Discussion 5

relational algebra and **joins**

note: these slides were designed to assist discussion and might not be as informative as notes.

Many pages adapted from
https://github.com/aimichelle/cs186
and
https://courses.cs.washington.edu/courses/cse544/07au/lecture-notes/lecture8/lecture8.pdf

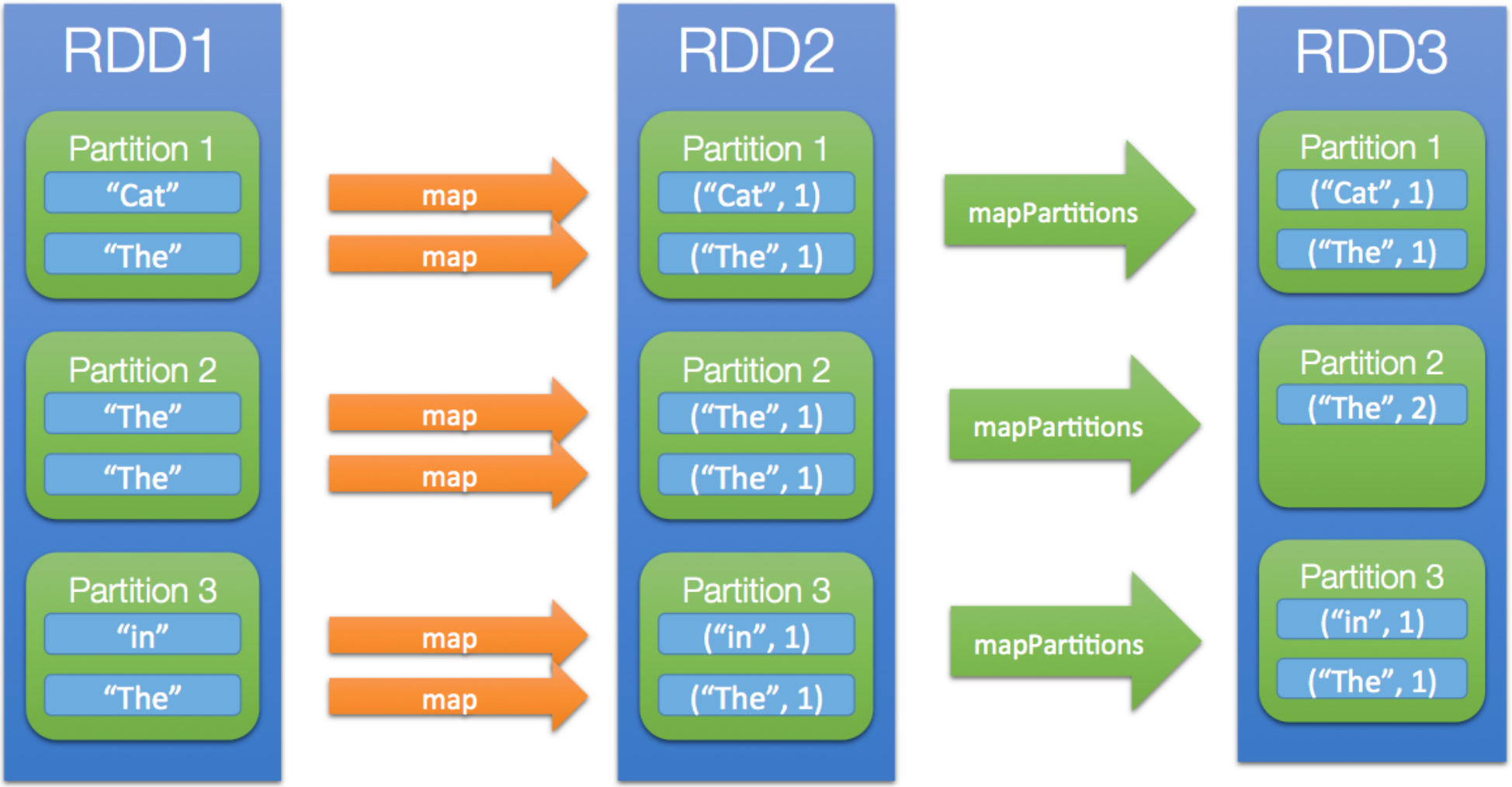
homework 3

mapPartitionsWithIndex

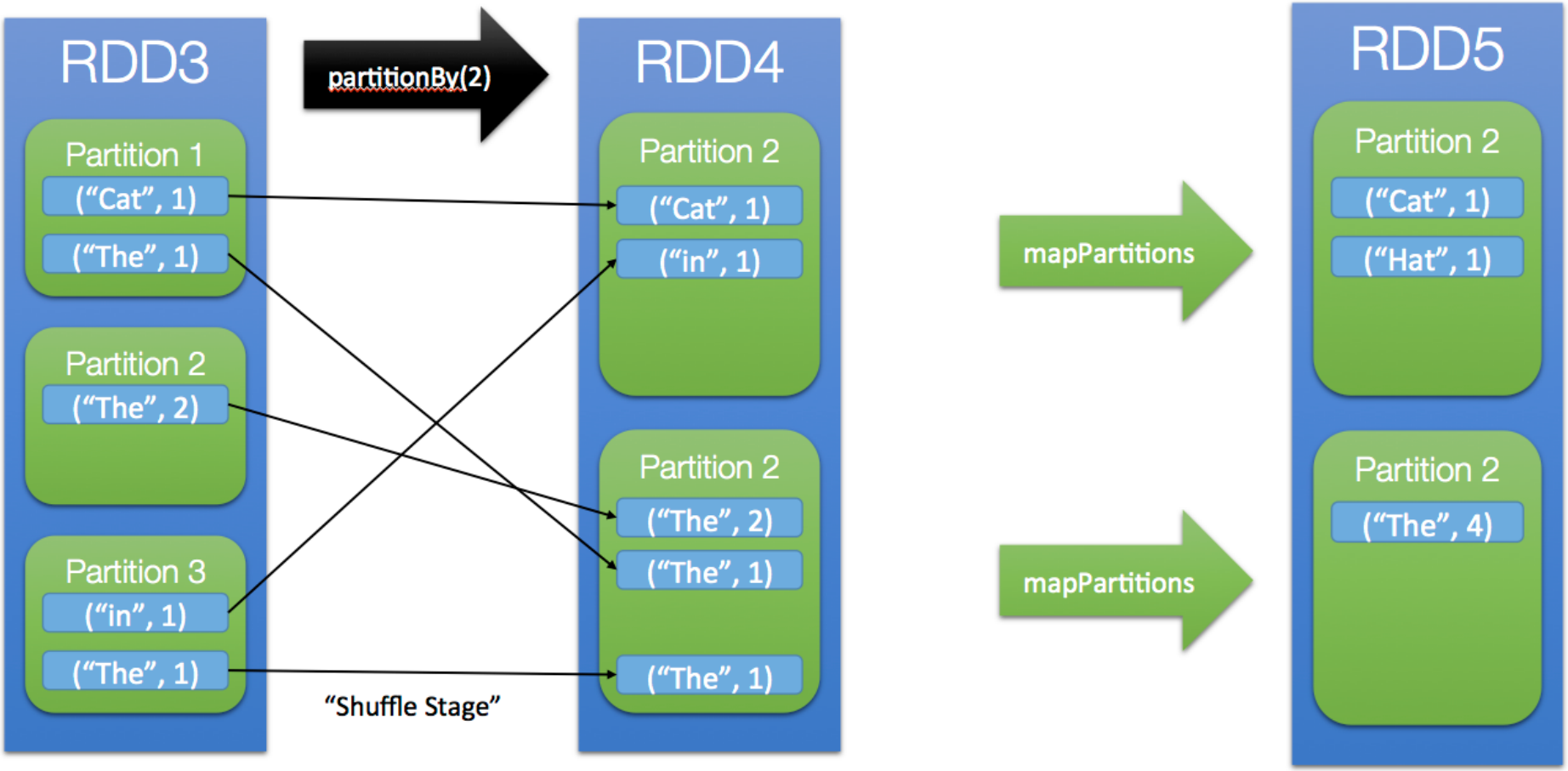
- "*Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition.*"
- **building block** for many functions.

.map(lambda x: (str(x), 1))

.mapPartitionsWithIndex(lambda _, iter: _combine(funcAdd(iter))

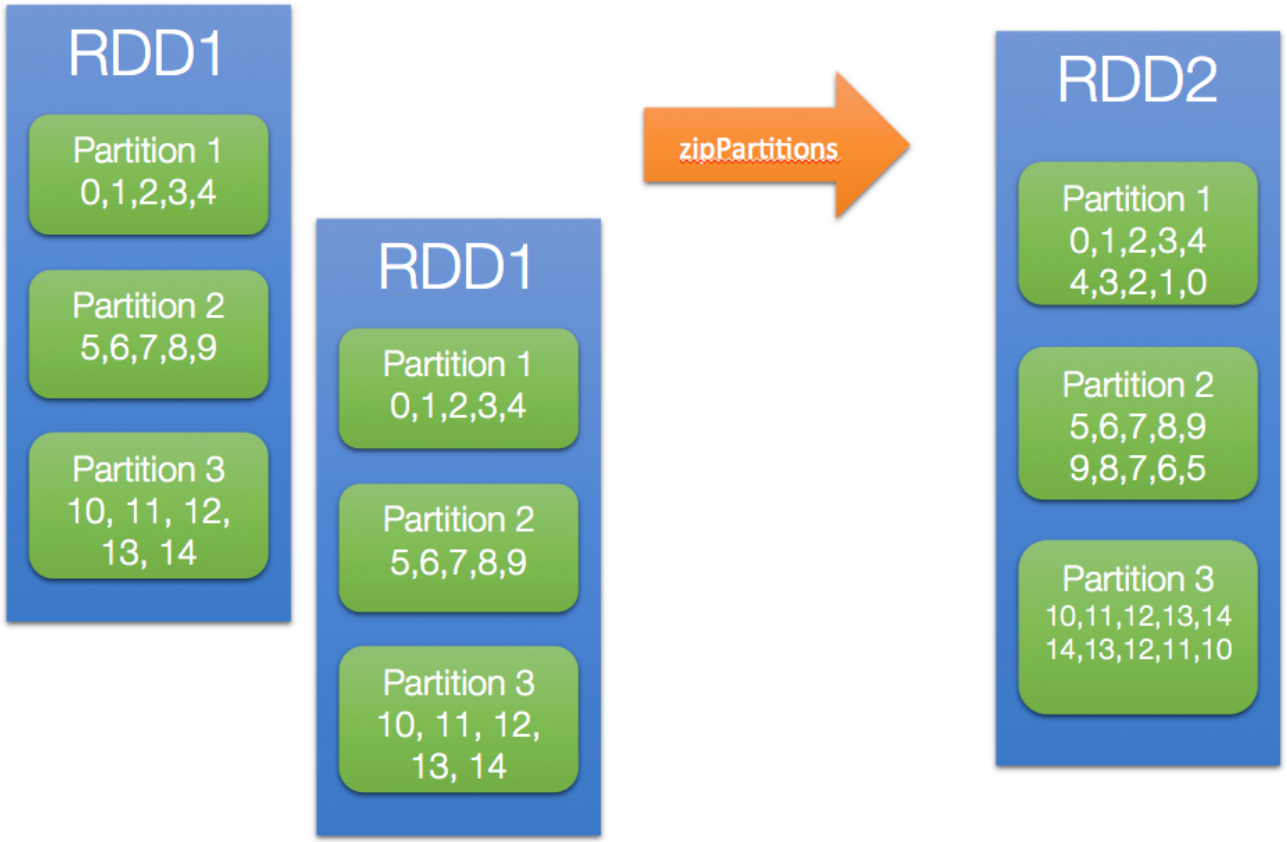


taken from Prof. Gonzalez's piazza post


RDD3
partitionBy(2)
RDD4
Partition 1
("Cat", 1)
("The", 1)
Partition 2
("The", 2)
Partition 3
("in", 1)
("The", 1)
"Shuffle Stage"
Partition 2
("Cat", 1)
("in", 1)
Partition 2
("The", 2)
("The", 1)
("The", 1)
mapPartitions
mapPartitions
RDD5
Partition 2
("Cat", 1)
("Hat", 1)
Partition 2
("The", 4)

```
rdd = sc.parallelize(range(20), 4)
zipper = rdd.zipPartitions(rdd, lambda lIter, rIter:
                iter(zip(list(lIter), list(reversed(list(rIter))))))
print zipper.collect()
```

```
[(0, 4), (1, 3), (2, 2), (3, 1), (4, 0), (5, 9), (6, 8), (7, 7), (8, 6), (9, 5), (10, 14),
 (11, 13), (12, 12), (13, 11), (14, 10), (15, 19), (16, 18), (17, 17), (18, 16), (19, 15)]
```



"collect() Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a ***sufficiently small!*** *subset of the data."*

comments

- the solution should be fairly short in length, once you
  - think through the **data flow**
  - understand how the **recommended utility functions** work
- things should be ~~fairly simple~~ simpler :)

midterm 1

- **review session**: sat
- **cheat sheet** (make sure you have all the equations from lecture down, and know what they mean!)
- until JOINs (ie this discussion)
- similar to previous years

Relational Algebra

- Relation -> Relation
- Sets

Operation	Symbol	Explanation
Selection	σ	Selects rows
Projection	π	Selects columns
Union	$\cup$	Tuples in r1 or r2
Intersection	$\cap$	Tuples in r1 and r2
Cross-product	$\times$	Combines two relations
Join	$\bowtie$	Conditional cross-product
Difference	-	Tuples in r1 not in r2

Selection & Projection

Example: $\pi_{\text{name, sid}}(\sigma_{\text{gpa} > 3.5}(R))$

name	sid	gpa
Bob	1	3.7
Sue	3	2.9
Ron	2	1.2
Al	4	4.0
Sally	5	3.6

Difference

- Takes rows in A that are not in B
- Example: $\sigma_{gpa > 3.5}(R) - \sigma_{sid\%2==0}(R)$

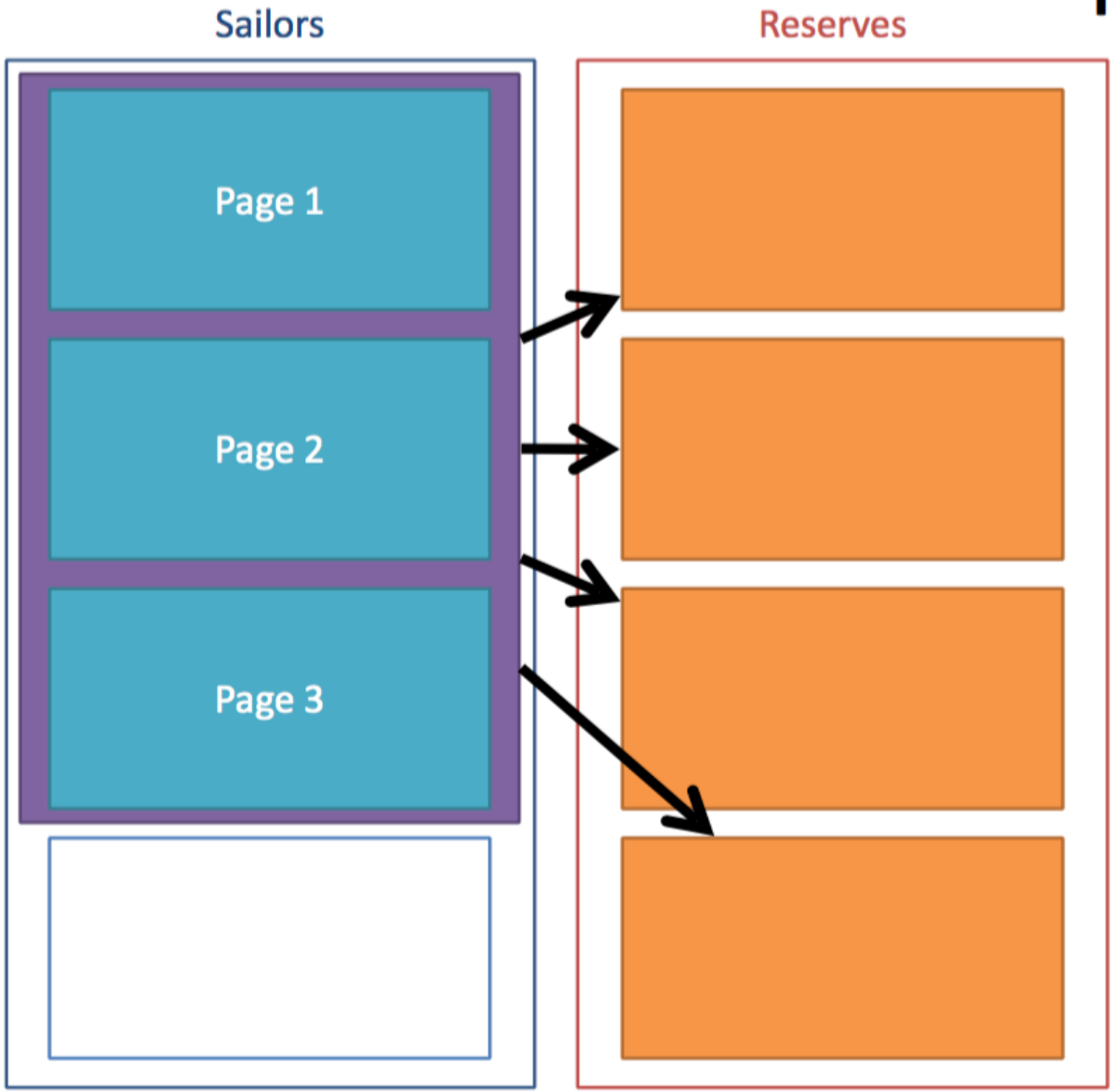
name	sid	gpa
Bob	1	3.7
Al	4	4.0
Sally	5	3.6

-

name	sid	gpa
Ron	2	1.2
Al	4	4.0

name	sid	gpa
Bob	1	3.7
Sally	5	3.6

Block Nested Loops Join

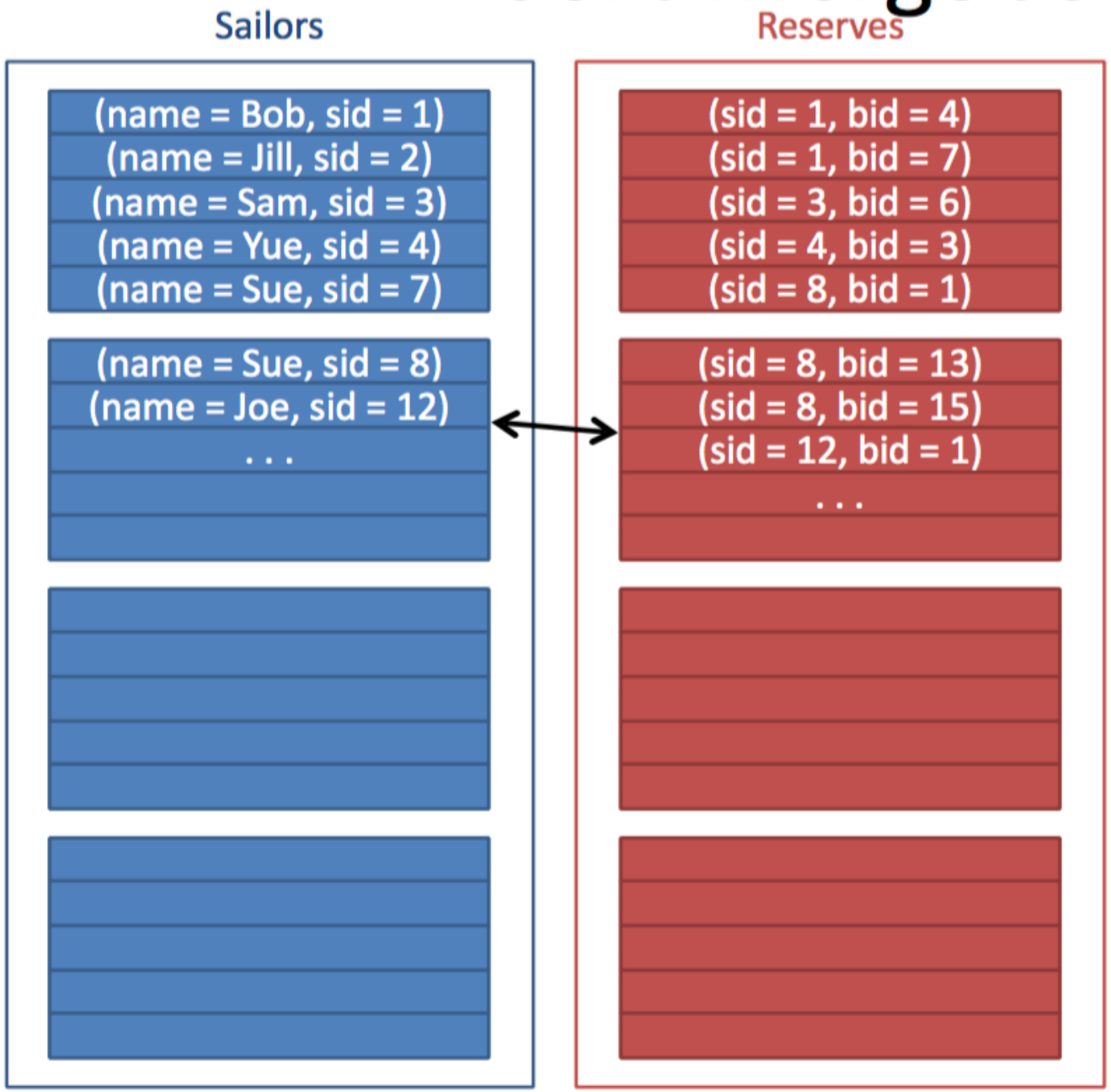


Key idea:
Take **k pages** of S and match with each page of R.

Steps:

1. Get **k** pages of S.
2. Iterate through each page in R.
3. Compare tuples in each.

Sort-Merge Join



Key idea:

Sort S and R **on join column**, then merge them!

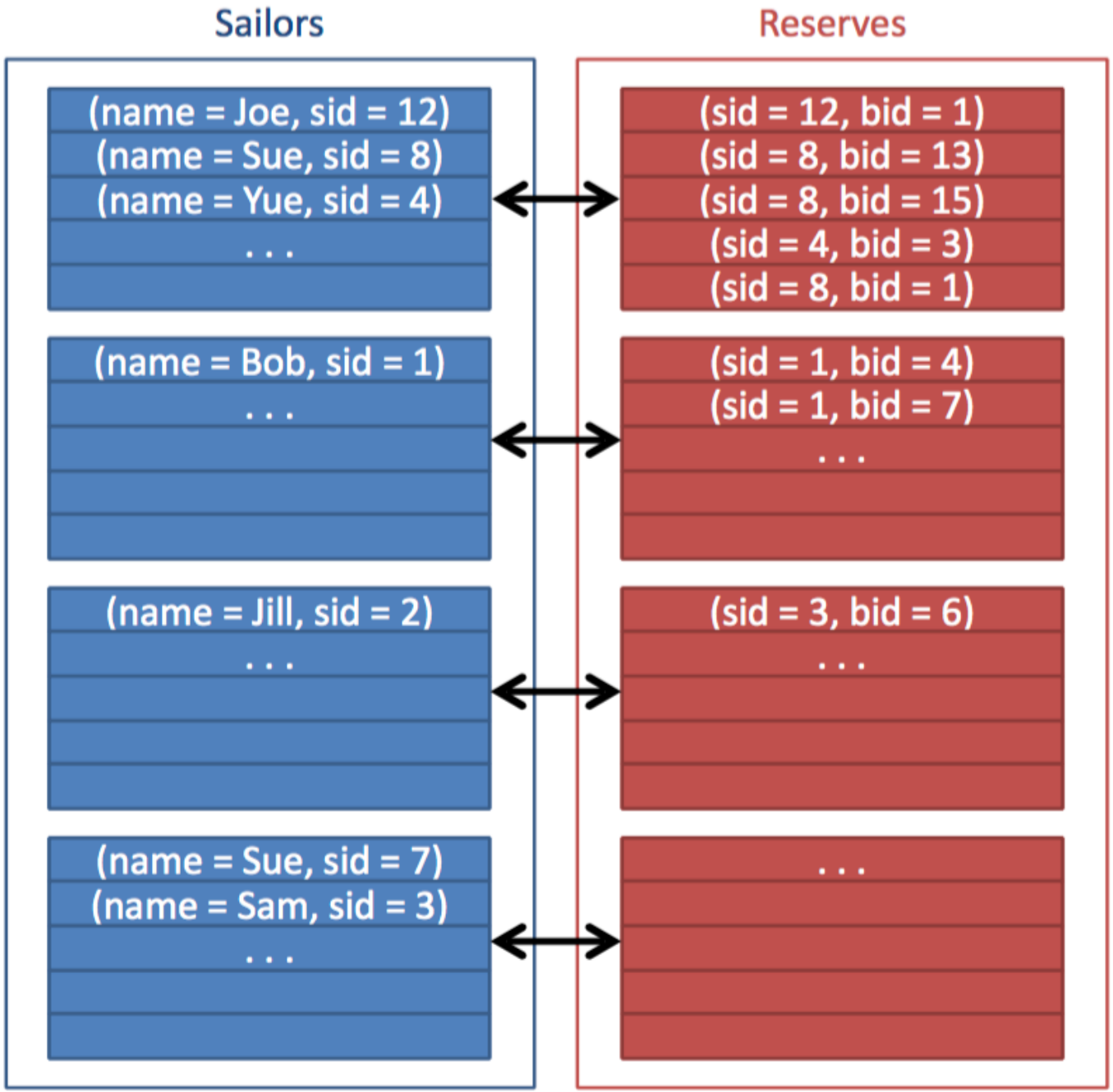
Steps:

1. Sort S and R.
2. "Zip" or merge.

Output:

(name = Bob, sid = 1, bid = 4)
(name = Bob, sid = 1, bid = 7)
(name = Sam, sid = 3, bid = 6)
. . .

Hash-Join



Key idea:

Partition S and R using same hash fn, then collect same partitions

Steps:

1. Partition S and R
2. Re-Hash, collect

GRACE

- also called "**partitioned**" (it's what you implement in homework!)
- insight: use hashing to increase "**rendezvous**" and *decreases passes*

$R \bowtie S$

- Step 1:
  - Hash S into M-1 buckets
  - Send all buckets to disk
- Step 2
  - Hash R into M-1 buckets
  - Send all buckets to disk
- Step 3
  - Join every pair of buckets

Hybrid

- Insight: use the data while it's in memory to save some additional I/Os

Assume we have extra memory available

Partition S into k buckets

t buckets S_1 , …, S_t stay in memory

k-t buckets S_{t+1}, …, S_k to disk

Partition R into k buckets

- First t buckets join immediately with S
- Rest k-t buckets go to disk

Finally, join k-t pairs of buckets:

(R_{t+1},S_{t+1}), (R_{t+2},S_{t+2}), …, (R_k,S_k)

worksheet

Spark & Relational

- Summary: "**In general, the SQL DBMSs were significantly faster and required less code to implement each task, but took longer to tune and load the data**."

- **Schema restriction**: "Parallel DBMSs require data to fit into the relational paradigm of rows and columns. In contrast, the MR model does not require that data files adhere to a schema defined using the relational data model."

- **Declarative**: "Anecdotal evidence from the MR community suggests that there is widespread sharing of MR code fragments to do common tasks, such as joining data sets. To alleviate the burden of having to re- implement repetitive tasks, the MR community is migrating high- level languages on top of the current interface to move such functionality into the run time."

write some algebra

Consider the schema:

```
Songs   (song_id,    song_name,    album_id,   weeks_in_top_40)
     Artists(artist_id, artist_name, first_year_active)
Albums  (album_id,   album_name,   artist_id, year_released, genre)
```

Write relational algebra expressions for the following query:

- Find the id of the artists who have albums of genre ‘pop’ or have spent over 10 weeks in the top 40.

$$\pi_{\text{Artists.artist\_id}} (\text{Artists} \bowtie (\sigma_{\text{Albums.genre} = \text{‘pop’}} \text{Albums}))$$

$$\cup$$

$$\pi_{\text{Albums.artist\_id}} (\text{Albums} \bowtie (\sigma_{\text{Songs.weeks\_in\_top\_40} > 10} \text{Songs}))$$

Consider the schema:

```
Songs   (song_id,   song_name,   album_id,  weeks_in_top_40)
     Artists(artist_id, artist_name, first_year_active)
Albums  (album_id,  album_name,  artist_id, year_released, genre)
```

Write relational algebra expressions for the following query:

- Find the names of all artists who do not have any albums.

$$\pi_{\text{Artists.artist\_name}} (\text{Artists} \bowtie ($$

$$(\pi_{\text{Artists.artist\_id}} \text{Artists}) - (\pi_{\text{Albums.artist\_id}} \text{Albums}))$$