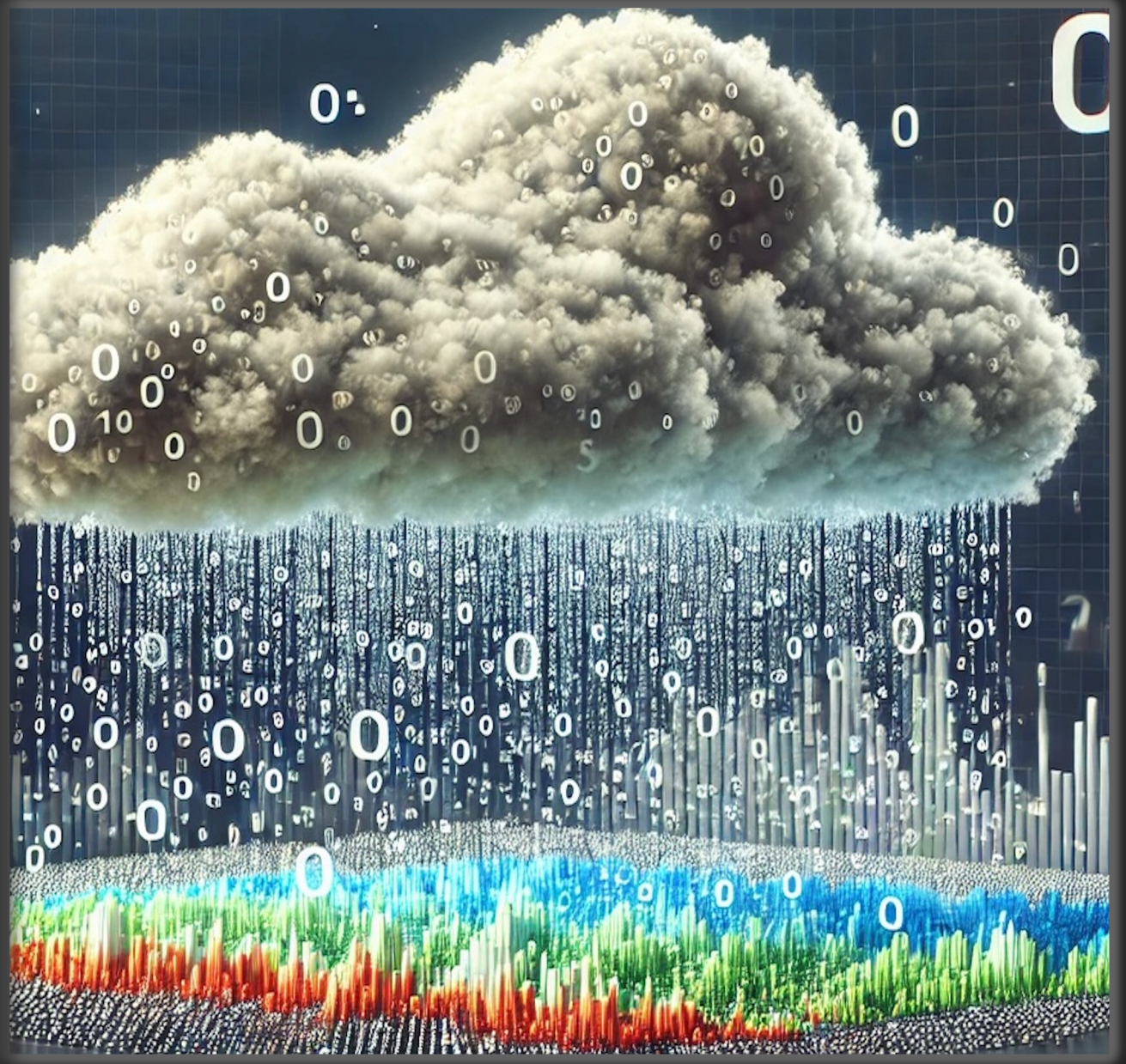
Carnegie Mellon University

Advanced Database Systems (15-721)

Lecture #15

Analytics at Scale: Dremel and BigQuery



Fall 2024

Prof. Jignesh Patel

Announcements

- Building Blocks Seminar: today, at 4:30 pm.
  Towards "Unified" Compute Engines: Opportunities and Challenges
  (Mehmet Ozan Kabak)
  https://db.cs.cmu.edu/events/
- Next lecture: Snowflake. The talk is over Zoom, and we will watch it in the classroom. I'll hang around after the lecture to answer any questions.
- Same for the lectures next week.

Google's Early Data Story

- At the turn of this century, Google had just about every data problem you can imagine: large volumes, real-time analytics, scalable OLTP, streaming … And on large volumes of data that was growing exponentially.
- Had to build their home-grown infrastructure, as existing systems did not scale to their needs.
- Early systems included GFS (File System), sharded MySQL (AdWords/OLTP/HTAP), MapReduce (data analytics), …
- All have evolved:
  Colossus cluster-level file system (Storage). The precursor, GFS, is deprecated.
  Spanner (OLTP): globally-consistent, scalable relational database.
  Big Query (OLAP): Built using Dremel as the execution engine.
  Borg: scalable job schedule -> influences Kubernetes.
  …

GOOGLE'S EARLY DATA STORY: NOSQL

- In early 2000's SQL was used in some parts (shared MySQL), but SQL was not seen as the way to interact with these structured and nested data.
- Scalable Key-Value stores and MapReduce were seen as the answer.

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber

{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

Google, Inc.

Abstract

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, Bigtable has successfully provided a flexible, high-performance solution for all of these Google products. In this paper we describe the simple data model provided by Bigtable, which gives clients dynamic control over data layout and format, and we describe the design and implementation of Bigtable.

1 Introduction

Over the last two and a half years we have designed, implemented, and deployed a distributed storage system for managing structured data at Google called Bigtable. Bigtable is designed to reliably scale to petabytes of data and thousands of machines. Bigtable has achieved several goals: wide applicability, scalability, high performance, and high availability. Bigtable is used by more than sixty Google products and projects, including Google Analytics, Google Finance, Orkut, Personalized Search, Writely, and Google Earth. These products use Bigtable for a variety of demanding workloads, which range from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. The Bigtable clusters used by these products span a wide range of configurations, from a handful to thousands of servers, and store up to several hundred terabytes of data.

In many ways, Bigtable resembles a database: it shares many implementation strategies with databases. Parallel databases [14] and main-memory databases [13] have achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and 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. Bigtable also treats data as uninterpreted strings, although clients often serialize various forms of structured and semi-structured data into these strings. Clients can control the locality of their data through careful choices in their schemas. Finally, Bigtable schema parameters let clients dynamically control whether to serve data out of memory or from disk.

Section 2 describes the data model in more detail, and Section 3 provides an overview of the client API. Section 4 briefly describes the underlying Google infrastructure on which Bigtable depends. Section 5 describes the fundamentals of the Bigtable implementation, and Section 6 describes some of the refinements that we made to improve Bigtable's performance. Section 7 provides measurements of Bigtable's performance. We describe several examples of how Bigtable is used at Google in Section 8, and discuss some lessons we learned in designing and supporting Bigtable in Section 9. Finally, Section 10 describes related work, and Section 11 presents our conclusions.

2 Data Model

A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.

(row:string, column:string, time:int64) → string

Google Today: SQL Everywhere

- SQL is critical across the Google data platforms, including Spanner (OLTP), BigQuery (OLAP), and BigTable (Key-value store).
- GoogleSQL: Complies with the ANSI SQL standard.

Bigtable transforms the developer experience with SQL support

August 2, 2024

Christopher Crosbie
Group Product Manager, Google

Gary Elliott
Engineering Manager, Bigtable

Bigtable is a fast, flexible, NoSQL database that powers core Google services such as Search, Ads, and YouTube, as well as critical applications for customers such as PLAID and Mercari. Today, we're announcing Bigtable support for GoogleSQL, an ANSI-compliant SQL dialect used by Google products such as Spanner and BigQuery. Now you can use the same SQL with Bigtable to write applications for AI, fraud detection, data mesh, recommendations, or any other application that would benefit from real-time data.

Bigtable SQL support allows you to query Bigtable data using the familiar GoogleSQL syntax, making it easier for development teams to work with Bigtable's flexibility and speed. With over 100 SQL functions at launch, Bigtable SQL support also makes it easy to analyze and process large amounts of data directly within Bigtable, unlocking its potential for a wider range of use cases, ranging from JSON manipulation for log analysis, hyperloglog for web analytics, or kNN for vector search and generative AI.

GOOGLE TODAY: SQL EVERYWHERE

- Some new ideas that clean up the SQL syntax.
- Example: Calculate the average sales for items that have above-average total sales.

Traditional way, with CTEs.

```
WITH item_sales AS (
  SELECT item, SUM(sales) AS total_sales
  FROM mydataset.produce
  GROUP BY item
),
avg_sales AS (
  SELECT AVG(total_sales) AS overall_avg
  FROM item_sales
)
SELECT item_sales.item, item_sales.total_sales
FROM item_sales, avg_sales
WHERE item_sales.total_sales > avg_sales.overall_avg;
```

With the new pipe syntax.

```
FROM mydataset.produce
|> AGGREGATE SUM(sales) AS total_sales GROUP BY item
|> EXTEND (SELECT AVG(total_sales) FROM UNNEST) AS overall_avg
|> WHERE total_sales > overall_avg
|> SELECT item, total_sales;
```

Shute et al.: SQL has problems. We can fix them: Pipe syntax in SQL. Proc. VLDB Endow, 2024.

Dremel

- Nested records. (Recall the discussion on extensible types in database systems.)
- Large data volumes.
- Need to scale.
- Need to be fault tolerant.

Nested Records

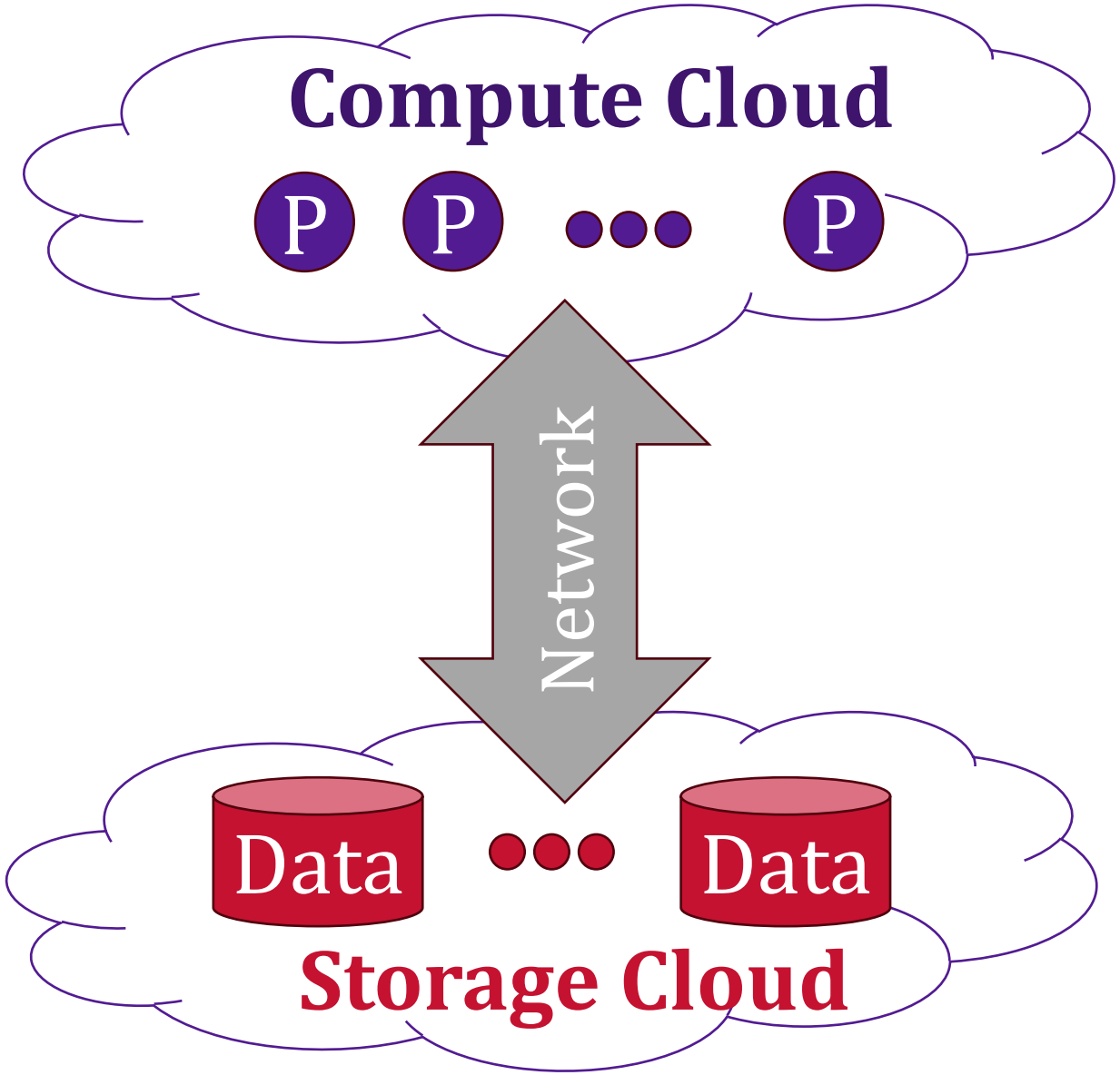
- Need to go beyond flat tables, and in-practice need a nested data model.
- Protocol Buffers (protobufs) are ubiquitious across Google, and effectively their record-level data model.
  - This idea carried over to similar data formats including Thrift (Facebook), and Avro (Hadoop).

```
{
  "order_id": 1001,
  "customer": {
    "customer_id": 5001,
    "name": "John Doe",
    "email": "johndoe@example.com",
    "address": {
      "street": "123 Elm St",
      "city": "Springfield",
      "state": "IL",
      "zip_code": "62701"
    }
  },
  "items": [
    {
      "item_id": 1,
      "name": "Laptop",
      "quantity": 1,
      "price": 999.99
    },
    {
      "item_id": 2,
      "name": "Wireless Mouse",
      "quantity": 2,
      "price": 19.99
    },
    {
      "item_id": 3,
      "name": "Keyboard",
      "quantity": 1,
      "price": 49.99
    }
  ],
  "order_date": "2024-10-30",
  "total_amount": 1089.96
}
```

```
syntax = "proto3";

package order;

// Define a message to represent an Order
message Order {
  int32 order_id = 1;
  Customer customer = 2;
  repeated OrderItem items = 3;
  string order_date = 4;
  double total_amount = 5;
}

// Define a nested message for Customer
message Customer {
  int32 customer_id = 1;
  string name = 2;
  string email = 3;
  Address address = 4;  // Nested customer address
}

// Define a nested message for OrderItem
message OrderItem {
  int32 item_id = 1;
  string name = 2;
  int32 quantity = 3;
  double price = 4;
}

// Define a nested message for Address
message Address {
  string street = 1;
  string city = 2;
  string state = 3;
  string zip_code = 4;
}
```

Dremel: Key Ideas

- Disaggregated storage (now called Lakehouse).
- Columnar storage even for nested data (now part of Parquet).
- Efficient query execution with a special shuffle infrastructure.



Nested Records and Querying

A Web Document record schema in protobuf (v1) for a web crawl.

```
DocId: 10                    r1
Links
  Forward: 20
  Forward: 40
  Forward: 60
Name
  Language
    Code: 'en-us'
    Country: 'us'
  Language
    Code: 'en'
  Url: 'http://A'
Name
  Url: 'http://B'
Name
  Language
    Code: 'en-gb'
    Country: 'gb'
```

```
message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward; }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country; }
    optional string Url; }}
```

```
DocId: 20                    r2
Links
  Backward: 10
  Backward: 30
  Forward:  80
Name
  Url: 'http://C'
```

User may want to refer to a nested field, such as Name.Language.Code in their queries.

Melnik et al.: Dremel: Interactive Analysis of Web-Scale Datasets. Proc. VLDB Endow. 3(1): 330-339 (2010)

Nested Records and Querying

A Web Document record schema in protobuf (v1) for a web crawl.

r_1

```
DocId: 10
Links
  Forward: 20
  Forward: 40
  Forward: 60
Name
  Language
    Code: 'en-us'
    Country: 'us'
  Language
    Code: 'en'
  Url: 'http://A'
Name
  Url: 'http://B'
Name
  Language
    Code: 'en-gb'
    Country: 'gb'
```

```
message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward; }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country; }
    optional string Url; }}
```

r_2

```
DocId: 20
Links
  Backward: 10
  Backward: 30
  Forward:  80
Name
  Url: 'http://C'
```

Columnar storage representation with repetition levels (r) and definition levels (d).

DocId

value	r	d
10	0	0
20	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2
80	0	2

Links.Backward

value	r	d
NULL	0	1
10	0	2
30	1	2

Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

Name.Language.Country

value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3
NULL	0	1

Melnik et al.: Dremel: Interactive Analysis of Web-Scale Datasets. Proc. VLDB Endow. 3(1): 330-339 (2010)

Nested Records and Querying

- SQL with nesting.
- Operators take as input one or more nested tables, and outputs a nested table (and the output schema).
- Notice the use of path expressions, e.g., Name.Language.Code and Name.Url, is allowed in the query.
- Also notice the within-record aggregation (COUNT).
- Model: nested record == a labeled tree. Selection prunes branches.

A Web Document in protobuf (v1), and a sample query.

```
SELECT DocId AS Id,
  COUNT(Name.Language.Code) WITHIN Name AS Cnt,
  Name.Url + ',' + Name.Language.Code AS Str
FROM t
WHERE REGEXP(Name.Url, '^http') AND DocId < 20;
```

t_1

```
Id: 10
Name
  Cnt: 2
  Language
    Str: 'http://A,en-us'
    Str: 'http://A,en'
Name
  Cnt: 0
```

```
message QueryResult {
  required int64 Id;
  repeated group Name {
    optional uint64 Cnt;
    repeated group Language {
      optional string Str; }}}
```

In the original version of Dremel, queries were 1-pass scan on a table, and aggregation (no joins).

Melnik et al.: Dremel: Interactive Analysis of Web-Scale Datasets. Proc. VLDB Endow. 3(1): 330-339 (2010)

Move towards elastic computing.

Data can be computed by multiple compute engines.
Don't lock the data format to a specific compute engine.
e.g.; the same data may be consumed via SQL, dataframes, MapReduce, …

Google's Dremel was one of the first systems that combined a set of architectural principles that have become a common practice in today's cloud-native analytics tools, including disaggregated storage and compute, in situ analysis, and columnar storage for semistructured data. In this paper, we discuss how these ideas evolved in the past decade and became the foundation for Google BigQuery.

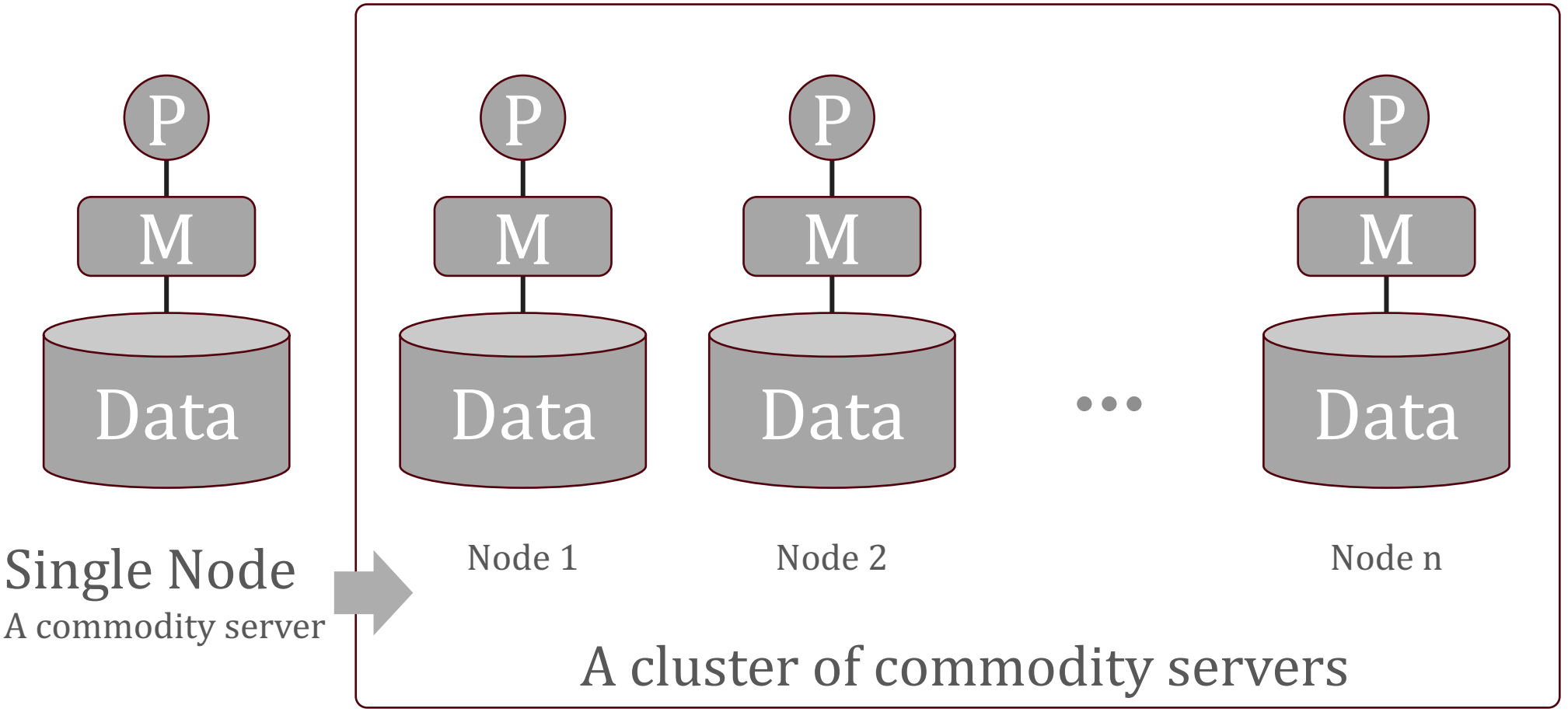
Columnar storage even for nested data.

Serverless: no-upfront provisioning.
Pay-as-you-go consumption model.

Also, adopted SQL as the query language.
(Moving away from Sawzall, which was a syntactic sugar layer over MapReduce.)

DREMEL ON A SHARED NOTHING ARCHITECTURE

- Till ~2009 data was managed in a cluster of commodity servers

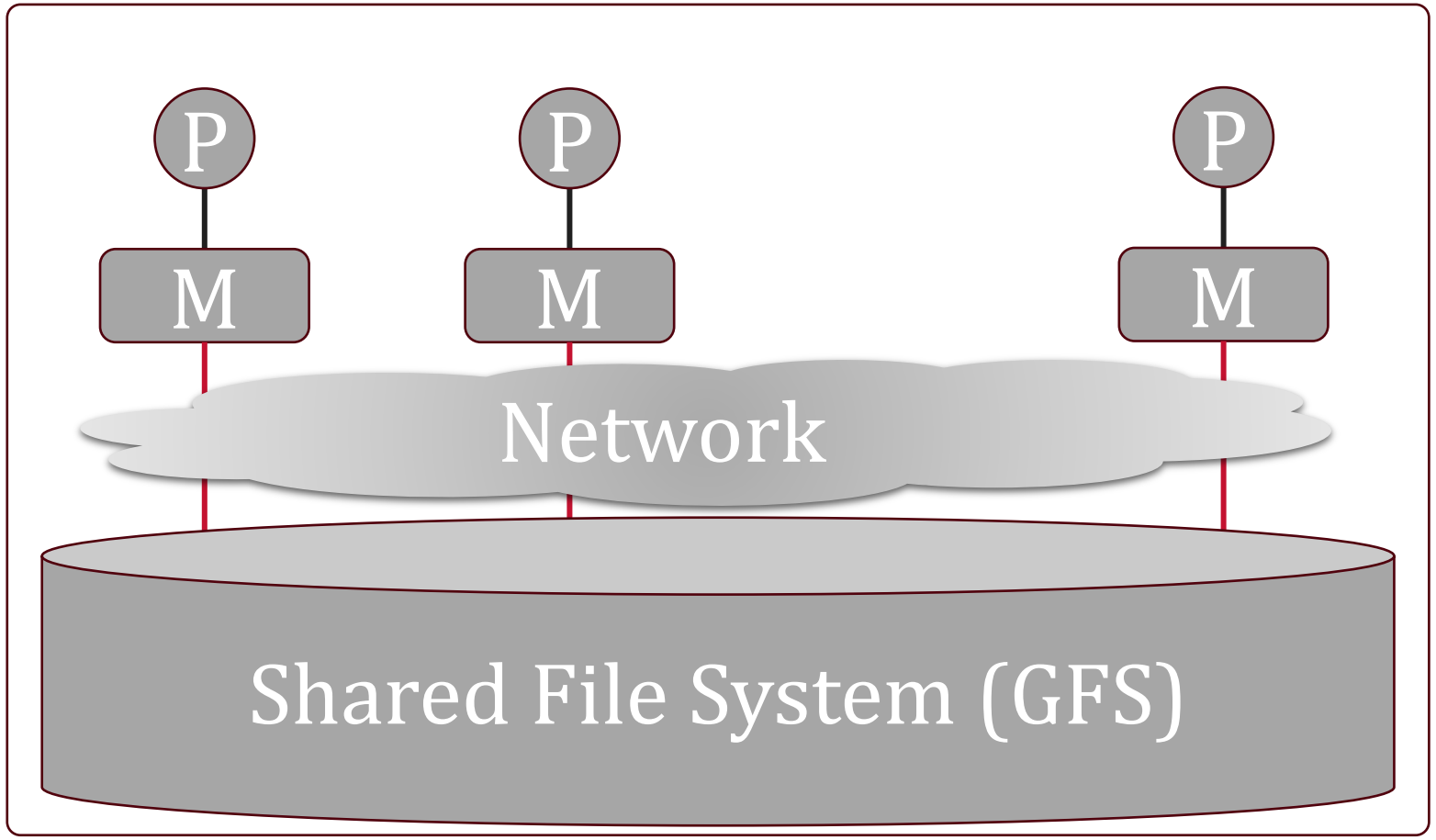


Disaggregated Architecture

- Till ~2009 data was managed in a cluster of commodity servers.
- Then Borg – a cluster management system – was introduced. (Precursor to Kubernetes).
- Why a cluster management system?
  1. Share the hardware across different platforms to improve utilization.
  2. Grow/shrink the cluster to deal with changes in workload.

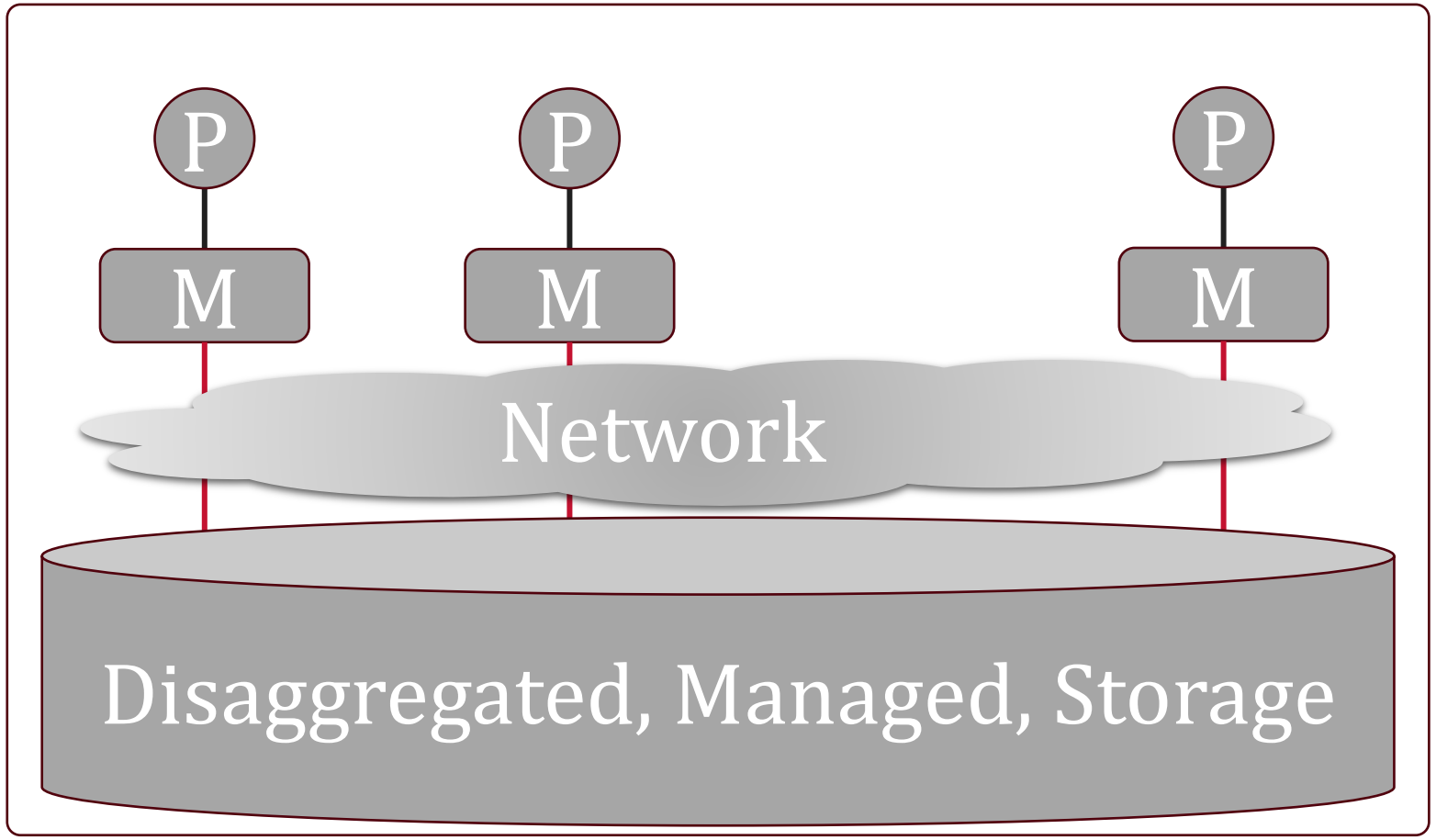
  Elasticity
- Now the hard disk (spindle) were shared by Dremel with other platforms.
  - Replicate the data (on local disks) for performance and fault-tolerance.
  - The algorithms now have to be replication aware → More complex development.
  - Also, resizing the system means having to move replicas to balance the system.
- The need for disaggregate storage starts to emerge.

Move from Shared-Nothing to Shared-Disk



- After moving to the shared file system (GFS), the system was much slower than on the shared nothing.
  - A table scan may require opening 100K+ files in GFS
  - Metadata access was also slow.
- Tune: storage format, metadata, query affinity, prefetching …

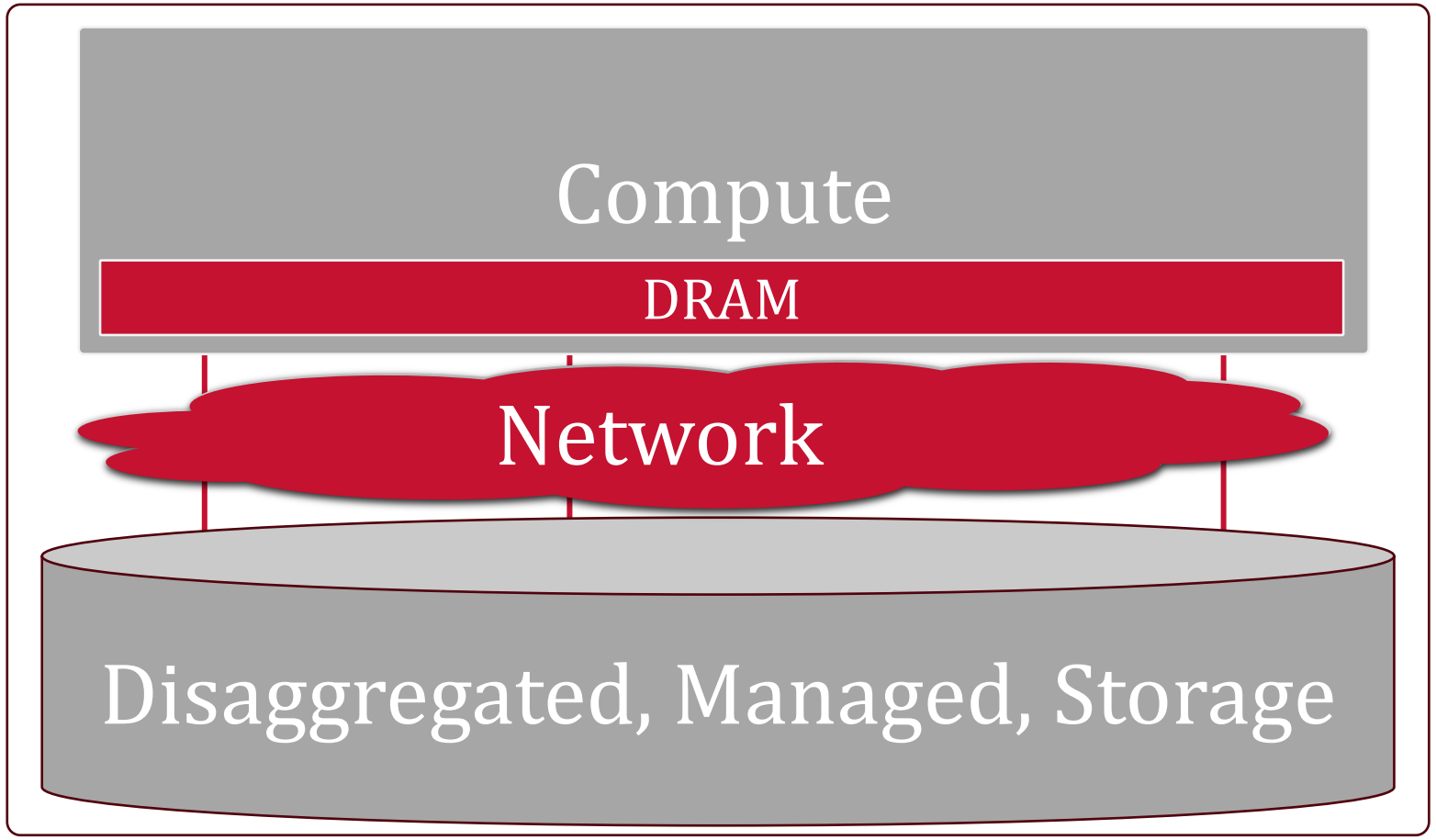
Move from Shared-Nothing to Disaggregated Storage



Advantages of managed storage:

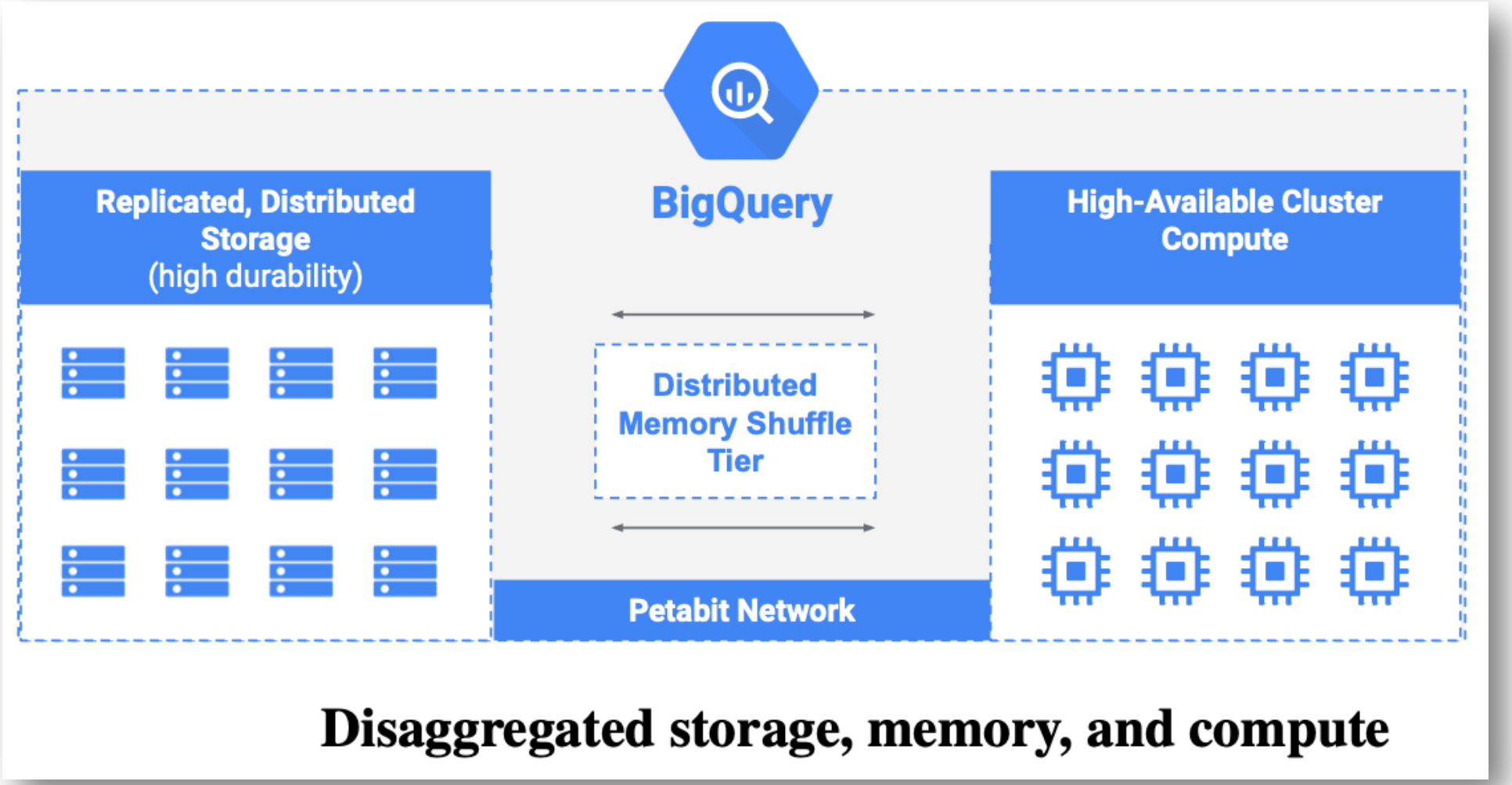
- SLOs in the storage are now **not** a responsibility of the Dremel team (a clean division of team responsibilities).
- Easier to "resize" the system to add new databases – just ask for more from the storage service.
- The whole system gets more robust as the storage system gets more robust.

Move to a Disaggregated Architecture



- Dremel initially did not have joins – it was used mainly for aggregate queries.
- As joins got added, need to add support to partition the inputs, aka. shuffle (in MapReduce parlance).
- Shuffle puts a lot of pressure on the DRAM and the network. Want to abstract the compute side too, especially for shuffle.

Move to Fully Disaggregated Architecture

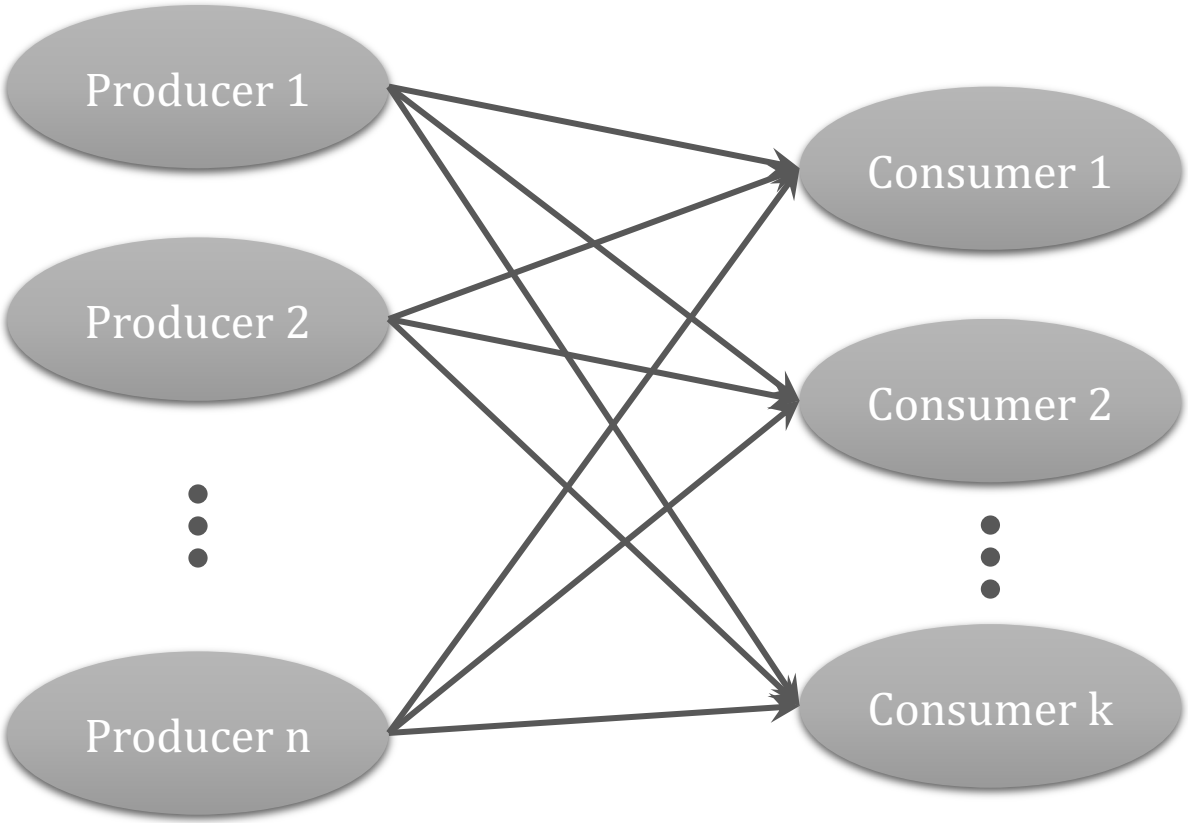


A new component in the disaggregated architecture: the memory shuffle tier.

This tier has memory and disk space to store intermediate shuffle data.

A TYPICAL SHUFFLE OPERATOR

- Remember Exchange … essentially that.



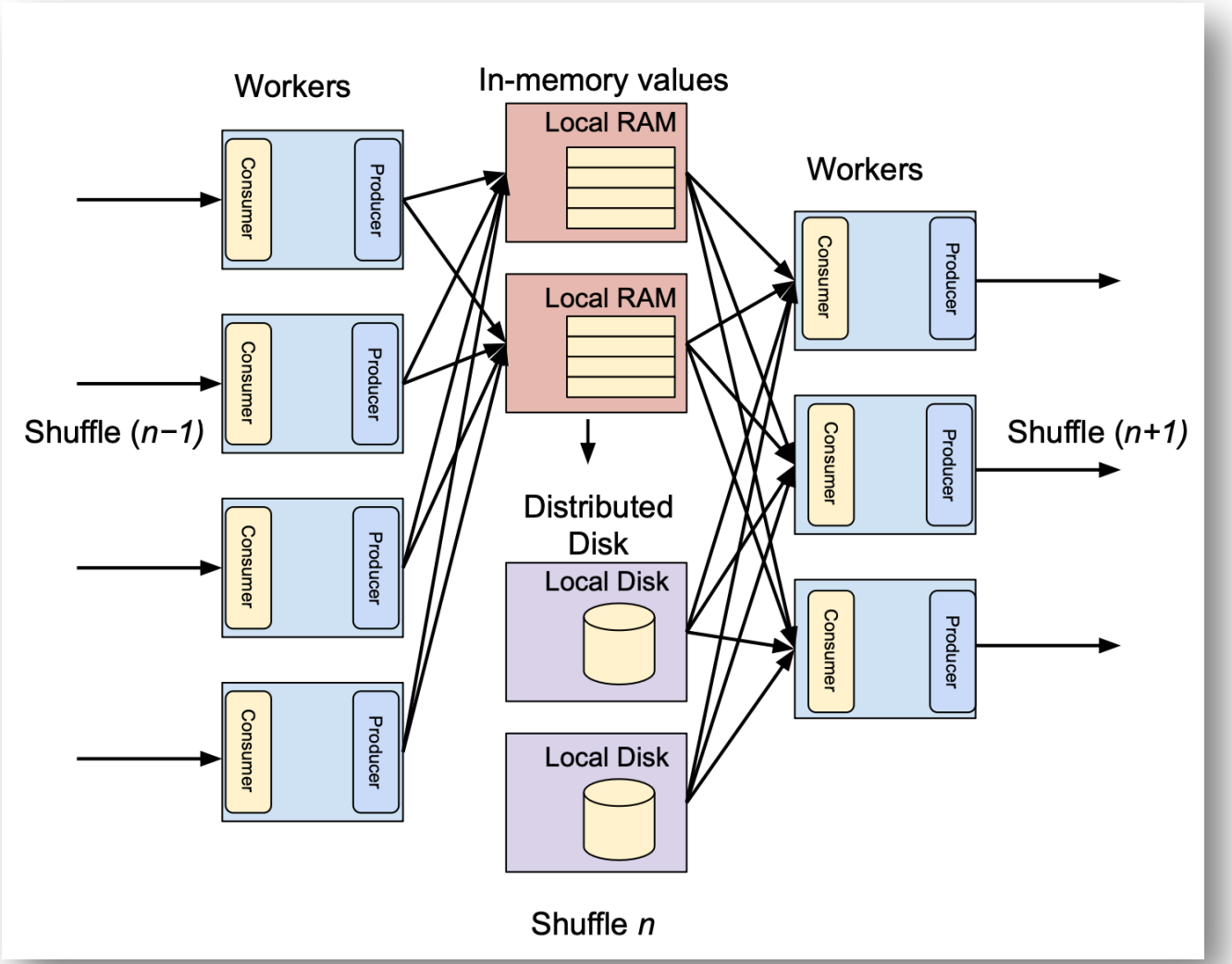
Two key issues with a typical shuffle:

1. The communication is $O(n^2)$.
2. MapReduce Shuffle: The shuffle has to complete before consumers can start.

   Was a big deal as MapReduce dominated the big data space for (way) too long.

https://cloud.google.com/blog/products/bigquery/in-memory-query-execution-in-google-bigquery

A TYPICAL SHUFFLE OPERATOR



Specialized shuffle infrastructure in Dremel/BigQuery.

Can optimize the memory/disk requirement for the shuffle infrastructure.

CLX can potentially help in the future?

https://cloud.google.com/blog/products/bigquery/in-memory-query-execution-in-google-bigquery

In situ Data Analysis

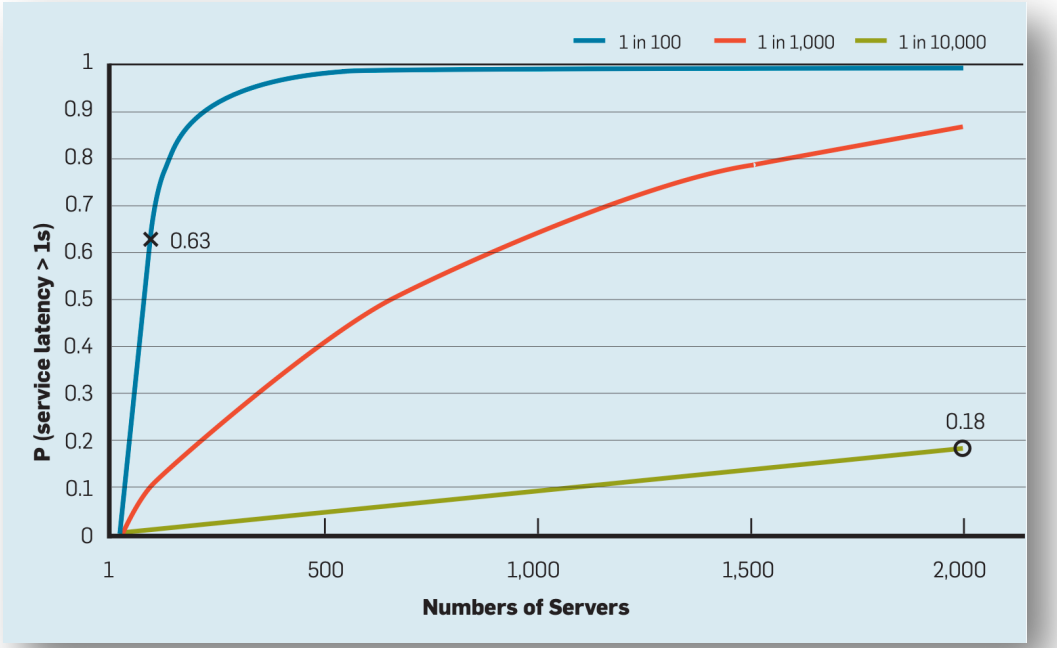
- Instead of loading data into the warehouse than then querying the data, what if the initial data was self-descriptive?
- Now can simply run queries on this data – no explicit ETL.
- There may be extra overhead, but the flexibility makes this worth while.
- Also, once the data engine does not need to "own the data," there is bigger emphasis on building ways to bring in external data via wrappers, or APIs to other data sources. This is a federated data systems now.

New issue: Data Governance.

New issue: QO may have no stats on data.

STRAGGLERS

- When you have 100s or 1000s of servers for a single query, high chance that some worker may fail or fall behind.

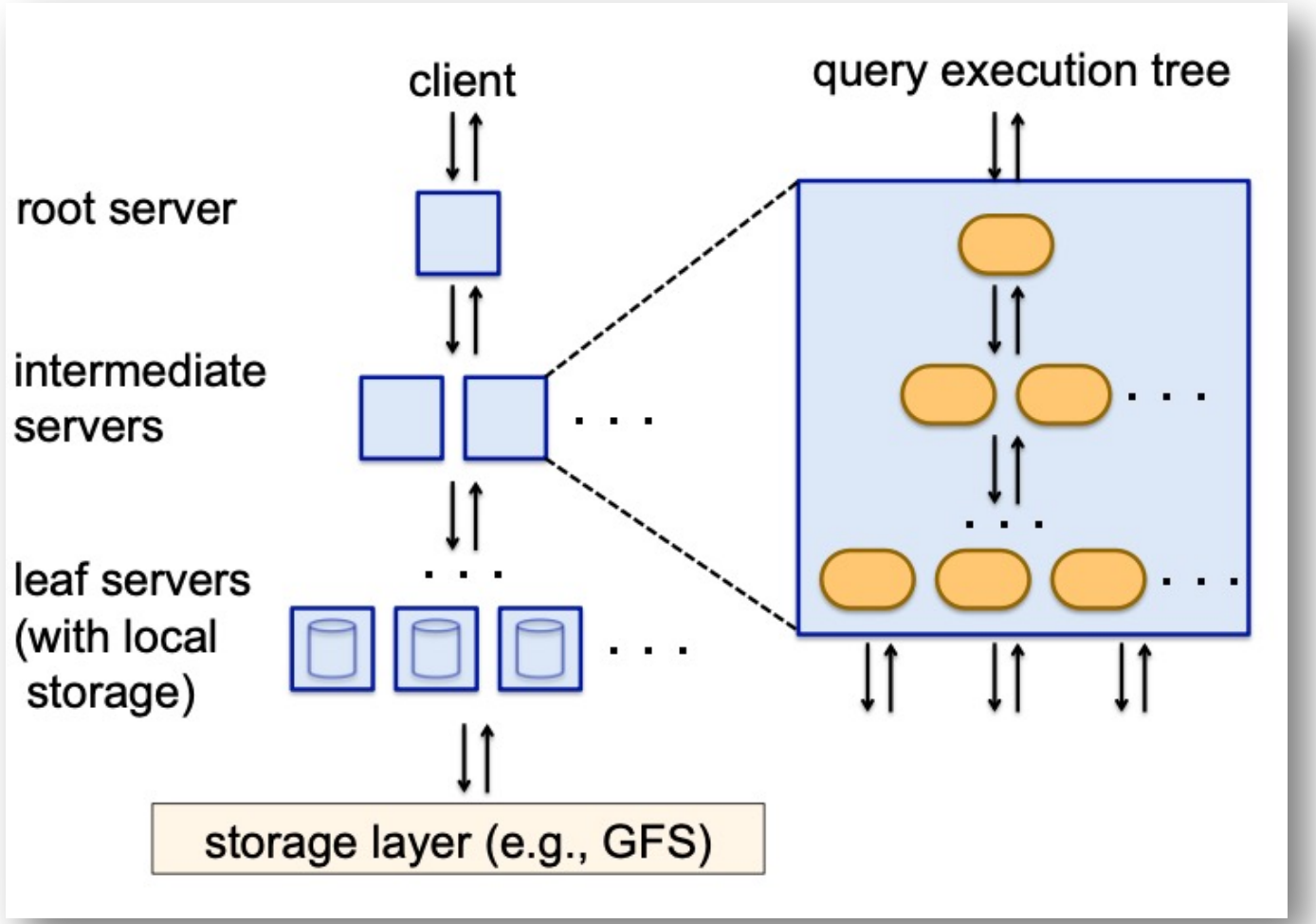


Jeffrey Dean, Luiz André Barroso: The tail at scale. Commun. ACM, 2013.

- For a server in a service, assume a 10ms response time but 99th percentile latency of 1 sec.
- If a task needs 100 server, 63% requests will need more than 1 sec., i.e. end up on a server that has a service time of 1 sec.
- Ways out:
  1. Detect a straggler, and assign that work to another worker. Need the task to be idempotent.
  2. Speculate and duplicate tasks, and when one of them completes, kill the other task.

Query Execution in Dremel

- Coordinator: Receives queries and uses a multi-level serving tree.
- Also, the root server can query the metadata server, instead of each leaf server doing that independently. The latter can overload the metadata server when the query first starts.



Original source of the following slides: Andy Pavlo

DREMEL: STORAGE

- DBMS relies on Google's distributed file system (Colossus) to scale out storage capacity.
- Relies on Capacitor's columnar encoding scheme for nested relational and semi-structured data.
  - Think of it as JSON/YAML without the slowness.
  - Capacitor also provides access libraries with basic filtering.
  - Similar to Parquet vs. ORC formats.
- Repetition and definition fields are embedded in columns to avoid having to retrieve/access ancestor attributes.

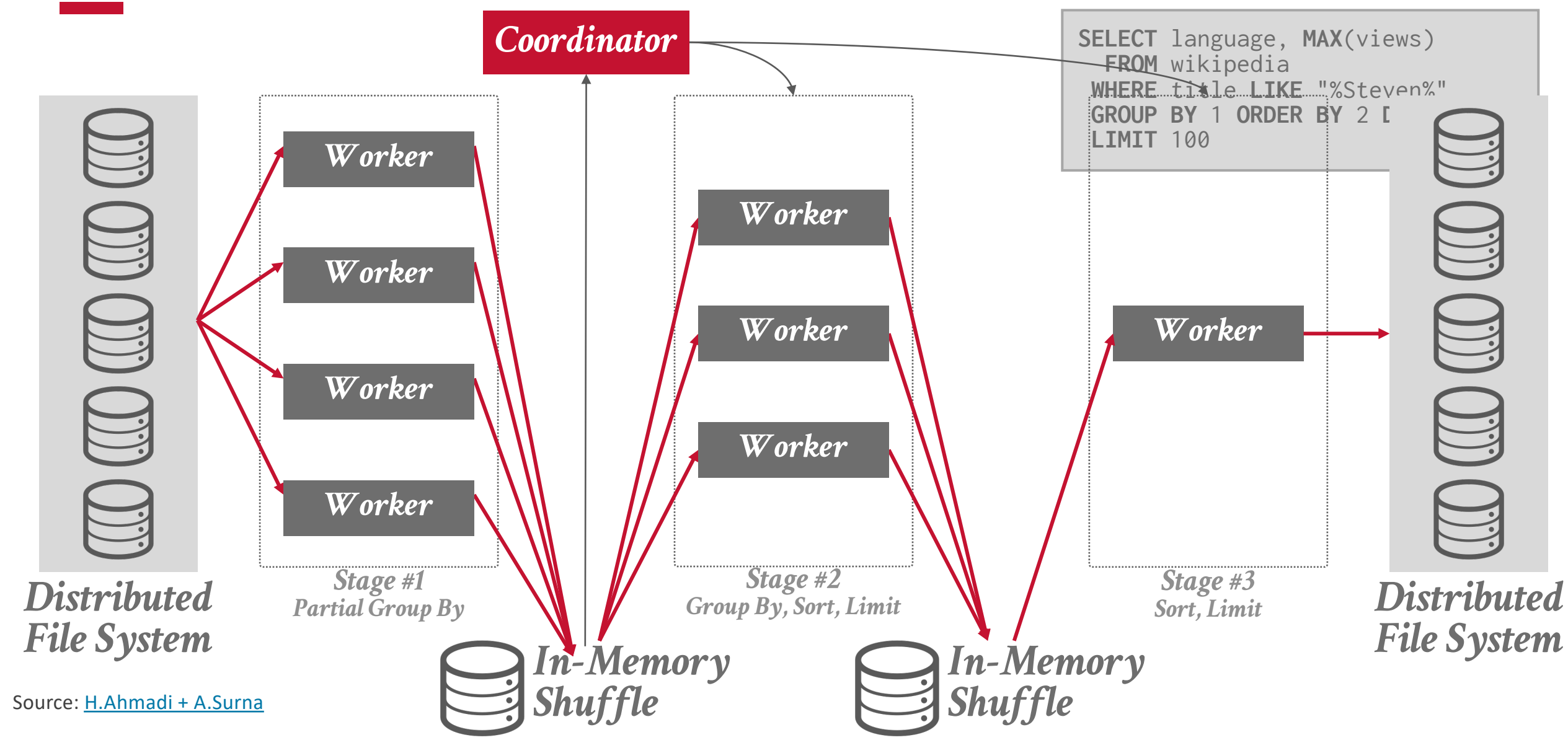
DREMEL: SCHEMA REPRESENTATION

- Dremel's internal storage format is self-describing
  - Everything the DBMS needs to understand what is in a file is contain within the file.
- But the DBMS must parse a file's embedded schema whenever it wants to read that a file.
  - Tables can have thousands of attributes. Most queries only need a subset of attributes.
- DBMS stores schemas in a columnar format to reduce overhead when retrieving meta-data.

DREMEL: QUERY EXECUTION

- DBMS converts a logical plan into **stages** (pipelines) that contain multiple parallel **tasks**.
  - Each task must be deterministic and idempotent to support restarts.
- Root node (Coordinator) retrieves all the meta-data for target files in a batch and then embeds it in the query plan.
- Each worker node has its own local memory and can spill to local disk if needed.

DREMEL: QUERY EXECUTION

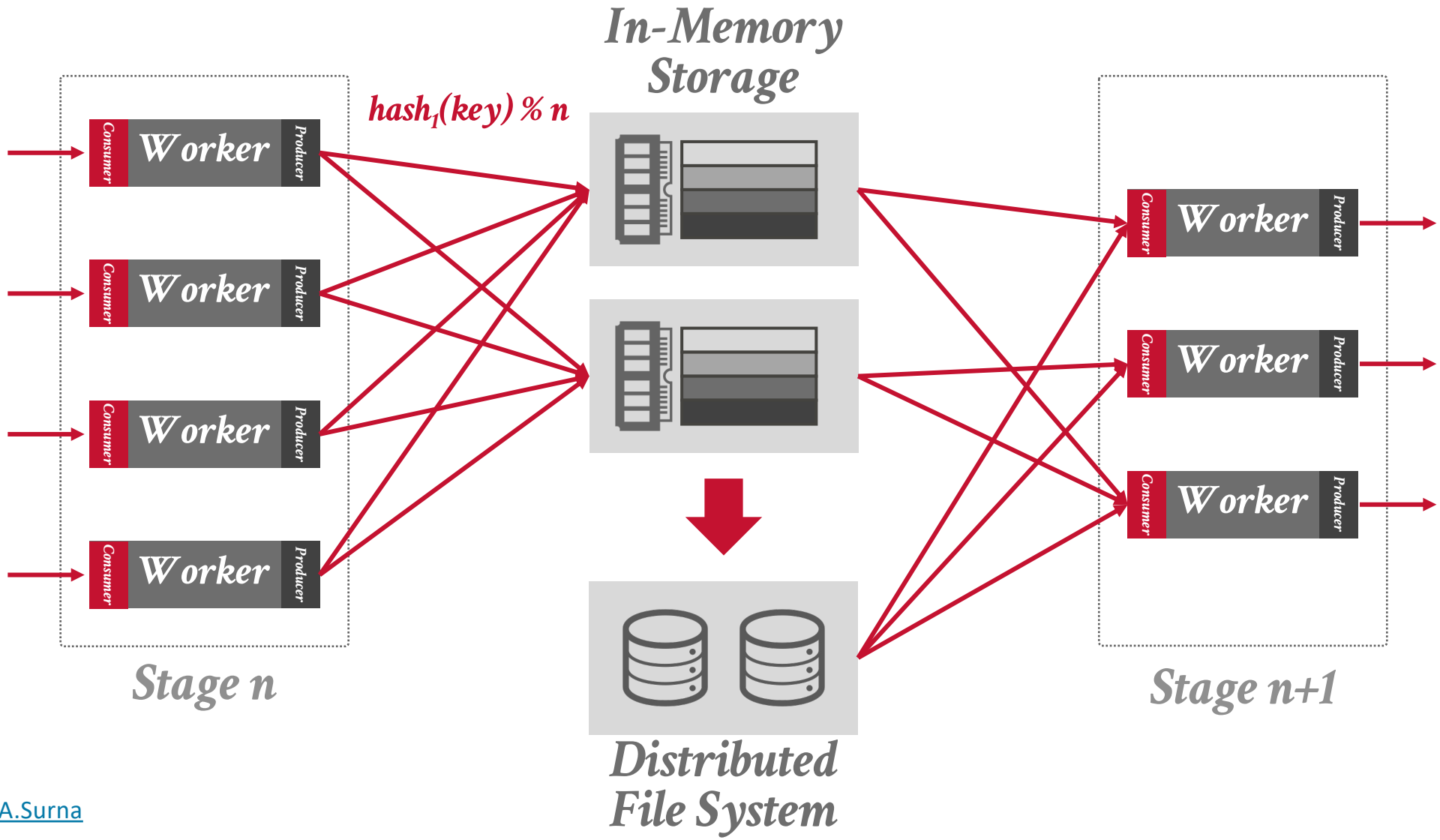


Source: [H.Ahmadi + A.Surna]

DREMEL: IN-MEMORY SHUFFLE

- Producer/consumer model for transmitting intermediate results from each stage to the next using dedicated nodes.
  - Workers send output to shuffle nodes.
  - Shuffle nodes store data in memory in hashed partitions.
  - Workers at the next stage retrieve their inputs from the shuffle nodes.
- Shuffle nodes store this data in memory and only spill to disk storage if necessary.

DREMEL: IN-MEMORY SHUFFLE

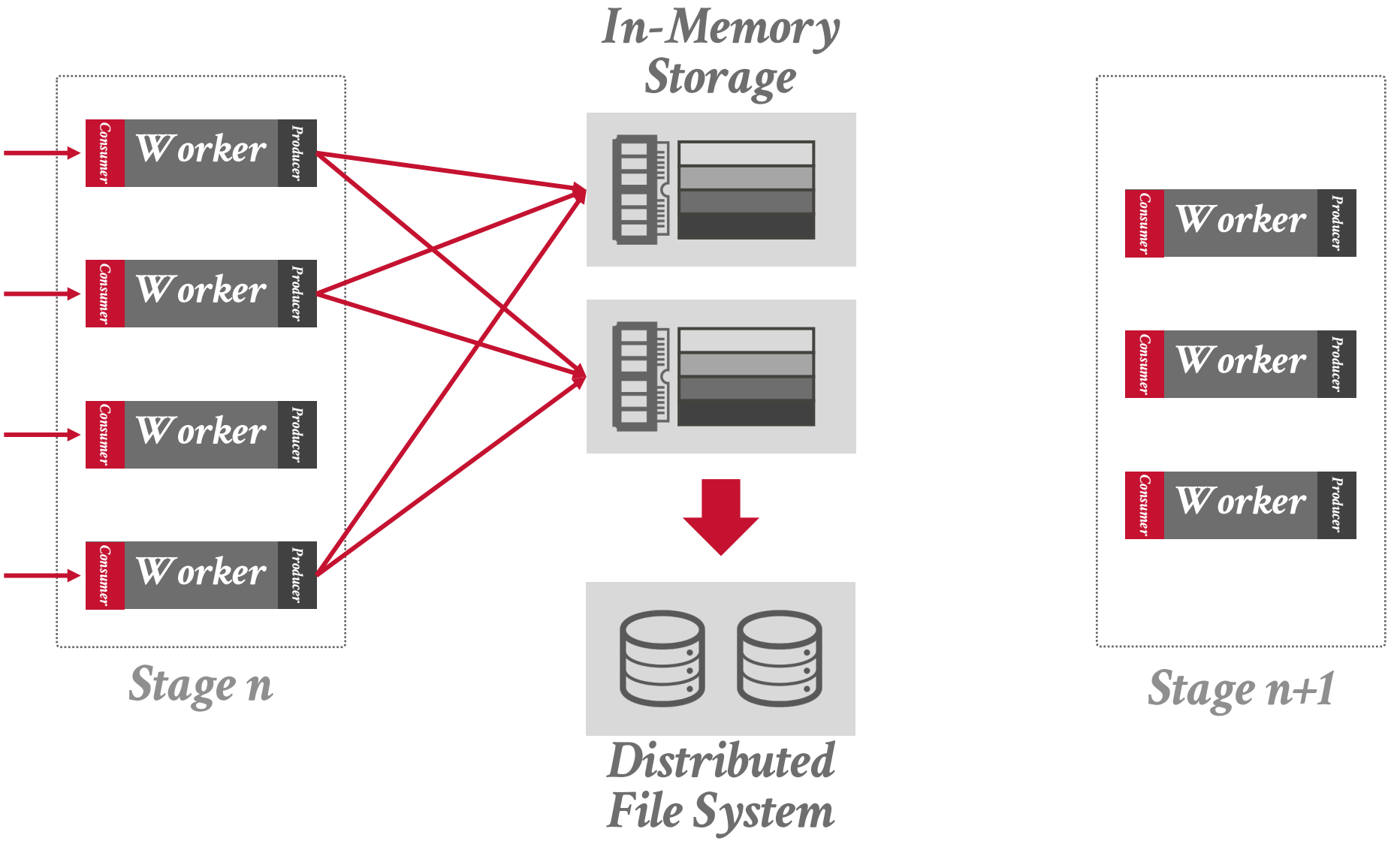


Source: [H.Ahmadi + A.Surna]

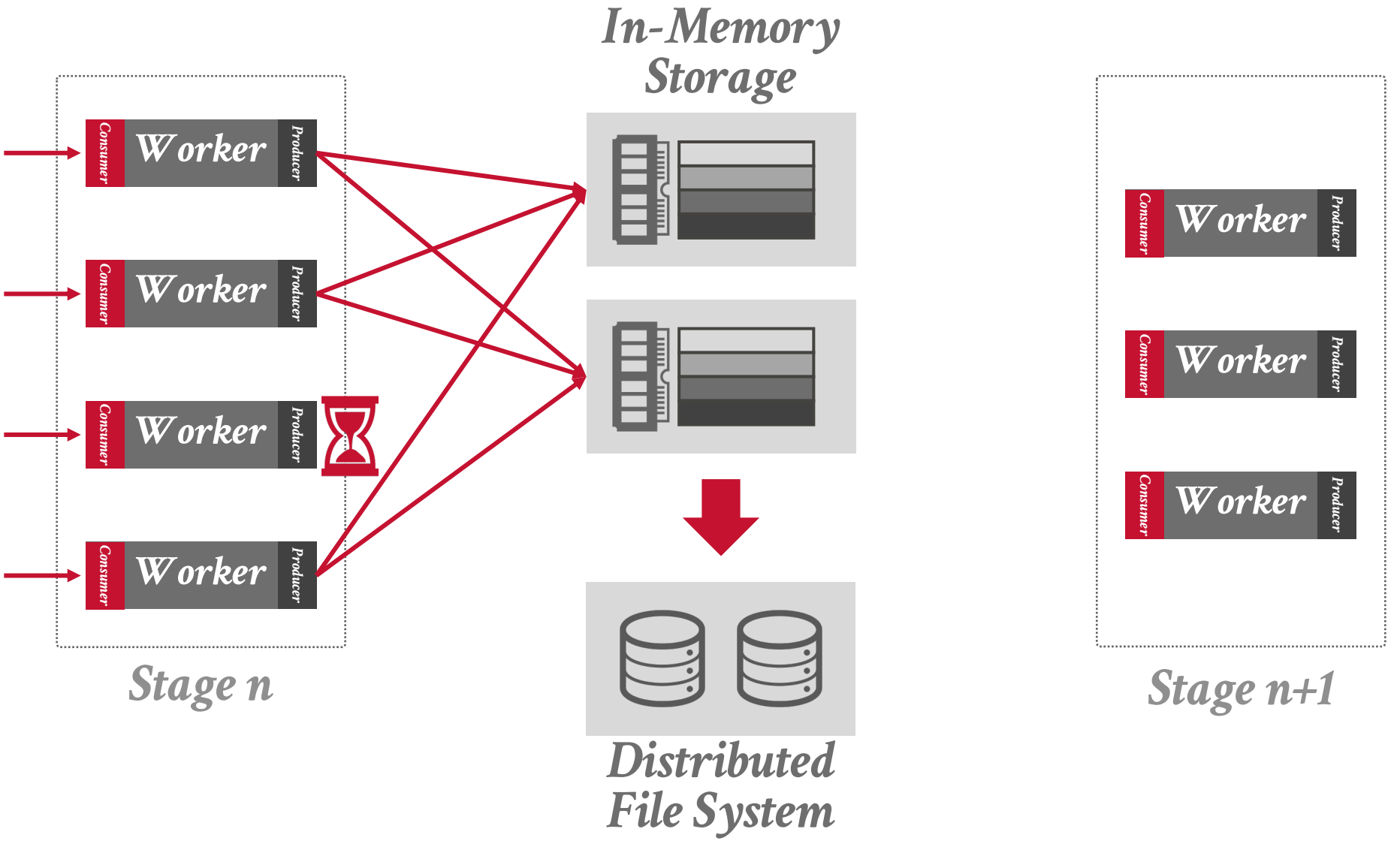
DREMEL: IN-MEMORY SHUFFLE

- The shuffle phases represent checkpoints in a query's lifecycle where that the coordinator makes sure that all tasks are completed.
- **Fault Tolerance / Straggler Avoidance:**
  - If a worker does not produce a task's results within a deadline, the coordinator speculatively executes a redundant task.
- **Dynamic Resource Allocation:**
  - Scale up / down the number of workers for the next stage depending size of a stage's output.

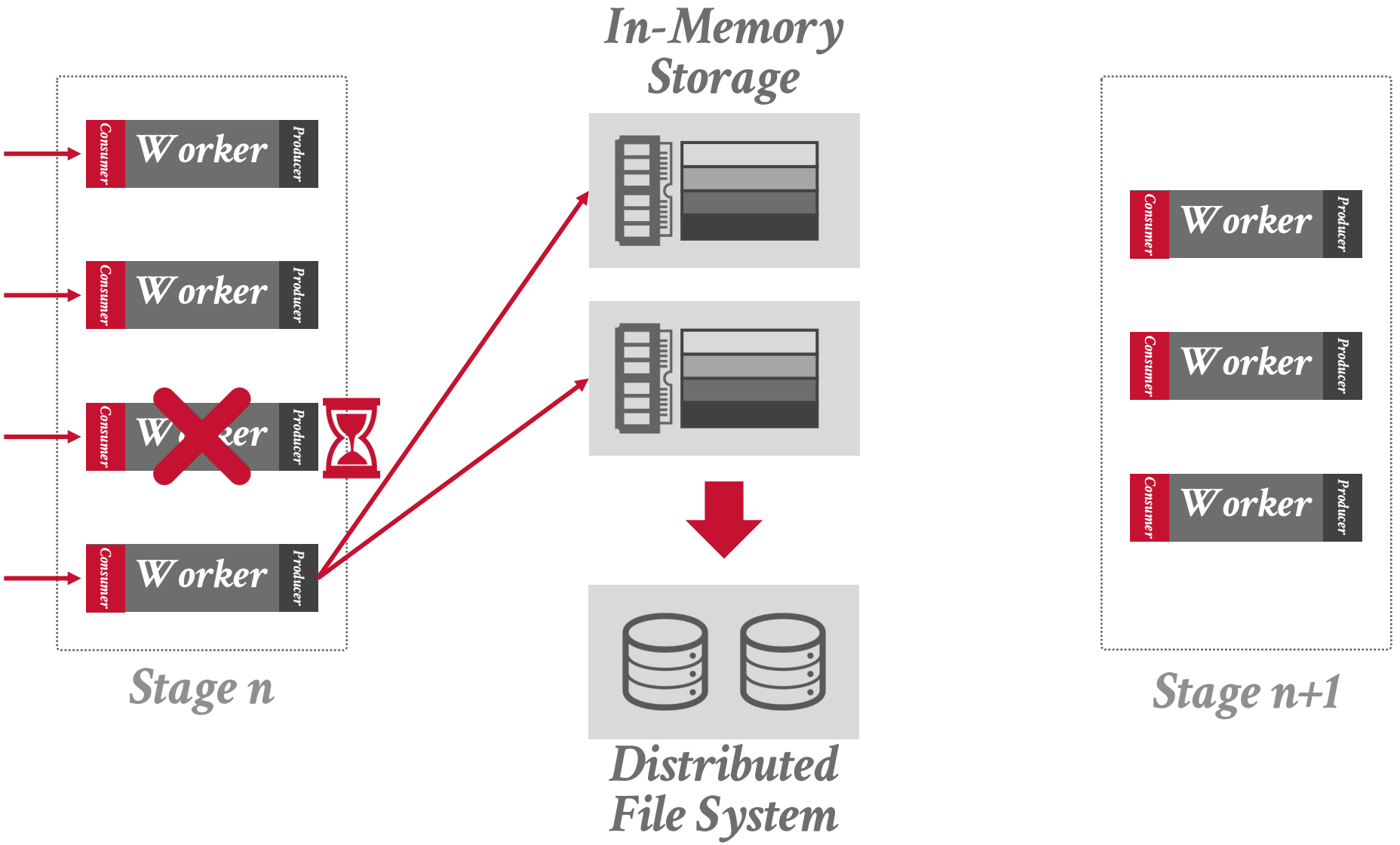
DREMEL: IN-MEMORY SHUFFLE



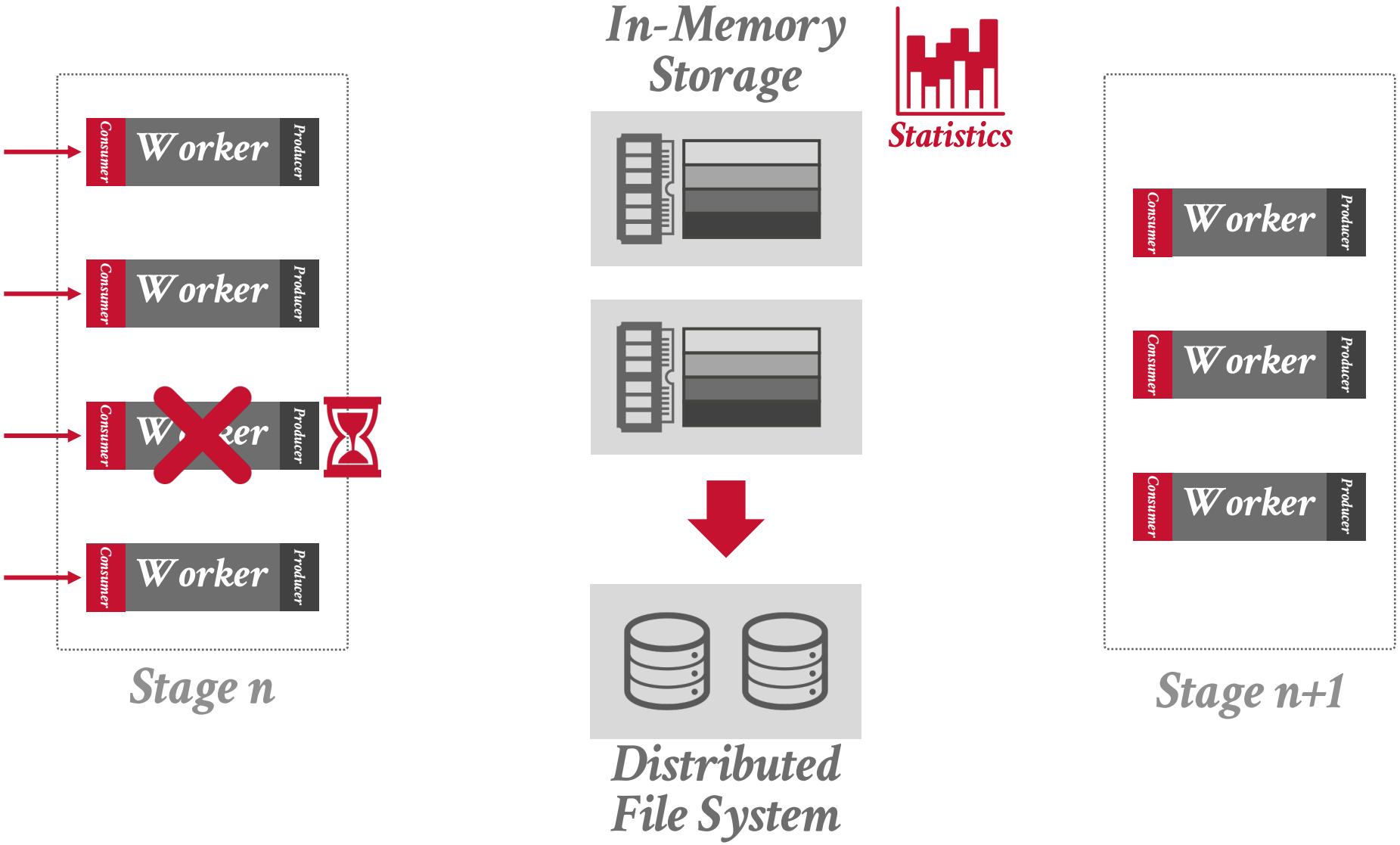
DREMEL: IN-MEMORY SHUFFLE



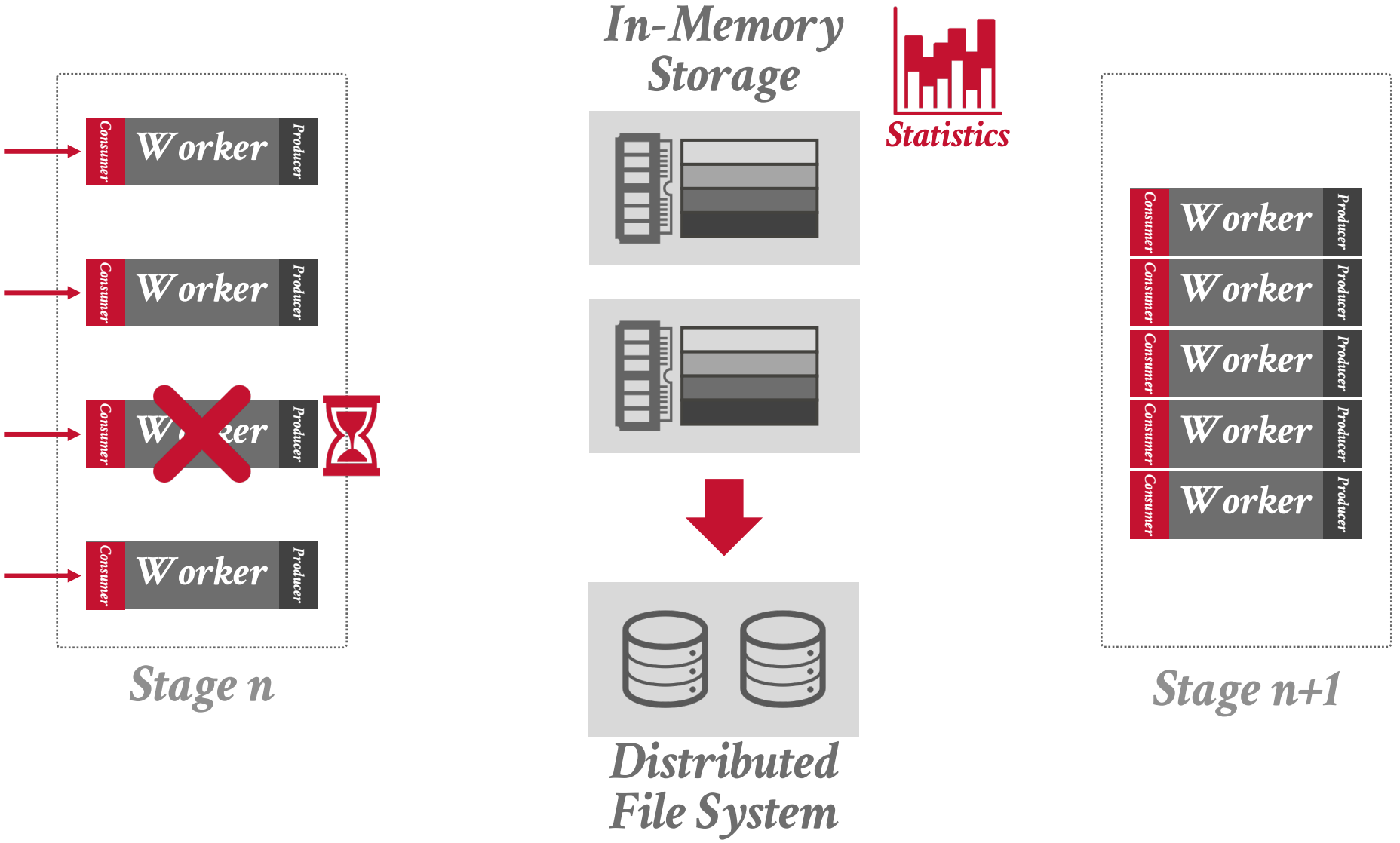
DREMEL: IN-MEMORY SHUFFLE



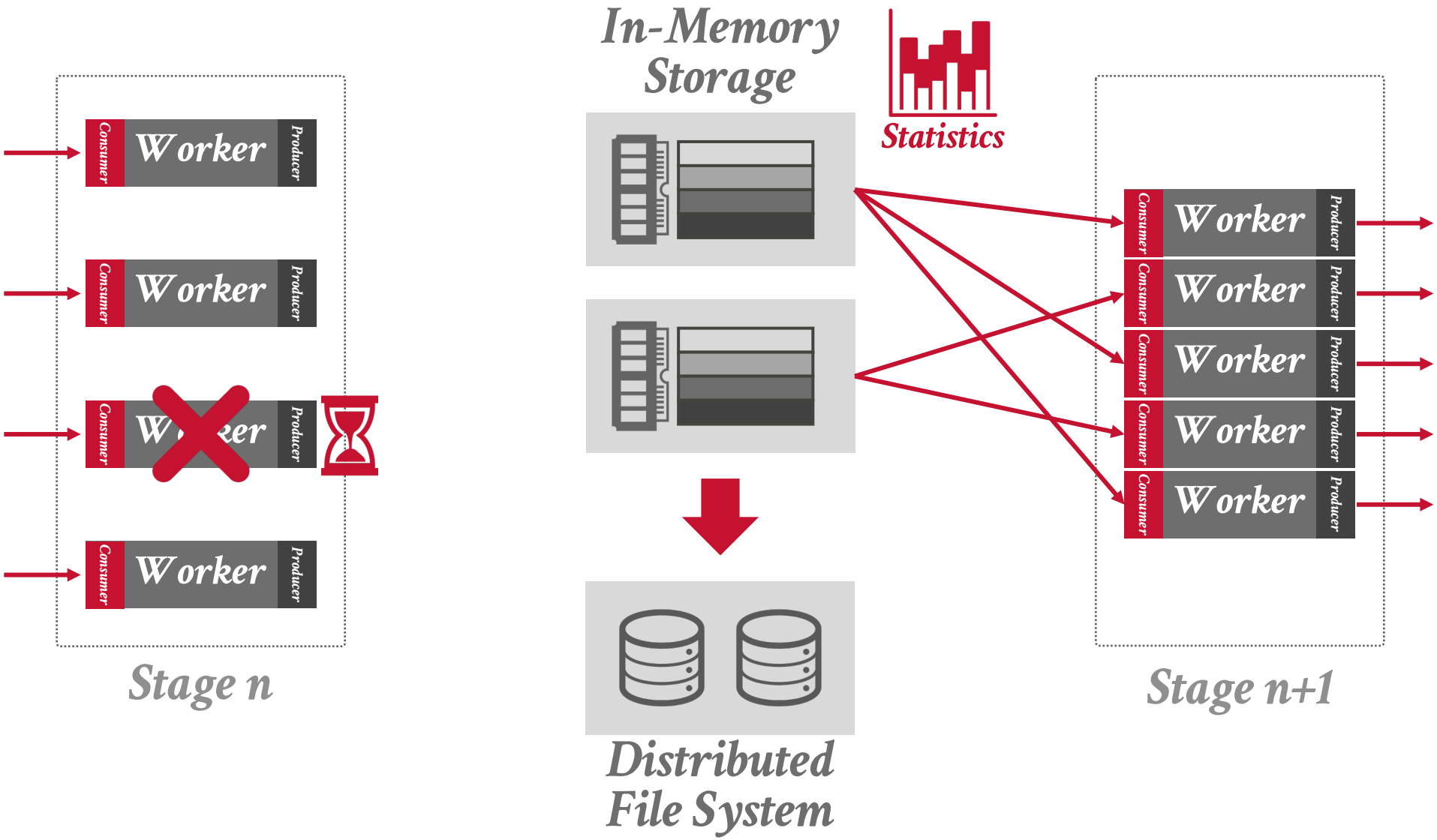
DREMEL: IN-MEMORY SHUFFLE



DREMEL: IN-MEMORY SHUFFLE



DREMEL: IN-MEMORY SHUFFLE



DREMEL: QUERY OPTIMIZATION

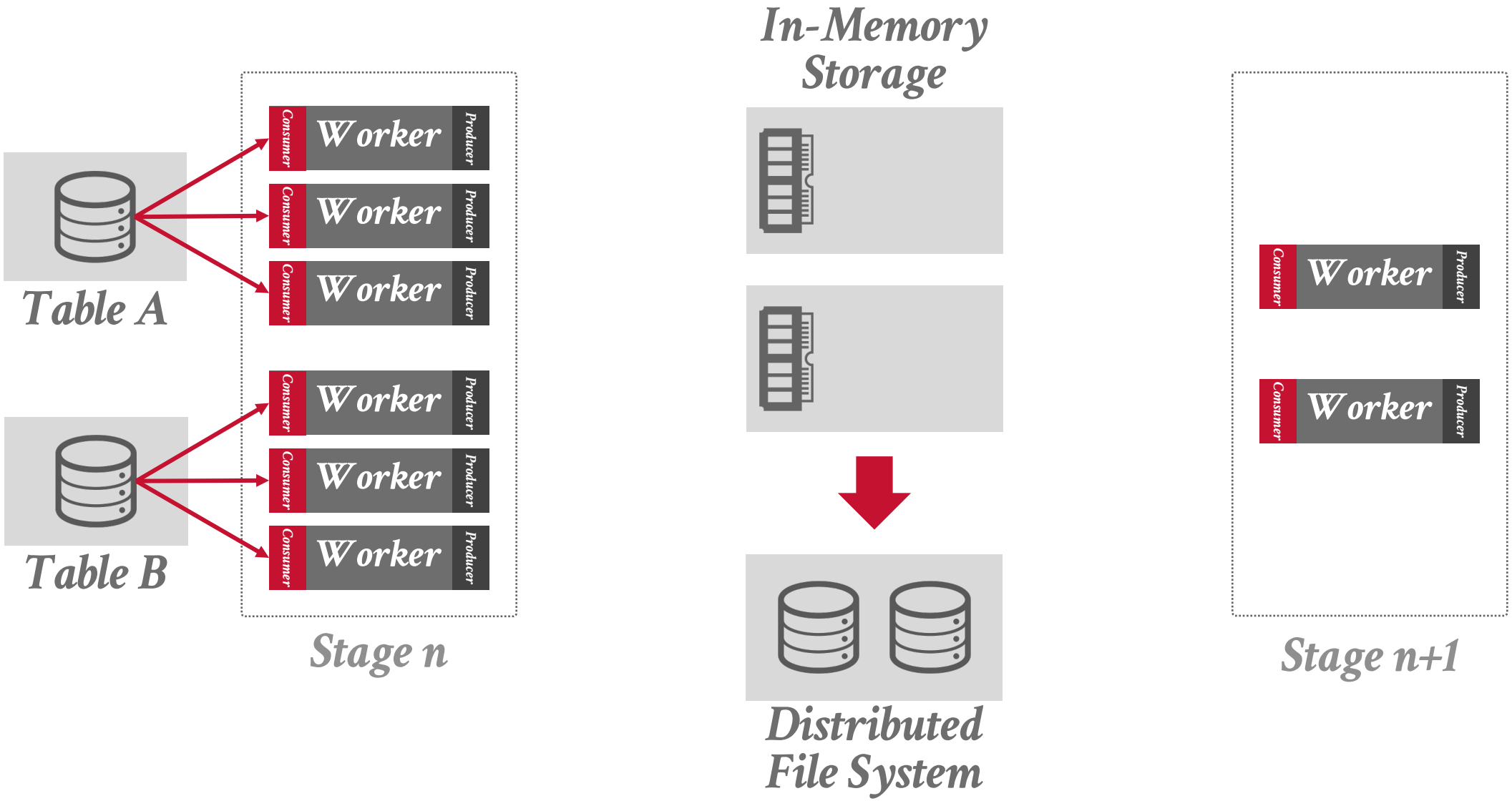
- Dremel's optimizer uses a stratified approach with rule-based + cost-based optimizer passes to generate a preliminary physical plan to start execution.
  - Rules for predicate pushdown, star schema constraint propagation, primary/foreign key hints, join ordering.
  - Cost-based optimizations only on data that the DBMS has statistics available (e.g., materialized views).
- To avoid the problems with bad cost model estimates, Dremel uses adaptive query optimization…

Source: H.Ahmadi + A.Surna

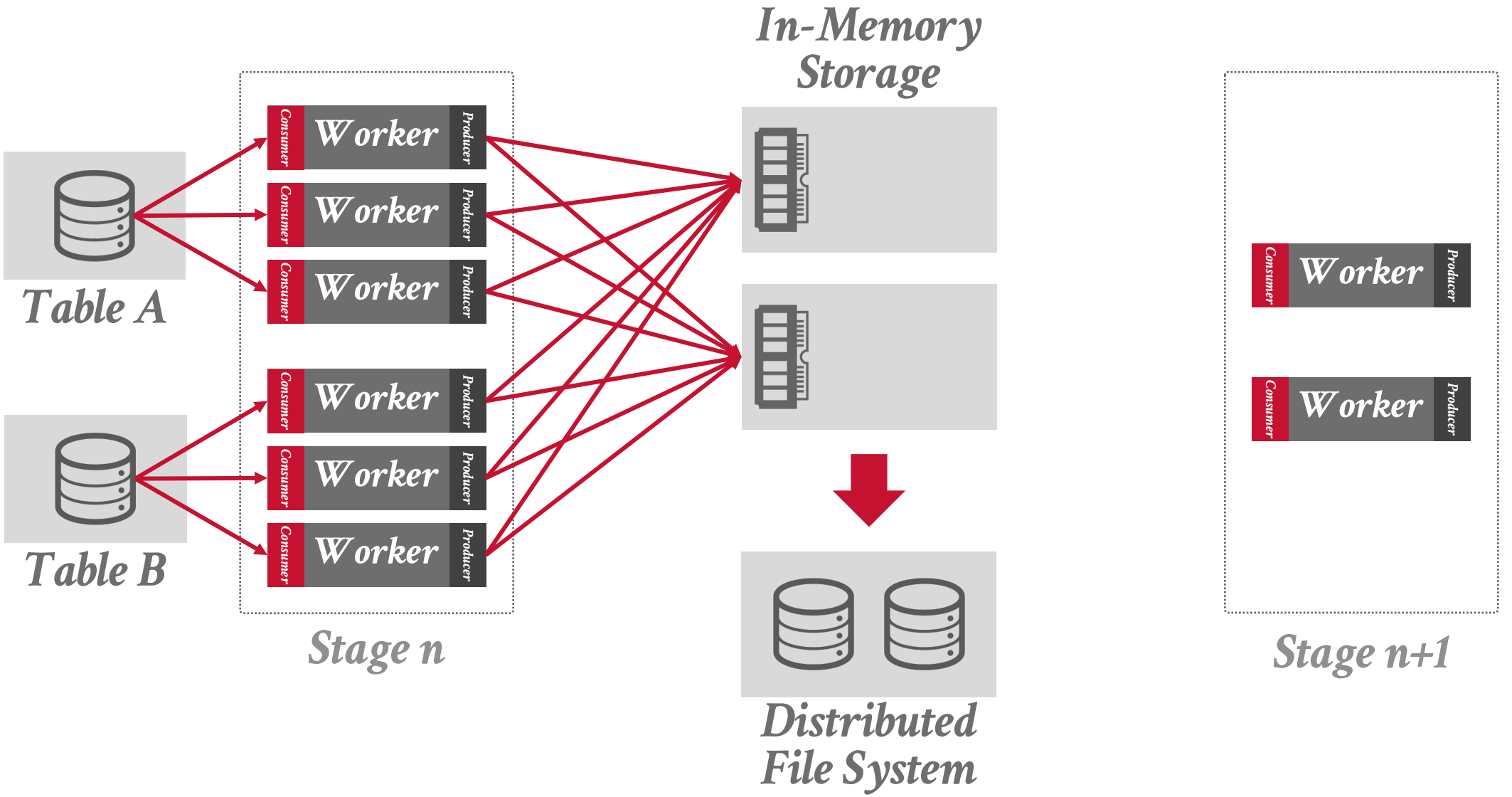
DREMEL: ADAPTIVE QUERY OPTIMIZATION

- Dremel changes the query plan before a stages starts based on observations from the preceding stage.
  - Avoids the problem of optimizer making decisions with inaccurate (or non-existing) data statistics.
- **Optimization Examples:**
  - Change the # of workers in a stage.
  - Switch between shuffle vs. broadcast join.
  - Change the physical operator implementation.
  - Dynamic repartitioning.

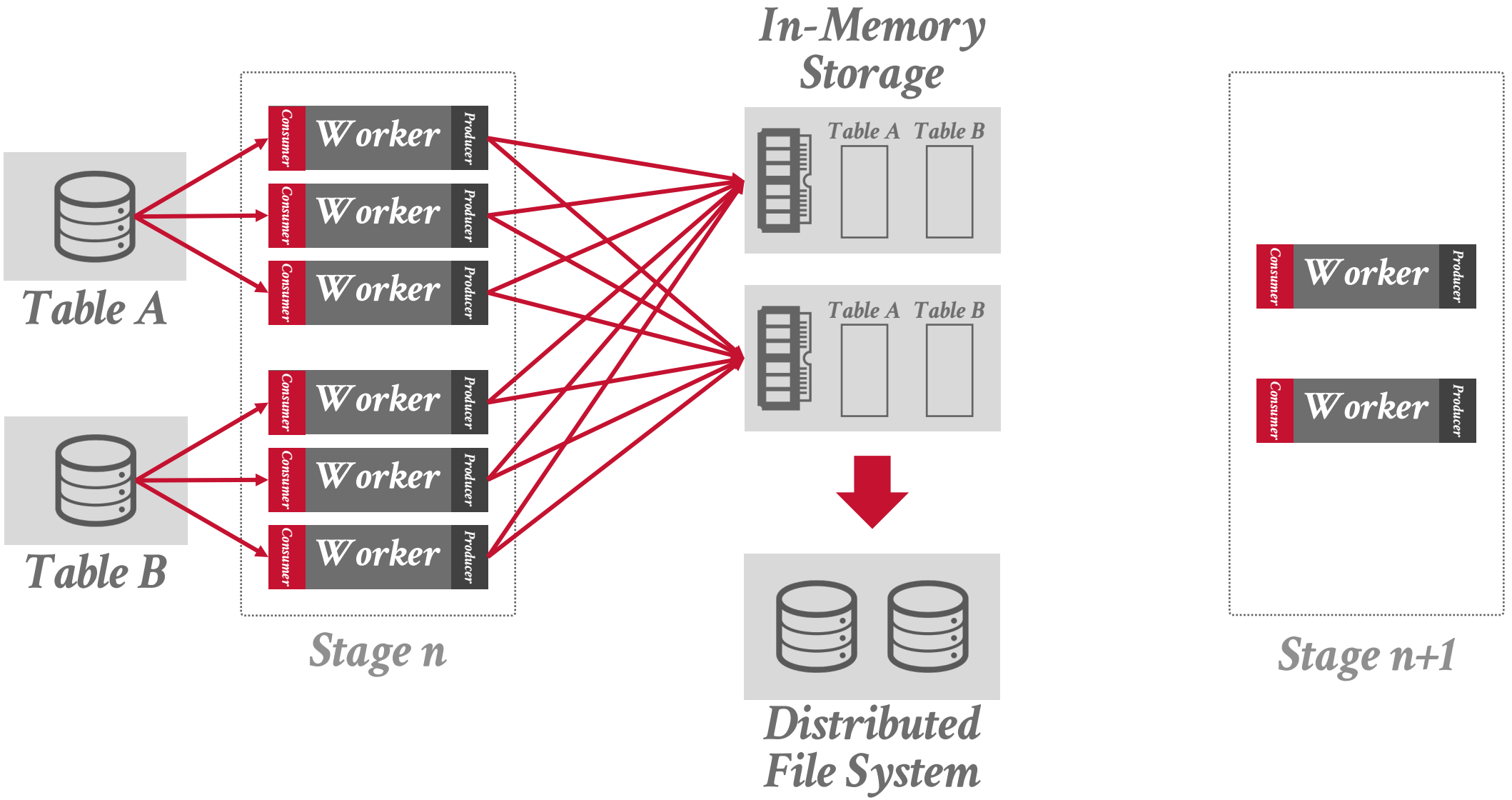
DREMEL: ADAPTIVE JOIN



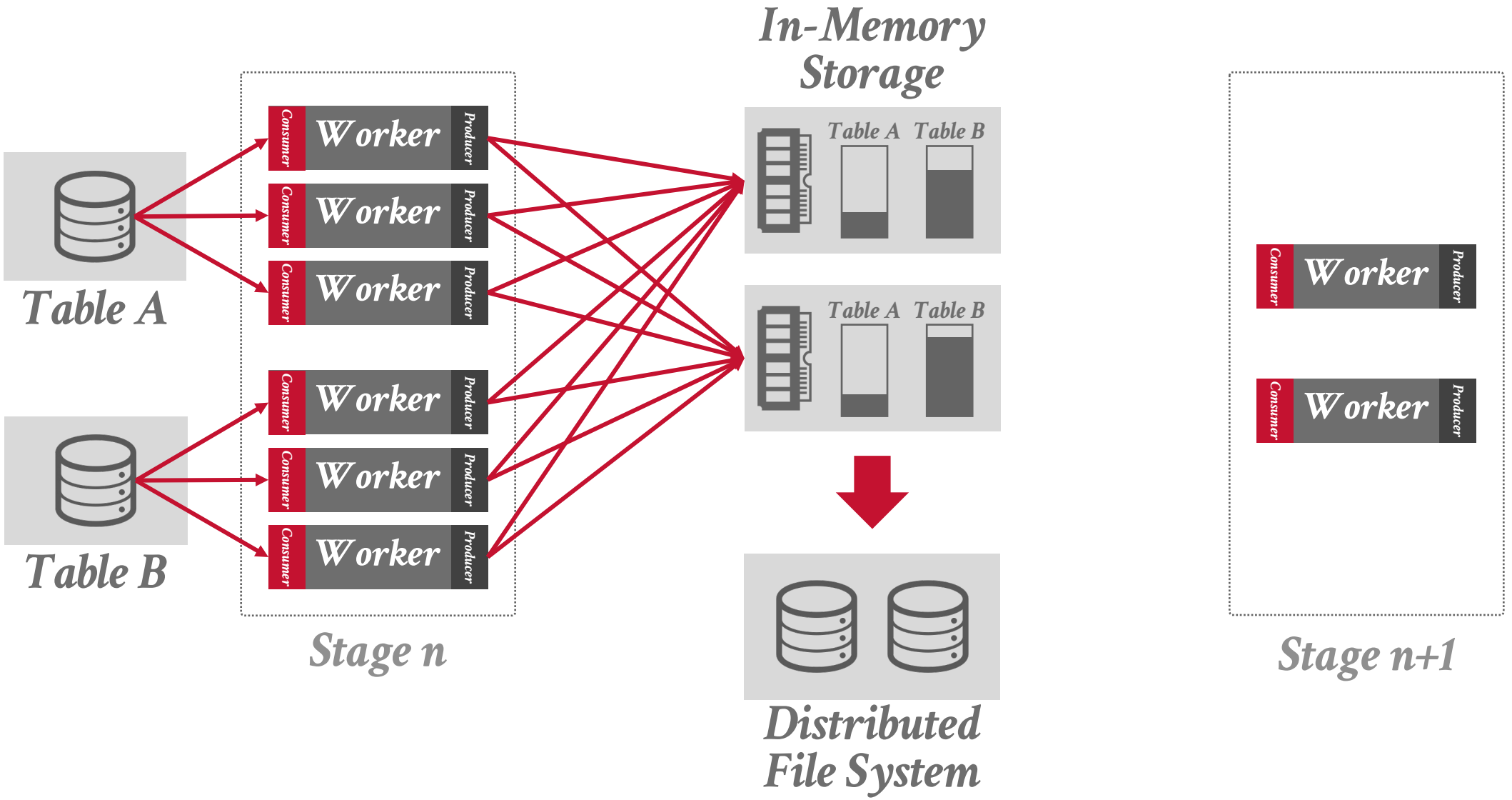
DREMEL: ADAPTIVE JOIN



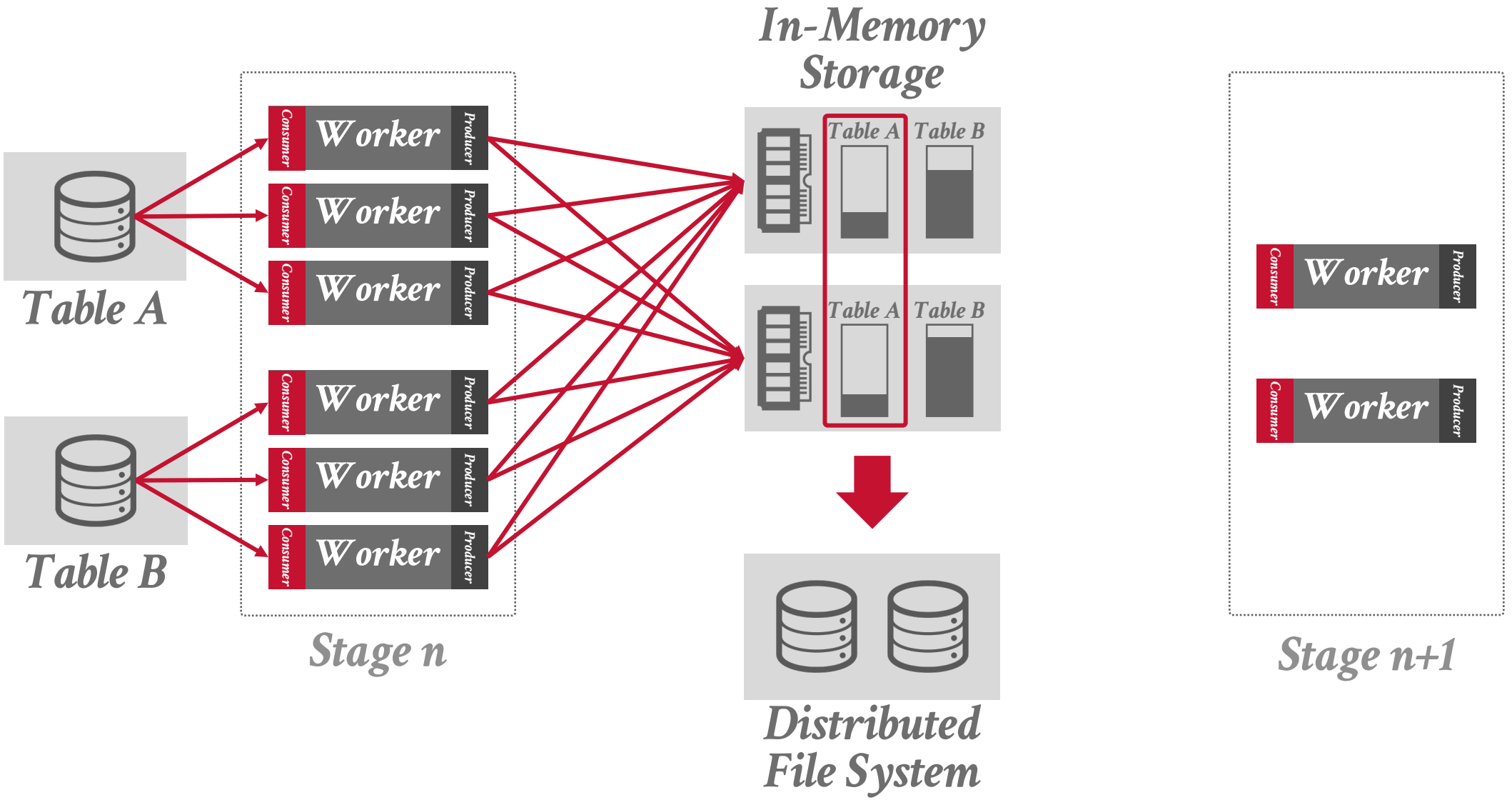
DREMEL: ADAPTIVE JOIN



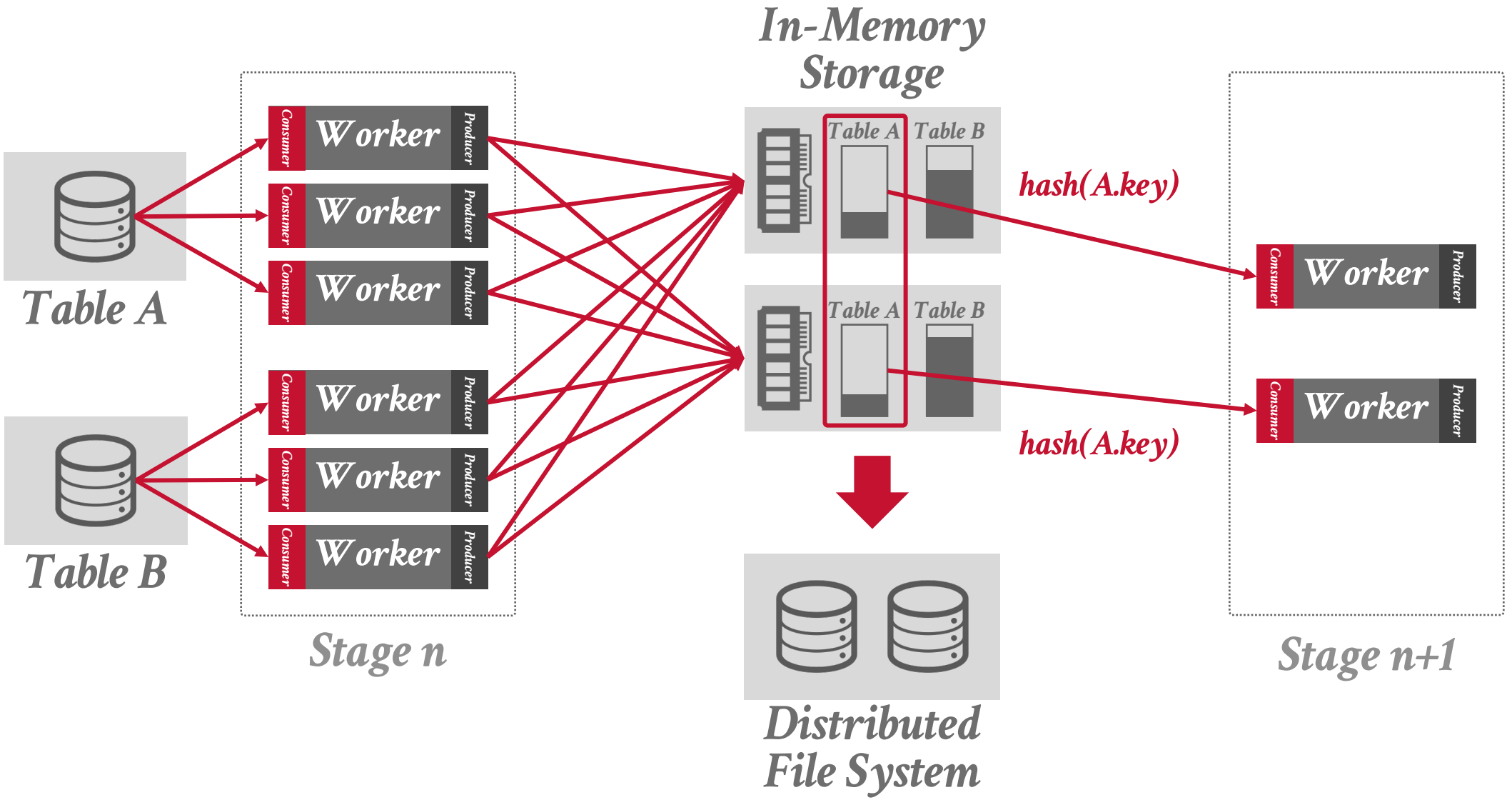
DREMEL: ADAPTIVE JOIN



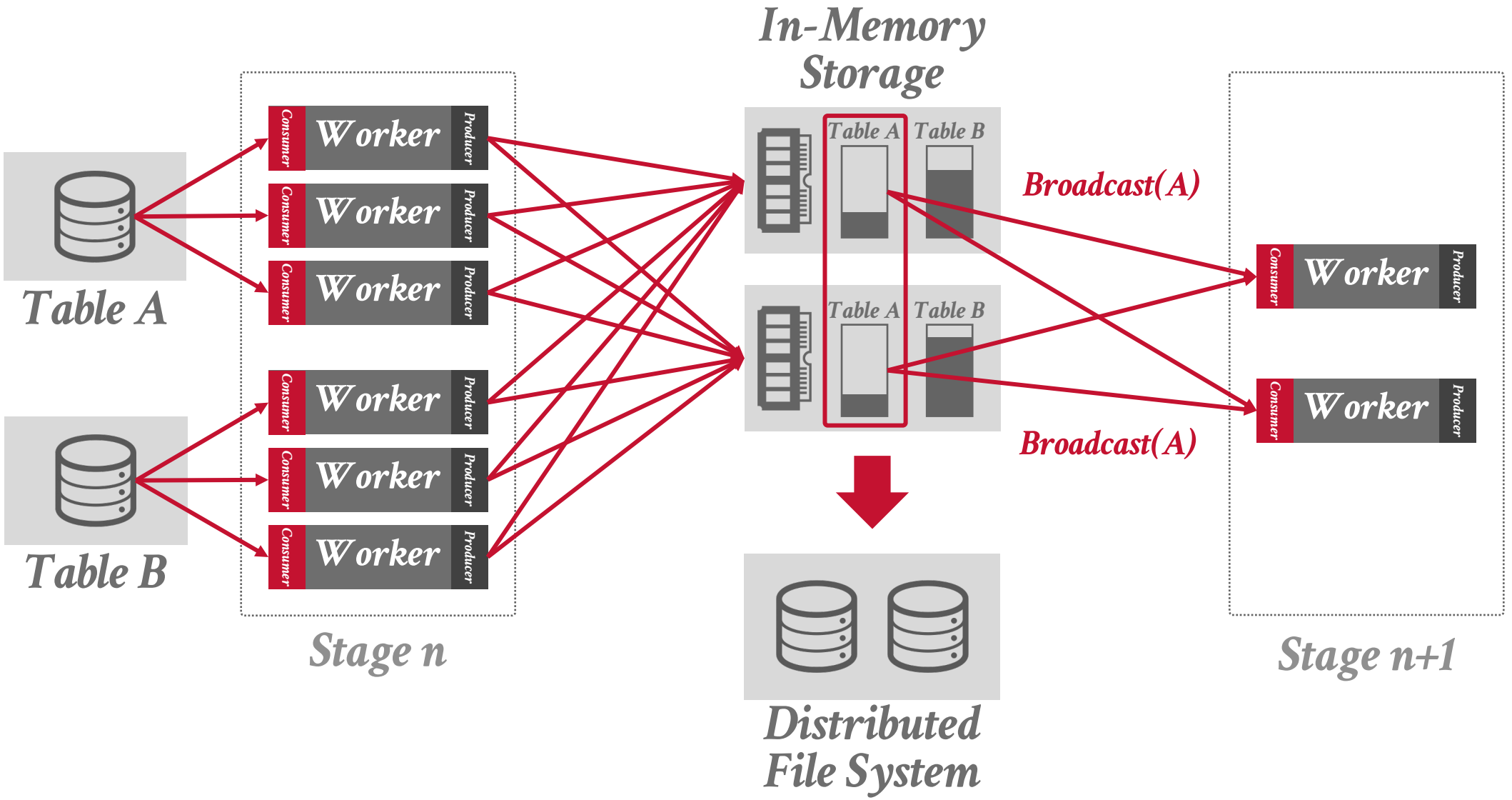
DREMEL: ADAPTIVE JOIN



DREMEL: ADAPTIVE JOIN

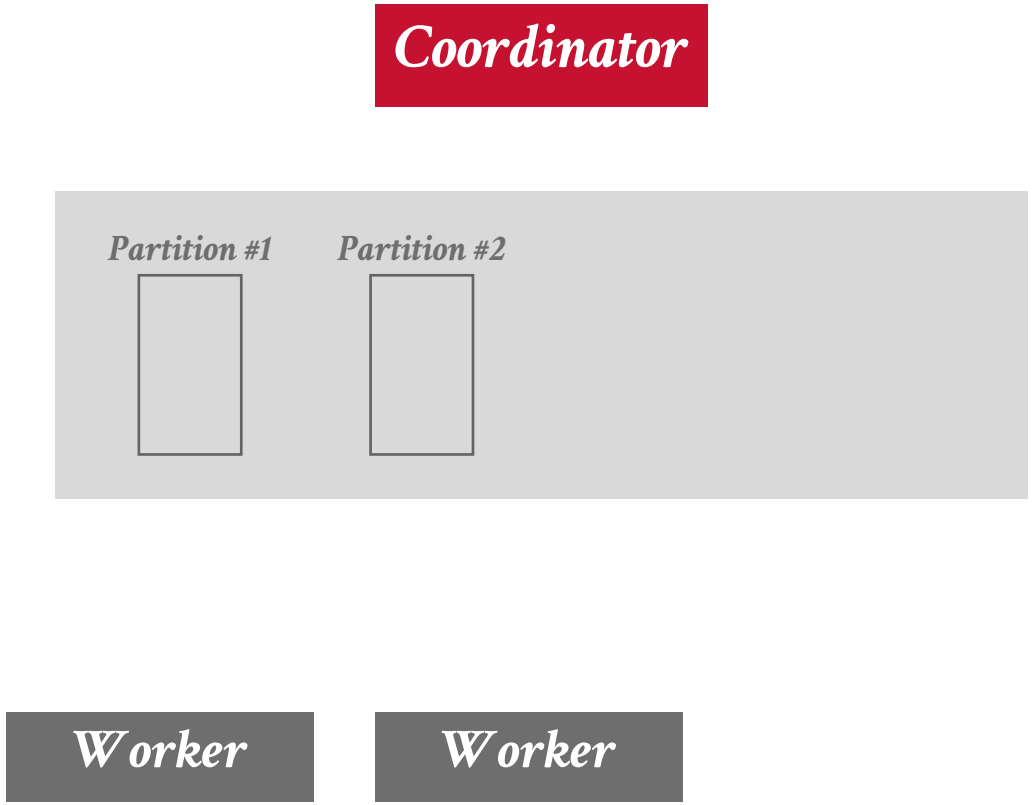


DREMEL: ADAPTIVE JOIN



DREMEL: DYNAMIC REPARTITIONING

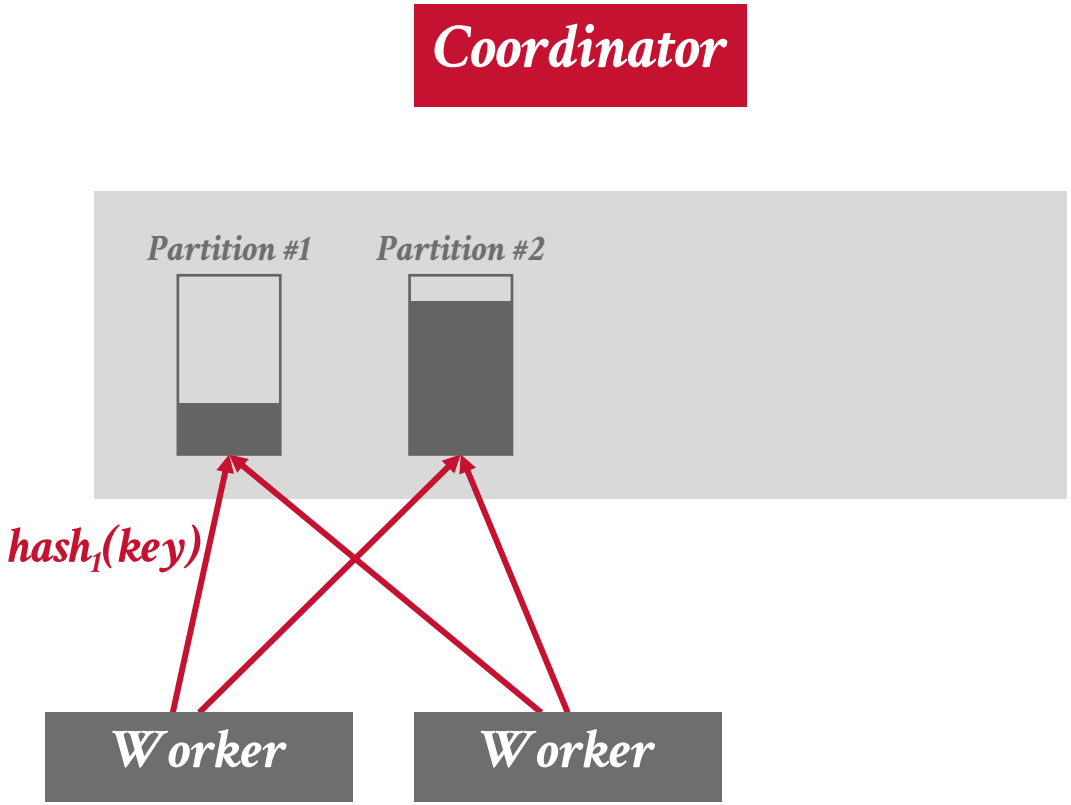
- Dremel dynamically load balances and adjusts intermediate result partitioning to adapt to data skew.
- DBMS detects whether shuffle partition gets too full and then instructs workers to adjust their partitioning scheme.



Source: H.Ahmadi + A.Surna

DREMEL: DYNAMIC REPARTITIONING

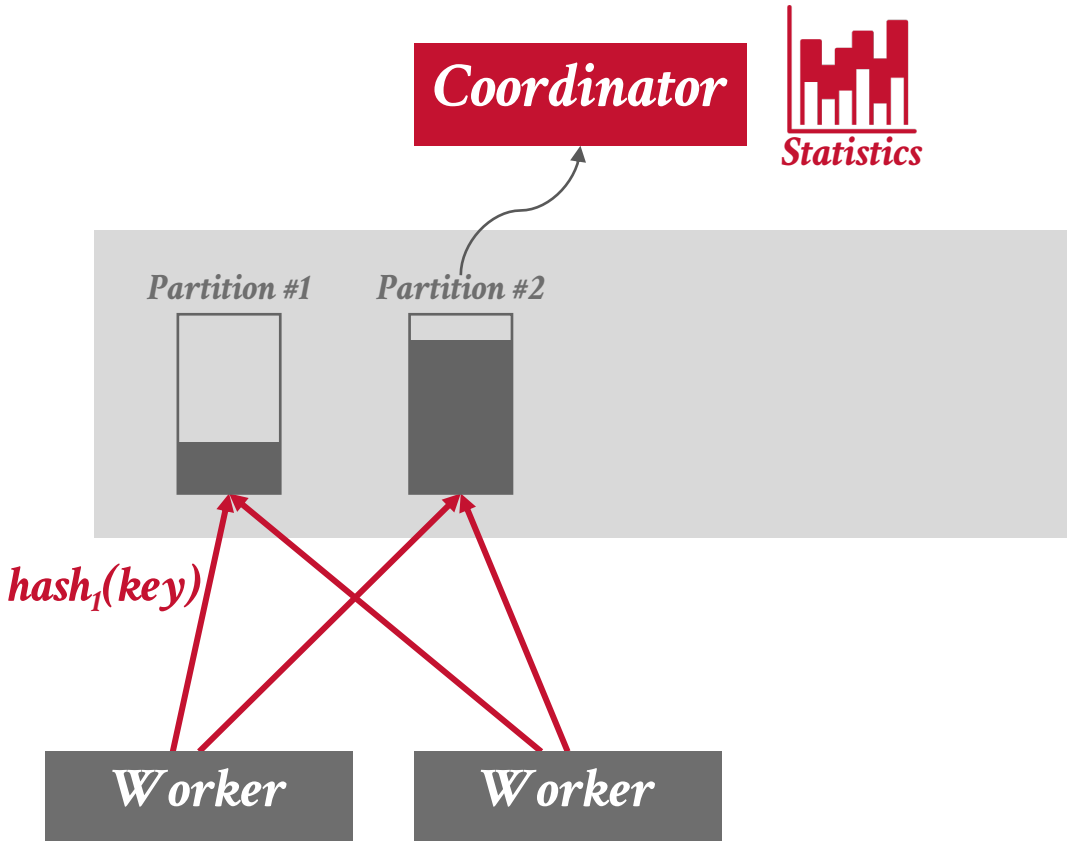
- Dremel dynamically load balances and adjusts intermediate result partitioning to adapt to data skew.
- DBMS detects whether shuffle partition gets too full and then instructs workers to adjust their partitioning scheme.



Source: H.Ahmadi + A.Surna

DREMEL: DYNAMIC REPARTITIONING

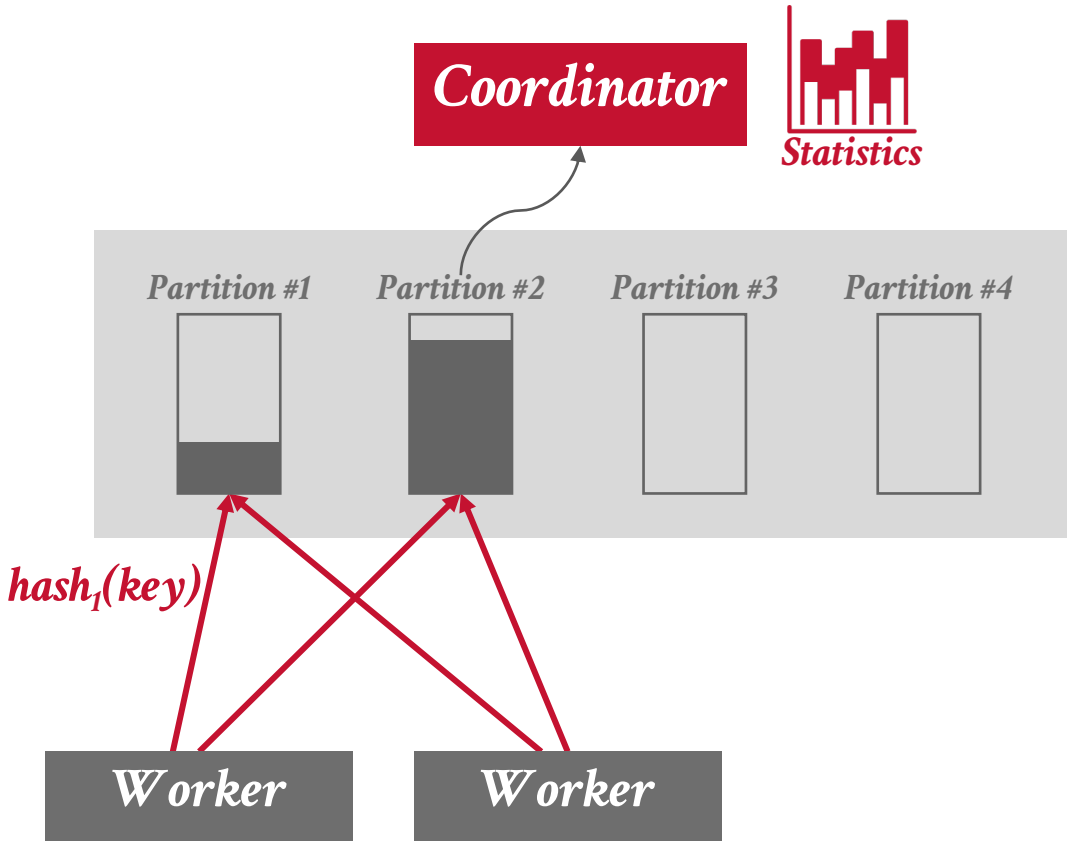
- Dremel dynamically load balances and adjusts intermediate result partitioning to adapt to data skew.
- DBMS detects whether shuffle partition gets too full and then instructs workers to adjust their partitioning scheme.



Source: H.Ahmadi + A.Surna

DREMEL: DYNAMIC REPARTITIONING

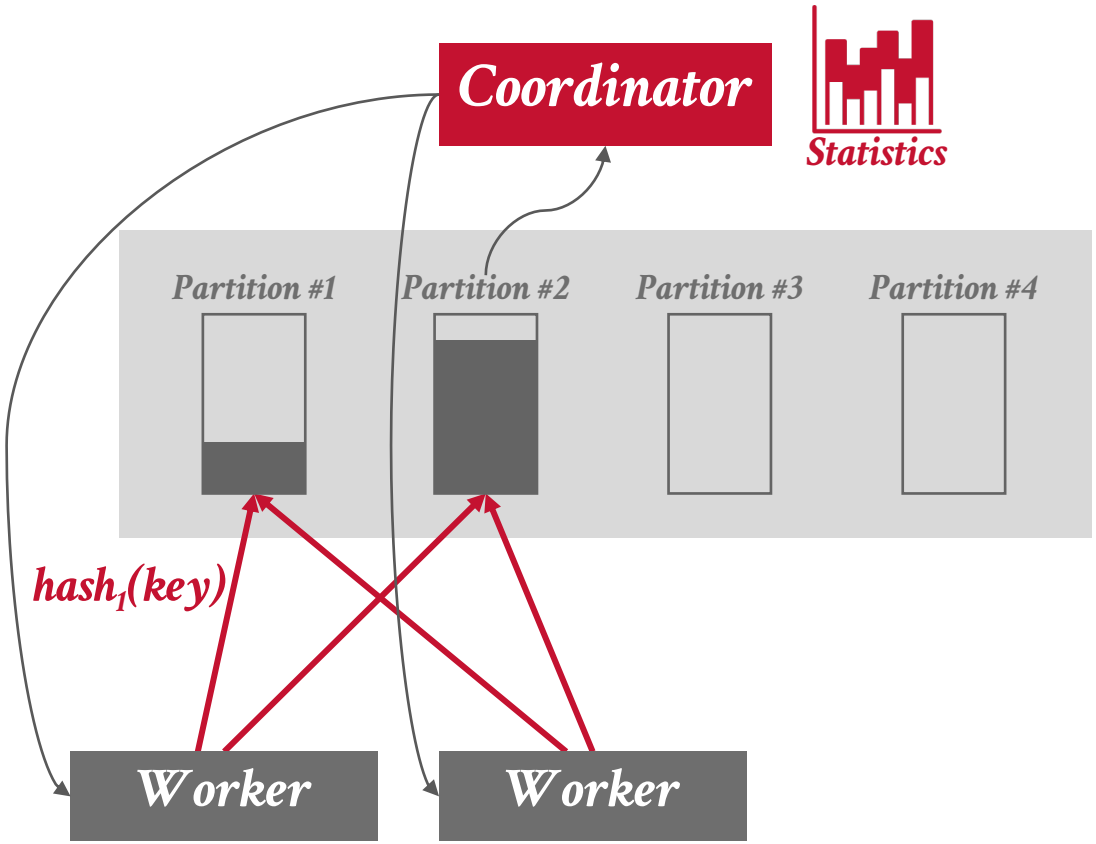
- Dremel dynamically load balances and adjusts intermediate result partitioning to adapt to data skew.
- DBMS detects whether shuffle partition gets too full and then instructs workers to adjust their partitioning scheme.



Source: H.Ahmadi + A.Surna

DREMEL: DYNAMIC REPARTITIONING

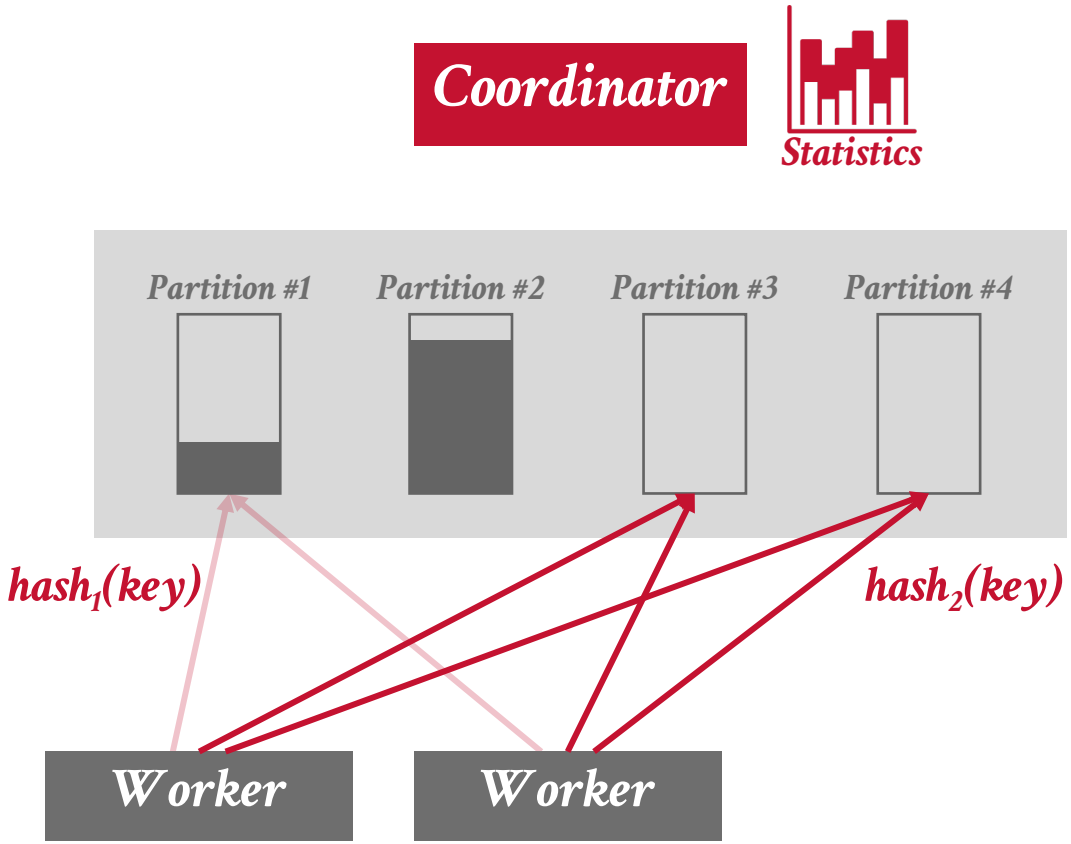
- Dremel dynamically load balances and adjusts intermediate result partitioning to adapt to data skew.
- DBMS detects whether shuffle partition gets too full and then instructs workers to adjust their partitioning scheme.



Source: H.Ahmadi + A.Surna

DREMEL: DYNAMIC REPARTITIONING

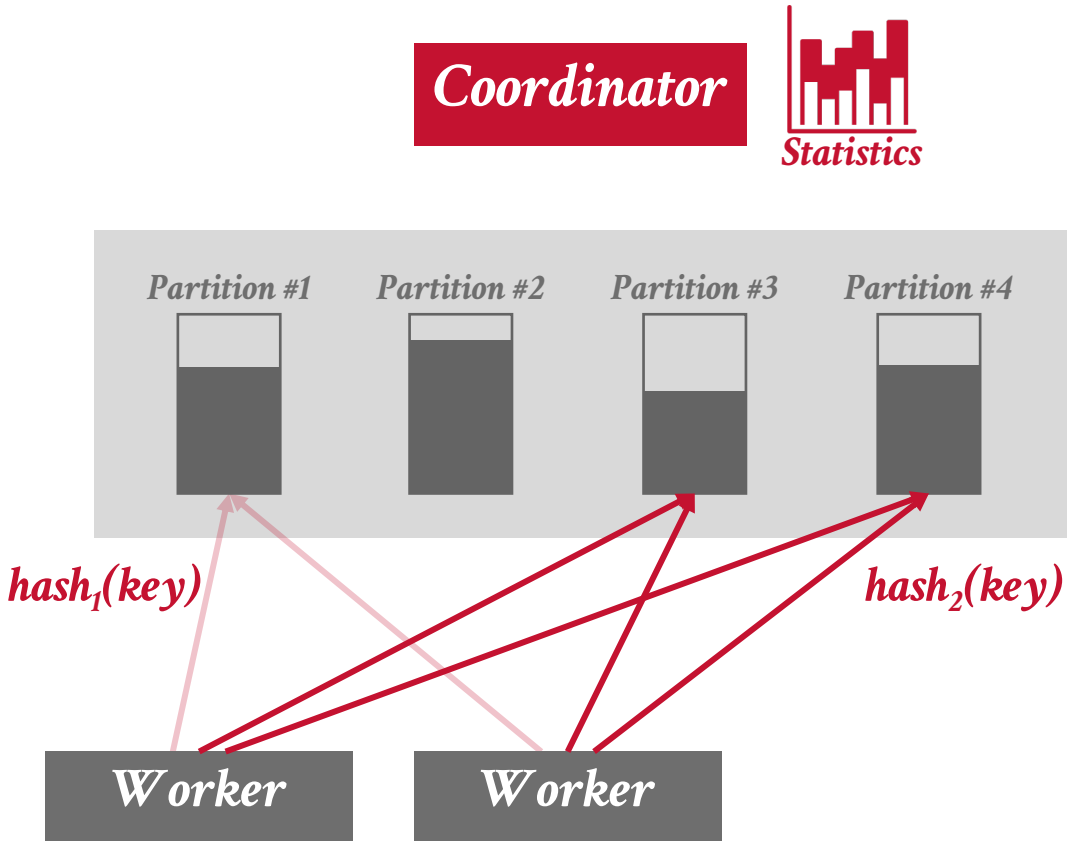
- Dremel dynamically load balances and adjusts intermediate result partitioning to adapt to data skew.
- DBMS detects whether shuffle partition gets too full and then instructs workers to adjust their partitioning scheme.



Source: H.Ahmadi + A.Surna

DREMEL: DYNAMIC REPARTITIONING

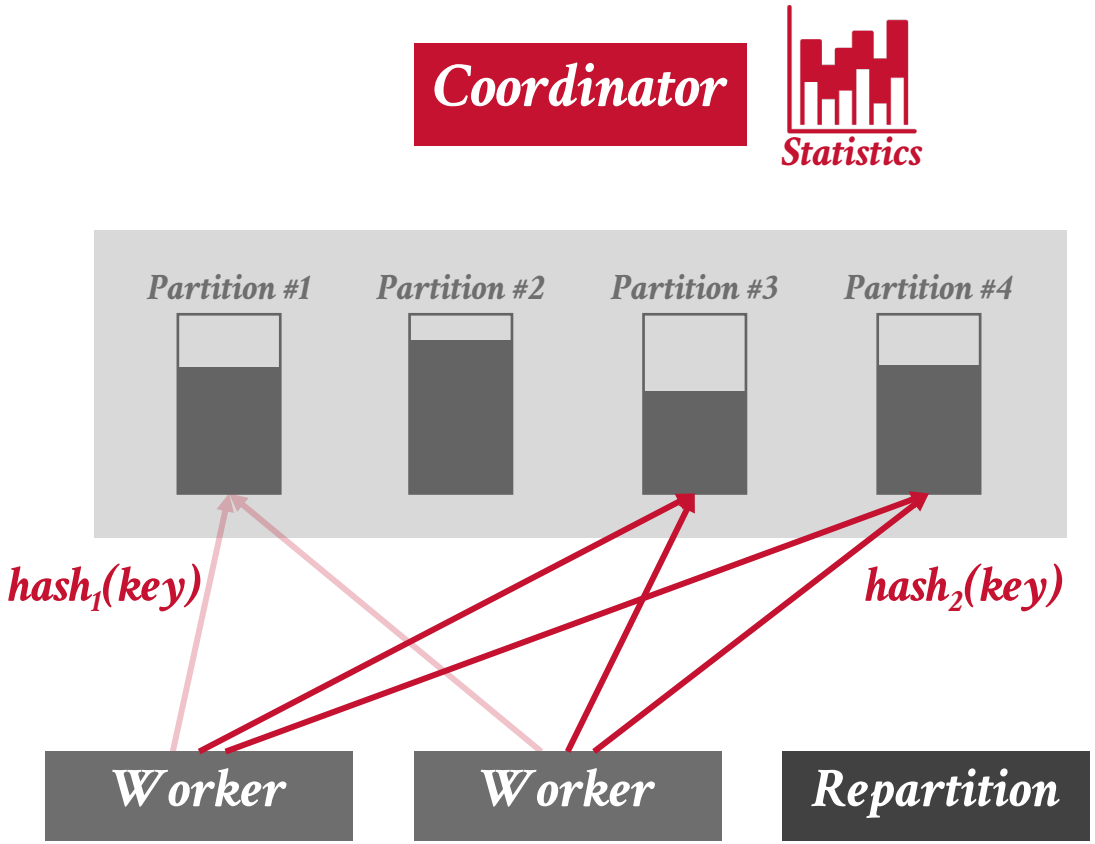
- Dremel dynamically load balances and adjusts intermediate result partitioning to adapt to data skew.
- DBMS detects whether shuffle partition gets too full and then instructs workers to adjust their partitioning scheme.



Source: H.Ahmadi + A.Surna

DREMEL: DYNAMIC REPARTITIONING

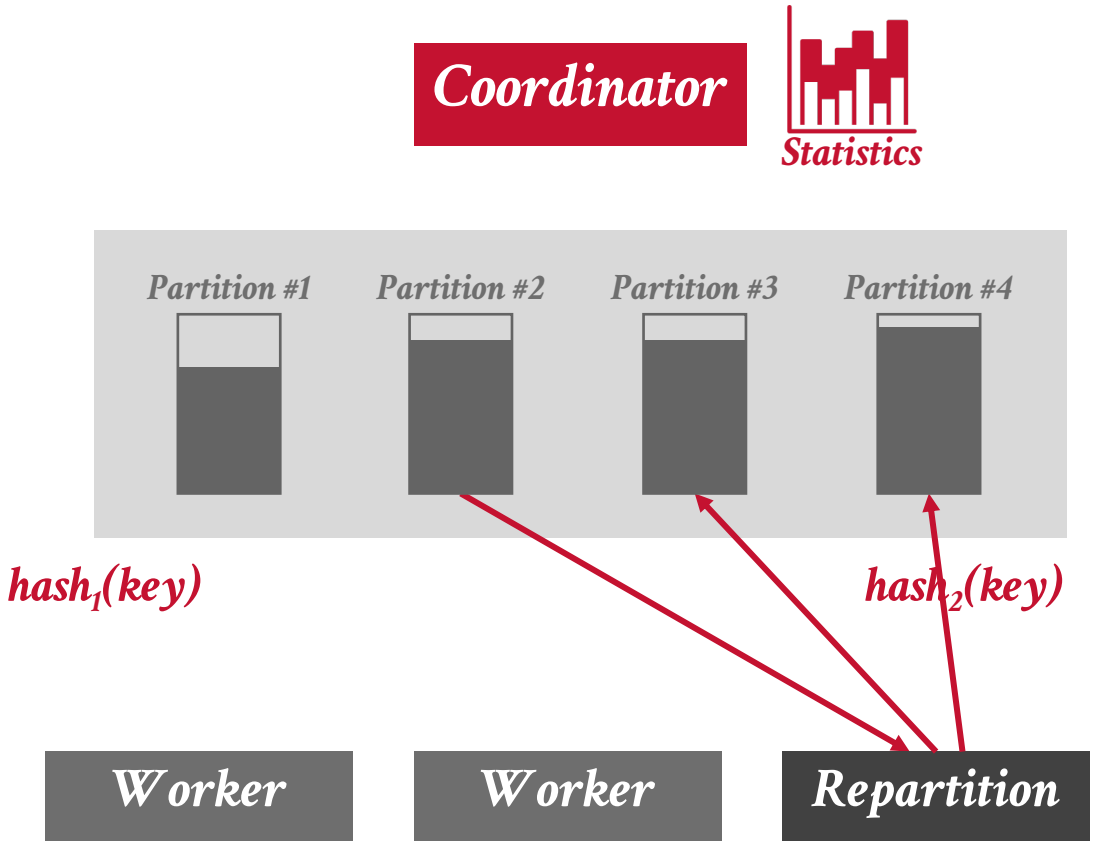
- Dremel dynamically load balances and adjusts intermediate result partitioning to adapt to data skew.
- DBMS detects whether shuffle partition gets too full and then instructs workers to adjust their partitioning scheme.



Source: H.Ahmadi + A.Surna

DREMEL: DYNAMIC REPARTITIONING

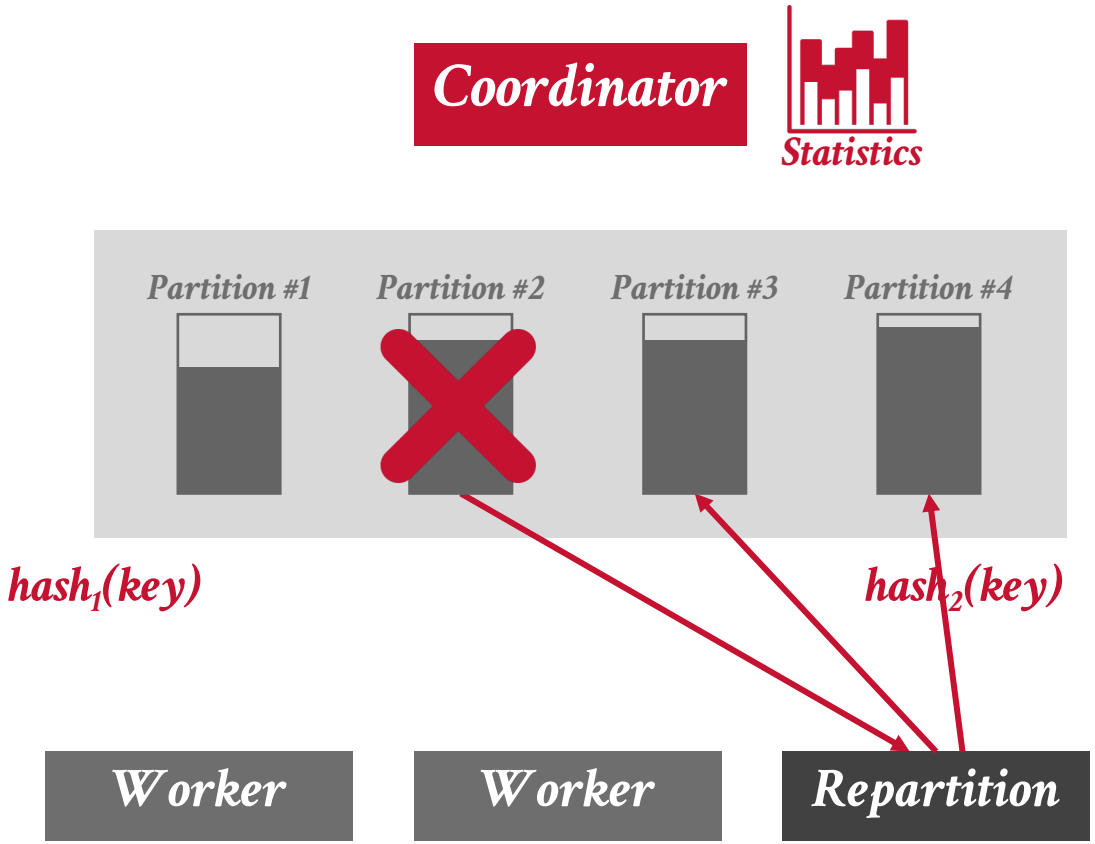
- Dremel dynamically load balances and adjusts intermediate result partitioning to adapt to data skew.
- DBMS detects whether shuffle partition gets too full and then instructs workers to adjust their partitioning scheme.



Source: H.Ahmadi + A.Surna

DREMEL: DYNAMIC REPARTITIONING

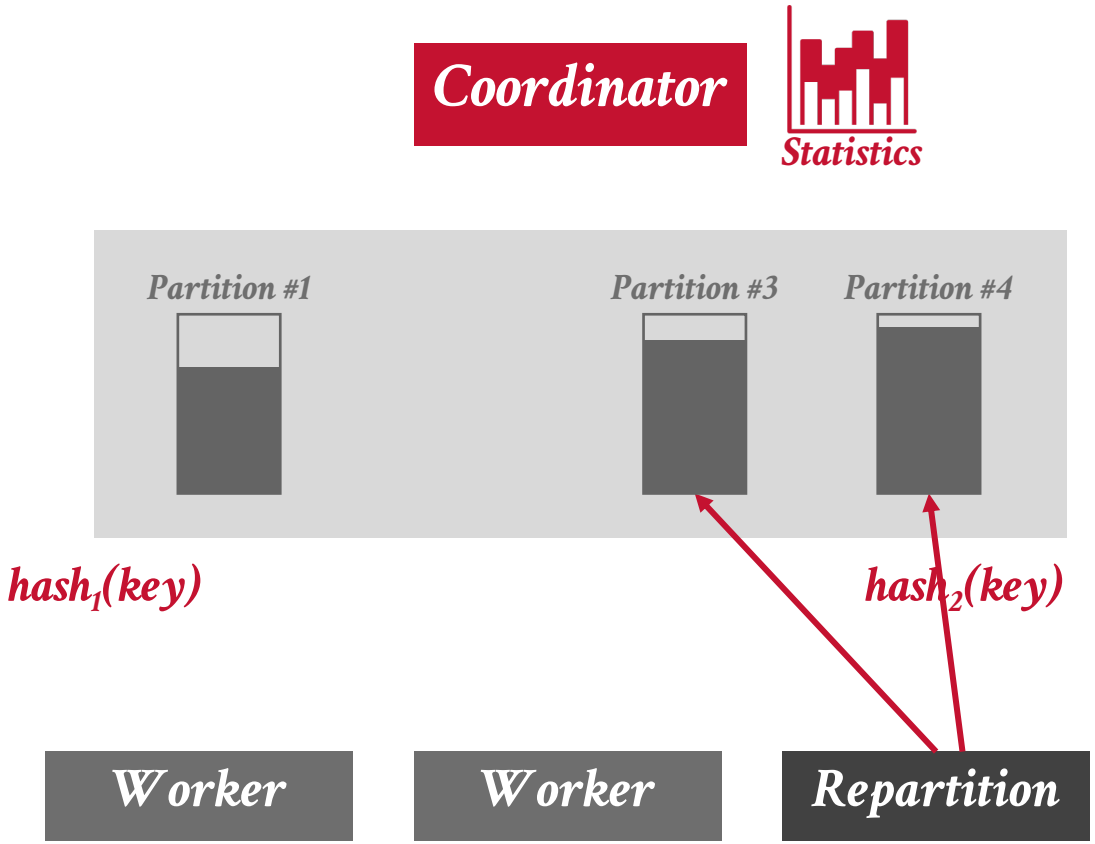
- Dremel dynamically load balances and adjusts intermediate result partitioning to adapt to data skew.
- DBMS detects whether shuffle partition gets too full and then instructs workers to adjust their partitioning scheme.



Source: H.Ahmadi + A.Surna

DREMEL: DYNAMIC REPARTITIONING

- Dremel dynamically load balances and adjusts intermediate result partitioning to adapt to data skew.
- DBMS detects whether shuffle partition gets too full and then instructs workers to adjust their partitioning scheme.



Source: H.Ahmadi + A.Surna

OBSERVATION

- Since the 2011 VLDB paper, there are DBMS projects that are copies or inspired by Dremel.
  - **Apache Drill** (MapR)
  - **Presto** (Meta)
  - **Apache Impala** (Cloudera)
  - **Dremio**
- There are also shuffle-as-a-service systems:
  - **Apache Celeborn** (Alibaba)
  - **Apache Uniffle** (Tencent)
  - **Remote Shuffle Service** (Uber)

Summary and Outlook

- Dremel became BigQuery, a key GCP offering.
- Lead the way in disaggregated: compute, storage, shuffle layer.
- Columnar storage for nested nested data. Mimicked in Parquet.
- In-situ data analytics, essentially the foundation for data lakes.
- Lead the way in making SQL the key query language for structured and nested data at Google and beyond. Including for Spanner, and other non-Google projects like Hive, Spark, Presto, …
- Snowflake recognized this trend early to build a cloud-native data warehousing solution … next class.

DREMEL: SQL

- In the early 2010s, many of Google's internal DBMS projects each had their own SQL dialect.
- The GoogleSQL project unified these redundant efforts to build a data model, type system, syntax, semantics, and function library.
- (Zombie?) Open-Source Version: ZetaSQL