# Параллельная обработка больших графов

**Занятие 4**

**А.С. Семенов**

dislab.org

НИЦ ВТ

DISLab

# Поиск всех кратчайших путей от заданной вершины в графе (Single Source Shortest Paths, SSSP)

$$G = (V, E, W), W : E \to R, directed$$

$$p(v_0, v_k) = \langle v_0, v_1, \ldots, v_k \rangle$$

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

$$\delta(u, v) = \begin{cases} \min\{w(p)\}, \exists p(u, v) \\ \infty, otherwise \end{cases}$$

# Алгоритмы решения
# Single Source Shortest Paths

- Дейкстры, 1959
- Беллмана-Форда, 1969
- Дельта-степпинг, 2003

# SSSP, Вспомогательные процедуры

```
InitializeSingleSource(G, s)
for all v ∈ V
    d[v] = inf
end for
d[s] = 0

Relax(u, v)
if d[v] > d[u] + w(u, v)
    d[v] = d[u] + w(u, v)
    p[v] = u
end if
```

# SSSP, Алгоритм Дейкстры

```
Dijkstra (G, w, s)
InitializeSingleSource(G, s)
S = {}
Q = V
while Q != {}
    u = GetMin (Q)
    S = S U {u}
    Q = Q / {u}
    for all v in Adj[u]
        Relax (u, v)
    end for
end while
```

- G(V, E, W) – ориентированный, w(e)>=0, w **in** W
- последовательный

Сложность $O(N^2+M)$, $O(N \lg N + M)$

# SSSP, Алгоритм Беллмана-Форда

```
BellmanFord (G, s)
InitializeSingleSource(G, s)
for i = 1 to |V| - 1
    for all (u, v) in E
        Relax(u, v)
    end for
end for
```

- G(V, E, W) – ориентированный, W: E -> R
- может определять наличие циклов с отрицательным весом

Сложность O(N M)

# SSSP, Алгоритм Беллмана-Форда

```
BellmanFord (G, s)
InitializeSingleSource(G, s)
for i = 1 to |V| - 1
    #pragma omp parallel for
    for all (u, v) in E
        Relax(u, v) // атомарно
    end for
end for
```

# SSSP, Алгоритм delta-stepping (2003, U. Meyer, P. Sanders) (1/4)

- G(V, E, W) – ориентированный, w(e)>=0, w **in** W
- delta **in** R, delta > 0

```
DeltaStepping (G, s, delta)
InitializeDeltaStepping
i = 0
while true
    processBucket(i)
    i = min(k > i : B_k != {} )
    if i == inf then break
end while
```

# SSSP, Алгоритм delta-stepping (2/4)

**InitializeDeltaStepping** (G, s)
**for all** u in V
    d(u)= inf
**end for**
d(s) = 0
$B_0$={s}
$B_{inf}$ = V / {s}
$B_1$=$B_2$=…= {}

$$B_i = \{v \in V : d[v] \in [i * delta; (i+1) * delta)\}$$

# SSSP, Алгоритм delta-stepping (3/4)

**ProcessBucket** (i)
$A = B_i$
**while** A != {}
    **for all** u **in** A, **for all** e = (u, v) **in** E
        Relax(u, v)
    **end for**
    A’ = {x : d(x) изменен в предыдущем цикле}
    $A = B_i \cap A'$
**end while**

# SSSP, Алгоритм delta-stepping (4/4)

**Relax** (u, v)

i = $\lfloor d(v)/delta \rfloor$ // старый bucket

$d(v) = \min(d(v), d(u) + w(u,v))$

j = $\lfloor d(v)/delta \rfloor$ // новый bucket

**if** j < i **then** переместить v из $B_i$ в $B_j$

# SSSP, Алгоритм delta-stepping

**DeltaStepping** (G, s, delta)
InitializeDeltaStepping
i = 0
**while** true
    processBucket(i)
    i = min(k > i : $B_k$ != {} )
    **if** i != inf **then break**
**end while**

- Для случайных графов «хороший» показатель delta = 1/d, где d – максимальная степень вершины, w равномерно распределены в [0; 1]
  - Сложность O(M log N)
  - Количество итераций $O(\frac{d_c}{delta} * \frac{\log n}{\log \log n})$
  - $d_c$ - максимальная длина пути

# SSSP, Алгоритм delta-stepping

**ProcessBucket** (i)
$A = B_i$
**while** A != {}
    **#pragma omp parallel for**
    **for all** u **in** A, **for all** e = (u, v) **in** E
        Relax(u, v) // **атомарно**
    **end for**
    A’ = {x : d(x) changed in previous for all}
    $A = B_i \cap A'$
**end while**

# Варианты заданий

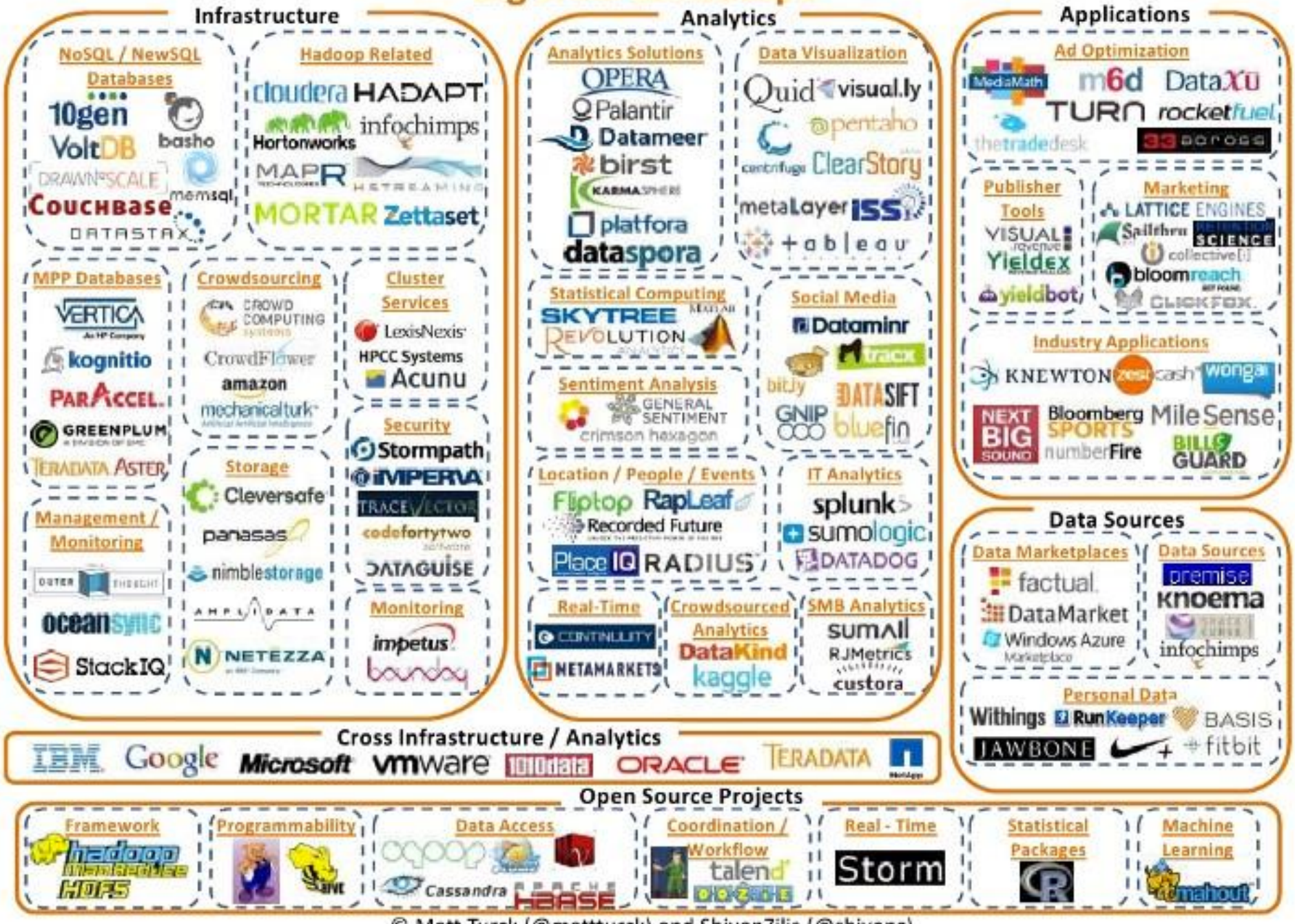
Реализовать алгоритм дельта-степпинг на распределенной памяти при помощи:

- Функции MPI-1
- mpi4py
- Односторонние коммуникации MPI-3
- Библиотека OpenSHMEM
- Charm++, charm4py
- Chapel
- Apache Spark, GraphX
- Использование ускорителей
- Combinatorial BLAS
- Neo4j, … ?

- Реализация должна быть масштабируемой **по памяти** и обладать **сильной масштабируемостью**
- Для экзамена автоматом реализация должна обладать особенностью

- **Дедлайны**: 1.04 – корректность
- 6.05 - масштабируемость

# Big Data Landscape



# Apache Hadoop

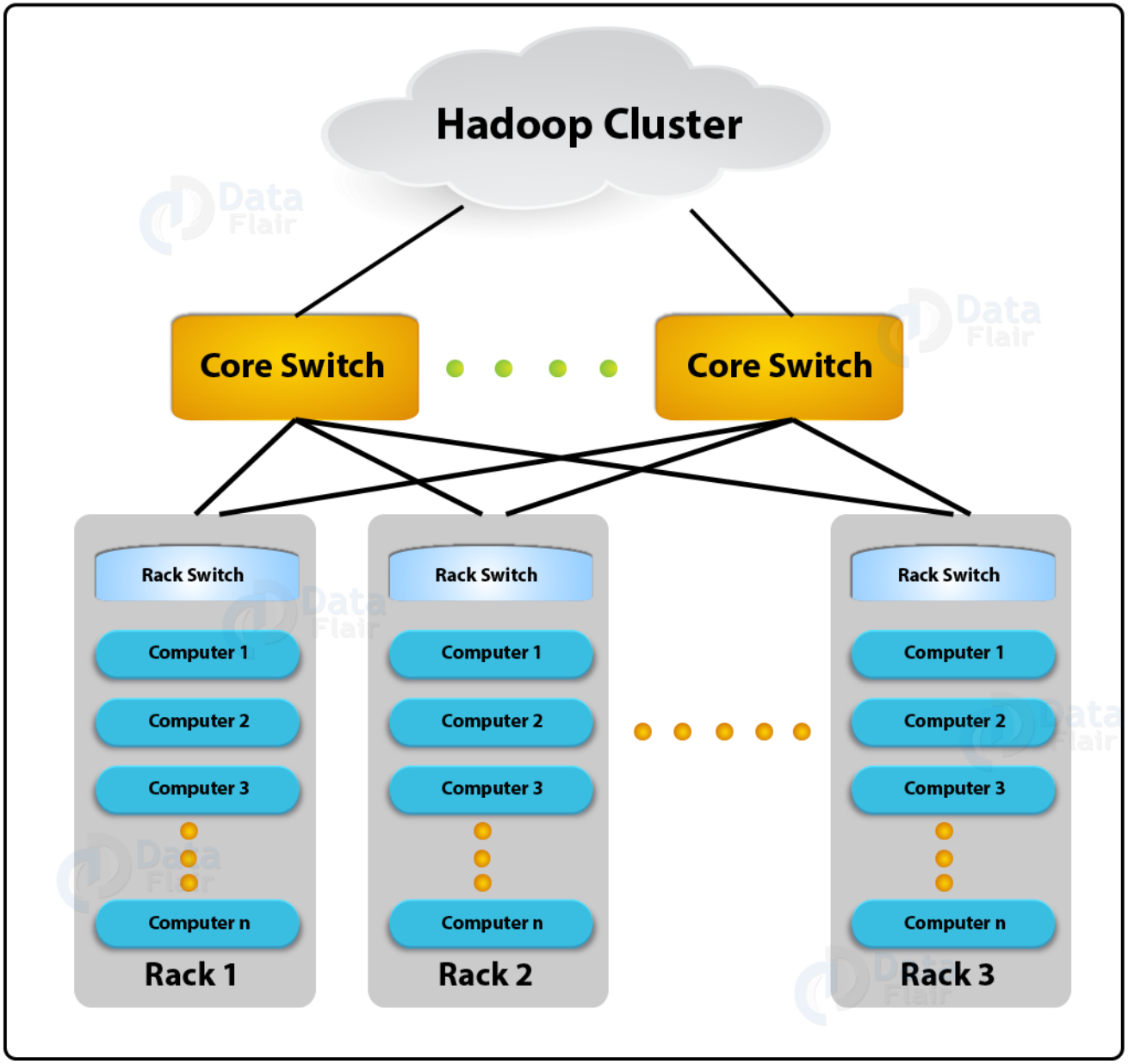

- 

- 

- 

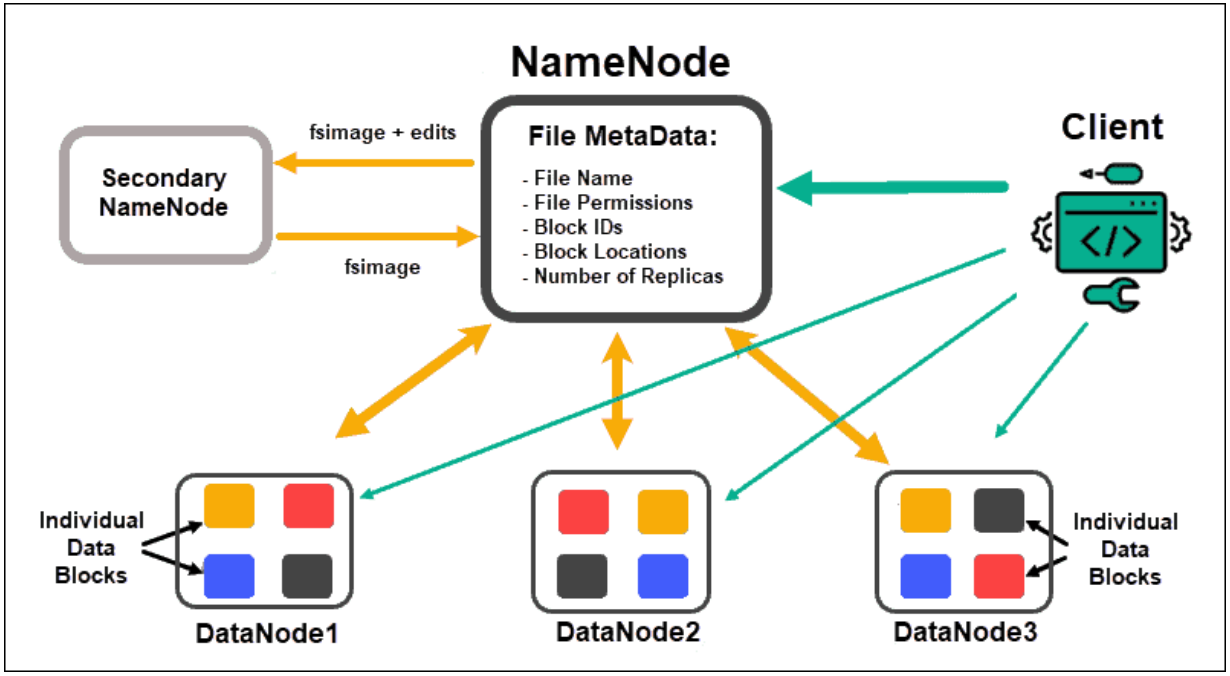

| | | |
|---|---|---|
| 1 | PB | Hadoop/Spark Cluster |
| 1 | TB | Hadoop/Spark Cluster |
| 100 | GB | Postgres, Hadoop/Spark Cluster |
| 10 | GB | pandas, Spark, Postgres |
| | GB | pandas, Spark, Postgres, CLI |
| | MB | Excel, pandas, Postgres, CLI |
| | KB | Excel, CLI |

# Hadoop Cluster



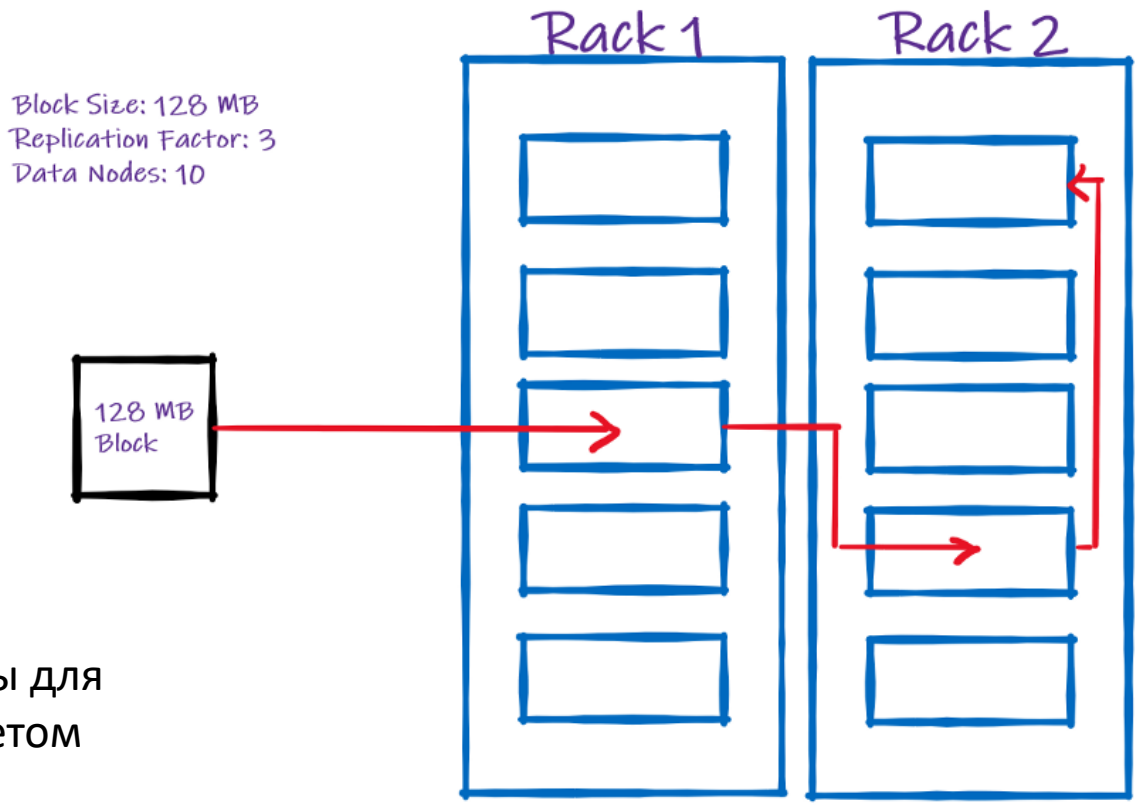
# HDFS

- Работает полностью в user-space
- Отказоустойчивая
- Горизонтально высокомасштабируемая
- Разработана для обработки небольшого числа больших файлов
- Обеспечивает потоковую обработку: 1 запись, большое число чтений




- Разбиение на блоки по size (64 МБ), коэффициент репликации
- Узел имен: метаданные - информация о иерархии, о распределении
  - Номер стойки
- Standby NameNode (Hadoop 2.0)

# HDFS – репликация




**Алгоритм репликации:**

- NameNode выбирает узлы для размещения реплик с учетом балансировки
- 1-я реплика размещается на первом узле из списка
- 2-я реплика копируется на другой узел
- И так далее

# LISP

- 1958 John McCarty (MIT) Lisp (LISt Processor)
- Lisp – Функциональный язык программирования
- **(map f list)**
- **(reduce q list)**

- **(map square ‘(1 2 3 4))**
  -> (1 4 9 16)
- **(reduce + ‘(1 4 9 16))**
  -> 30
- **(reduce + (map square ‘(1 2 3 4)))**
  -> 30

# LISP

- 1958 John McCarty (MIT) Lisp (LISt Processor)
- Lisp – Функциональный язык программирования
- **(map f list)**
- **(reduce q list)**

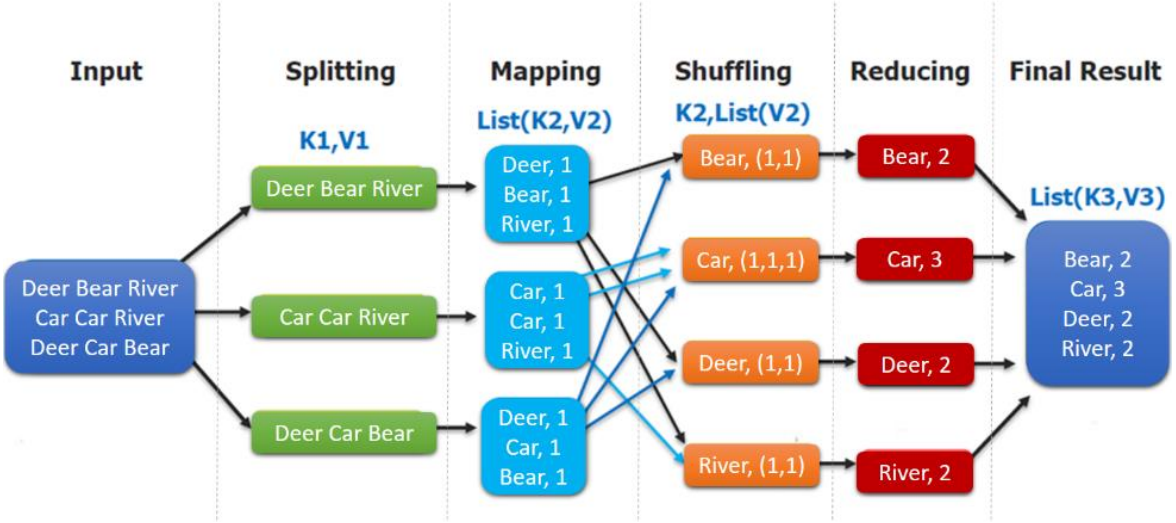
- **(map square ‘(1 2 3 4))**
  -> (1 4 9 16)
- **(reduce + ‘(1 4 9 16))**
  -> 30
- **(reduce + (map square ‘(1 2 3 4)))**
  -> 30

**Функциональное программирование:**

- Функция – объект 1 класса (можно сохранять как значение, передавать как аргумент, возвращать значение)
- Отсутствие side-эффектов
- Неизменяемость значений переменных

# MapReduce 2002 [Dean, 2004]

**1. Input.** Получение входных данных на главном узле (master node)
**2. Split.** Главный узел делит входные данные на части (K1) и передает рабочим узлам (worker node)
**3. Map.** Каждый рабочий узел применяет функцию Map к каждому K1 (локальные данные) и записывает результат в формате «ключ-значение» (K2, V2) во временное хранилище




**4. Shuffle (&Sort).** Рабочие узлы перераспределяют данные на основе ключей, чтобы все данные с одним ключом K2 лежали на одном рабочем узле.
**5. Reduce.** Рабочие узлы применяют функцию Reduce к каждой группе результатов с одинаковым K2 и передает в главный узел

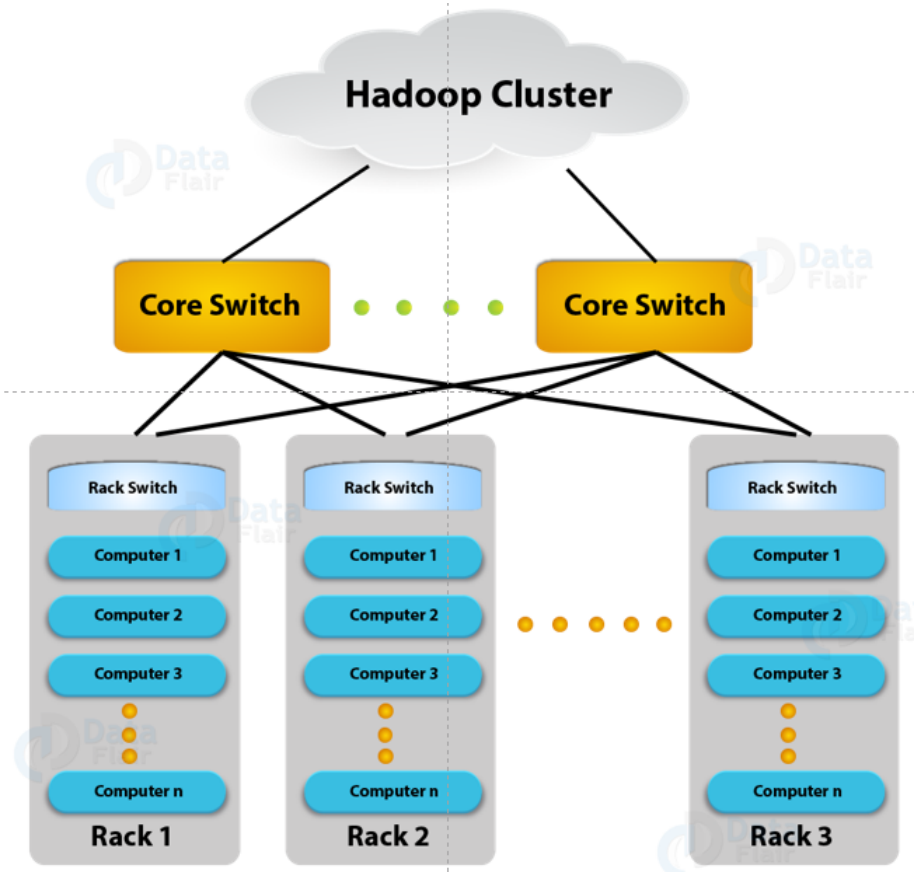

[Dean, 2004] J. Dean, S. Ghemawat MapReduce: Simplified Data Processing on Large Clusters
https://www.usenix.org/legacy/event/osdi04/tech/full_papers/dean/dean_html/

# MapReduce

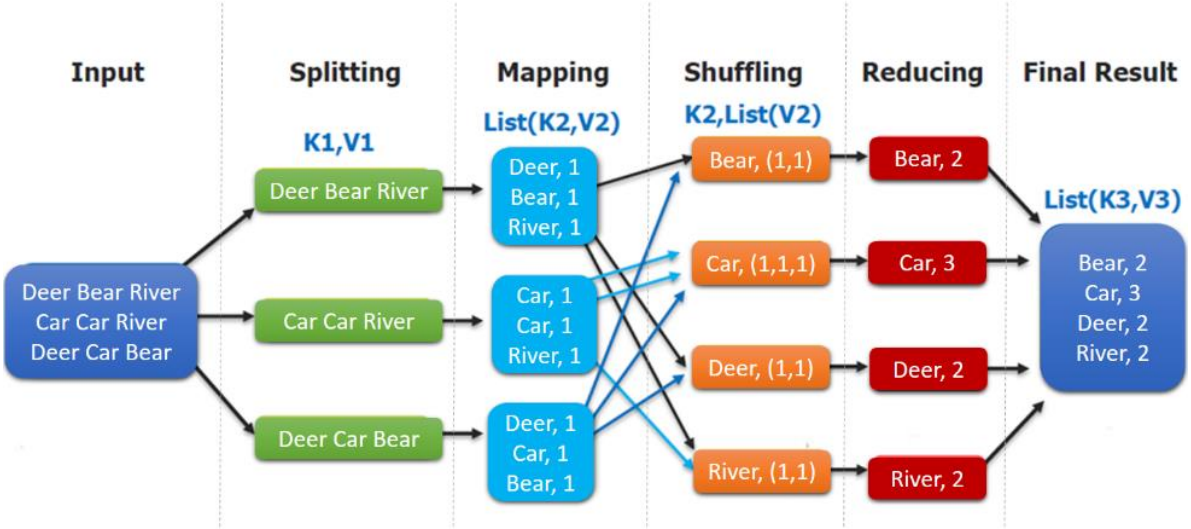
- **map (in_key, in_value)**
  -> (out_key, intermediate_value) list
- **reduce (out_key, intermediate_value list)**
  -> out_value list

**Основные преимущества:**

- Автоматическое распределение данных и распараллеливание вычислений
- Балансировка нагрузки
- Отказоустойчивость
  - Повтор выполнения операций в случае отказов



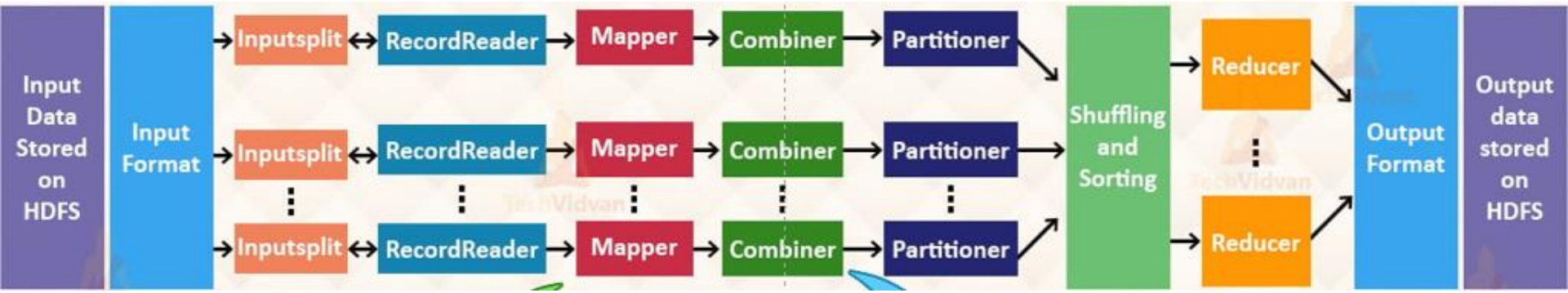
# Пример




**Java**

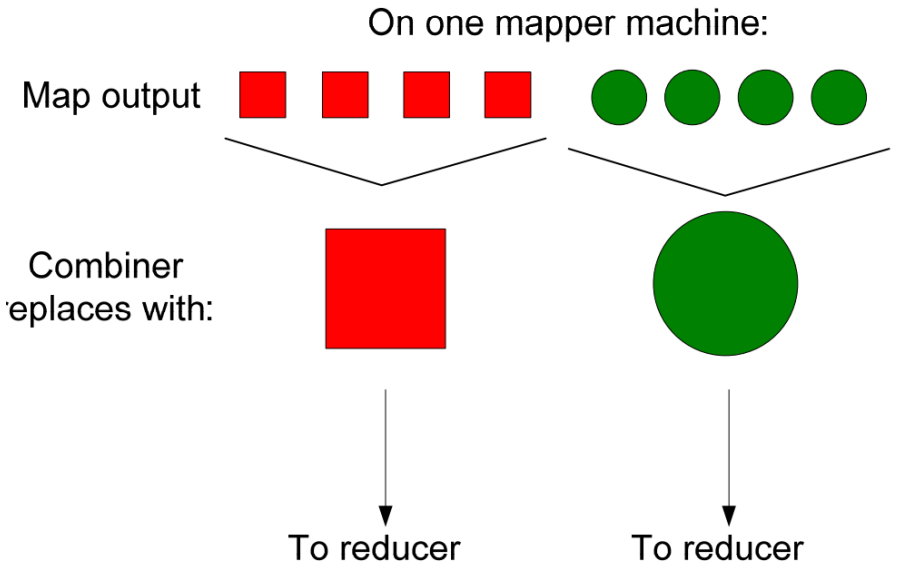
```
public void map(LongWritable key, Text value, Context con) throws … {
        String line = value.toString();
        String[] words=line.split(",");
        for (String word: words ) {
                Text outputKey = new Text(word.toUpperCase().trim());
                IntWritable outputValue = new IntWritable(1);
                con.write(outputKey, outputValue);
}}}

public void reduce(Text word, Iterable<IntWritable> values, Context con) throws … {
        int sum = 0;
        for (IntWritable value : values)   {
                sum += value.get();   }
        con.write(word, new IntWritable(sum));
}
```

# MapReduce




**Оптимизации:**

- Combiner – запуск на узлах mapper после фазы map (локальный mini reduce) для экономии пропускной способности



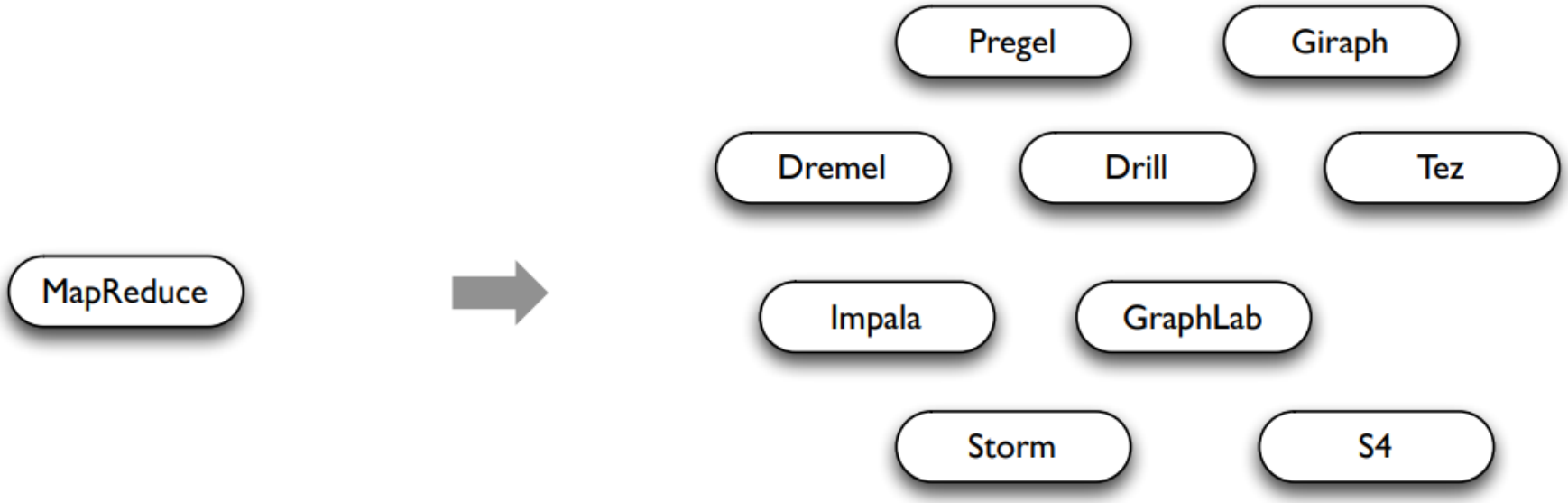
# MapReduce

**Реализации MapReduce:**

- Google File System
- Apache Hadoop
- Yahoo
- Amazon AWS

**Недостатки MapReduce:**

- Трудоемкость программирования в чистой парадигме MapReduce
- Каждый раз результаты складываются на диск, производительность

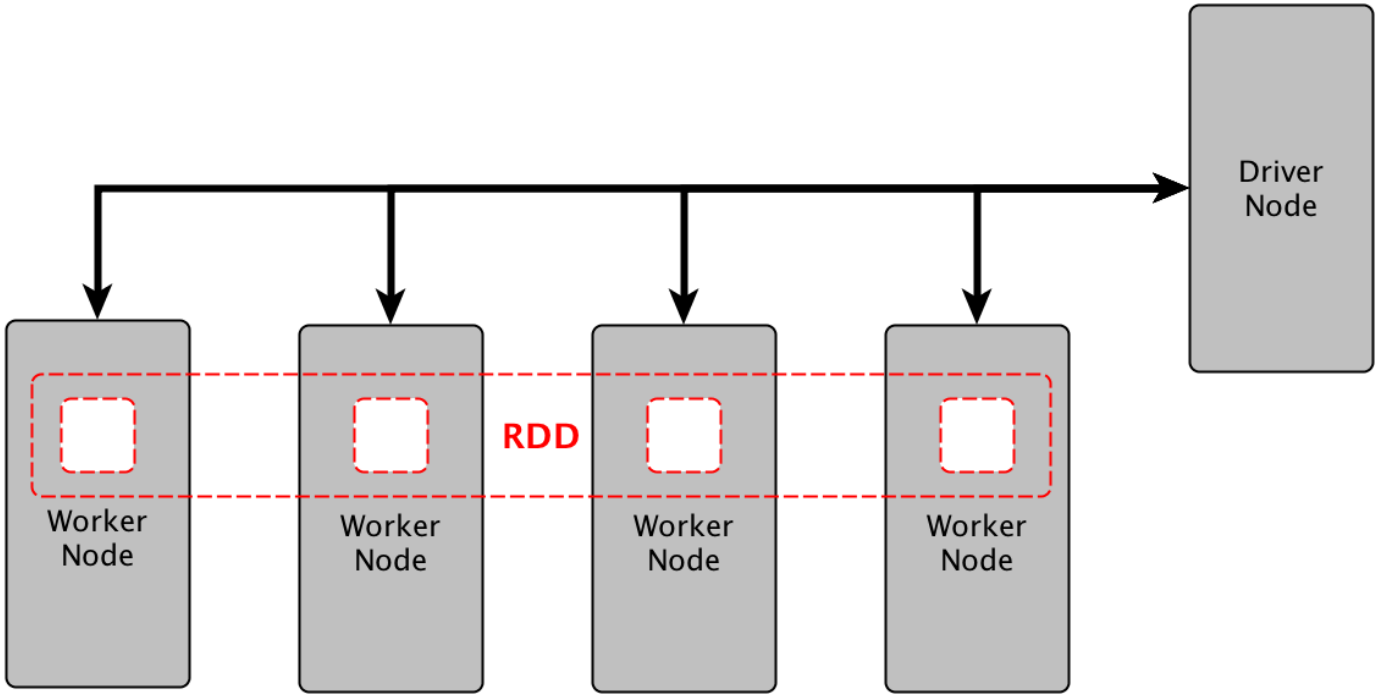



**General Batch Processing**

**Specialized Systems:**
iterative, interactive, streaming, graph, etc.

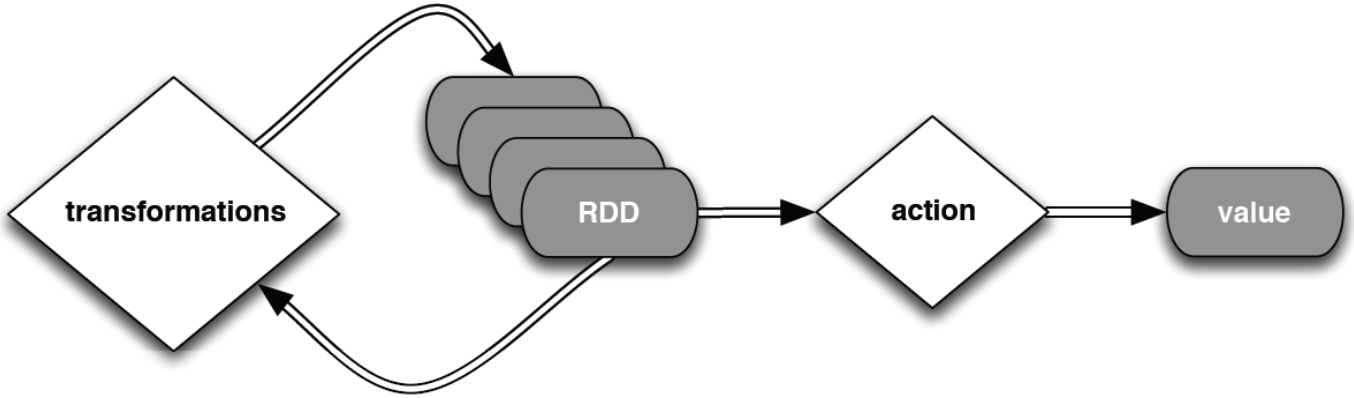
# Spark [Zaharia, 2010]

- RDD (Resilient Distributed Dataset) – главная абстракция в Spark:
  - коллекция элементов
  - отказоустойчивая
  - параллельная обработка
  - read-only




[Zaharia, 2010] Zaharia M. et al. Spark: Cluster computing with working sets //2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10). – 2010.
https://www.usenix.org/legacy/event/hotcloud10/tech/full_papers/Zaharia.pdf

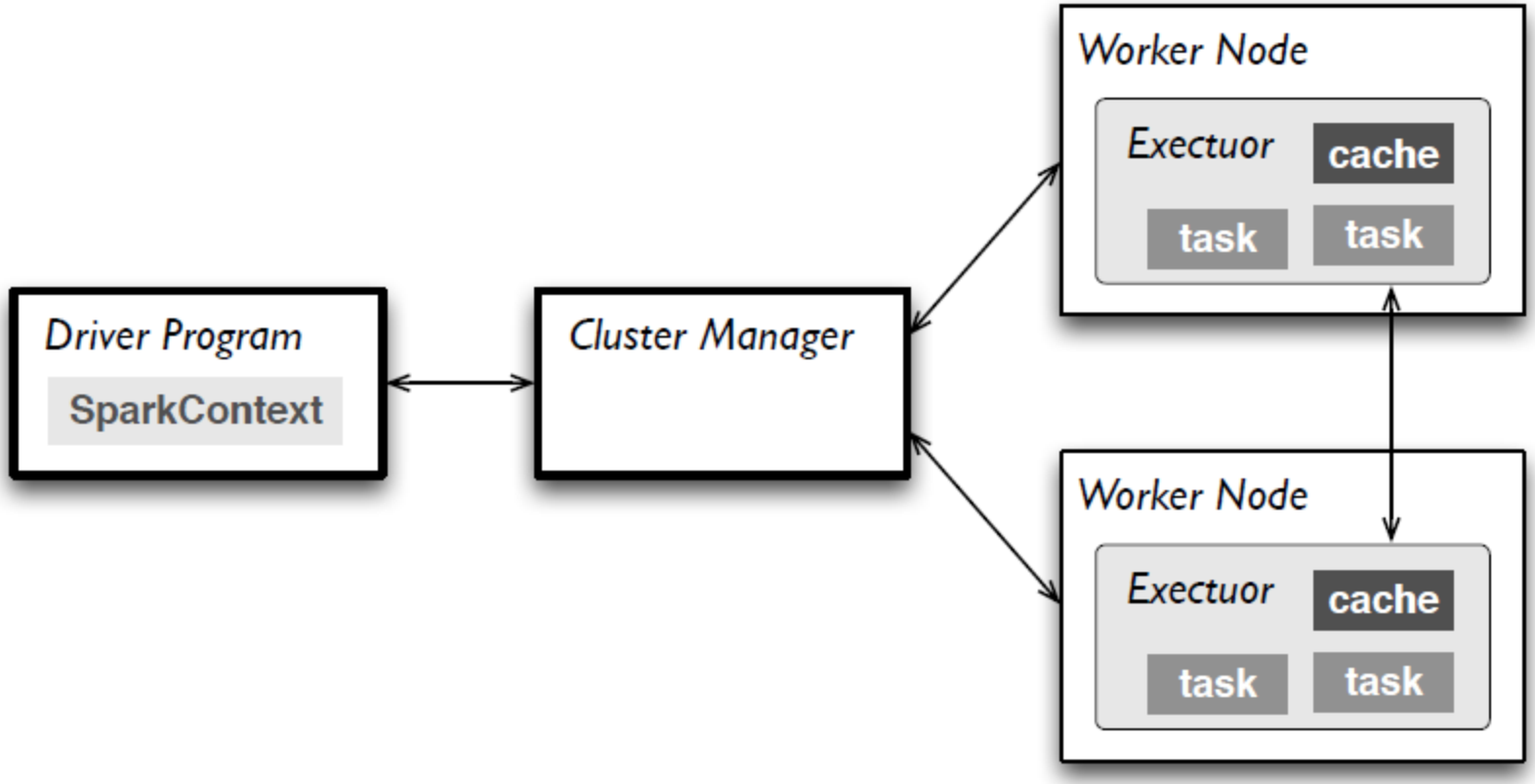
# Spark



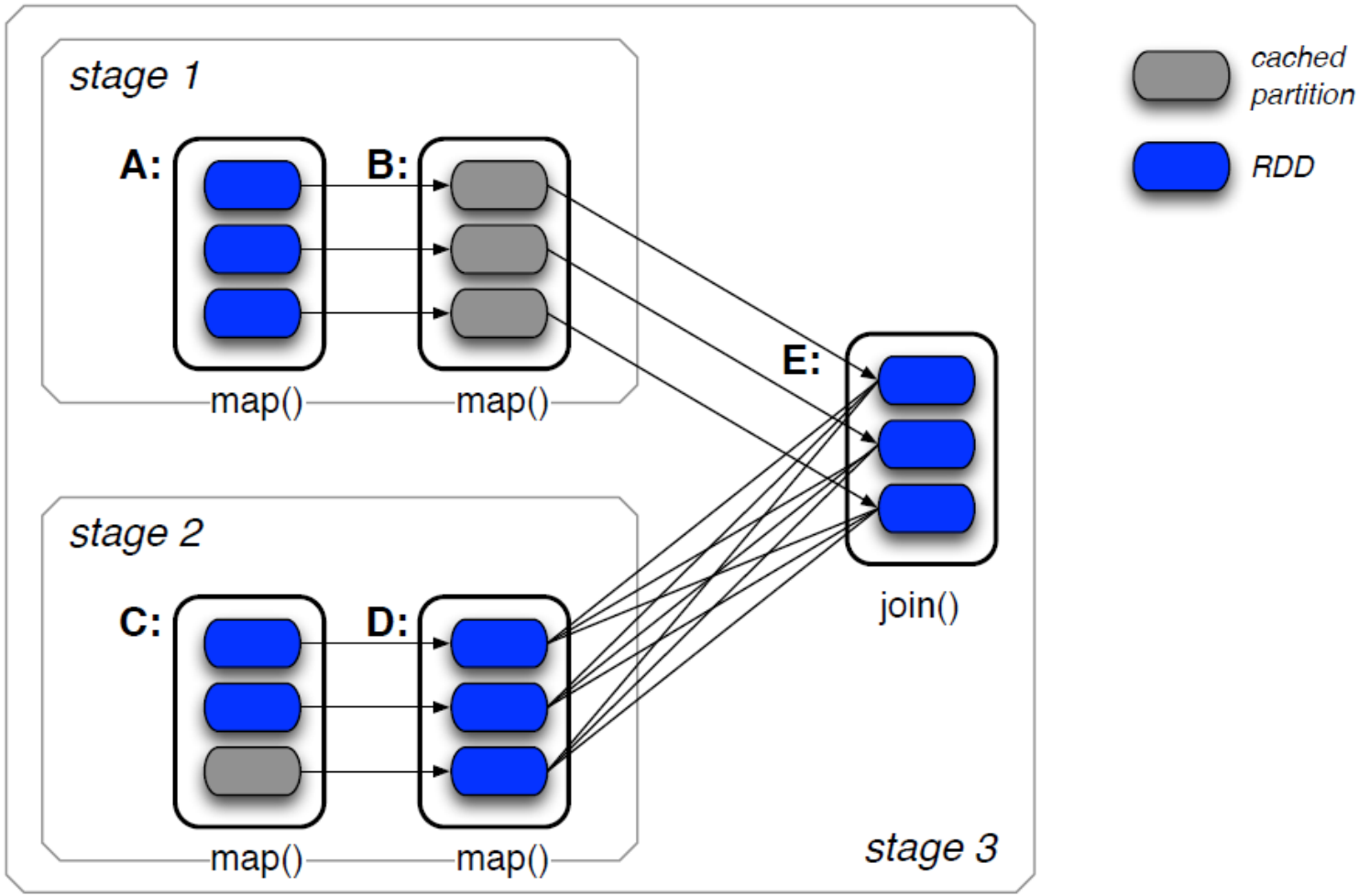

- Transformations
  - вычисления **ленивые** (оптимизация плана вычислений, восстановление)
  - map, filter, flatMap, union, distinct, groupByKey, sortByKey, reduceByKey, join
- Actions
  - reduce, count, collect, saveAs
- Persistence
  - cache – оставляет ленивым, но изменяет место хранения после выполнения вычислений: MEMORY_ONLY, MEMORY_AND_DISK, MEMORY_ONLY_SER, MEMORY_AND_DISK_SER, DISK_ONLY, MEMORY_ONLY_2, MEMORY_AND_DISK_2
- Разделяемые переменные
  - broadcast (read-only)
  - accumulators

  (только модификация)

# Spark: cluster



# Spark: граф выполнения операций

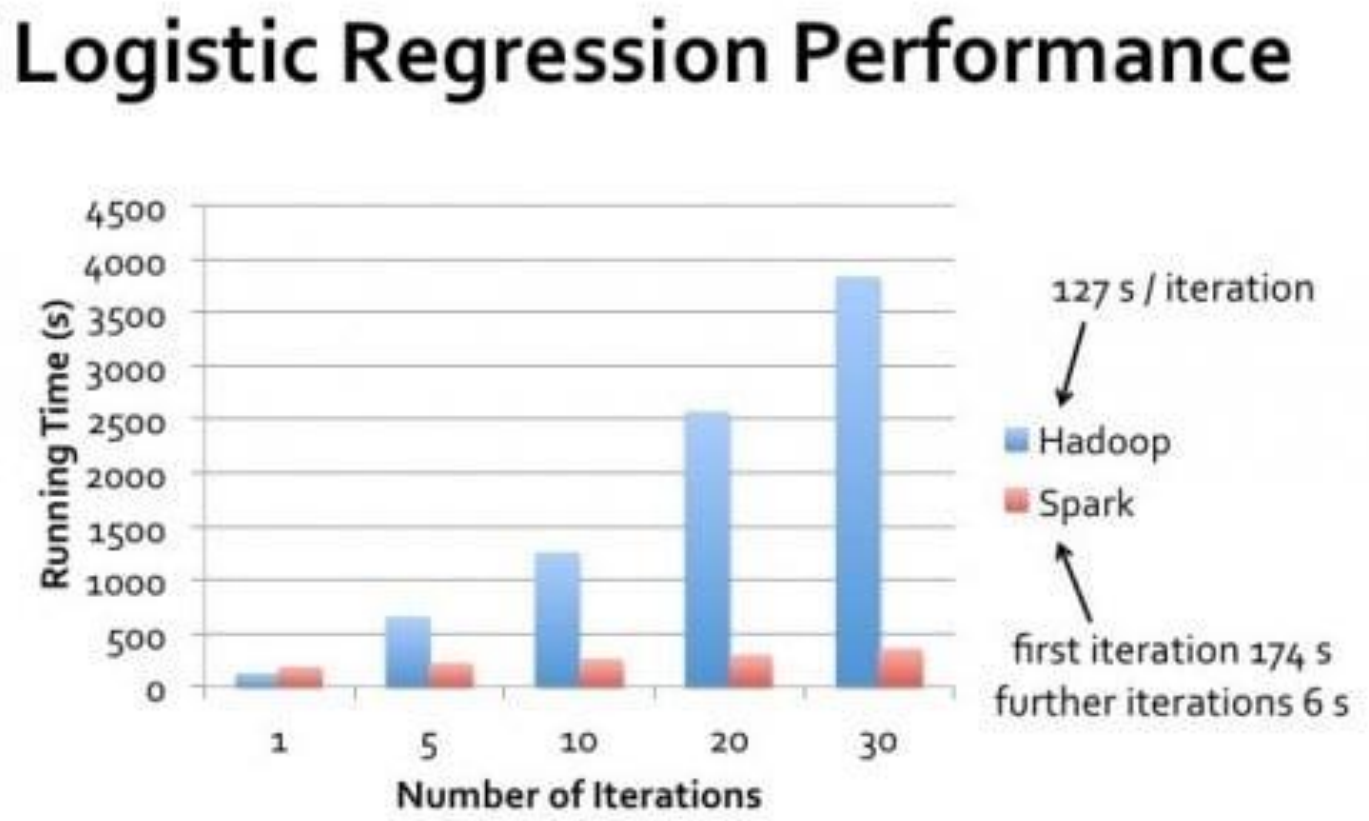


# Пример

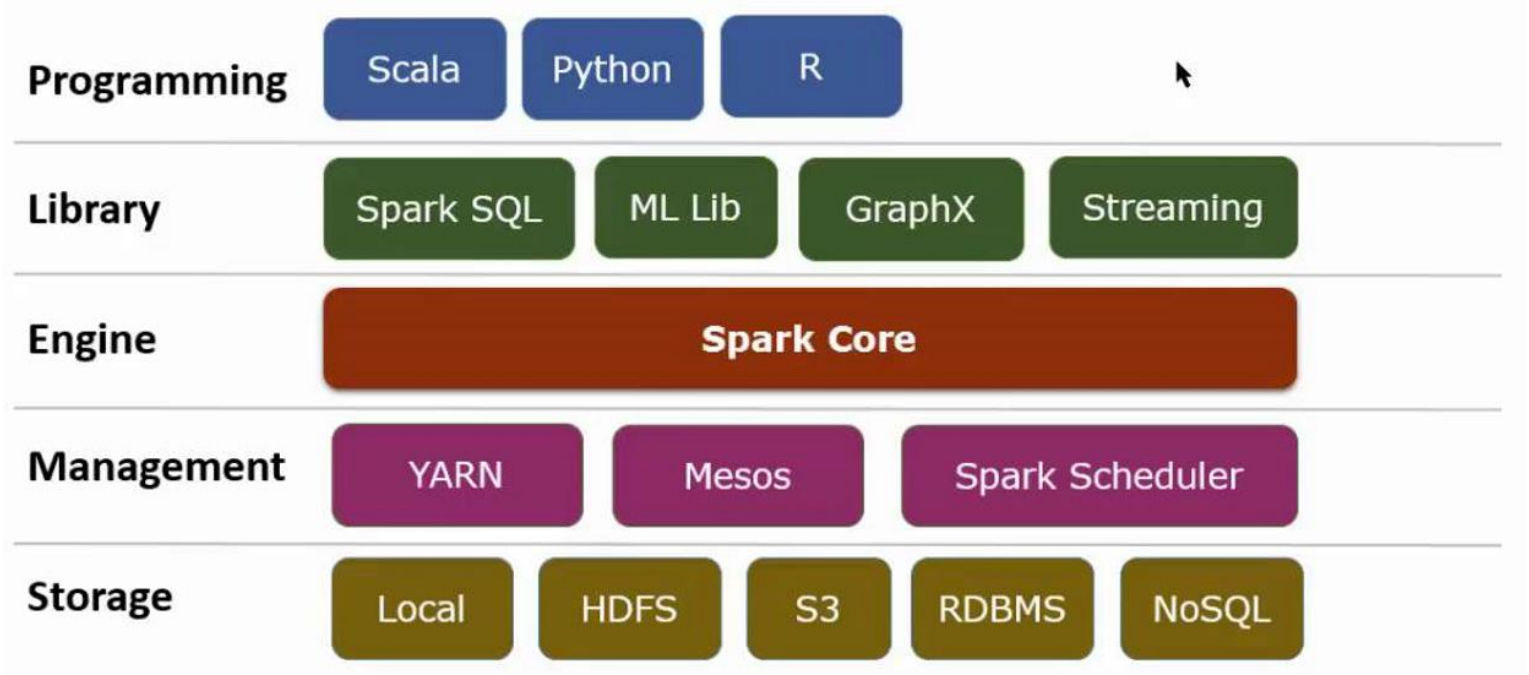
**Scala:**

```
val f = sc.textFile(“Input.file”)
val wc = f.flatMap(l => l.split(“ ”)).map(word => (word, 1)).reduceByKey(_ + _)
wc.saveAsTextFile(“wc_count.txt”)
```

# Spark


Logistic Regression Performance
Running Time (s)
4500
4000
3500
3000
2500
2000
1500
1000
500
0
1
5
10
20
30
Number of Iterations
127 s / iteration
Hadoop
Spark
first iteration 174 s
further iterations 6 s

# Spark


Programming
Scala
Python
R
Library
Spark SQL
ML Lib
GraphX
Streaming
Engine
Spark Core
Management
YARN
Mesos
Spark Scheduler
Storage
Local
HDFS
S3
RDBMS
NoSQL