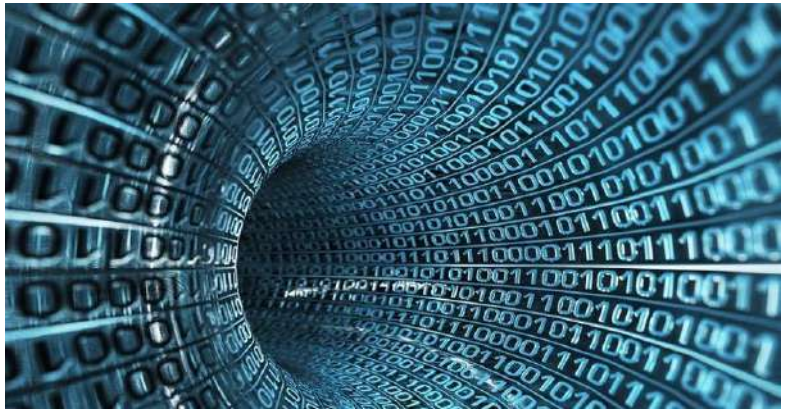
Sistemi Informativi Evoluti e Big Data



Tecnologie per i Big Data – Hadoop e HDFS

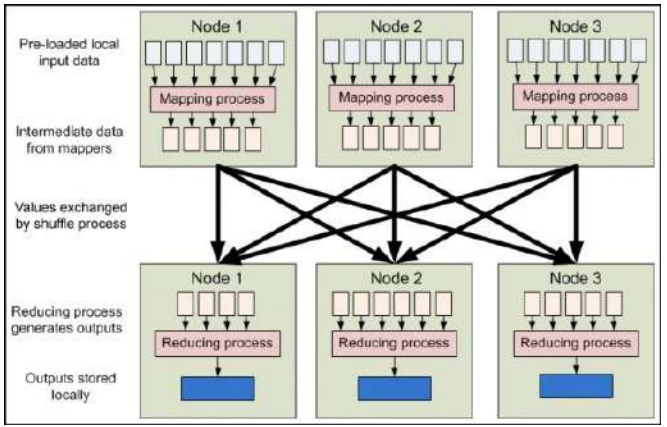
Università degli Studi di Brescia

Dipartimento di Ingegneria dell'Informazione

UNIVERSITY OF BRESCIA

Risorse e servizi distribuiti

- Architetture distribuite
  - Cluster di macchine
  - Scaling orizzontale
- Tolleranza ai guasti
  - Replicazione delle risorse
  - Eventual consistency
- Processing distribuito
  - Modello *shared-nothing*
  - Nuovi paradigmi di programmazione distribuita



Teorema di Brewer (o teorema CAP)

- Dimensione dei data set (crea problemi per gli odierni DBMS)
- I dati andrebbero distribuiti -> il **teorema CAP o di Brewer** è fondamentale per capire il comportamento di **sistemi SW distribuiti**, e progettarne l'architettura in modo da rispettare requisiti non funzionali stringenti
  - Un sistema informatico distribuito non può fornire simultaneamente tutte e tre le seguenti garanzie:
    1. **Consistency** (tutti i nodi vedono gli stessi dati nello stesso momento)
    2. **Availability** (ogni richiesta riceve una risposta su ciò che è riuscito/fallito)
    3. **Partition tolerance** (il sistema continua a funzionare nonostante arbitrarie perdite di messaggi)

Teorema di Brewer (o teorema CAP)

Consistency (*Totale coerenza*)

Un sistema distribuito è **completamente coerente** se preso un dato che viene scritto su un **nodo A** e viene letto da un altro **nodo B**, il sistema ritornerà l'ultimo valore scritto (quello *consistente*)

- Se si considera la cache di un singolo nodo la totale consistenza è garantita, così come la tolleranza alle partizioni
- Non si hanno però sufficiente disponibilità (fault-tolerance) e buone performance
- Se la cache è distribuita su due o più nodi, aumenta la disponibilità, ma vanno previsti dei meccanismi complessi che permettano ad ogni nodo di accedere ad un repository virtuale distribuito (e leggere lo stesso valore di dato)

Teorema di Brewer (o teorema CAP)

Availability (*disponibilità*)

Un sistema distribuito è **completamente disponibile** se ogni nodo **è sempre in grado di rispondere ad una query ed erogare i propri servizi** a meno che non sia *indisponibile*

- Banalmente un singolo nodo non garantisce la continua disponibilità
- Una cache distribuita mantiene nei vari nodi delle aree di backup in cui sono memorizzati i dati presenti su altri nodi
- Per realizzare la **continua disponibilità** si ricorre alla **ridondanza dei dati (su più nodi)**; ciò però richiede meccanismi per garantire la consistenza e problematiche riguardo la tolleranza alle partizioni

Teorema di Brewer (o teorema CAP)

Partition-Tolerance (*Tolleranza alle partizioni*)

È la capacità di un sistema di essere tollerante ad una aggiunta o una rimozione di un nodo nel sistema distribuito (*partizionamento*) o alla perdita di messaggi sulla rete.

1. Si consideri una configurazione in cui un solo cluster è composto da nodi su due diversi data center
2. Supponiamo che i data center perdano la connettività di rete; i nodi del cluster non riescono più a sincronizzare lo stato del sistema
3. I nodi si riorganizzano in sotto-cluster, tagliando fuori quelli dell'altro data center

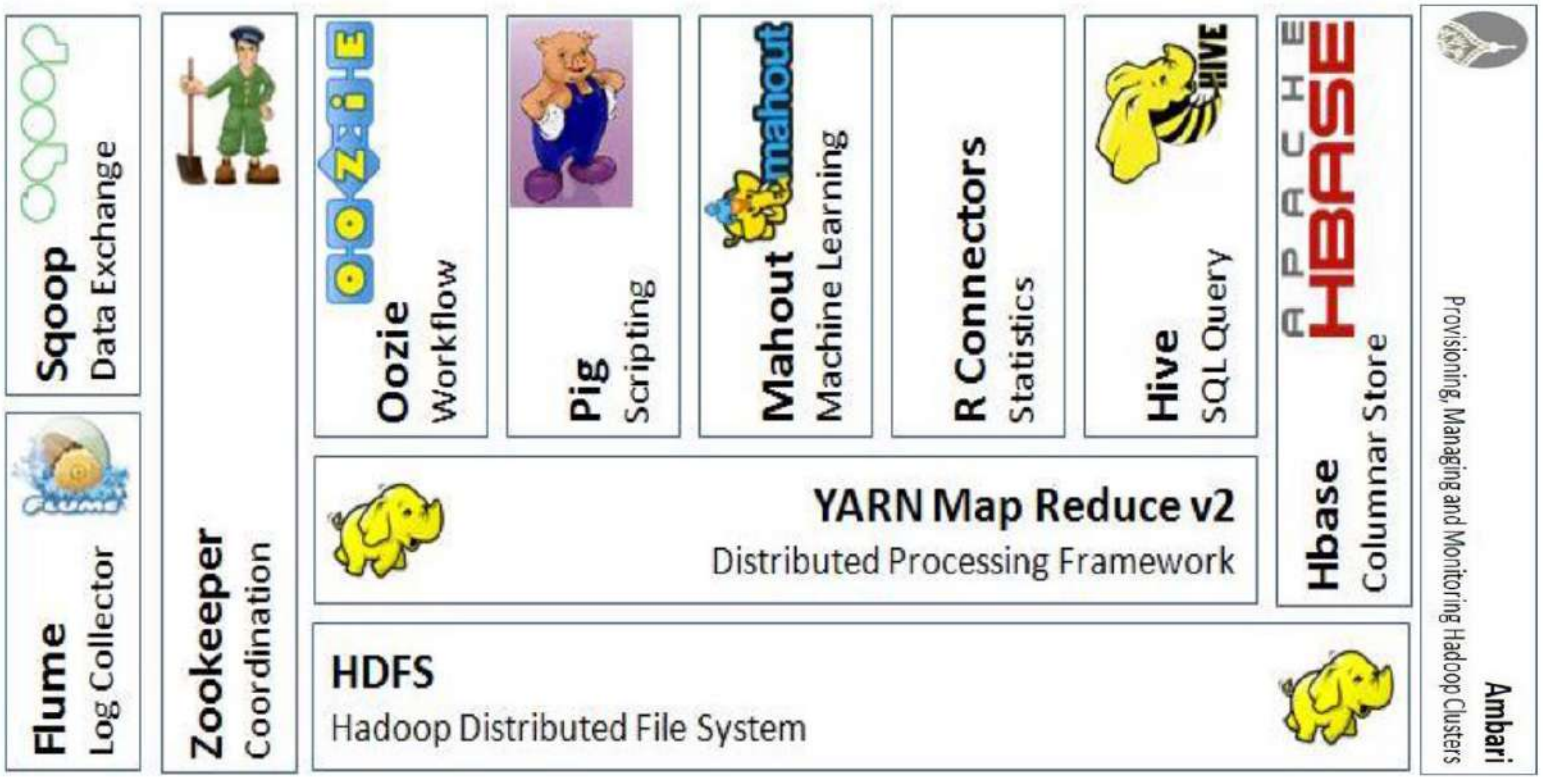
Il sistema continuerà a funzionare, in modo non coordinato e con possibile perdita di dati (es. assegnazione della stessa prenotazione a clienti diversi)

L'ecosistema Hadoop

- Un nuovo ambiente di programmazione parallela che lavora su cluster di macchine (commodity hardware) piuttosto che su supercomputer
- Creato da Google e inizialmente sviluppato da Yahoo!
- Uno stack di applicativi e di soluzioni con alla base un **file system distribuito** (HDFS)
- Principali scenari di utilizzo:
  - **Analytics (batch)** – collezione, trasformazione e modellazione di dati allo scopo di estrarre conoscenza e informazioni utili al decision-making; append-only I/O
  - **Streaming (near-real-time)** - Elaborazione di stream data (sequenze di dati resi disponibili incrementalmente nel tempo) allo scopo di monitorare e analizzare dati al volo su finestre temporali; stream I/O
  - **Interactive (real-time)** - Elaborazione dei dati per ritornare velocemente risultati (e-commerce, search engine, booking…); read-write I/O
- Diverse distribuzioni e configurazioni ad opera di grandi vendor (IBM, Microsoft, Oracle, Cloudera, Hortonworks, etc.)

Nota – Un discorso a parte merita Apache Spark

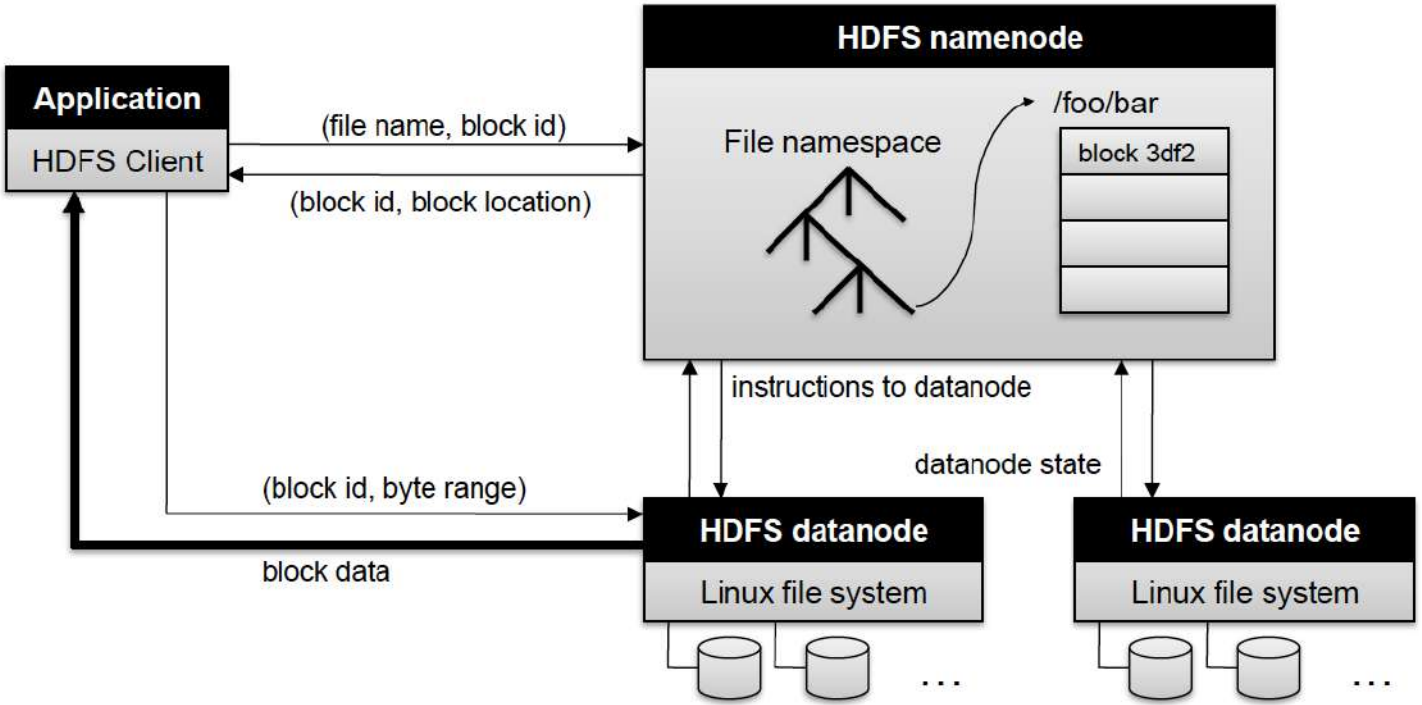
Hadoop Ecosystem



Organizzazione di un DFS

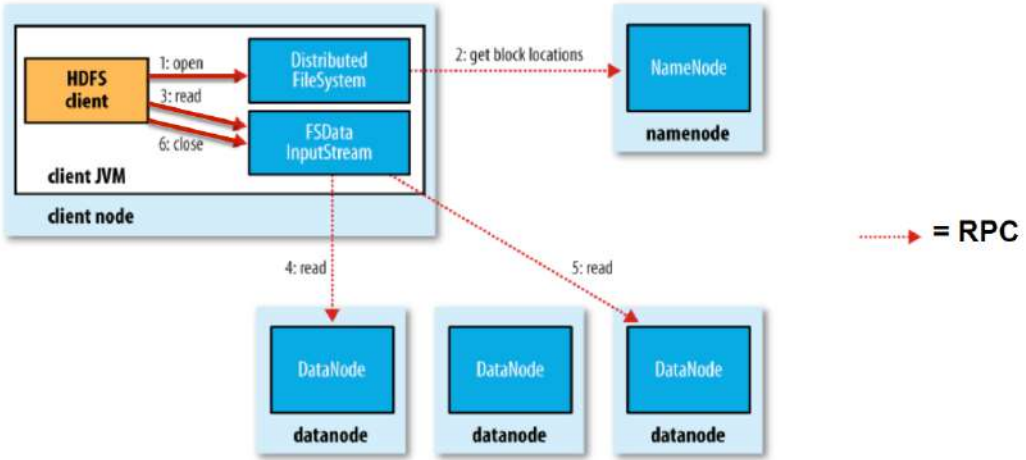
- I dati sono divisi in ***chunks***
  - Blocchi di grande entità (diversi GB) sono scoraggiati (in Hadoop non si va mai oltre i 64/128MB)
- I chunk sono replicati su più nodi (fattore di replica tipicamente 3+, ma configurabile), possibilmente appartenenti a rack diversi
- Un nodo speciale (***master node***) contiene le informazioni su dove sono salvati i vari chunk
- Il master node è a sua volta replicato (***no single point of failure***)
- Diverse implementazioni di un DFS
  - Google File System (GFS), il primo della categoria
  - Hadoop Distributed File System (HDFS), il più utilizzato
    - Master node = NameNode
    - Nodo = DataNode
    - Chunk = blocco

HDFS - Architettura



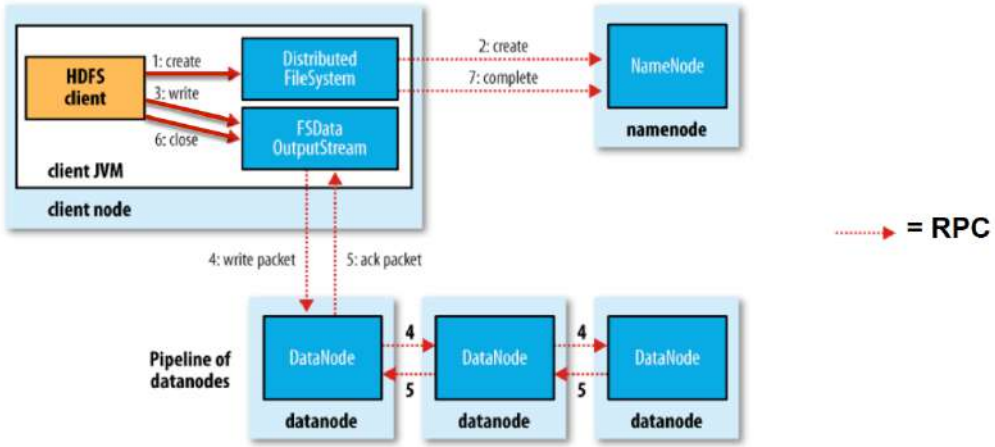
HDFS – Lettura dei file

1. Il nodo *client* che intende leggere un file (o parte di esso) manda una richiesta al *NameNode*
2. Il *NameNode* risponde indicando quali sono i blocchi (chunk) che contengono i dati richiesti e quali *DataNode* contengono tali blocchi
3. Il nodo *client* contatta i *DataNode* senza passare più dal *NameNode*
4. I *DataNode* leggono i blocchi e rispondono al nodo *client*
5. …
6. Il nodo *client* termina la procedura

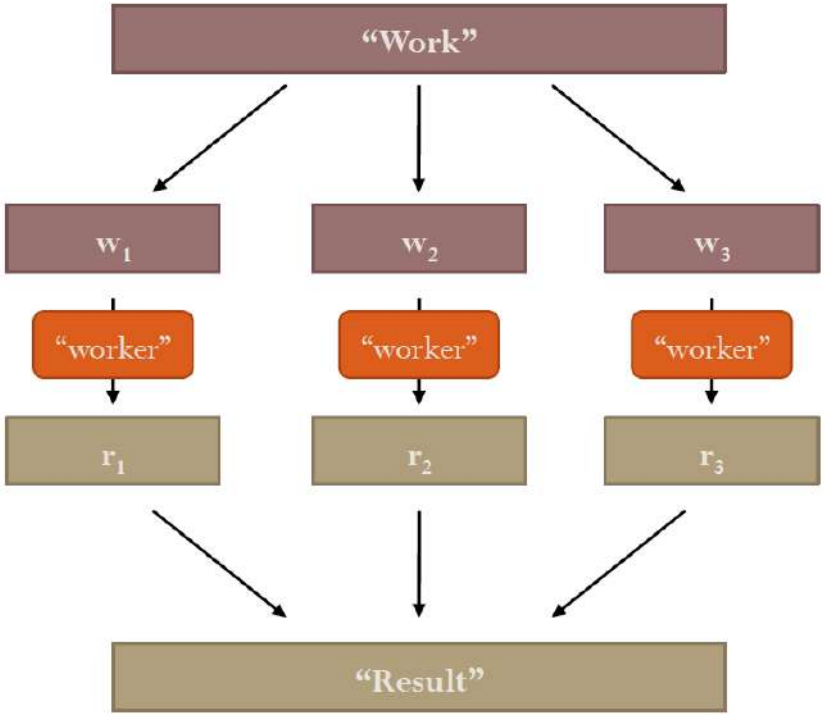


HDFS – Scrittura dei file

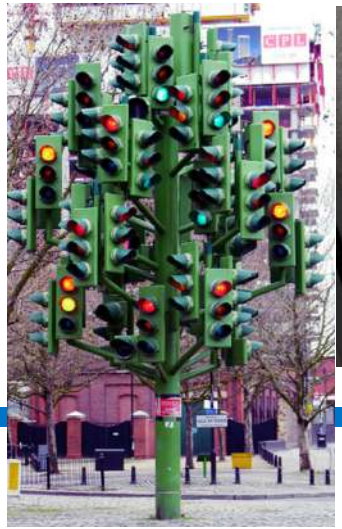
1. Il nodo *client* contatta il *NameNode* che identifica il *DataNode* primario e i *DataNode* che conterranno le repliche dei blocchi da scrivere
2. Il *NameNode* include nella risposta tutti i *DataNode* identificati
3. Il nodo client manda i blocchi da scrivere su tutti i *DataNode* in qualsiasi ordine; i blocchi inviati sono salvati in buffer sui *DataNode*
4. Il nodo client manda un “commit” al *DataNode* primario, che decide l’ordine di scrittura delle repliche
5. Dopo che tutti i *DataNode* hanno scritto, il *DataNode* primario manda una segnalazione al nodo client
6. Il nodo client dichiara chiusa l’operazione di scrittura
7. Tutti i cambiamenti effettuati sui *DataNode* sono memorizzati nel *NameNode*



Distributed computing: una concezione superata

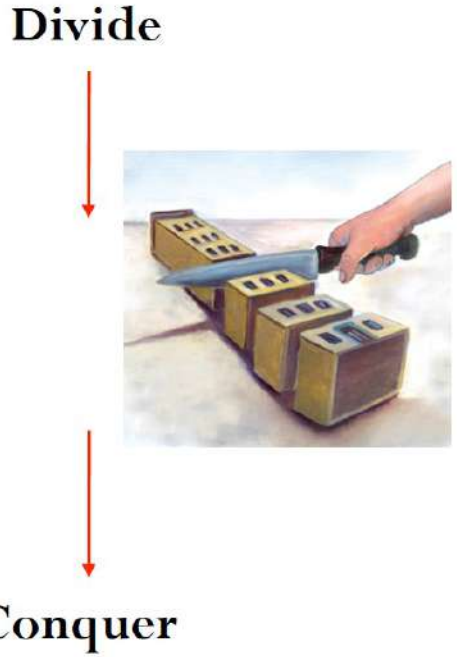


- Come suddividere il lavoro se le unità di lavoro sono più dei workers?
- Come condividere/combinare i risultati parziali?
- Che succede se un worker cade?



Cos'è MapReduce

- Un **paradigma di programmazione** per l'elaborazione distribuita di grandi quantità di dati
- Un **framework** per sviluppare ed eseguire i programmi pensati per tale elaborazione
  - Esecuzione di molteplici task in parallelo
  - Ridondanza e fault-tolerance
  - Principio della **data locality**
- Varie implementazioni disponibili: Google, Hadoop, …



Struttura tipica di un problema

- Iterare le elaborazioni su un numero elevato di record in parallelo
- Estrarre qualcosa di interessante ad ogni iterazione
- Combinare (*shuffle*) e riordinare i risultati intermedi di diverse iterazioni concorrenti
- Aggregare i risultati intermedi nel risultato finale

map

reduce

L’idea dietro MapReduce

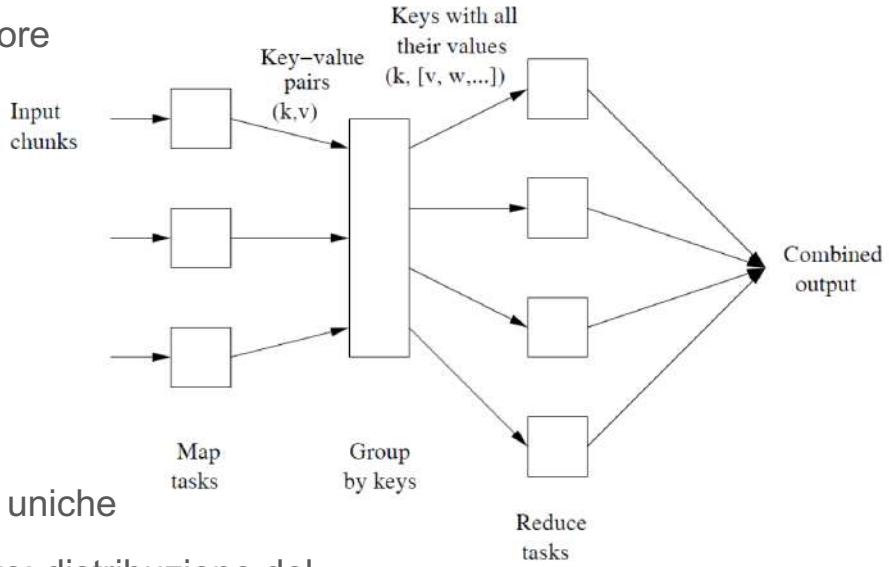
- L’idea principale è nascondere i dettagli di basso livello al programmatore
  - Problemi legati alla sincronizzazione, alla distribuzione delle risorse, etc.
  - Il programmatore deve preoccuparsi solo di implementare le funzioni di ***`map`*** e ***`reduce`*** ispirandosi alla *programmazione funzionale*
  - Il framework sottostante si occupa dell’esecuzione distribuita della computazione, secondo i principi di
    - ***data locality*** – (a differenza delle soluzioni *HPC – High Performance Computing*) i dati risiedono sui nodi dove sono elaborati, che si trovano in corrispondenza dei *DataNode* HDFS
    - ***shared nothing architecture*** – ogni nodo è indipendente e autosufficiente

Concetti chiave del modello “MapReduce”

- Legata al concetto di “chiave-valore”
- La funzione MAP processa i dati in input come coppia *(k,v)* e produce un insieme intermedio di coppie chiave-valore $(k_1,v_1)(k_2,v_2)\ldots(k_n,v_n)$
- Il framework raccoglie le coppie con la stessa chiave $(k,[v_1,\ldots v_n])$
- La funzione *reduce* processa i risultati intermedi, $(k,[v_1,\ldots v_n])$ combinandoli per chiave e generando il risultato finale

MapReduce programming (I)

- La struttura dati elementare è la coppia chiave-valore
- Il programmatore deve specificare due funzioni
  - **map** (k_1,v_1) -> $[(k_2,v_2)]$
  - **reduce** $(k_1,[v_1])$ -> $[(k_2,v_2)]$
- Dove
  - *(k,v)* rappresenta una coppia *(chiave,valore)*
  - *[…]* rappresenta una lista
  - Le chiavi non devono necessariamente essere uniche
- Il framework di esecuzione si occupa di tutto il resto: distribuzione del carico di lavoro, gestione della sincronizzazione (gestione dei risultati intermedi, *shuffle*), gestione errori e fault



MapReduce programming (II)

- Un programma MapReduce, solitamente indicato come *job*, è composto da
  - Il codice per le funzioni **map** e **reduce**
  - I parametri di configurazione (dove si trovano gli input, dove salvare gli output)
  - Gli input da sottomettere alla funzione di **map**
- Ogni job MapReduce viene tradotto in diversi *task*, ovvero unità più piccole di lavoro
  - **map** task
  - **reduce** task
- Input e output del job MapReduce sono salvati nel sottostante file system distribuito
- Diversi job MapReduce possono essere combinati in sequenza per le operazioni più complesse
- A livello architetturale, il **JobTracker** si occupa della gestione del job (in maniera trasparente per l'utente), mentre sui singoli nodi i **TaskTracker** eseguono i task sotto la supervisione del **JobTracker**

Un esempio in pratica: WordCount

- **Problema**: conteggiare il numero di occorrenze di ciascuna parola in una collezione di documenti
- **Input**: un repository di documenti, dove ogni documento è un elemento
- **Map**: legge un singolo documento ed emette una sequenza di coppie chiave-valore, dove le chiavi sono le parole del documento e i valori sono tutti uguali a 1

$$(w_1,1), (w_2,1), \ldots (w_n,1)$$

- **Shuffle**: raggruppa per chiave

$$(w_1,[1,1,1,\ldots1]), \ldots (w_n, [1,1,1,\ldots1])$$

- **Reduce**: somma i valori corrispondenti a ciascuna chiave

$$(w_1,k), \ldots (w_n, h)$$

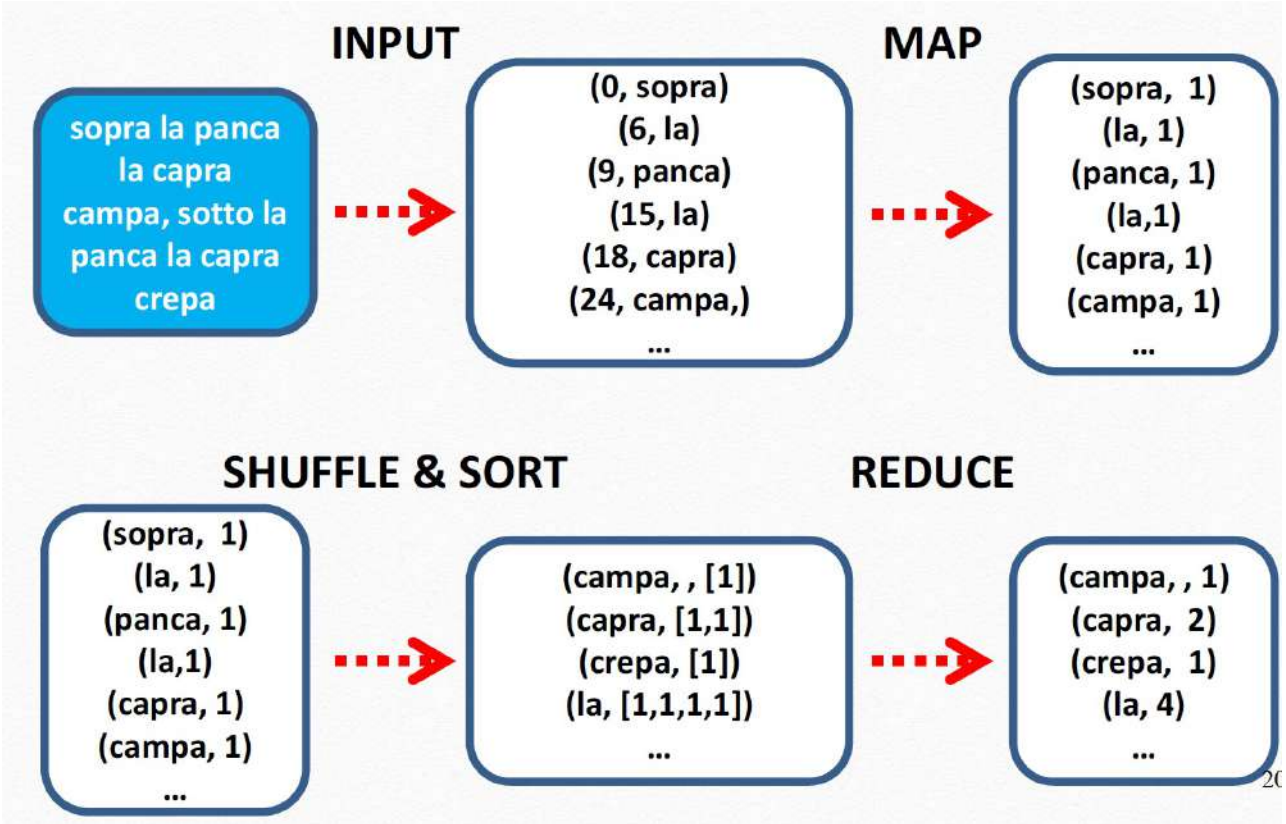
- **Output**: coppie *(w,m)*, dove *w* è una parola che appare almeno una volta in uno qualsiasi dei documenti in input e *m* è il numero di volte che la parola *w* appare in tutti i documenti del repository

Un esempio in pratica: WordCount

```
map(String docid, String text):
    for each word w in text:
        emit (w, 1);

reduce(String term, counts[]):
    int sum = 0;
    for each c in counts:
        sum += c;
    emit (term, sum);
```

WordCount: flusso logico



Un altro esempio: WordLengthCount

- **Problema**: conteggiare quante parole di una certa lunghezza esistono in una collezione di documenti
- **Input**: un repository di documenti, dove ogni documento è un elemento
- **Map**: legge un singolo documento ed emette una sequenza di coppie chiave-valore, dove ogni chiave è la lunghezza di una parola e il valore è la parola stessa

$$(i,w_1),\ \ldots\ (j,w_n)$$

- **Shuffle and sort**: raggruppa per chiave

$$(1,[w_1,\ \ldots w_k]),\ \ldots\ (n,[w_r,\ \ldots w_s])$$

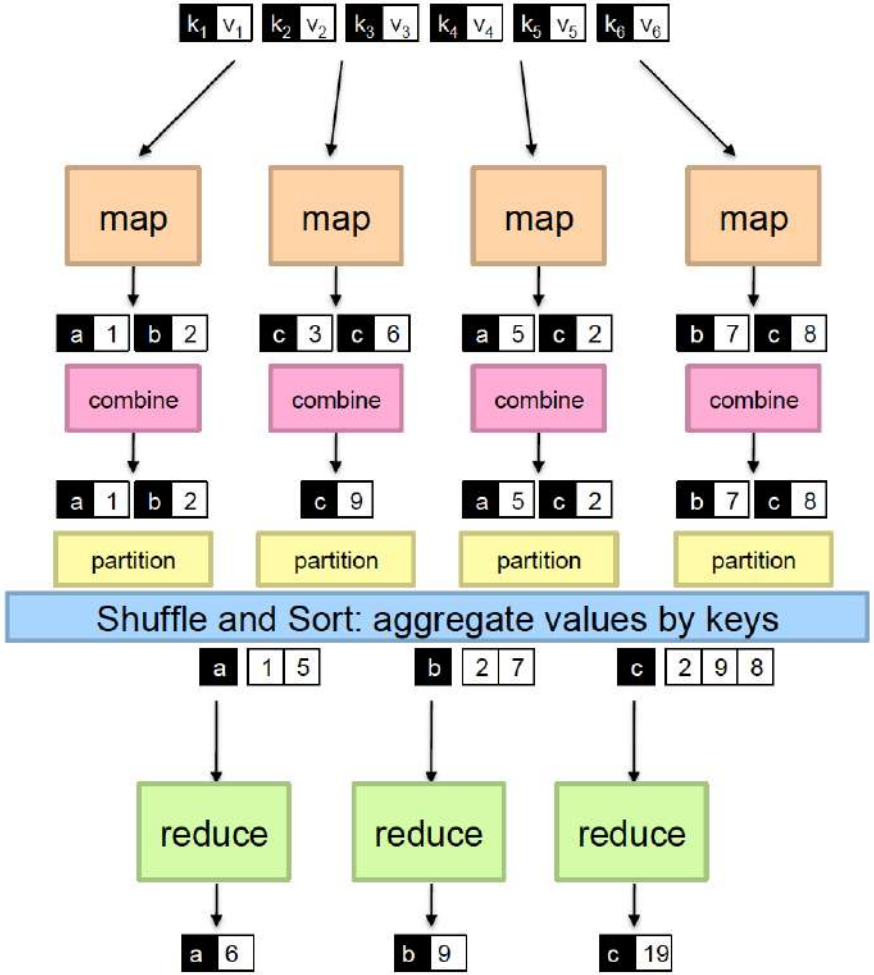
- **Reduce**: conta il numero di parole in ciascuna lista

$$(1,k),\ \ldots\ (n,\ h)$$

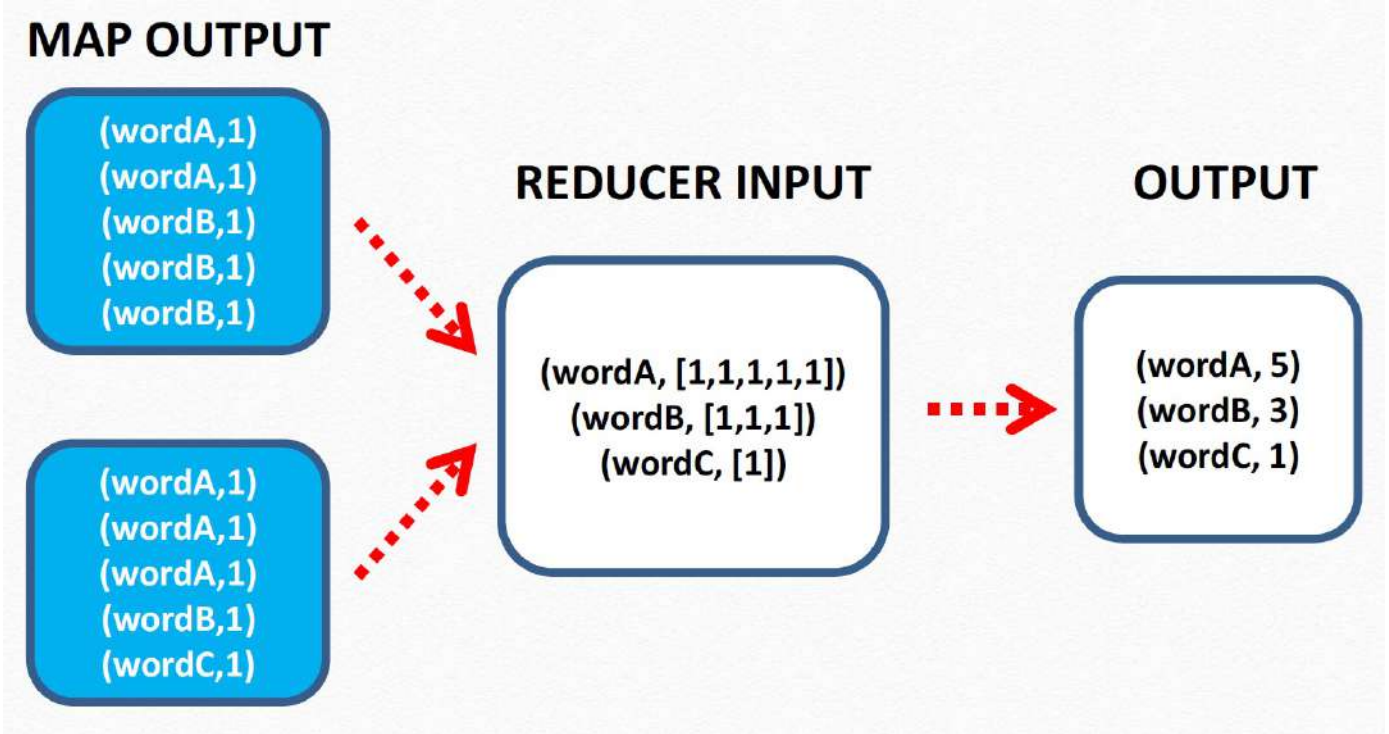
- **Output**: coppie *(l,n)*, dove *l* è una lunghezza e *n* è il numero di parole di lunghezza *l* in tutti i documenti del repository

Utilizzo dei *combiners*

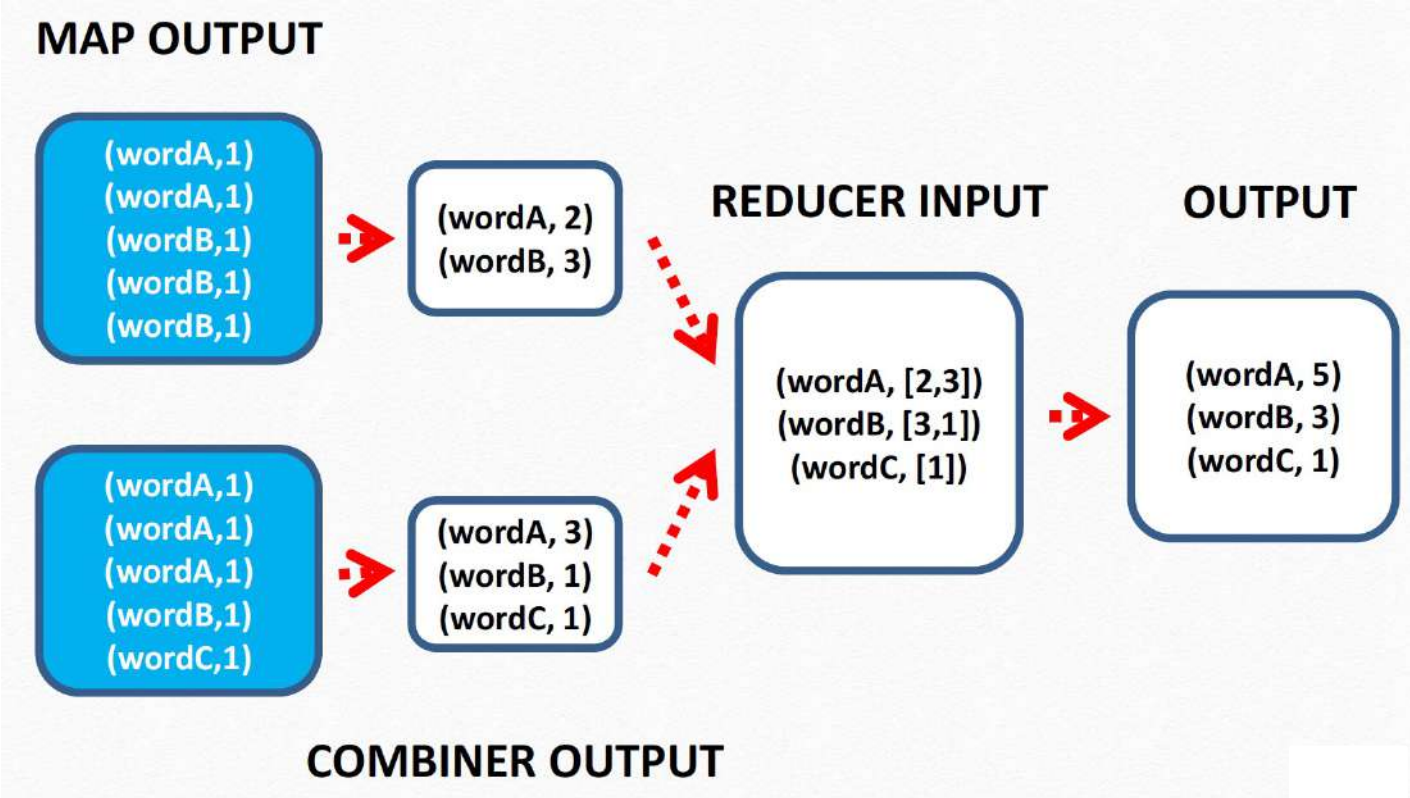
- Quando la funzione **reduce** è associativa e commutativa
- Molto spesso la stessa funzione usata come **reduce** e come **combine**
- La funzione **combine** non sostituisce le operazioni di **shuffle** e **sort**
- La funzione **combine** riduce il volume dei risultati intermedi e il traffico di rete



WordCount: senza Combiner



WordCount: utilizzo di un Combiner

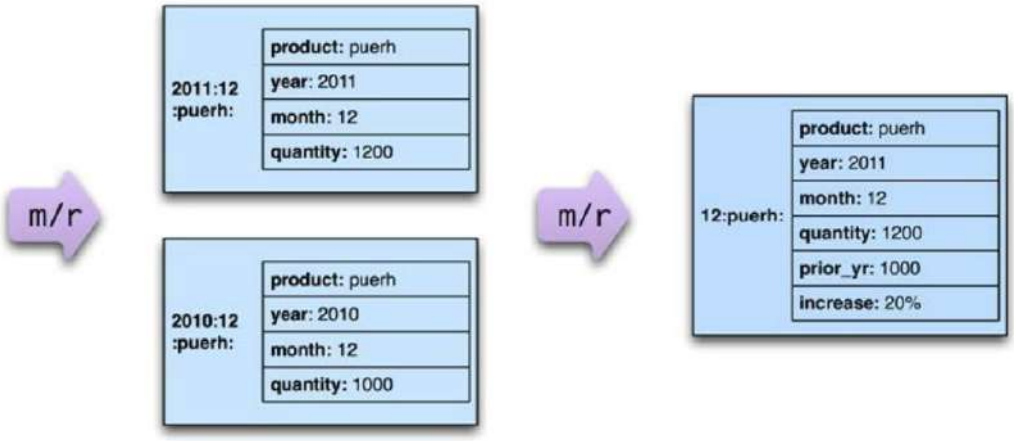


Sequenza di passi *MapReduce*

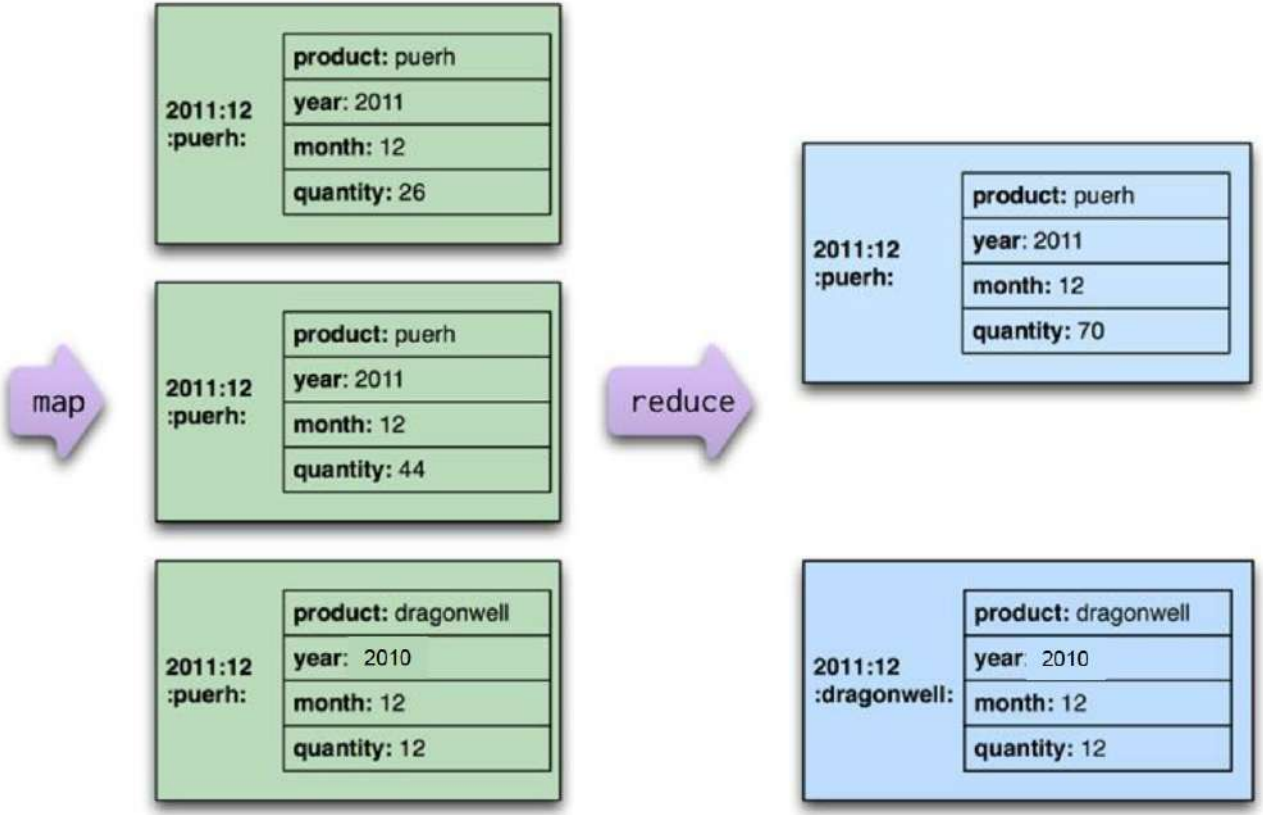
- Se l'operazione di *MapReduce* diventa molto complessa, può essere comodo decomporla in diversi passaggi, dove in ogni passaggio viene applicato il paradigma *MapReduce*; gli output di un passaggio diventano input del successivo
- I risultati intermedi potrebbero anche essere salvati nel file system distribuito e riutilizzati molte volte, facilitando il ***riuso***
- Le primissime fasi di operazioni *MapReduce* complesse sono le più costose da un punto di vista computazionale, quindi il meccanismo di riuso potrebbe portare ulteriori vantaggi

Two-stages MapReduce: un esempio

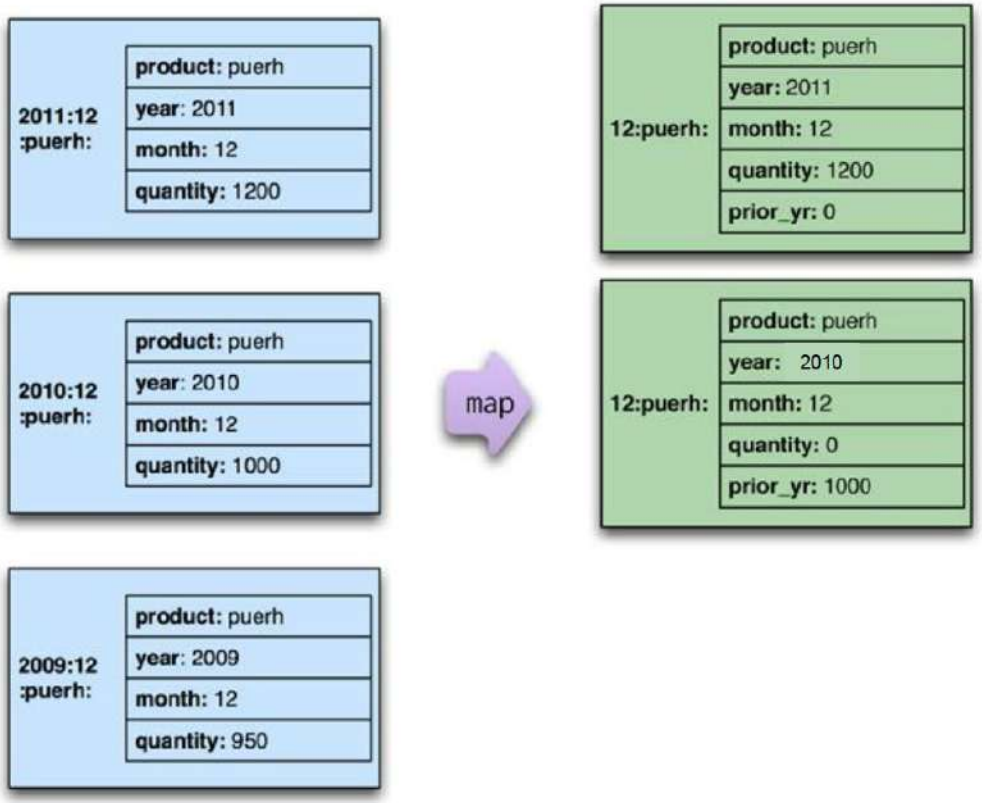
- **Obiettivo**: confrontare mese per mese le vendite di prodotti nell'anno 2011 con quelle dell'anno precedente
- ***Primo passaggio***: produce i record delle vendite per ogni singolo prodotto e ogni singolo mese dell'anno 2011 e del precedente
- ***Secondo passaggio***: produce il risultato finale per ogni prodotto confrontando il valore del 2011 con quello del 2010



Primo passaggio: ***map*** *e* ***reduce***



Secondo passaggio: ***map***



Secondo passaggio: ***reduce***