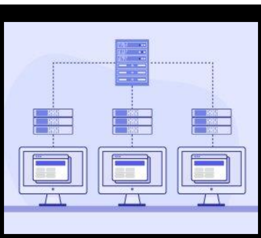
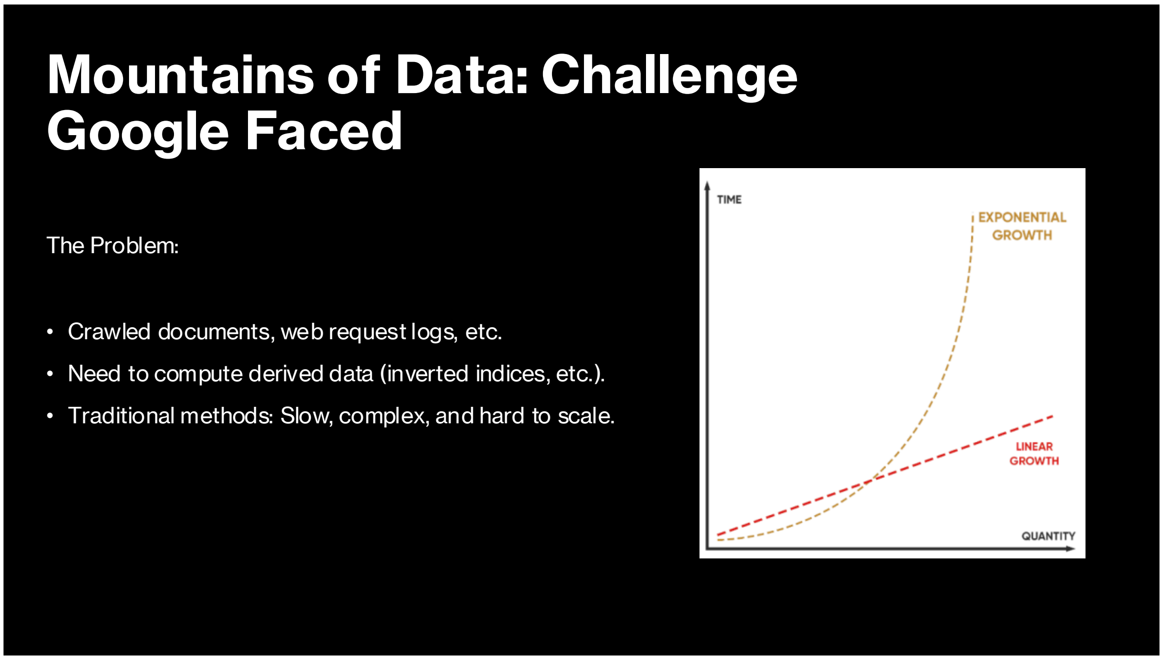
MapReduce Breakthrough Big Data



Author : Jeffrey Dean and Sanjay Ghemawat

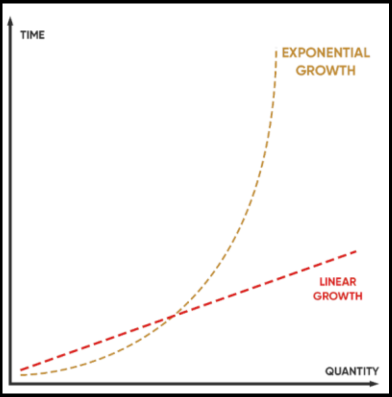
Presentation Prepared by:
Name : Mukesh Ravichandran
CWID : 50380788



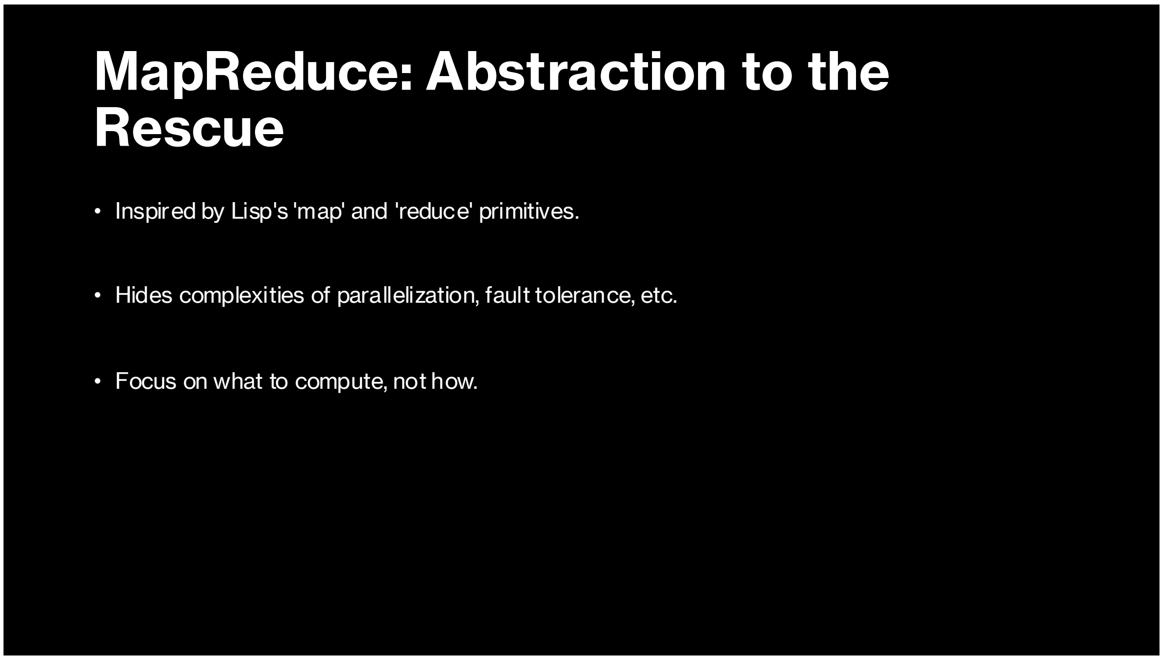
Mountains of Data: Challenge Google Faced

The Problem:

- Crawled documents, web request logs, etc.
- Need to compute derived data (inverted indices, etc.).
- Traditional methods: Slow, complex, and hard to scale.



Imagine Google in the early 2000s, grappling with an explosion of data—billions of web pages, countless user queries, and a growing wave of digital information. To refine their search engine, target ads effectively, and enhance their services, they needed a way to process vast amounts of data efficiently.
However, existing tools and techniques fell short. Handling such massive datasets was slow, cumbersome, and required extensive infrastructure management. The challenge wasn't just the computations themselves—many were conceptually simple—but rather the complexity of parallel processing, data distribution, and fault tolerance.
This is where MapReduce came in. It streamlined these intricate tasks, allowing Google to process data at an unprecedented scale while abstracting away the messiness of distributed computing



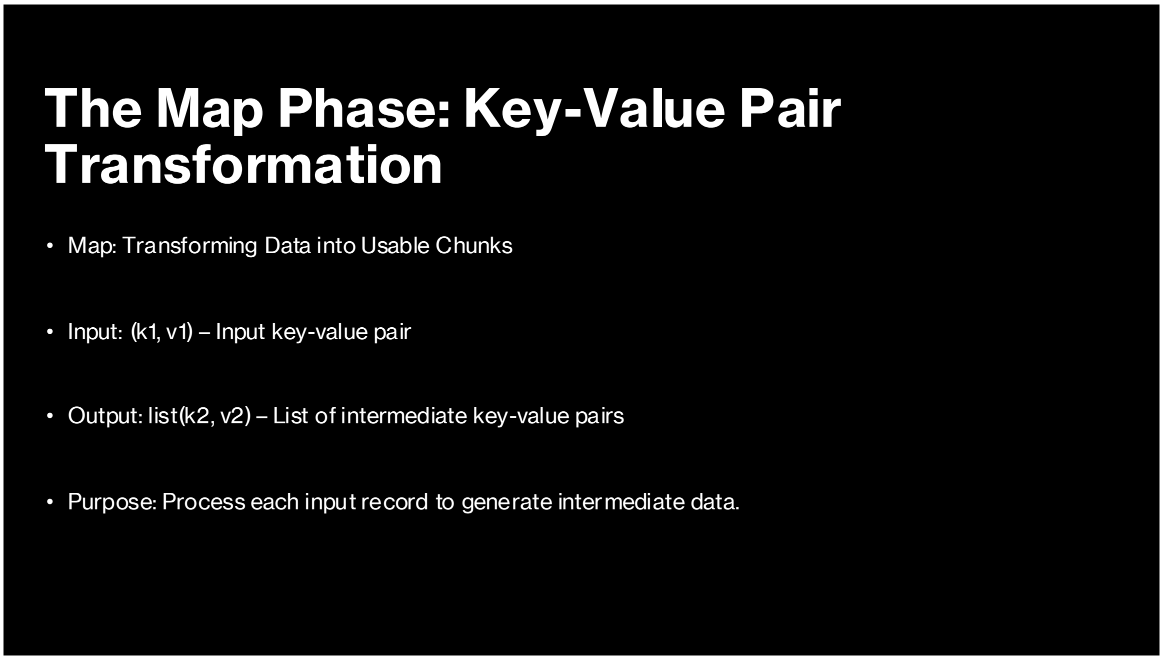
MapReduce: Abstraction to the Rescue

- Inspired by Lisp's 'map' and 'reduce' primitives.
- Hides complexities of parallelization, fault tolerance, etc.
- Focus on what to compute, not how.

To tackle this complexity, Google developed MapReduce, an abstraction inspired by the 'map' and 'reduce' functions in Lisp and other functional languages.

The key idea was to provide a simple programming model that hides all the messy details of parallelization, fault tolerance, data distribution, and load balancing. This allowed programmers to focus on what they wanted to compute, rather than how to compute it on a massive scale.

The paper highlights that MapReduce enables automatic parallelization and distribution of large-scale computations.



The Map Phase: Key-Value Pair Transformation

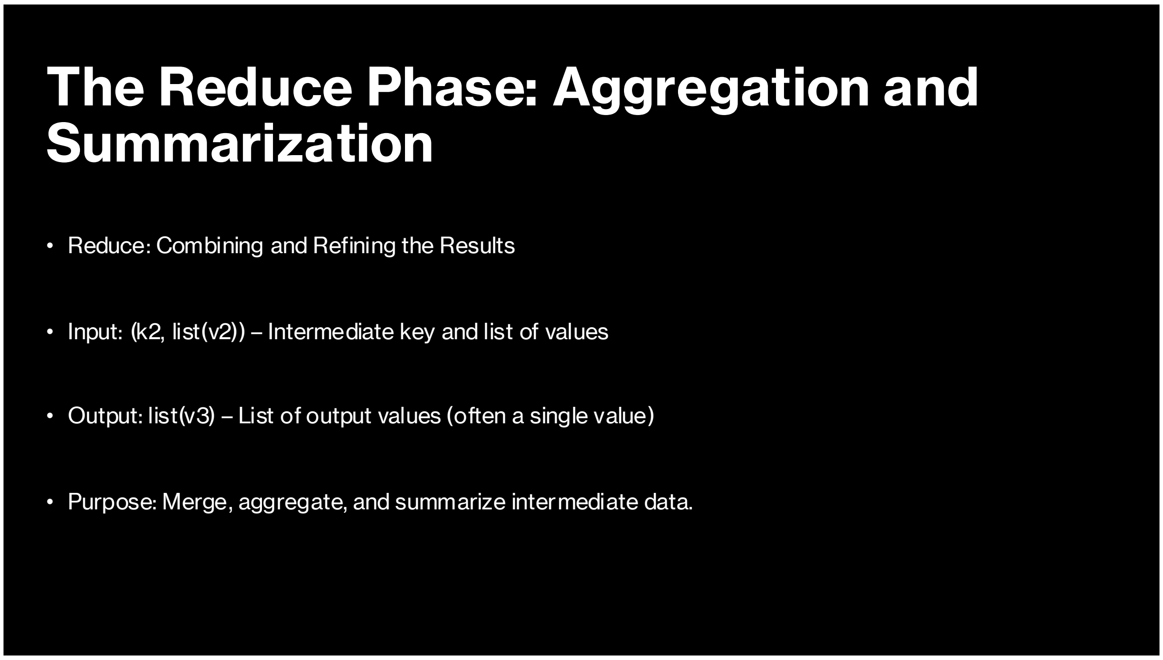
- Map: Transforming Data into Usable Chunks
- Input: (k1, v1) – Input key-value pair
- Output: list(k2, v2) – List of intermediate key-value pairs
- Purpose: Process each input record to generate intermediate data.

The Map phase is where the transformation begins. The user provides a Map function that takes an input key-value pair (k1, v1) and produces a list of intermediate key-value pairs (k2, v2).

Think of the input as the raw data and the output as processed, structured data, ready for aggregation.

The word count example in the paper is perfect here. Given a document, the map function extracts each word and emits a (word, "1") pair.

The paper mentions that strings are passed to and from the user-defined functions, so users must convert between strings and appropriate types. This offers flexibility.



The Reduce Phase: Aggregation and Summarization

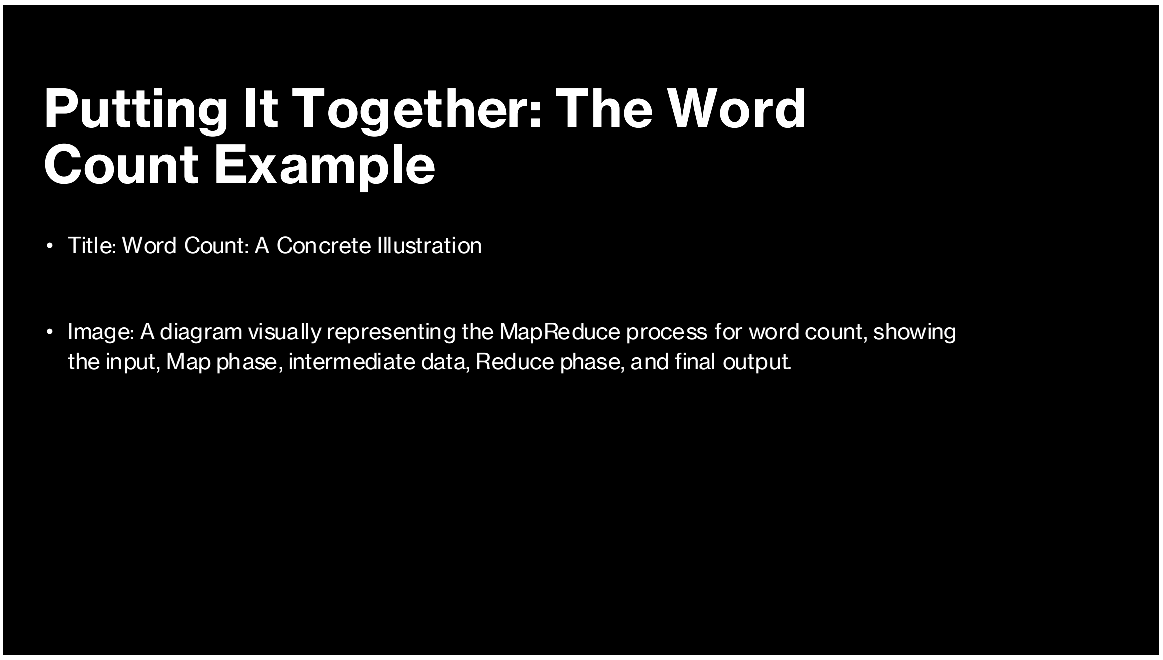
- Reduce: Combining and Refining the Results
- Input: (k2, list(v2)) – Intermediate key and list of values
- Output: list(v3) – List of output values (often a single value)
- Purpose: Merge, aggregate, and summarize intermediate data.

The Reduce phase takes the intermediate key-value pairs produced by the Map phase and combines the values associated with the same key.

The Reduce function takes an intermediate key (k2) and a list of values (list(v2)) and produces a list of output values (list(v3)).

In the word count example, the Reduce function sums up all the "1"s for each word, giving the total count for that word.

The paper specifies that the reduce function receives the intermediate values via an iterator, which allows the system to handle very large lists of values that do not fit in memory.

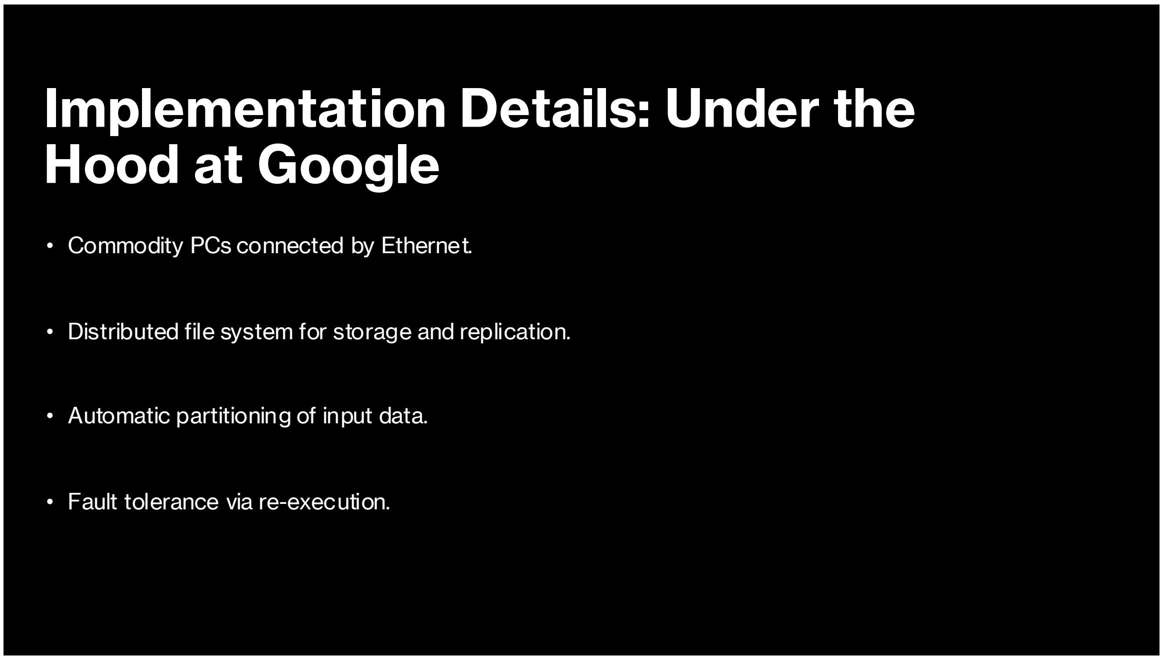


Putting It Together: The Word Count Example

- Title: Word Count: A Concrete Illustration
- Image: A diagram visually representing the MapReduce process for word count, showing the input, Map phase, intermediate data, Reduce phase, and final output.

Let's revisit the word count example to solidify our understanding. The Map function tokenizes each document and emits (word, "1"). The MapReduce framework groups these pairs by word, and the Reduce function sums the counts for each word.

The result is a list of (word, count) pairs, providing the frequency of each word in the entire document collection.



Implementation Details: Under the Hood at Google

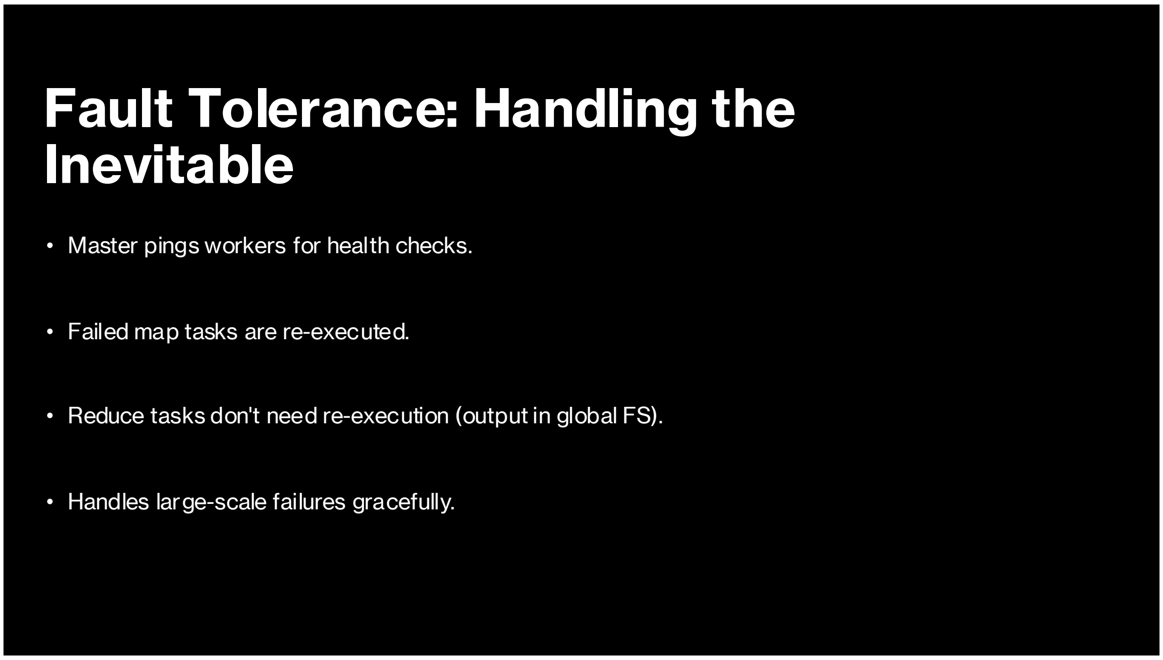
- Commodity PCs connected by Ethernet.
- Distributed file system for storage and replication.
- Automatic partitioning of input data.
- Fault tolerance via re-execution.

The paper describes Google's specific implementation of MapReduce, designed for their cluster environment.

They used commodity PCs connected by Ethernet, with a distributed file system for storage and replication. The system automatically partitions the input data into M splits and distributes the computation across the cluster.

Fault tolerance is a key feature. If a worker fails, the master re-executes the task on another worker. Completed map tasks are re-executed since the output is stored on the local disk of the failed machine.

The paper mentions that the master pings every worker periodically, and if there is no response, the worker is marked as failed.



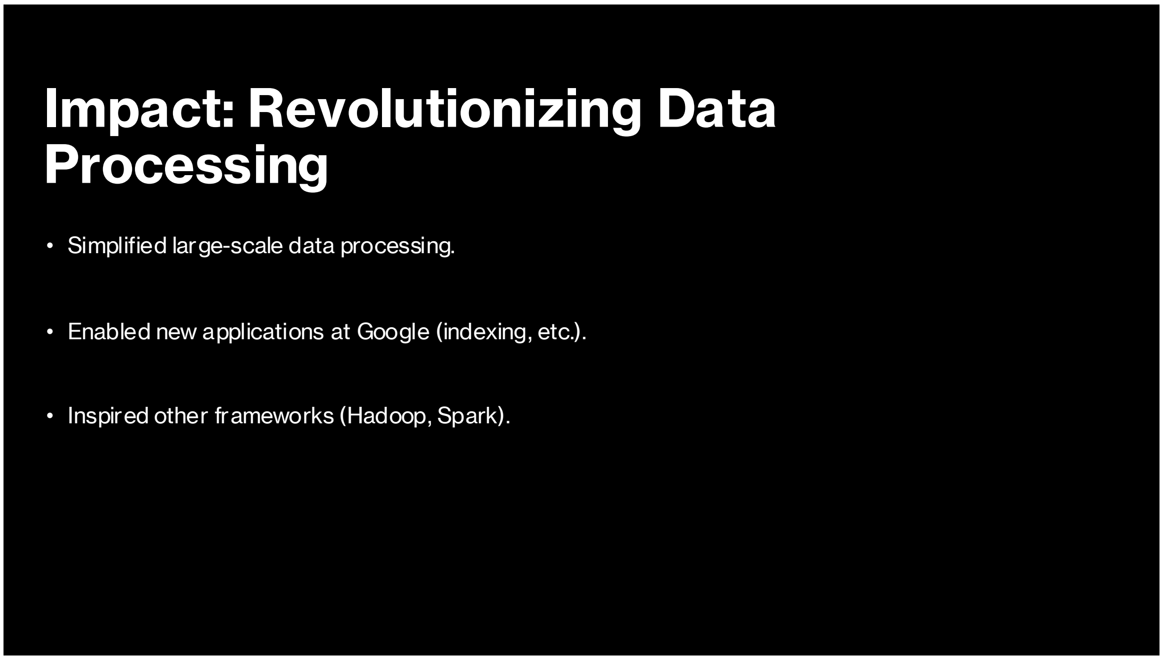
Fault Tolerance: Handling the Inevitable

- Master pings workers for health checks.
- Failed map tasks are re-executed.
- Reduce tasks don't need re-execution (output in global FS).
- Handles large-scale failures gracefully.

Because MapReduce is designed to run on a large number of machines, machine failures are inevitable. The system is designed to tolerate these failures gracefully.

The master node is responsible for monitoring the health of the worker nodes. If a worker node fails, the master node will re-assign any tasks that were running on that worker node to another worker node.

Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible.



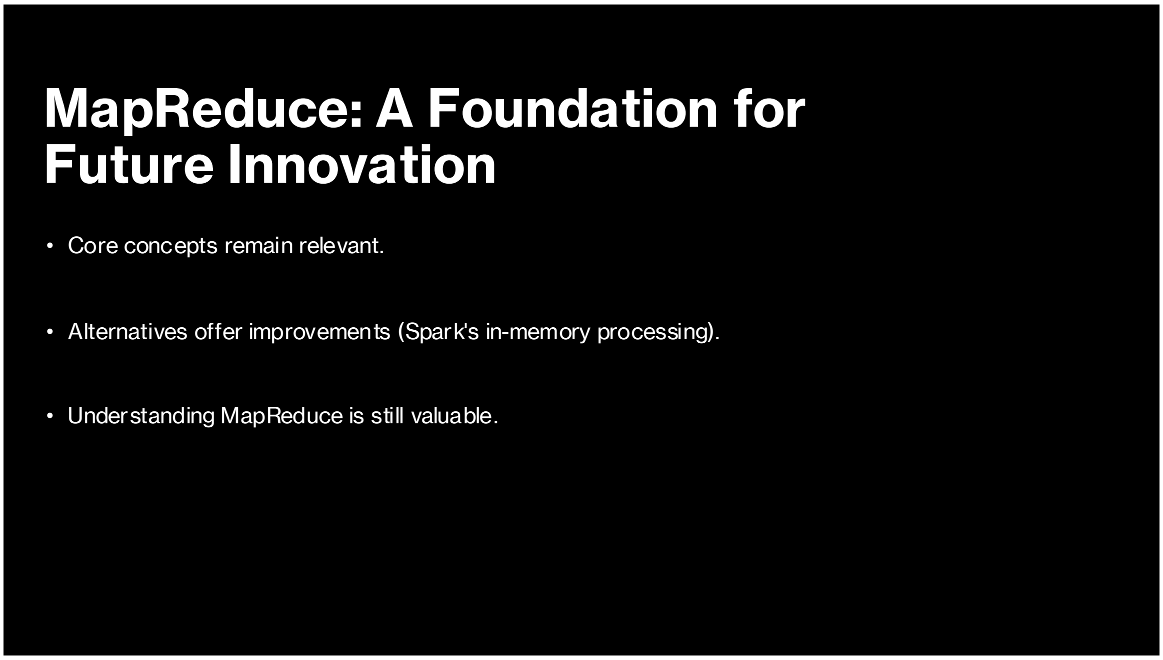
Impact: Revolutionizing Data Processing

- Simplified large-scale data processing.
- Enabled new applications at Google (indexing, etc.).
- Inspired other frameworks (Hadoop, Spark).

MapReduce had a profound impact on the field of data processing. It simplified the development of large-scale data processing applications and enabled new applications at Google.

It also inspired the development of other frameworks, such as Hadoop and Spark, which are widely used today.

While MapReduce may not be the best choice for every application today, its legacy is undeniable. It demonstrated the power of a simple, scalable, and fault-tolerant programming model for big data processing.



MapReduce: A Foundation for Future Innovation

- Core concepts remain relevant.
- Alternatives offer improvements (Spark's in-memory processing).
- Understanding MapReduce is still valuable.

MapReduce was a breakthrough, but technology has moved on. Frameworks like Spark offer in-memory processing and other advantages.

However, understanding MapReduce is still valuable. It provides a solid foundation for learning other big data technologies, and its core concepts remain relevant.