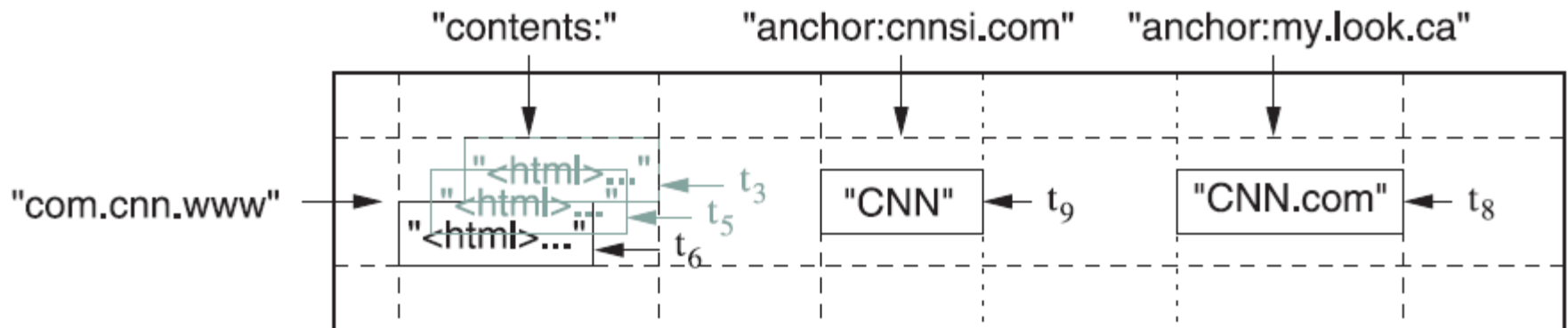
Timestamps

- **Used to store different versions of data in a cell**
 - New writes default to current time, but timestamps for writes can also be set explicitly by clients
- **Garbage Collection**
 - Per-column-family settings to tell Bigtable to GC
 - *“Only retain most recent K values in a cell”*
 - *“Keep values until they are older than K seconds”*
- **API: Create / delete tables and column families**



USC Viterbi
School of Engineering

Data Model – WebTable Example (1 of 7)

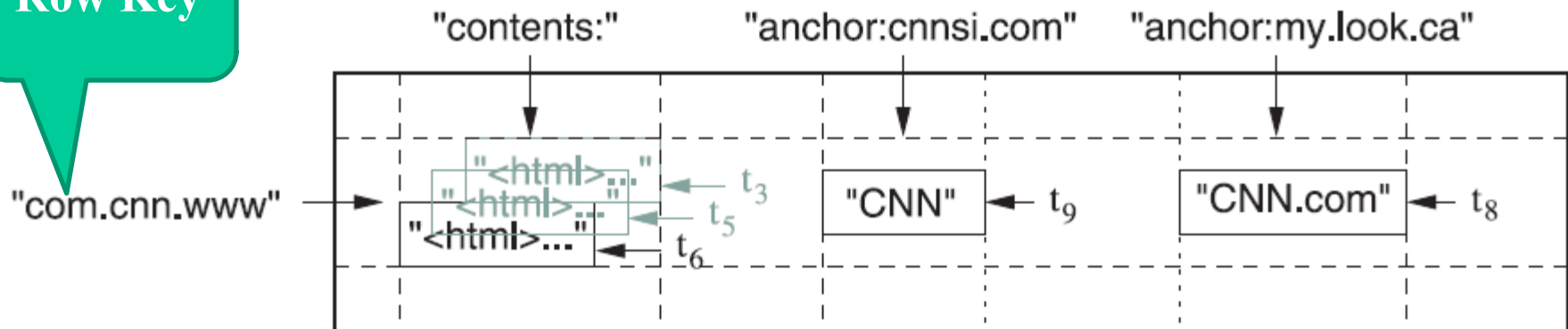


A large collection of web pages and related information



Data Model – WebTable Example (2)

Row Key

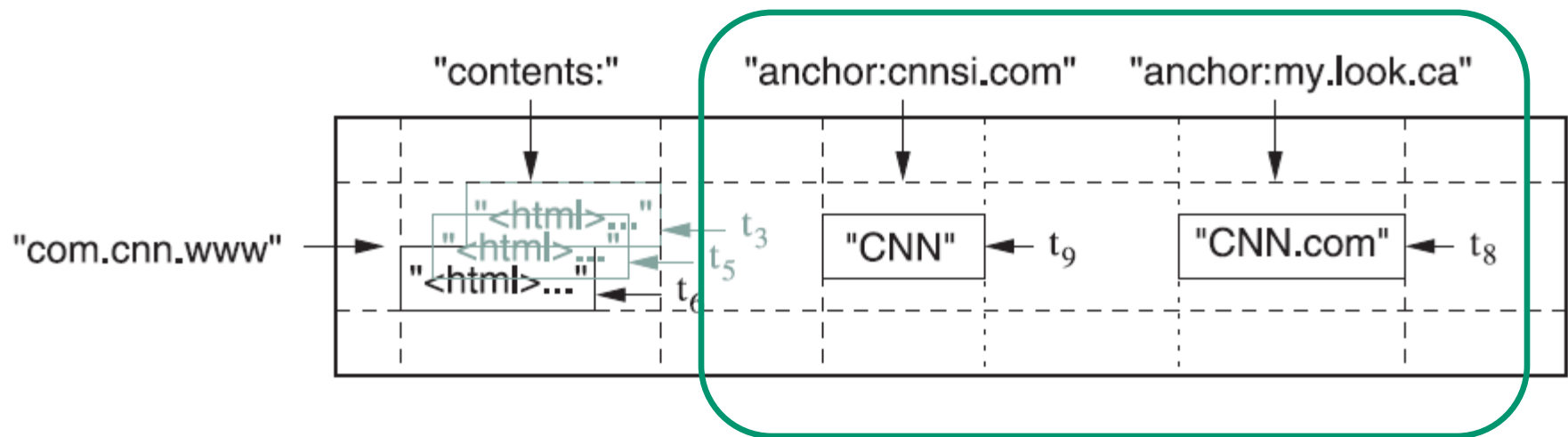


Tablet - Group of rows with consecutive keys.

Unit of Distribution

Bigtable maintains data in lexicographic order by row key

Data Model – WebTable Example (3)

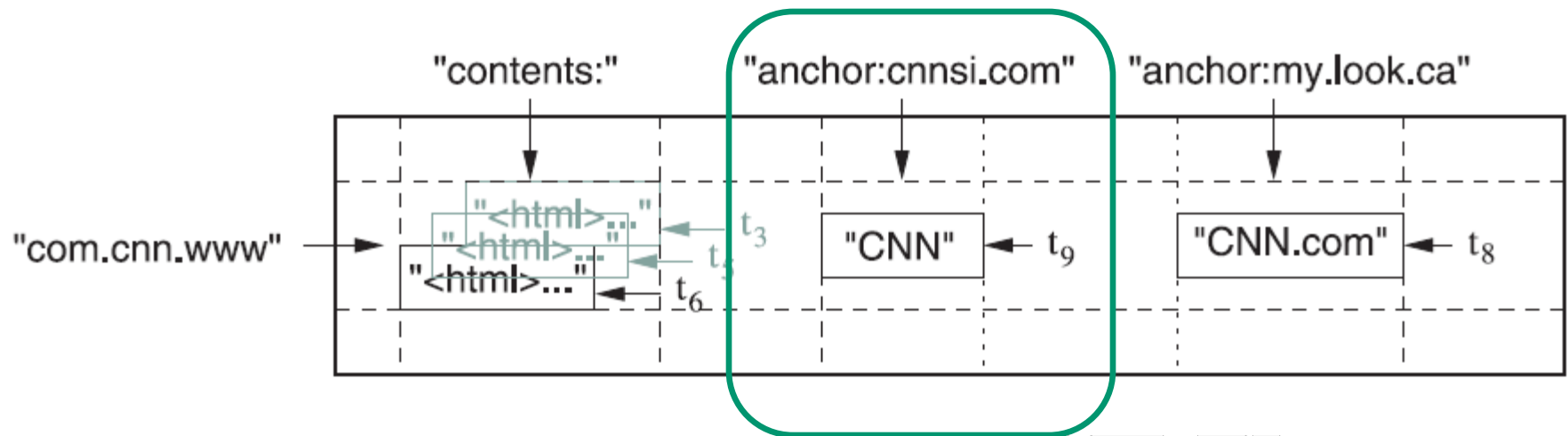


Column family is the unit of access control

**Column
Family**



Data Model – WebTable Example (4)



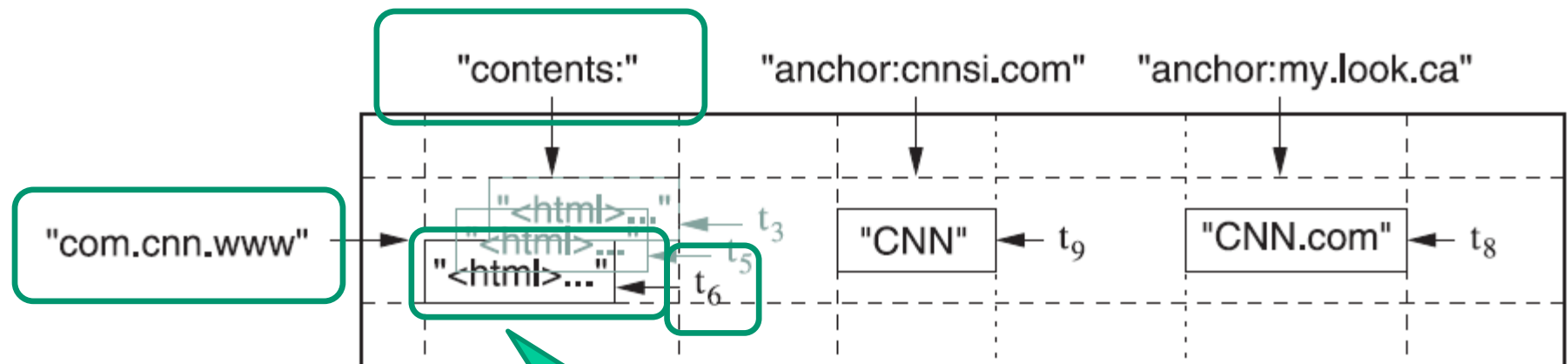
Column key is specified by
“**Column family:qualifier**”

Column

The diagram illustrates a sequence model processing a URL. The input "com.cnn.www" is processed by a "contents:" block, which outputs a sequence of tokens: "<html>...", "<html>...", and "<html>...". These tokens are then processed by an "anchor:cnnsi.com" block, which outputs "CNN". The "CNN" token is then processed by an "anchor:my.look.ca" block, which outputs "CNN.com". The final output is "CNN.com".

Column

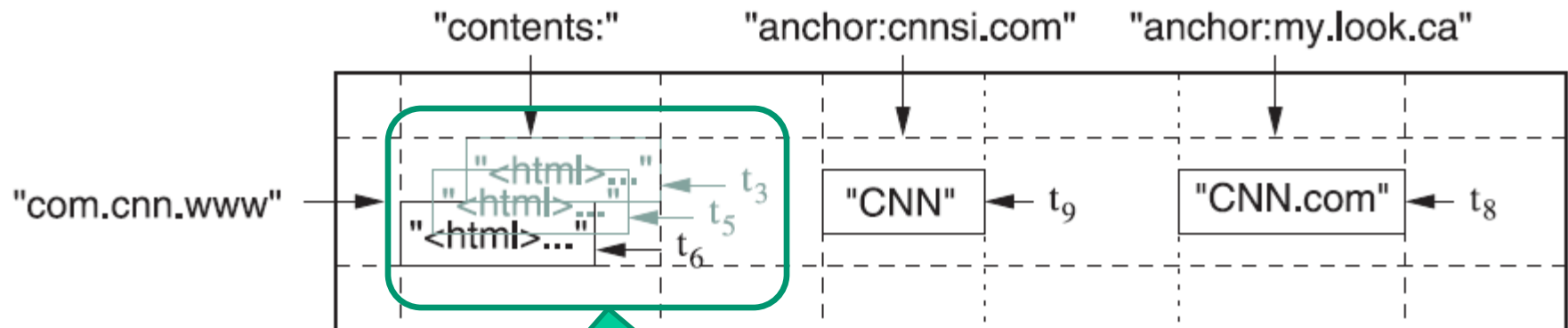
Data Model – WebTable Example (6)



Cell

Cell: the storage referenced by a particular **row key**, **column key**, and **timestamp**

Data Model – WebTable Example (7)



Different cells in a table
can contain multiple
versions indexed by
timestamp



BigTable is Used on many Real Applications

Project name	Table size (TB)	Compression ratio	# Cells (billions)	# Column Families	# Locality Groups	% in memory	Latency-sensitive?
<i>Crawl</i>	800	11%	1000	16	8	0%	No
<i>Crawl</i>	50	33%	200	2	2	0%	No
<i>Google Analytics</i>	20	29%	10	1	1	0%	Yes
<i>Google Analytics</i>	200	14%	80	1	1	0%	Yes
<i>Google Base</i>	2	31%	10	29	3	15%	Yes
<i>Google Earth</i>	0.5	64%	8	7	2	33%	Yes
<i>Google Earth</i>	70	–	9	8	3	0%	No
<i>Orkut</i>	9	–	0.9	8	5	1%	Yes
<i>Personalized Search</i>	4	47%	6	93	11	5%	Yes



References

- **Google Videos on map/reduce**
<https://www.youtube.com/watch?v=yjPBkvYh-ss> (Lecture 1, 46 min)
<https://www.youtube.com/watch?v=-vD6PUdf3Js> (Lecture 2, 52 min)
- **Wikipedia**, <http://en.wikipedia.org/wiki/MapReduce>
- *Data-Intensive Text Processing with MapReduce*, Jimmy Lin and Chris Dyer, Morgan & Claypool Synthesis Lectures on Human Language Technologies, 2010
<http://www.umiacs.umd.edu/~jimmylin/MapReduce-book-final.pdf>
- **Hadoop** is an open source implementation of MapReduce
<http://hadoop.apache.org/>
- MapReduce: Simplified Data Processing on Large Clusters, by Jeffrey Dean and Sanjay Ghemawat, <http://research.google.com/archive/mapreduce.html>



Other Useful Videos

- **How does Google search work?**
 - <https://www.youtube.com/watch?v=KyCYyoGusqs> (7 min)
- **How does Google decide when to display multiple results from the same website**
 - <https://www.youtube.com/watch?v=AGpEdyIcZcU> (6 min)*
- **Larry Page of the future directions of Google**
 - <http://www.youtube.com/watch?v=mArrNRWQEso> (3/2014, 23 min)
- **How Search Works by Matt Cutts**
 - <http://www.youtube.com/watch?v=BNHR6IQJGZs> (3 min)