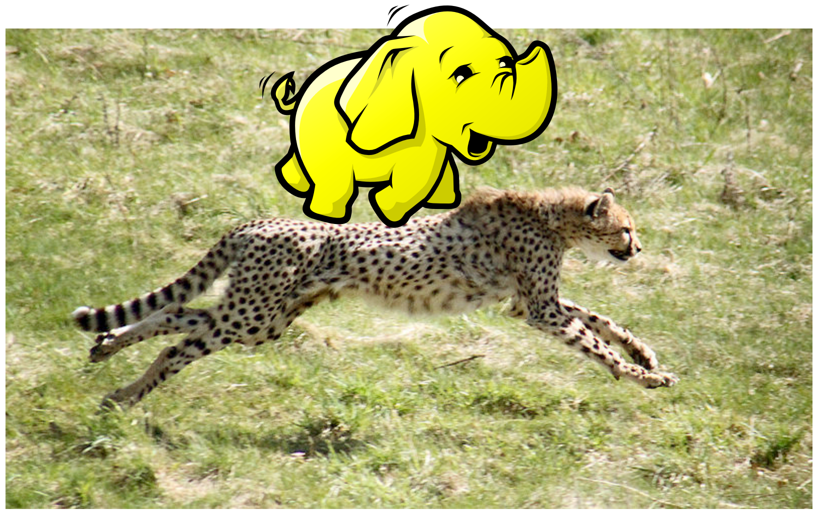
Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing)



Jens Dittrich[1] Jorge-Arnulfo Quiané-Ruiz[1] Alekh Jindal[1,2]

Yagiz Kargin[2] Vinay Setty[2] Jörg Schad[1]

[1]Information Systems Group,
Saarland University
http://infosys.cs.uni-saarland.de

[2]International Max Planck
Research School for Computer Science
http://www.imprs-cs.de/

The Parallel DBMS vs MapReduce Debate

	Parallel DBMS	MapReduce
licensing costs	usually high	none
administration	difficult	easy
upfront schema	must have	not required
user	advanced	beginner
scalability	10-100es of nodes	>10,000 nodes
failover, large clusters	suboptimal	very good
performance	very good	suboptimal

- see also [Pavlo etal, SIGMOD 2009] comparison
  - benchmark to compare Parallel DBMS with MapReduce
  - showed superiority of Parallel DBMS over MapReduce

MapReduce ≠ MapReduce ≠ MapReduce

SAARLAND UNIVERSITY
COMPUTER SCIENCE
Information Systems Group

- but, MapReduce is **three different** things:

(1) a **programming paradigm**:

- it allows users to specify analytical tasks
- need to provide two functions only: `map`() and `reduce`()

(2) a description of a **processing pipeline and system**:

- that system computes the result to a MapReduce-job
- MapReduce-job: `map`(), `reduce`(), and some input data
- scales to very large clusters, > 10,000 nodes

(3) several implementations of (2):

- Google‘s proprietary MapReduce, Hadoop, ...

Related Work

		(1) Programming Paradigm		
		MapReduce	SQL	Hybrid
(2) Processing pipeline and system	MapReduce	Hadoop	Hive	
	PDBMS	Greenplum Vertica		
	Hybrid			HadoopDB

Related Work

SAARLAND UNIVERSITY
COMPUTER SCIENCE
Information Systems Group

		(1) Programming Paradigm		
		MapReduce	SQL	Hybrid
(2) Processing pipeline and system	MapReduce	Hadoop	Hive	
	PDBMS	Greenplum Vertica		
	Hybrid			HadoopDB

back to initial interface hurdle

SAARLAND UNIVERSITY
COMPUTER SCIENCE
Information Systems Group

Related Work

		(1) Programming Paradigm		
		MapReduce	SQL	Hybrid
(2) Processing pipeline and system	MapReduce	Hadoop	Hive	
	PDBMS	Greenplum Vertica		
	Hybrid			HadoopDB

proprietary, expensive

back to initial interface hurdle

Related Work

		(1) Programming Paradigm		
		MapReduce	**SQL**	**Hybrid**
(2) Processing pipeline and system	**MapReduce**	Hadoop	Hive	
	PDBMS	Greenplum Vertica		
	Hybrid			HadoopDB

proprietary, expensive

back to initial interface hurdle

admin costs?

Related Work

	(1) Programming Paradigm		
	MapReduce	SQL	Hybrid

Research Challenge:

Can we invent a system that:

(1) keeps the MapReduce programming paradigm **and** the MapReduce execution engine?

(2) approaches Parallel DBMSs in performance?

Related Work

		(1) Programming Paradigm		
		MapReduce	**SQL**	**Hybrid**
(2) Processing pipeline and system	**MapReduce**	Research Challenge: Can we invent a system that: (1) keeps the MapReduce programming paradigm and the MapReduce execution engine? (2) approaches Parallel DBMSs in performance? Hadoop++	Hive	
	PDBMS	Greenplum Vertica		
	Hybrid			HadoopDB

proprietary, expensive

back to initial interface hurdle

admin costs?

SAARLAND UNIVERSITY
COMPUTER SCIENCE
Information Systems Group

Hadoop++ System Vision

(1) **MapReduce programming paradigm**

map(), reduce()

MapReduce program analysis
e.g. [Cafarella and Ré, WebDB2010]
[Iu and Zwaenepoel, EuroSys 2010]

logical plan

Optimization
e.g. cost models [Morton et.al. SIGMOD 2010]

optimized plan

MapReduce program generation
this paper, Hadoop++

map‘(), reduce‘()

(2) **MapReduce processing pipeline and system**

Features of Hadoop++

(1) we **do not change** the existing Hadoop framework at all

advantage: no need to maintain and test Hadoop code changes

advantage: future improvements of Hadoop orthogonal to Hadoop++

(2) **inject** our technology inside Hadoop, hide it

advantage: clear layering

advantage: no extra operators, no pipeline changes

(3) **do not change** the MapReduce programming paradigm

advantage: nothing changes from the user-side

(4) still trick Hadoop into using **more efficient plans**

advantage: improve runtime performance considerably

How do we do this?

Well, let‘s first better understand the existing Hadoop processing pipeline....

Analysis: The Hadoop Plan

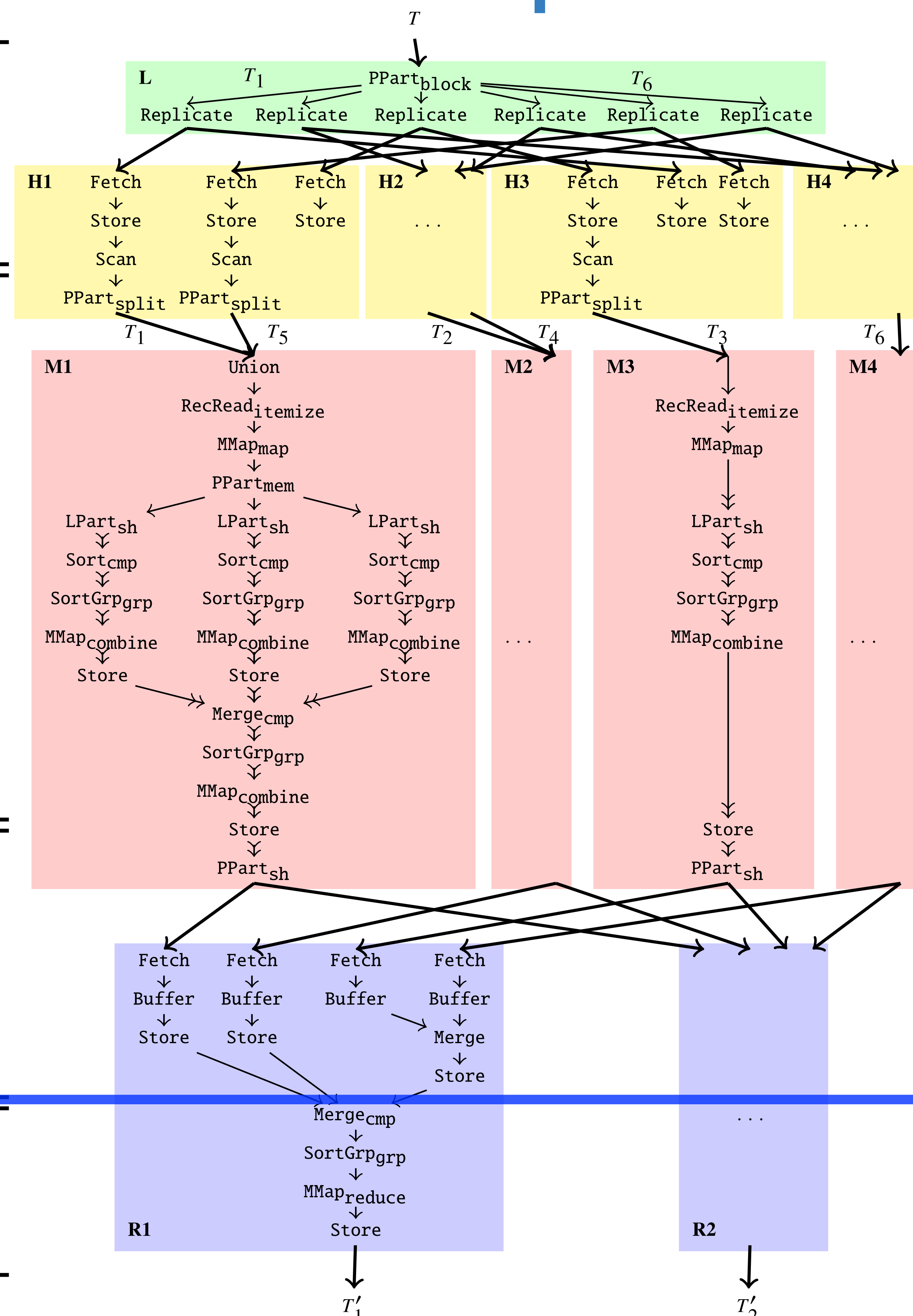
partition load map reduce

Data Load Phase

Map Phase

Shuffle Phase

Reduce Phase



- partition data into blocks
- replicate data to nodes
- store data

- scan input data blocks
- form splits
- send data to processing nodes
- break data into records
- call `map()` for each record
- pregroup and preaggregate output
- store output locally

- redistribute data over processing nodes
- merge subsets belonging to same reducer into single file

- perform final grouping
- call `reduce()` for each group
- store output

figure shows example with 4 mappers and 2 reducers

Observations on The Hadoop Plan

SAARLAND UNIVERSITY
COMPUTER SCIENCE
Information Systems Group

- again: no real operators, all hard-coded
- large distributed external merge sort
- sort in order to do a sort-based grouping
- full scan access at all times
- not only two functions, i.e. `map` and `reduce`,
- but...

Ten User-Defined Functions

partition load map reduce

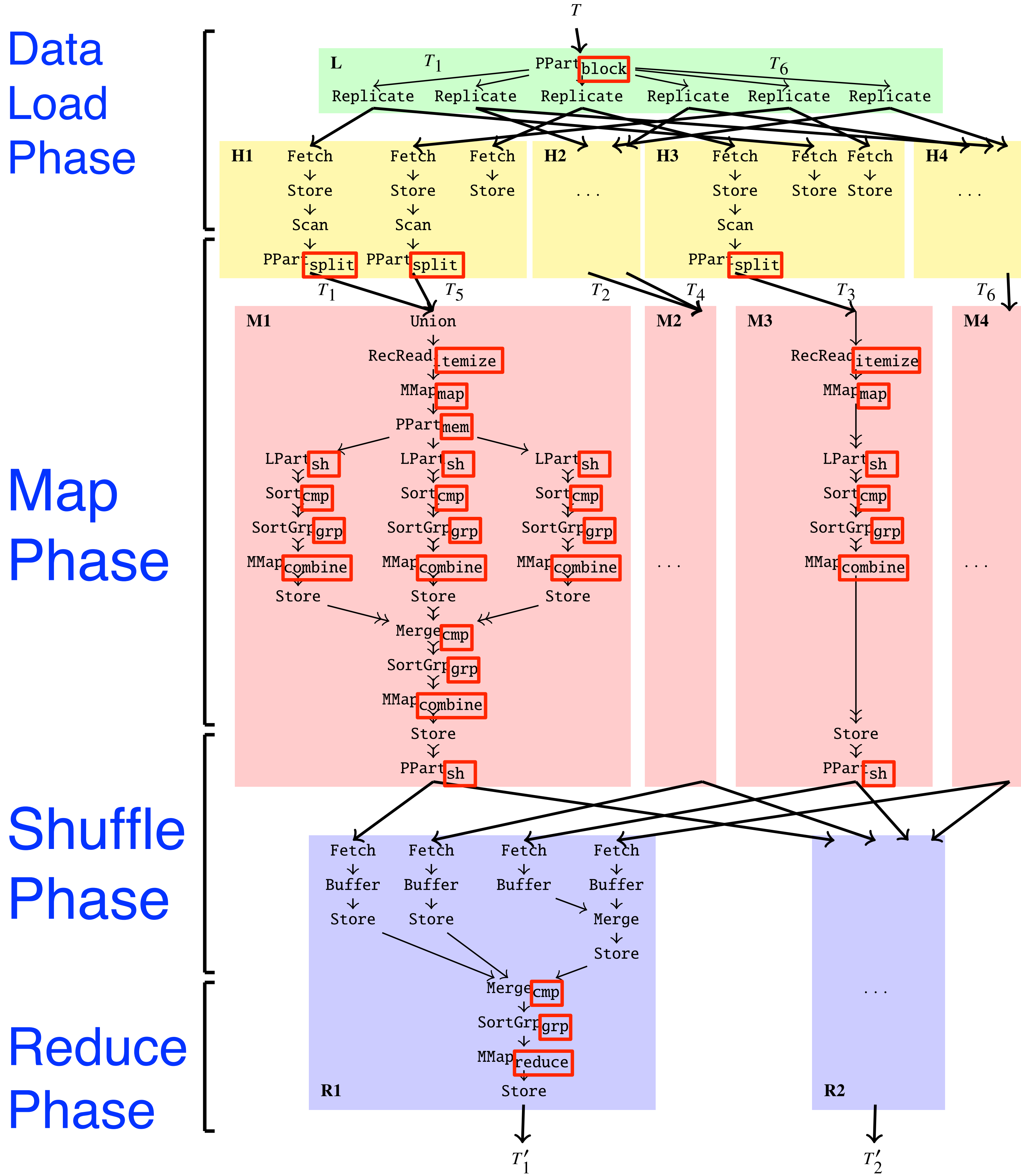


figure shows example with 4 mappers and 2 reducers

- **The Hadoop Plan has ten user-defined functions (UDFs):**

```
block
split
itemize
mem
map
sh
cmp
grp
combine
reduce
```

Hadoop++ Approach: Trojan Techniques

- **Trojan Index:**
  - at data load time: create index
  - at query time: use index access plan

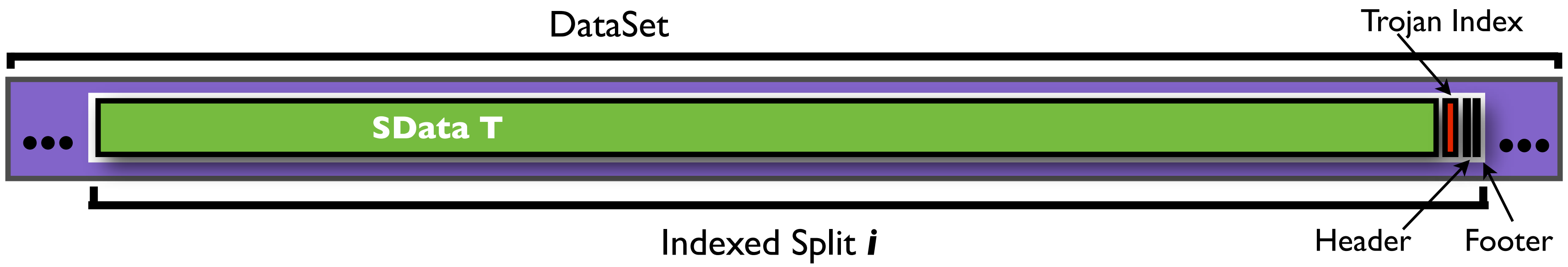
- **Trojan Join:**
  - at data load time: create co-partitions
  - at query time: compute all join results locally

Trojan Index Creation

SAARLAND UNIVERSITY
COMPUTER SCIENCE
Information Systems Group

Desired layout:

e.g. 8MB of index for 1GB of data



- **Index Creation Algorithm**:
  - read input split
  - add small clustered Trojan index (we use a CSS-tree)
  - add some metadata
- **Implementation**:
  - a MapReduce program

Trojan Index Creation

partition load map reduce

$\mathtt{map}(\text{key } k, \text{value } v) \mapsto$

$[(\mathtt{getSplitID}() \oplus \mathtt{prj}_a\,(k \oplus v), k \oplus v)]$

form intermediate key with splitID and index key *a*

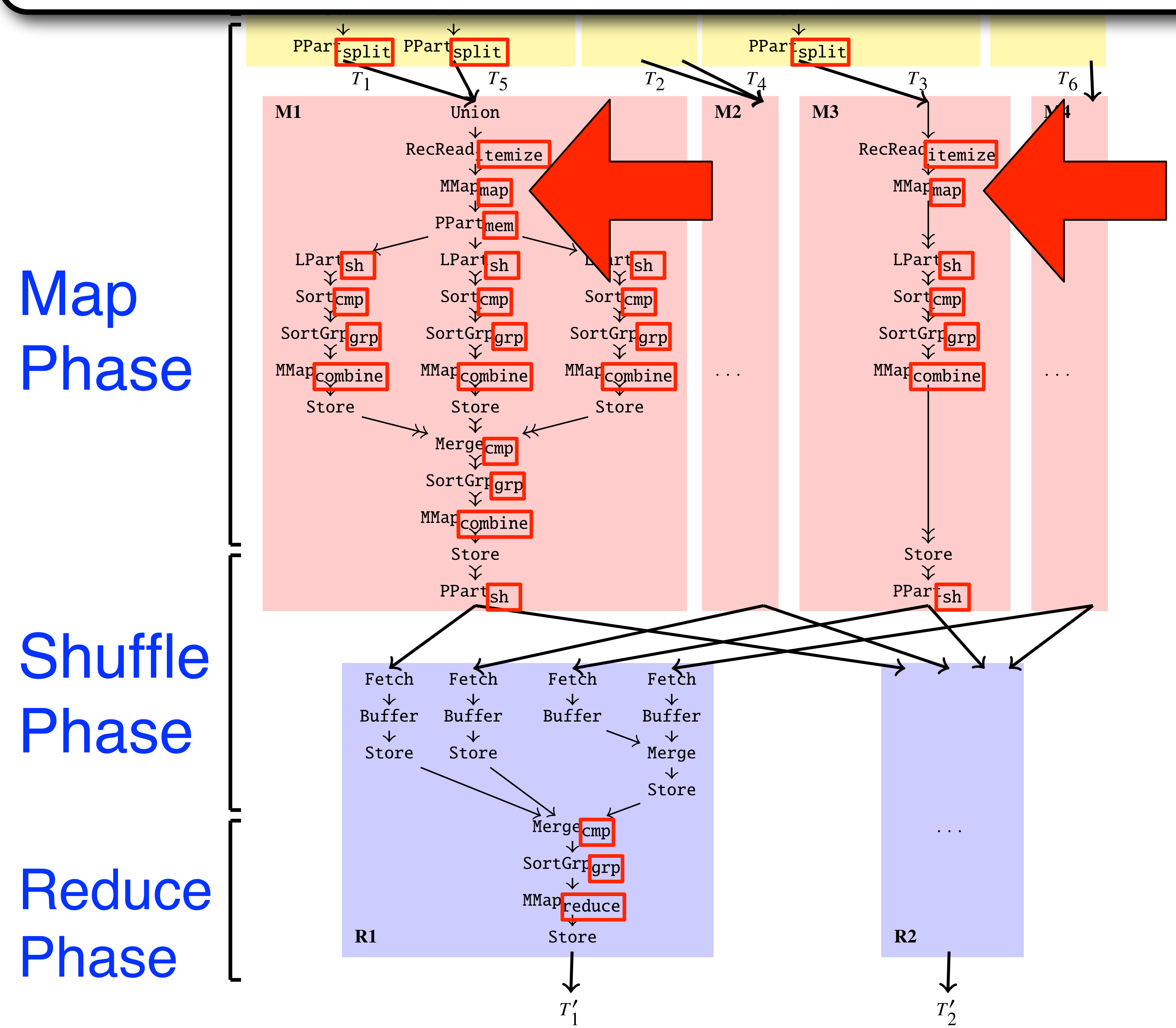


figure shows example with 4 mappers and 2 reducers

$\oplus$: concatenate schemas

Trojan Index Creation

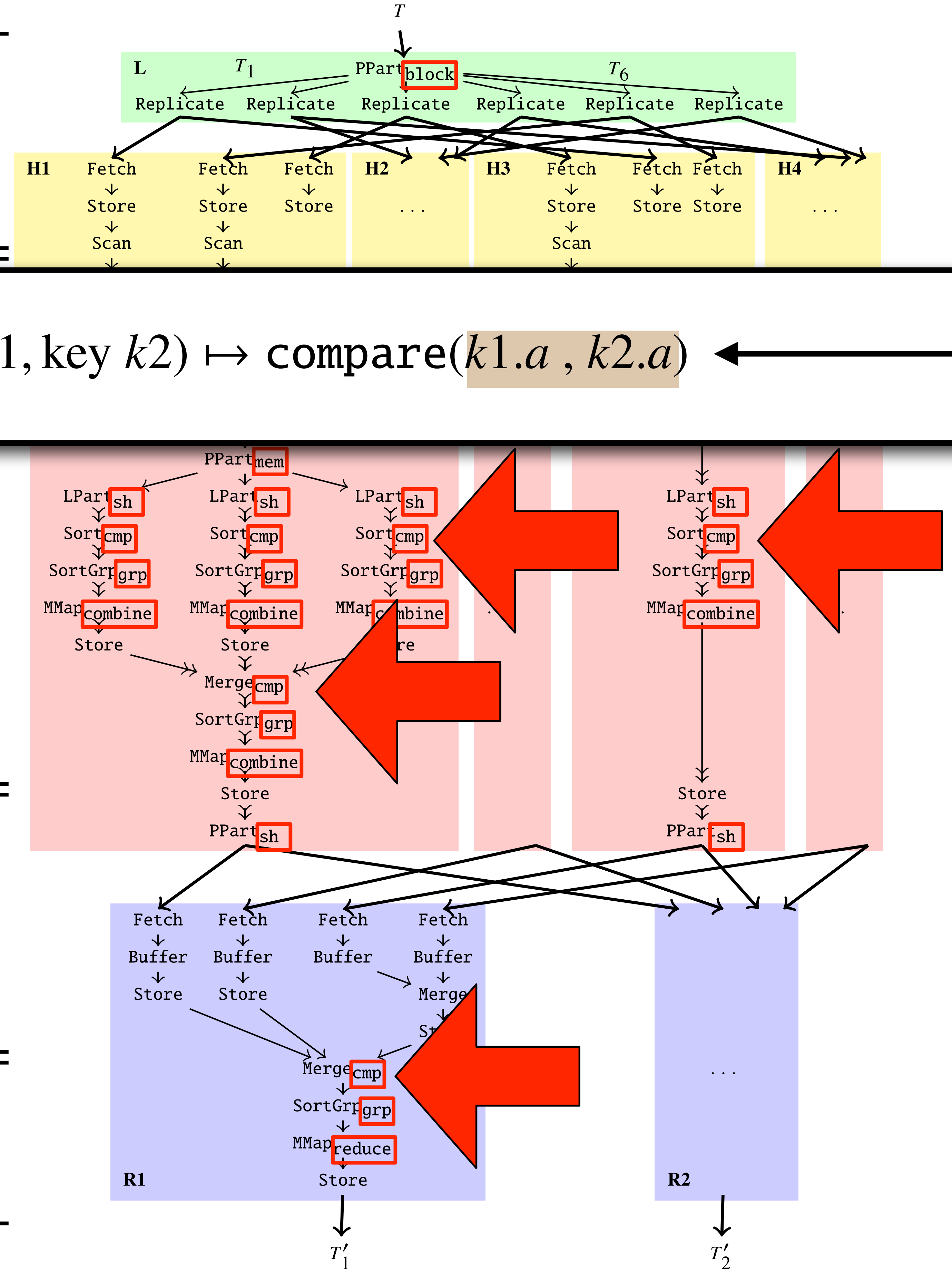
partition load map reduce

Data Load Phase

Map Phase

Shuffle Phase

Reduce Phase



$\mathtt{cmp}(\text{key } k1, \text{key } k2) \mapsto \mathtt{compare}(k1.a \,,\, k2.a)$

sort on index key only

figure shows example with 4 mappers and 2 reducers

$\oplus$: concatenate schemas

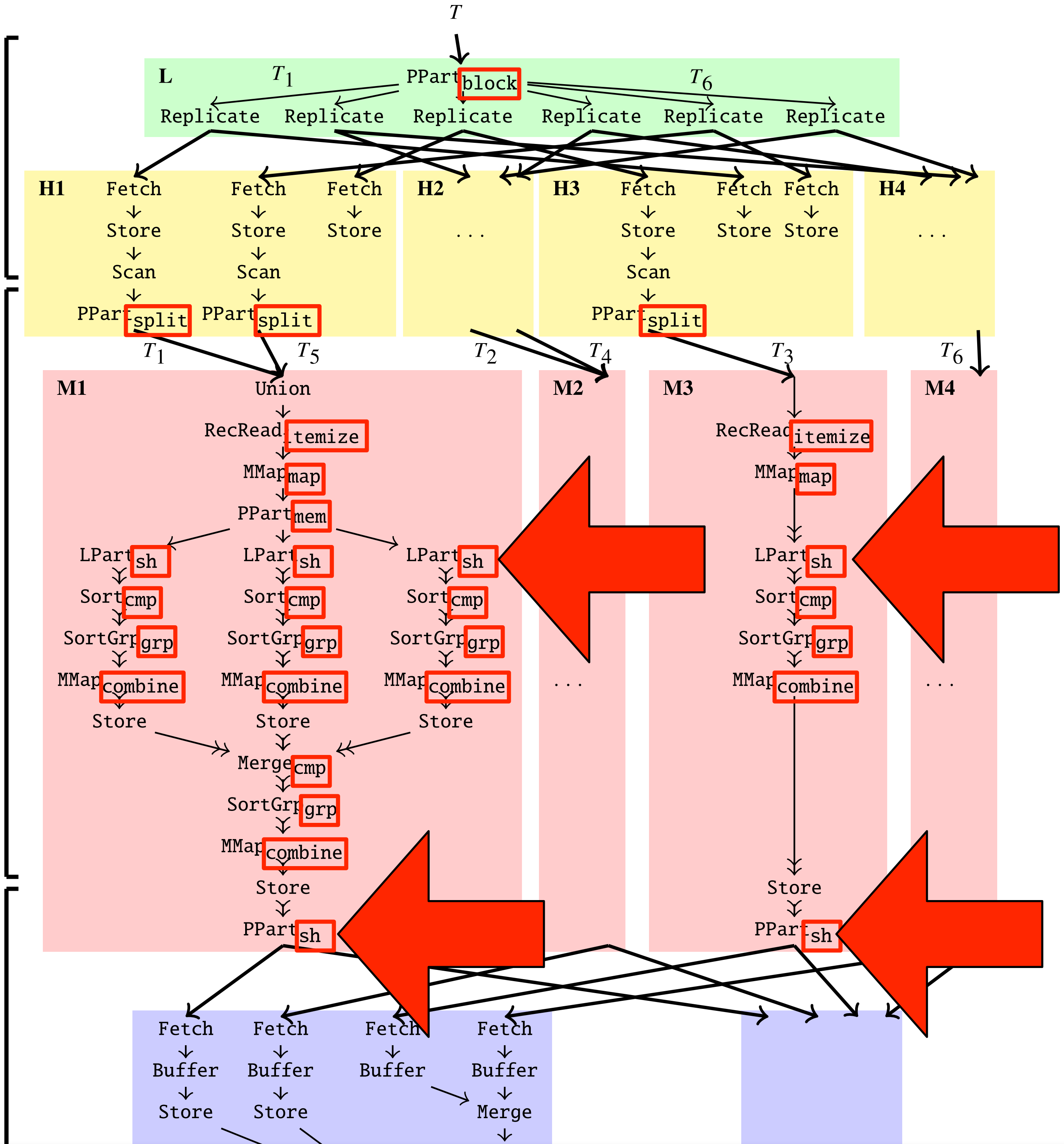
Trojan Index Creation

partition load map reduce

Data Load Phase

Map Phase

Shuffle Phase



sh(key k, value v, int $numPartitions$) $\mapsto$

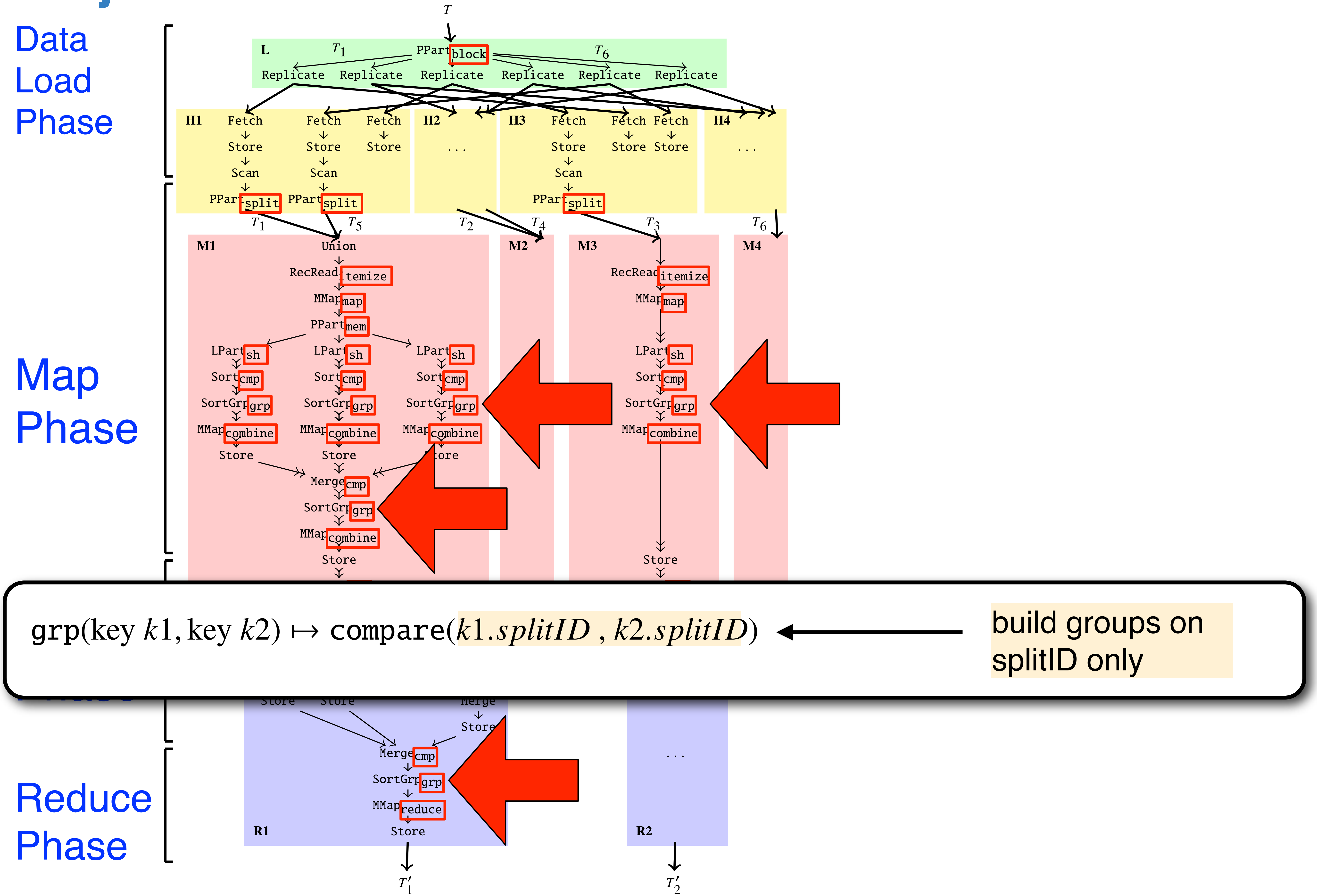
$k.splitID$ % $numPartitions$ ← shuffle on splitID only

T'_1 T'_2

figure shows example with 4 mappers and 2 reducers

$\oplus$: concatenate schemas

Trojan Index Creation



$\texttt{grp}(\text{key } k1, \text{key } k2) \mapsto \texttt{compare}(k1.splitID\ , k2.splitID)$

build groups on splitID only

figure shows example with 4 mappers and 2 reducers

$\oplus$: concatenate schemas

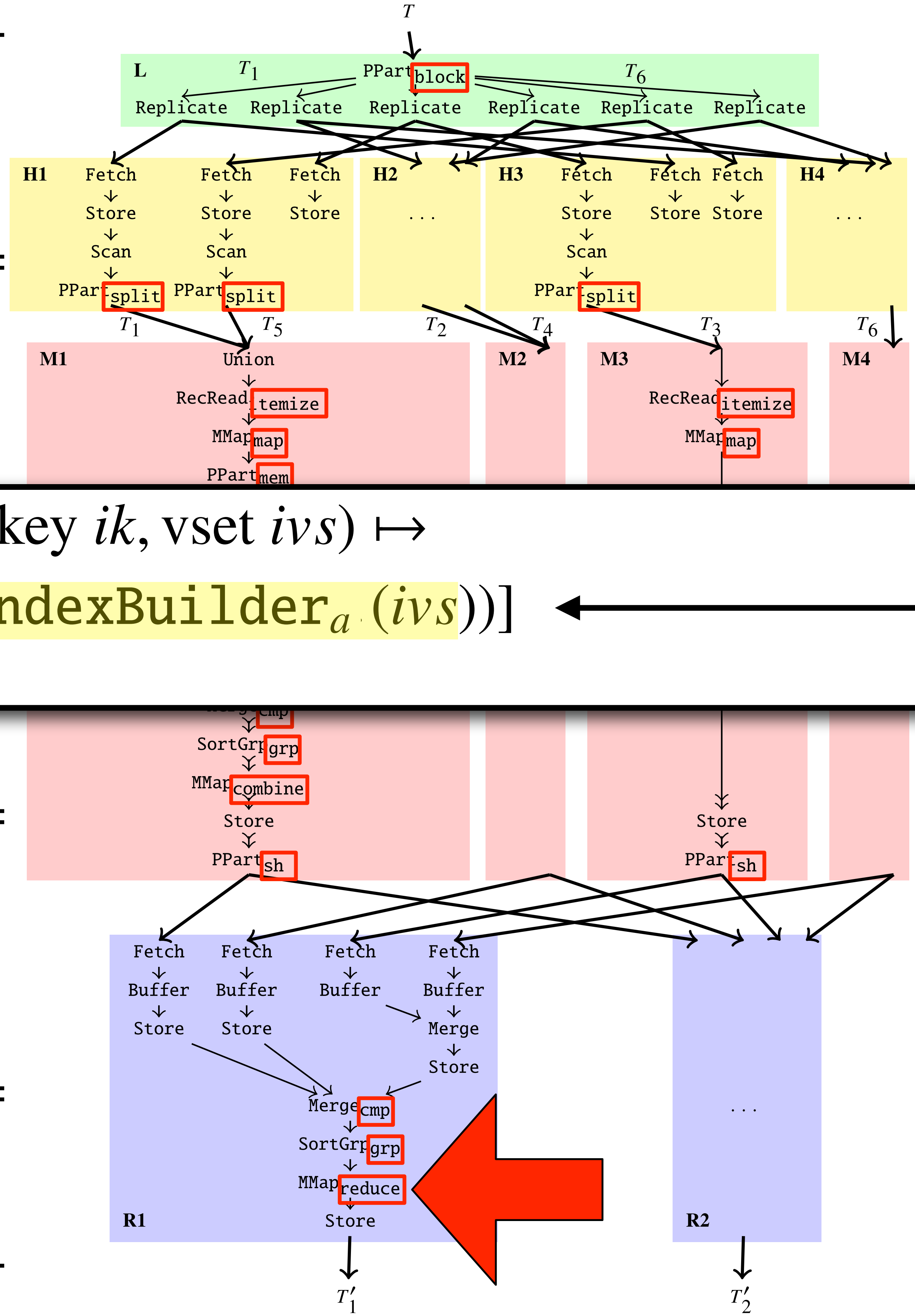
Trojan Index Creation

partition load map reduce

Data Load Phase

Shuffle Phase

Reduce Phase



$\texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto$

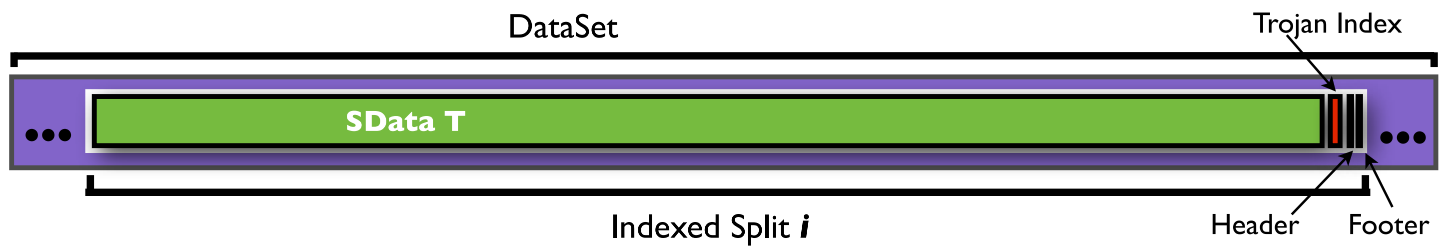
$[(ivs \oplus \texttt{indexBuilder}_a(ivs))]$

build CSS-tree for each ivs set

figure shows example with 4 mappers and 2 reducers

$\oplus$: concatenate schemas

Trojan Index Query Processing

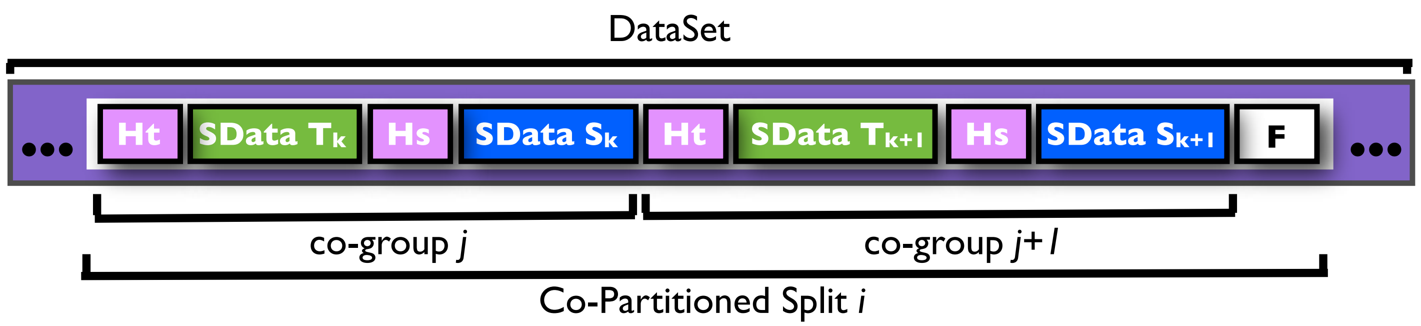


- **Query Algorithm**:
  - for each split:
    - read footer to obtain split size
    - read header to obtain [key_{min}, key_{max}]-range of index
    - if search key overlaps [key_{min}, key_{max}]-range:
      - read CSS-tree into main memory
      - read only records qualifying for search predicate
      - only pass those records to map()
    - else
      - skip this split
- **Implementation**:
  - a MapReduce program
  - provide `split` and `itemize` UDF
  - everything else unchanged

Trojan Join Co-Partitioning

Desired layout:

join T.a=S.b



- **Co-Partition Creation Algorithm**:
  - read input data
  - create co-partitioned data based on join keys of two relations
  - add some metadata
- **Implementation**:
  - a MapReduce program

Trojan Join Co-Partitioning Details

partition load map reduce

$$\mathtt{map}(\text{key } k, \text{value } v) \mapsto \begin{cases} [(\mathtt{prj}_a\ (k \oplus v)), k \oplus v)] & \text{if input}(k \oplus v) = T, \\ [(\mathtt{prj}_b\ (k \oplus v)), k \oplus v)] & \text{if input}(k \oplus v) = S. \end{cases}$$

form intermediate key with join key a from T and b from S

Map Phase

Shuffle Phase

Reduce Phase

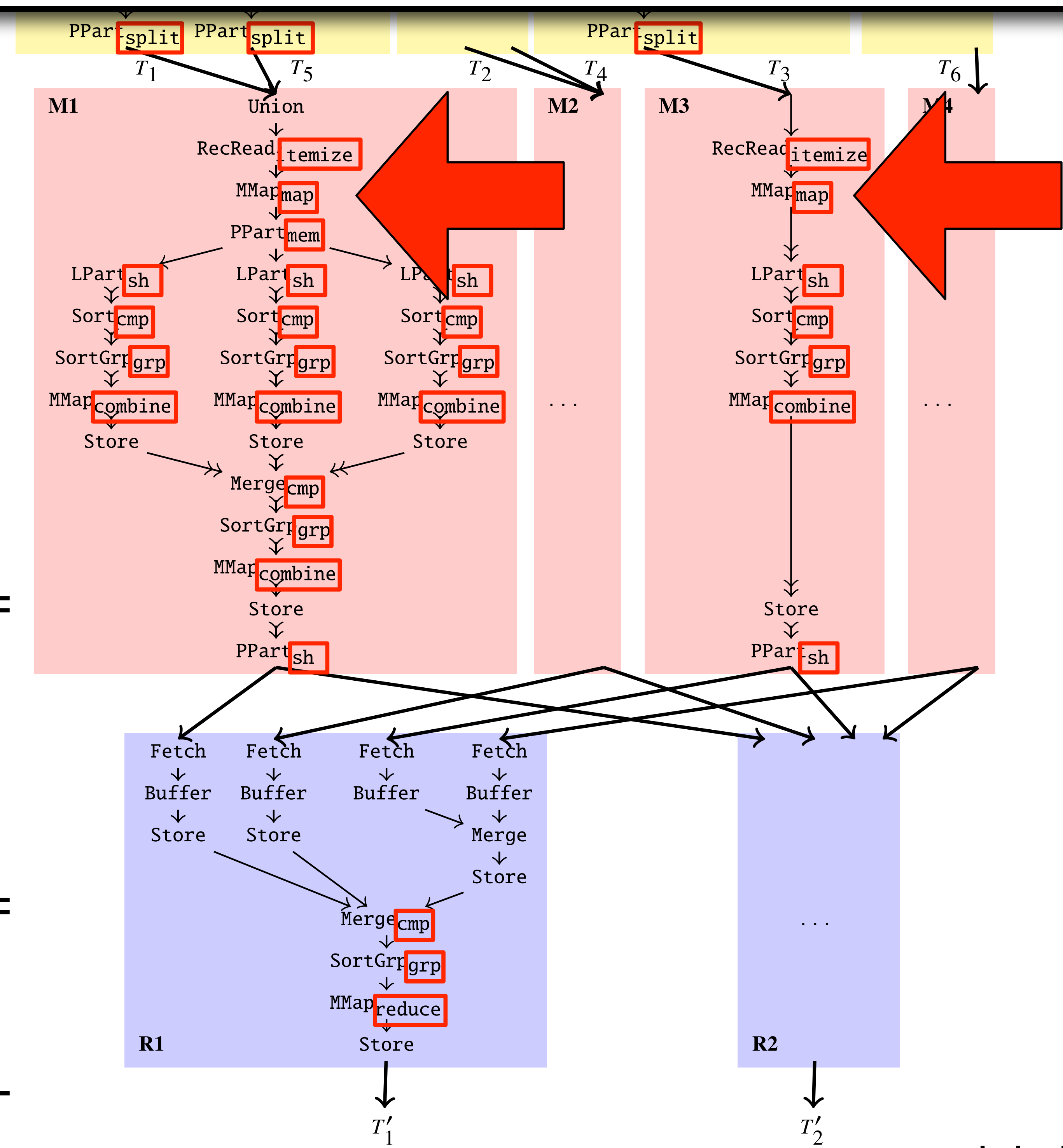


figure shows example with 4 mappers and 2 reducers

join T.a=S.b

$\oplus$: concatenate schemas

Trojan Join Co-Partitioning Details

partition | load | map | reduce

cmp(key $k1$, key $k2$) | use default ← sort on join key only

Map Phase

Shuffle Phase

Reduce Phase

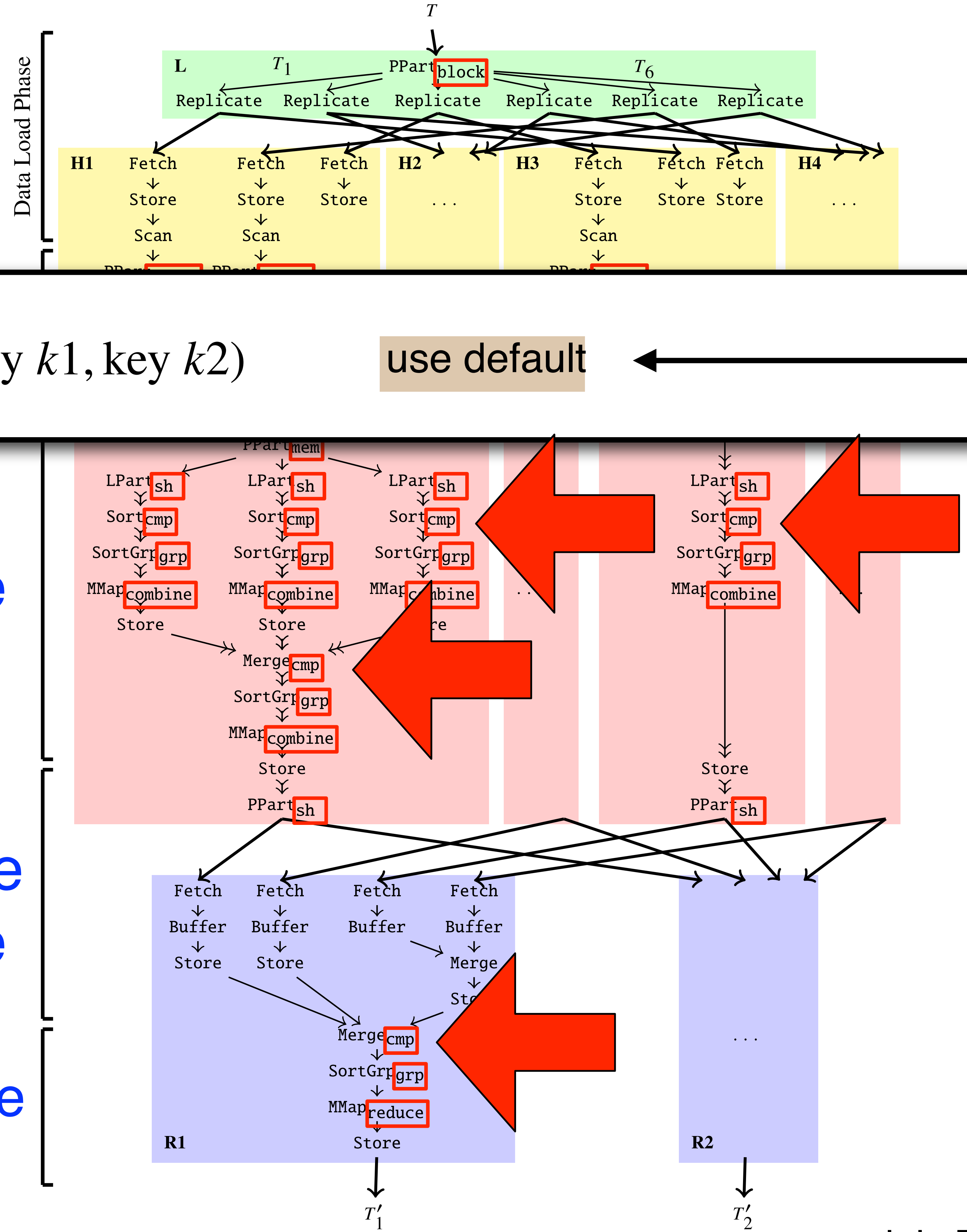


figure shows example with 4 mappers and 2 reducers

join T.a=S.b

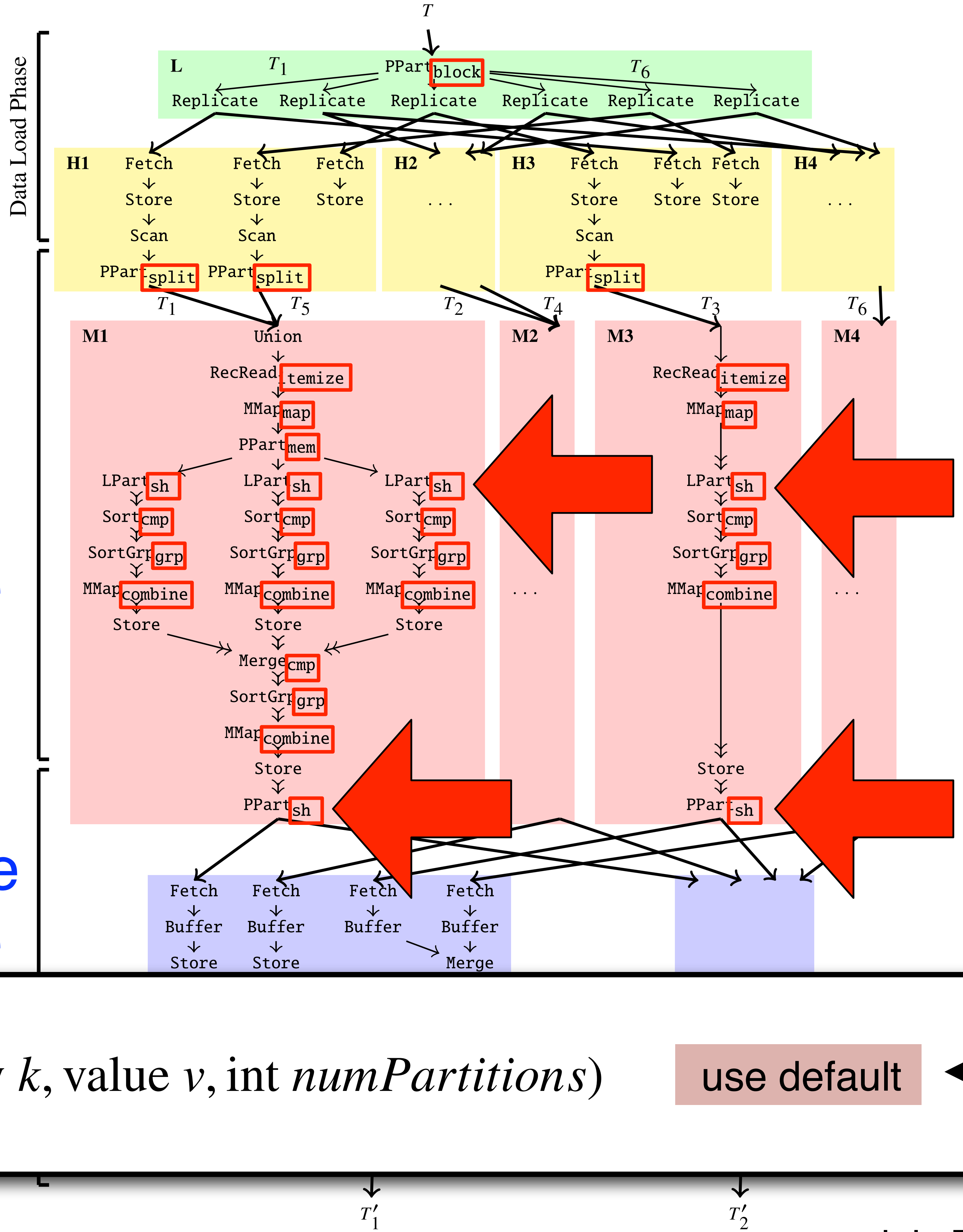
⊕: concatenate schemas

Trojan Join Co-Partitioning Details

partition load map reduce

Map Phase

Shuffle Phase



sh(key k, value v, int $numPartitions$) use default ← shuffle on join key only

figure shows example with 4 mappers and 2 reducers

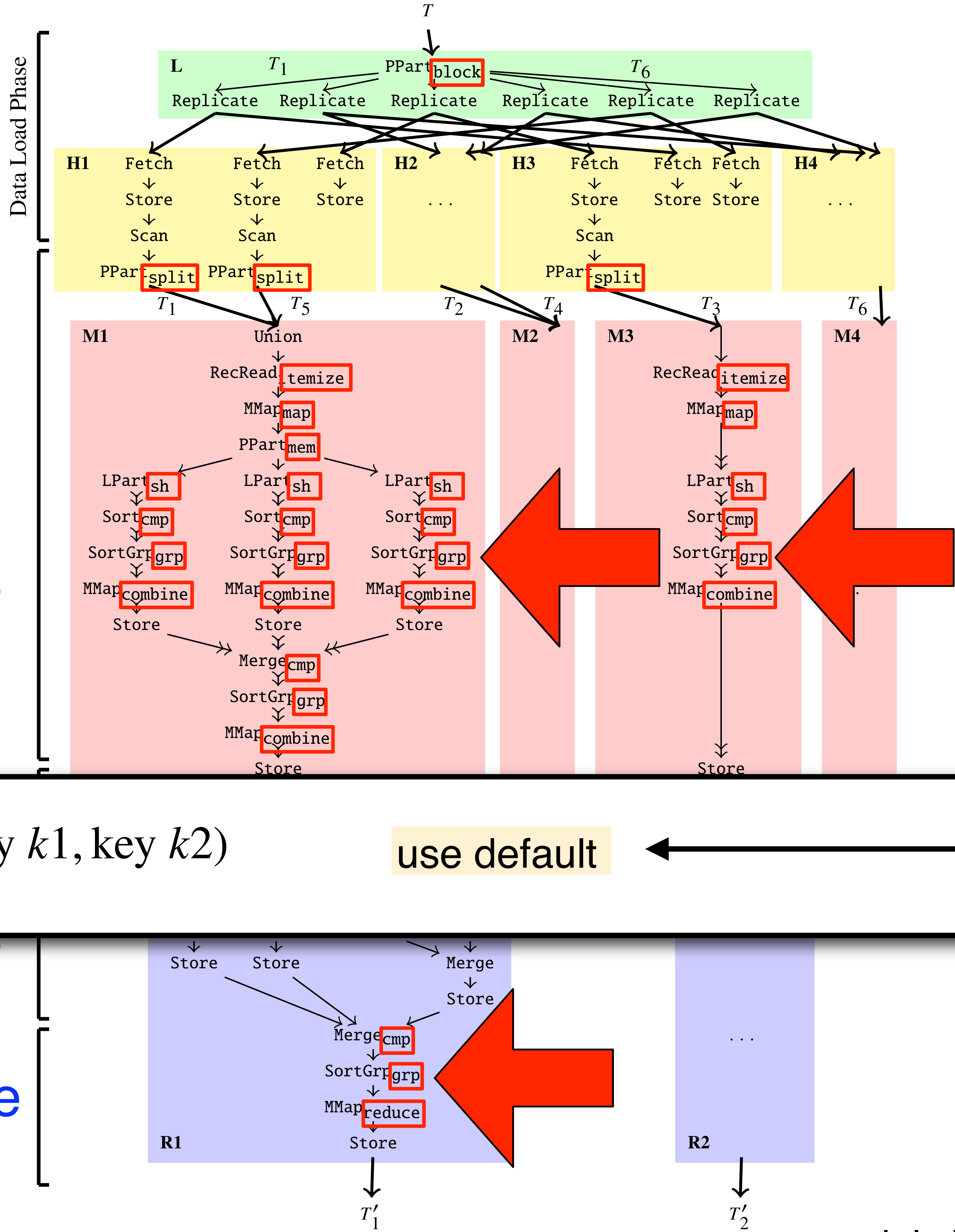
join T.a=S.b

$\oplus$: concatenate schemas

Trojan Join Co-Partitioning Details

partition load map reduce

Map
Phase



grp(key $k1$, key $k2$)

use default

build groups on
join key only

Phase

Reduce
Phase

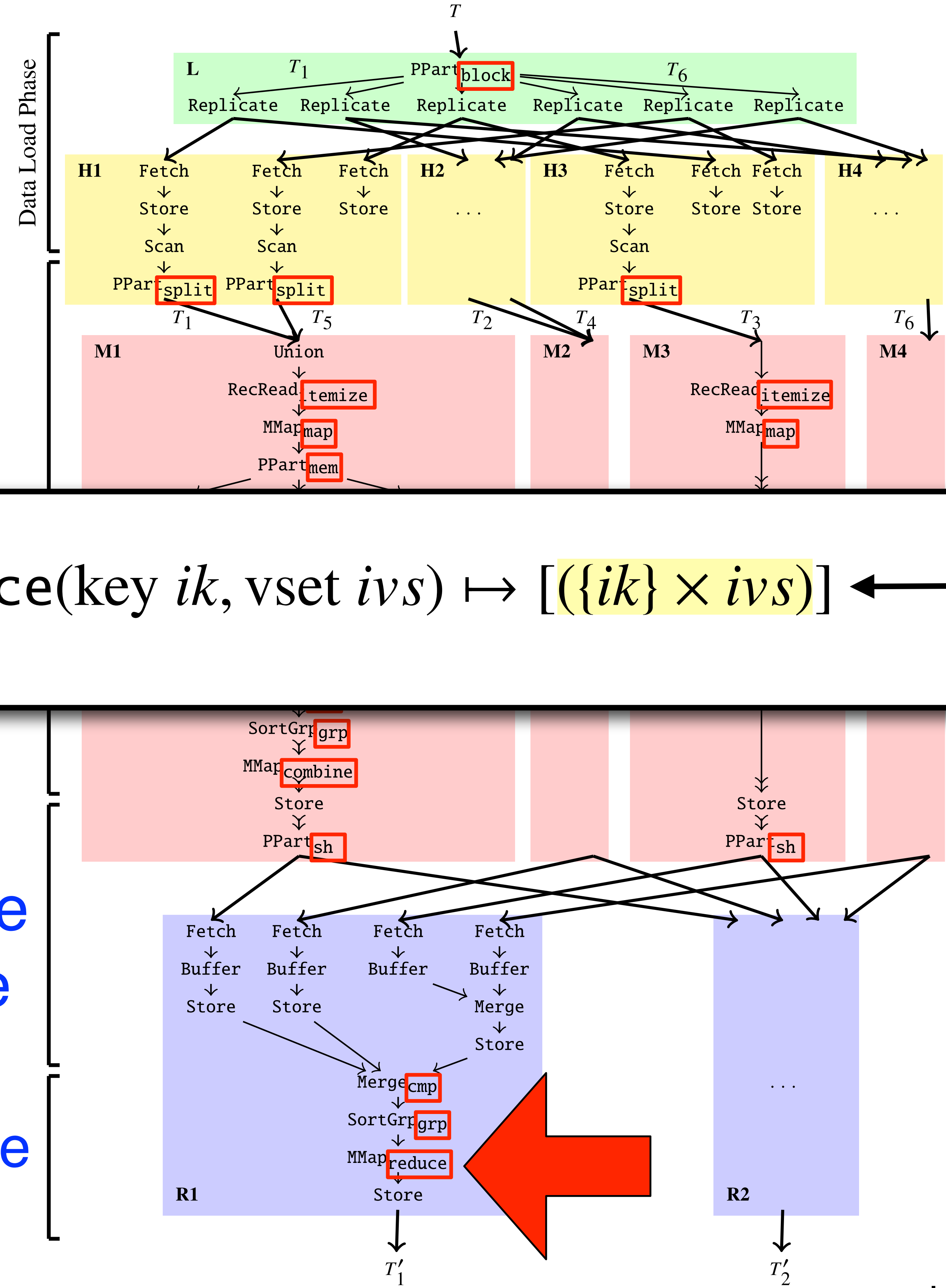
figure shows example with 4 mappers and 2 reducers

join T.a=S.b

$\oplus$: concatenate schemas

Trojan Join Co-Partitioning Details

partition load map reduce



Shuffle Phase

Reduce Phase

$$\texttt{reduce}(\text{key } ik, \text{vset } ivs) \mapsto [(\{ik\} \times ivs)]$$

build one co-group for each join value in a split

figure shows example with 4 mappers and 2 reducers

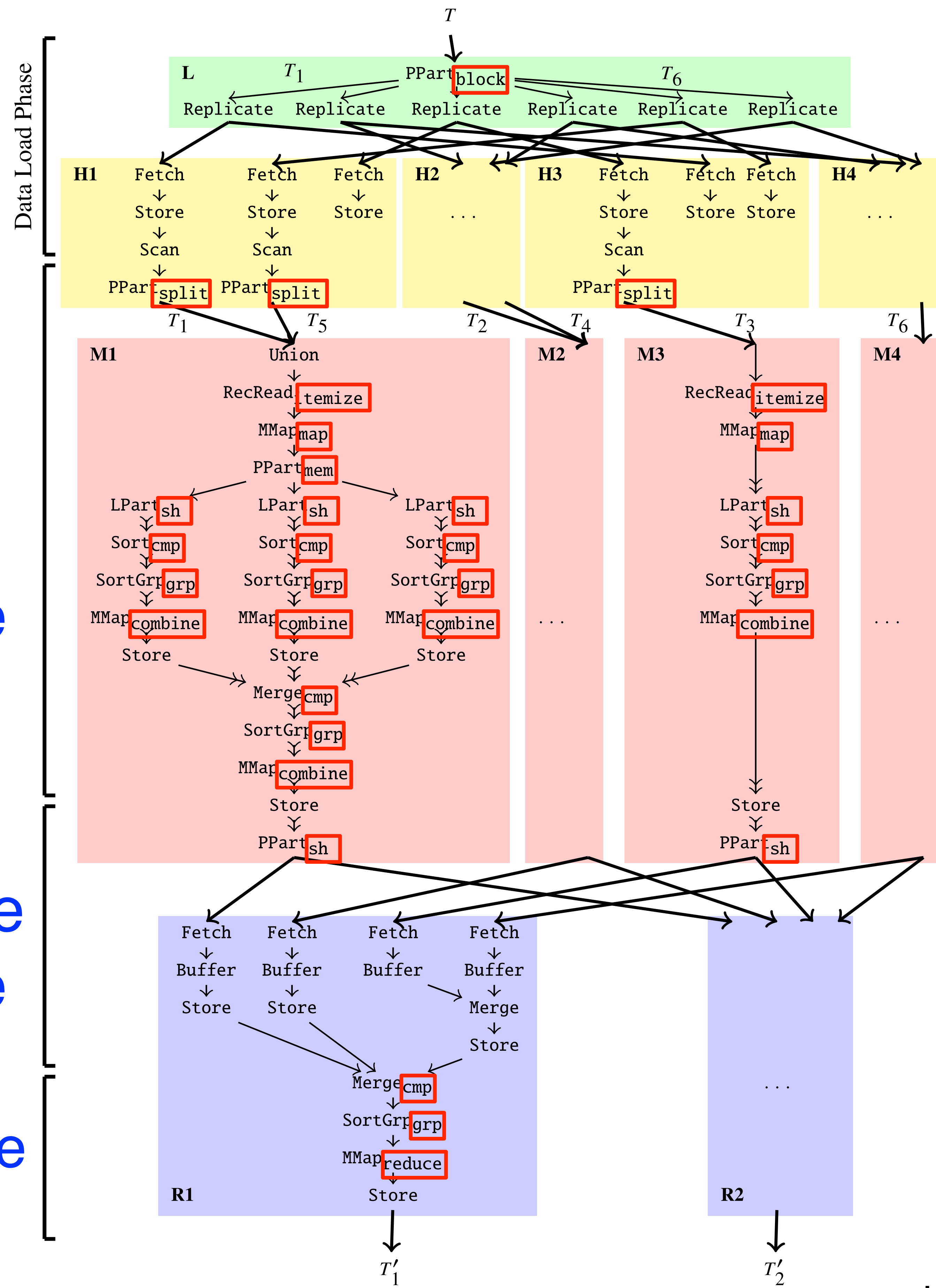
join T.a=S.b

$\oplus$: concatenate schemas

Trojan Join Co-Partitioning Details

partition load map reduce

Map Phase



Shuffle Phase

Reduce Phase

figure shows example with 4 mappers and 2 reducers

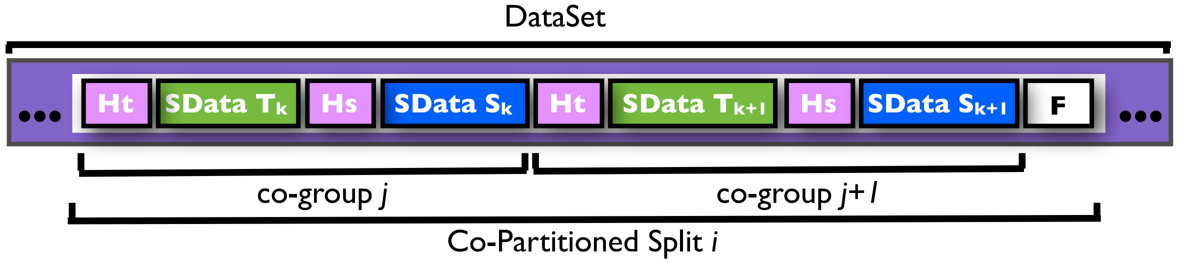
Notice. Write-up of these UDFs in the CR has a small bug. See note on our website:

http://infosys.cs.uni-saarland.de/publications/DQJ+10CRv1correction.pdf

join T.a=S.b

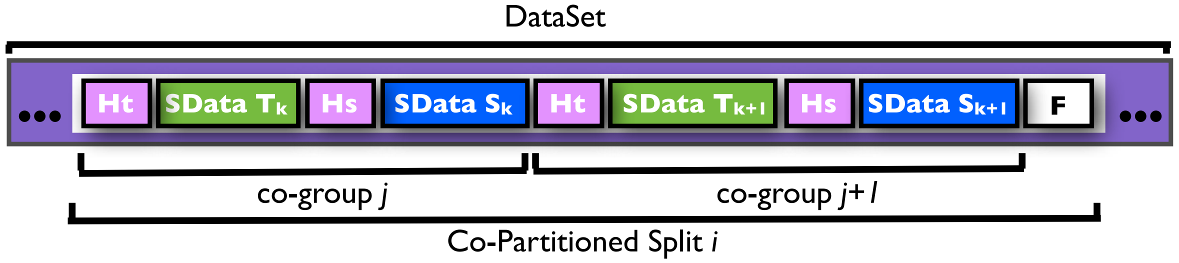
$\oplus$: concatenate schemas

Trojan Join Query Processing



- **Query Algorithm**:
  - read footer of each input split to determine split size
  - read records from each co-group in ascending order
  - build cross product for each co-group
- **Implementation**:
  - a MapReduce program
  - provide `split` UDF
- **Option 1: map-side join**
  - trick: map function keeps some state
  - perform local join in `map`()
  - advantage: no Reduce Phase (see paper)
  - drawback: need to keep some state in `map`() for sort-based grouping

Option 2: state-less map-side join



- **Algorithm**:
  - change `itemize` to return [joinkey, **entire co-group**]
  - then `map` is being called with the data belonging to an entire co-group
  - inside `map`: break co-group into tuples and compute cross product
- Advantages:
  - no Reduce Phase as in Option 1
  - but also: **no** need to keep state in `map`
- **in fact:** we exploited an interesting order plus `itemize` to **semantically reduce** data in `map`!

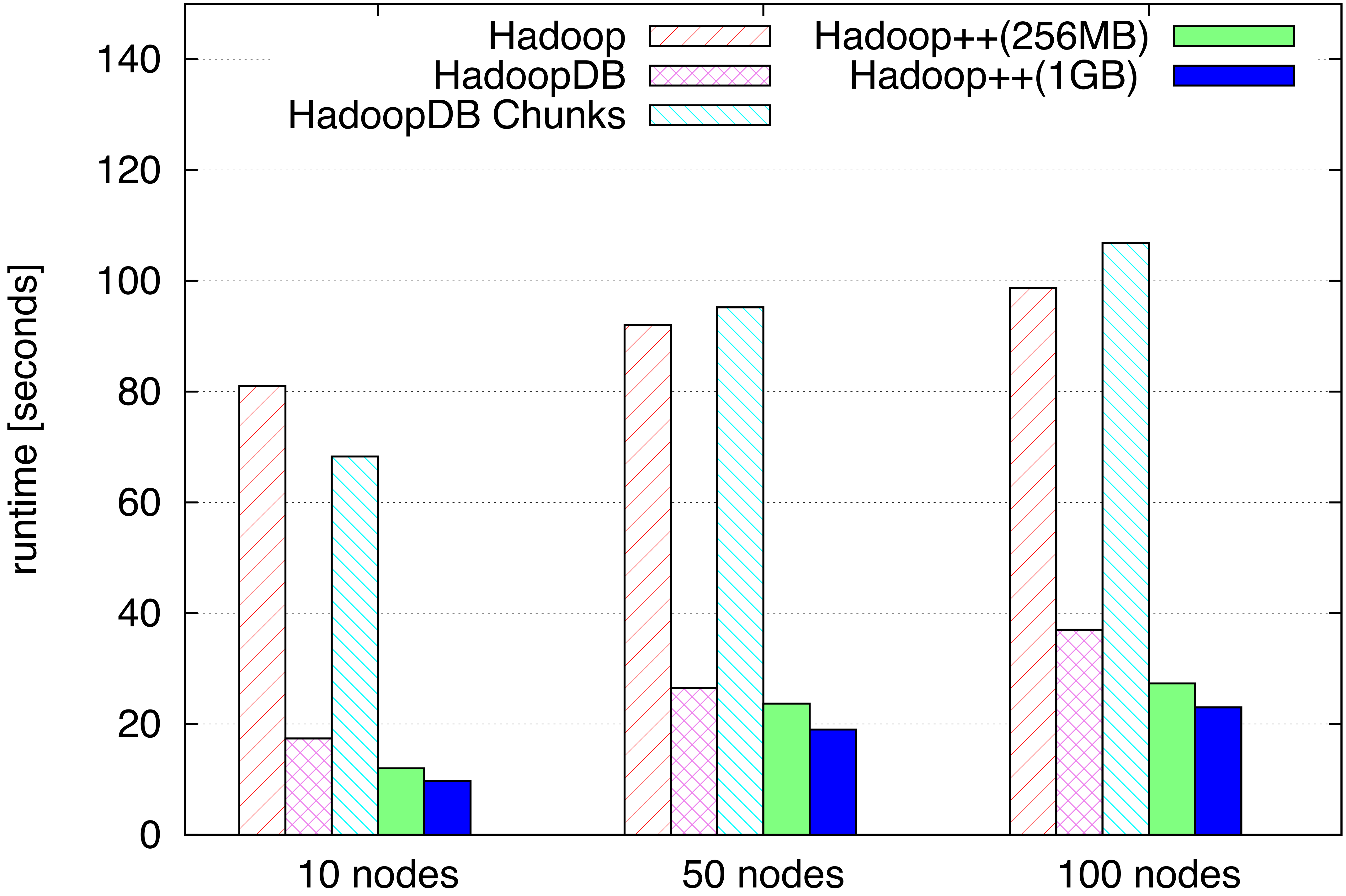
Trojan Index plus Trojan Join

- may combine both techniques
- may use index on join key
- may use index on different key
- may create multiple indexes inside the split
- in any case:
  - **both** scan access **and** index access paths possible

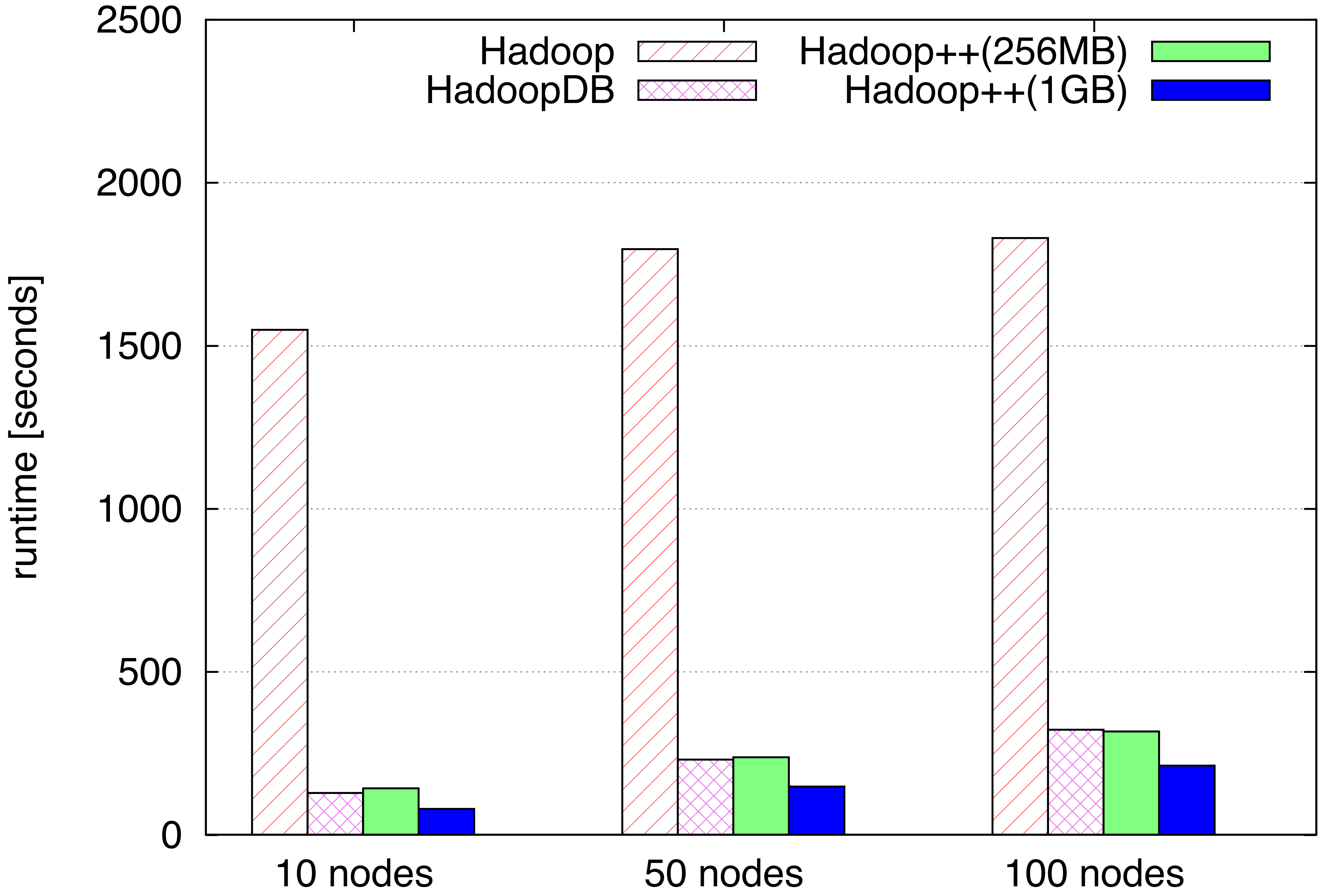
Experiments

- used benchmark as proposed in [Pavlo etal, SIGMOD 2009]
- benchmark defines several tasks
- two of them related to indexing and join processing
  - Selection Task
  - Join Task
- used up to 100 EC2 nodes as in HadoopDB-paper [Abouzeid etal, VLDB 2009]
- report average of three executions
- Some twist, see our paper:
  Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance
  Jörg Schad, Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz
  VLDB 2010
  Research Session-14 : Experimental Analysis and Performance (i.e., yesterday)
- **therefore:** also executed scaled-down experiments on small local cluster to verify

Selection Task

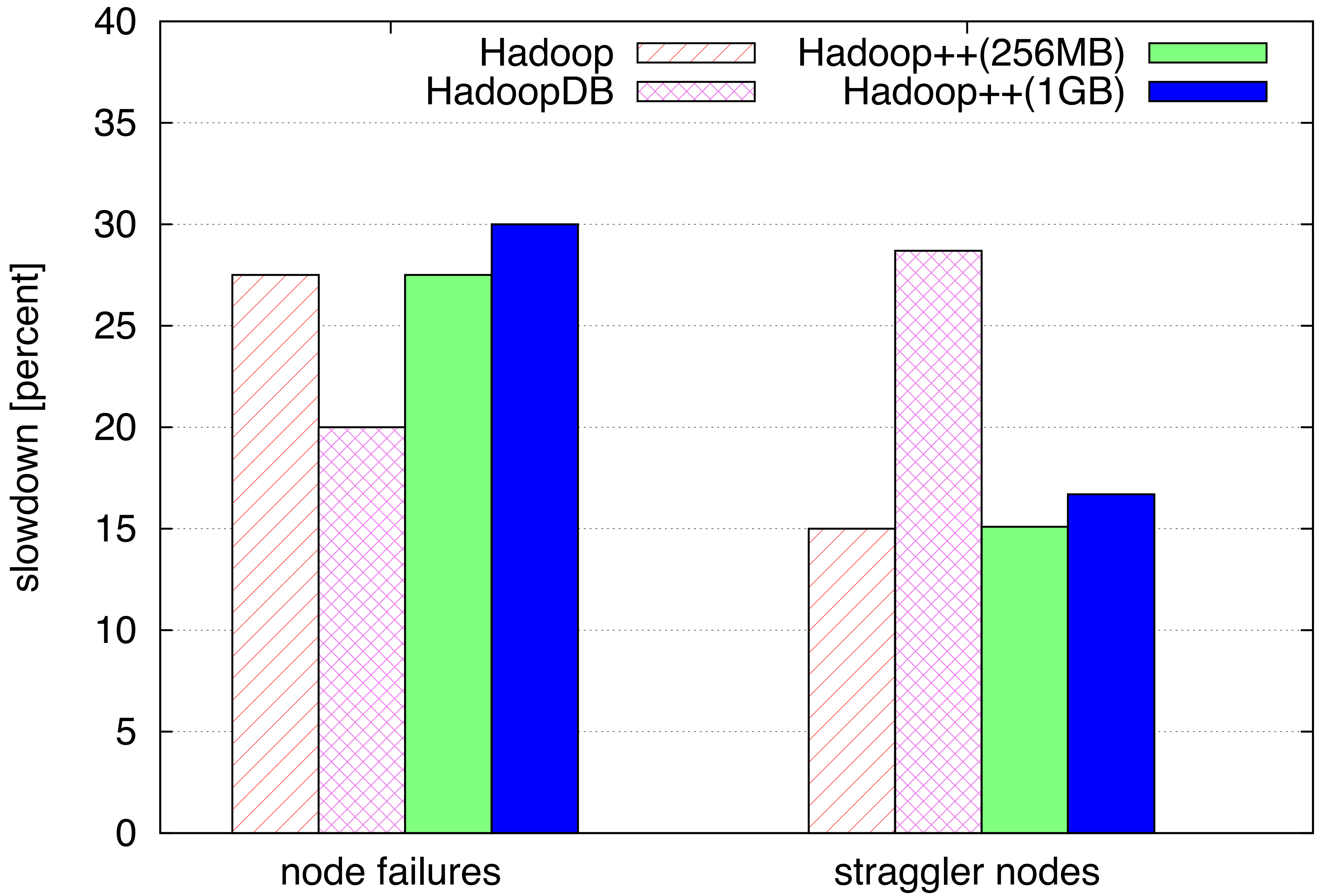


Join Task



runtime [seconds]
2500
2000
1500
1000
500
0
Hadoop
HadoopDB
Hadoop++(256MB)
Hadoop++(1GB)
10 nodes
50 nodes
100 nodes

Failover



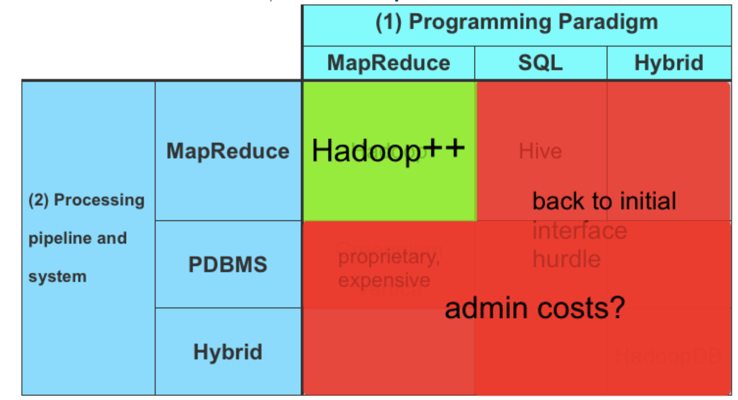
- we inherit fault tolerance from Hadoop!
- the Trojan effect!

Lessons Learned for our Community

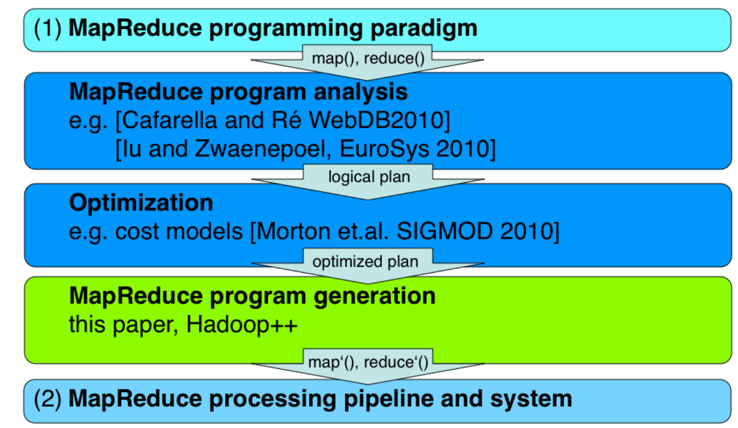
- indexing, co-partitioning, preprocessing, etc....
- ...are **not exclusive** to database management systems
- all these techniques may be successfully used in **any** data processing system, not only DBMS
- just one thing matters:
- **“Do we know anything about the schema and the anticipated workload in advance?“**
- if **yes**, we may:
  - create appropriate indexes
  - create co-partitions
  - etc.
- this holds for **both**
  - DBMS
  - and MapReduce/Hadoop

Conclusions

- we proposed Hadoop++
- a new approach to large scale data analysis

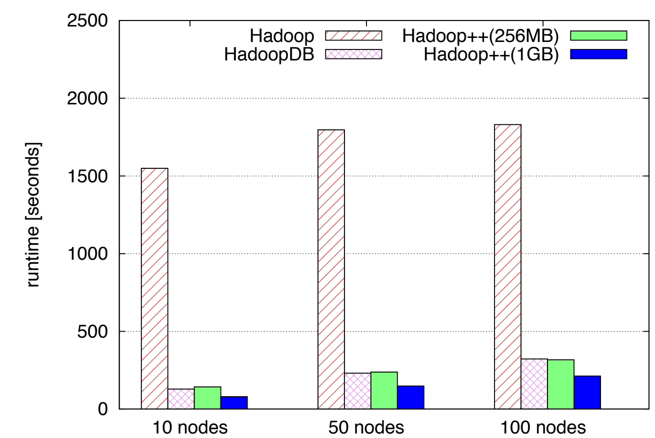


- keep the MapReduce interface
  and the MapReduce execution engine
- still: rewrite incoming MapReduce programs to more efficient ones

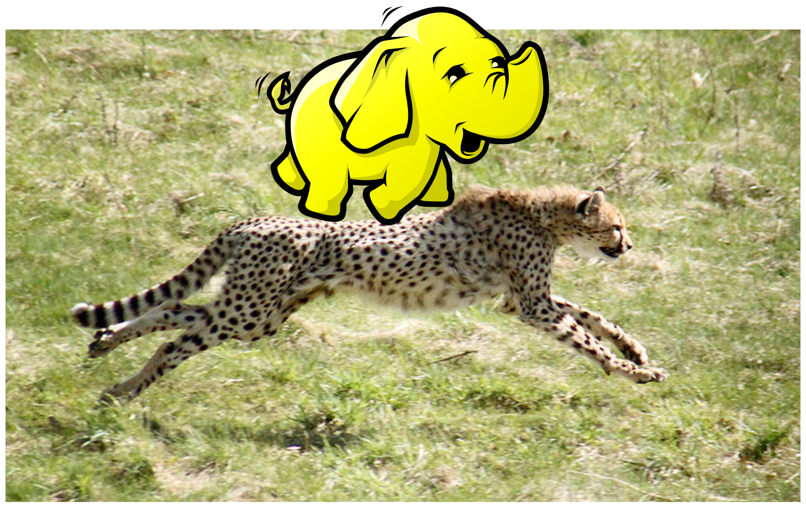


- inject code through **Trojan techniques**
- execute plans using existing MapReduce pipeline unchanged

- experimens with SIGMOD 2009 benchmark
- strong improvements in selection and join tasks
- up to a factor of 18 better than Hadoop



Future Work



- other Trojan techniques

  ongoing

- research challenges when executing MapReduce on the Cloud

  Flying Yellow Elephant: Predictable and Efficient MapReduce in the Cloud
  Jörg Schad
  VLDB PhD Workshop 2010 (see VLDB USB stick or online)

- marry Hadoop++ with OctopusDB* one-size-fits-all DBMS

  The Mimicking Octopus: Towards a one-size-fits-all Database Architecture
  Alekh Jindal
  VLDB PhD Workshop 2010 (see VLDB USB stick or online)

*patent pending