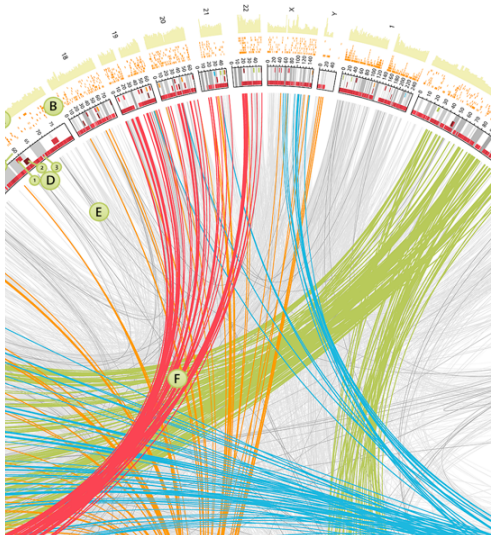
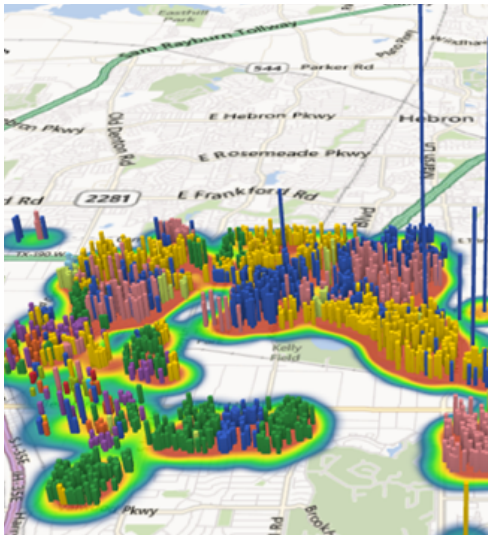
Big Data Analytics - Relational Operations with MapReduce



Prof. Nastaran Fatemi | HES-SO Yverdon-les-Bains (HEIG-VD)

Plan

- Relational databases and MapReduce
- Relational algebra
- MapReduce algorithms
  - Selection
  - Projection
  - Set operators
    - Intersection
    - Union
    - Difference
  - GroupBy and Aggregation
  - Natural Join
    - Reduce-side join
    - Map-side join
    - In-memory join

Introduction

- Role of relational databases in today's organizations
  - Where does MapReduce fit in?
- MapReduce algorithms for processing relational data
  - How do I perform a join, etc.?
- Evolving roles of relational databases and MapReduce
  - What's in store for the future?

Big Data Analysis

- Peta-scale datasets are everywhere:
  - Facebook has 2.5 PB of user data + 15 TB/day
  - eBay has 6.5 PB of user data + 50 TB/day
  - …
- A lot of these datasets are (mostly) structured
  - Query logs
  - Point-of-sale records
  - User data (e.g., demographics)
  - …
- How do we perform data analysis at scale?
  - Relational databases and SQL
  - MapReduce (Hadoop)

Relational Databases vs. MapReduce

- Relational databases:
  - Multipurpose: analysis and transactions; batch and interactive
  - Data integrity via ACID transactions
  - Lots of tools in software ecosystem (for ingesting, reporting, etc.)
  - Supports SQL (and SQL integration, e.g., JDBC)
  - Automatic SQL query optimization

- MapReduce (Hadoop):
  - Designed for large clusters, fault tolerant
  - Data is accessed in “native format”
  - Supports many query languages
  - Programmers retain control over performance
  - Open source

Database workloads

- OLTP (online transaction processing)
  - Typical applications: e-commerce, banking, airline reservations
  - User facing: real-time, low latency, highly-concurrent
  - Tasks: relatively small set of “standard” transactional queries
  - Data access pattern: random reads, updates, writes (involving relatively small amounts of data)

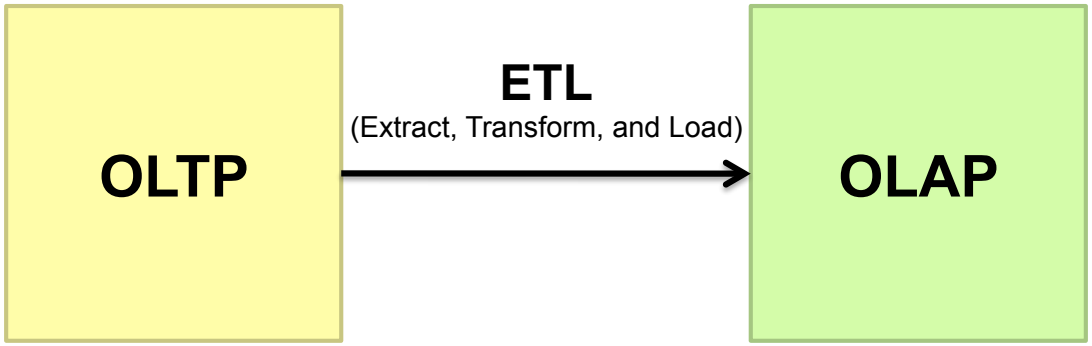
- OLAP (online analytical processing)
  - Typical applications: business intelligence, data mining
  - Back-end processing: batch workloads
  - Tasks: complex analytical queries, often ad hoc
  - Data access pattern: table scans, large amounts of data involved per query

One Database or Two?

- Downsides of co-existing OLTP and OLAP workloads
  - Poor memory management
  - Conflicting data access patterns
  - Variable latency

- Solution: separate databases
  - User-facing OLTP database for high-volume transactions
  - Data warehouse for OLAP workloads
  - How do we connect the two?

OLTP/OLAP Architecture



OLTP/OLAP Integration

- OLTP database for user-facing transactions
  - Retain records of all activity
  - Periodic ETL (e.g., nightly)

- Extract-Transform-Load (ETL)
  - Extract records from source
  - Transform: clean data, check integrity, aggregate, etc.

- Load into OLAP database
  - OLAP database for data warehousing
  - Business intelligence: reporting, ad hoc queries, data mining, etc.
  - Feedback to improve OLTP services

Business Intelligence

- Premise: more data leads to better business decisions
  - Periodic reporting as well as ad hoc queries
  - Analysts, not programmers (importance of tools and dashboards)

- Examples:
  - Slicing-and-dicing activity by different dimensions to better understand the marketplace
  - Analyzing log data to improve OLTP experience
  - Analyzing log data to better optimize ad placement
  - Analyzing purchasing trends for better supply-chain management
  - Mining for correlations between otherwise unrelated activities

Common database query primitives

- A number of operations on large-scale data are realized in database queries.
- In many traditional database applications queries involve retrieval of small amounts of data, even though the database itself may be large.
  - For example, a query may ask for the bank balance of one particular account.
  - Such queries are not useful applications of map-reduce.
- However, there are many operations on data that can be described easily in terms of the **common database-query primitives**, even if the queries themselves are not executed within a database management system.
- Thus, a good starting point for seeing applications of map-reduce is by considering the standard operations on relations.

Web Links Example
Scenario

- The relation *Links* describes the structure of the Web.
  - There are two attributes, *From* and *To*.
  - A row, or tuple, of the relation is a pair of URL's, such that there is at least one link from the first URL to the second.

From	*To*
url1	url2
url1	url3
url2	url3
url2	url4
...	...

- A relation, however large, can be stored as a file in a distributed file system. The elements of this file are the tuples of the relation.

Web Links Example
Problem: Paths of length two

- Problem: Find the paths of length two in the Web, using the relation *Links*.
  - That is, we want to find the triples of URL's *(u,v,w)* such that there is a link from *u* to *v* and a link from *v* to *w*.

Social Networking Example
Problem: friends statistics

- Imagine that a social-networking site has a relation *Friends(User, Friend)*
  - This relation has tuples that are pairs *(a, b)* such that b is a friend of a.
- Problem: We want to develop statistics about the number of friends members have.
  - Our first step would be to compute a count of the number of friends of each user.
  - How can we solve this problem using relational algebra (RA) operators?
  - Can you give the RA expression or the SQL query?

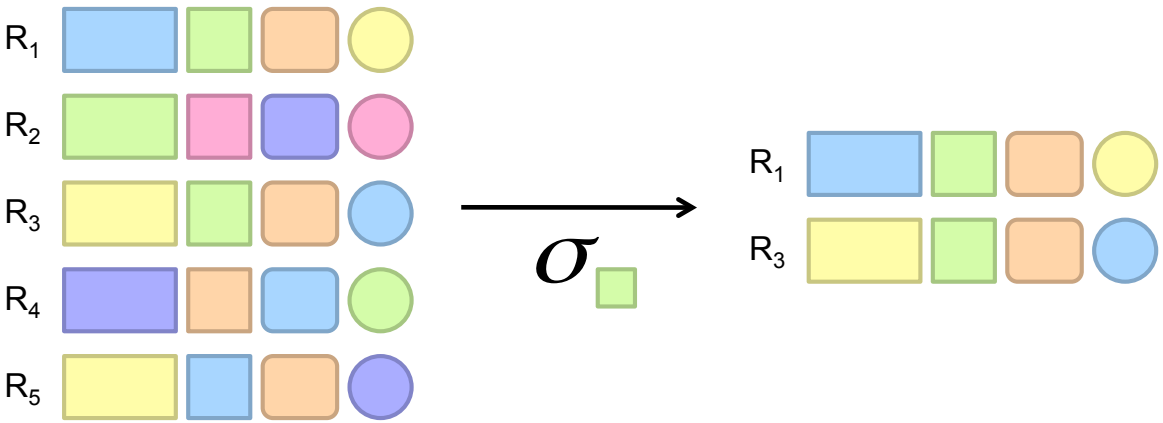
Relational databases

- A relational database is comprised of tables
- Each table represents a relation = collection of tuples (rows)
- Each tuple consists of multiple fields

Relational Algebra

- Primitives
  - Selection ()
  - Projection ()
  - Cartesian product ()
  - Set union ()
  - Set difference ()
  - Rename ()
- Other operations
  - Join (⋈)
  - Group by… aggregation
  - …

Selection



Selection in MapReduce

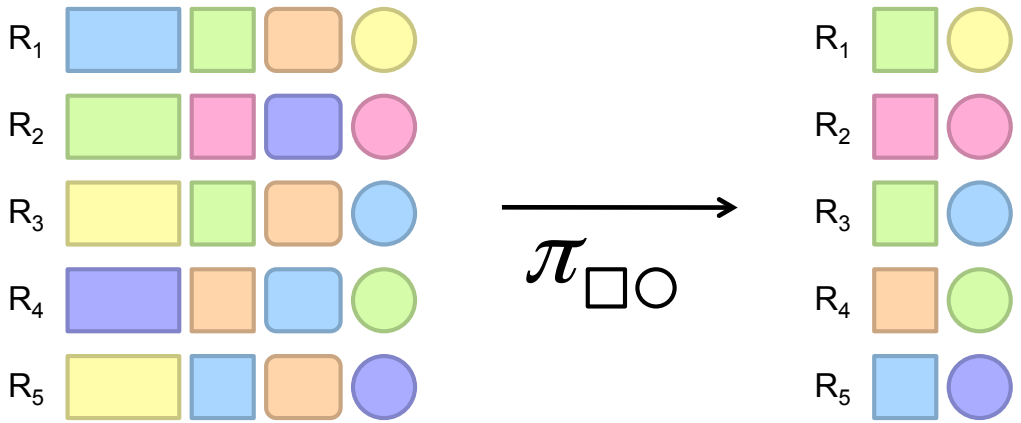
- Selections really do not need the full power of map-reduce.
  - They can be done most conveniently in the map portion alone.
- Implementation of of selection

```
class Mapper

   method Map(rowkey key, tuple t)
      if t satisfies the predicate
         Emit(tuple t, null)
```

- Identity reducer

Projection



Projection in MapReduce

- Projection is also a simple operation in terms of MapReduce.
  - Because projection may cause the same tuple to appear several times, the Reduce function must eliminate duplicates.

- Implementation of of projection

```
class Mapper
   method Map(rowkey key, tuple t)
      tuple g = project(t)  // extract required fields to tuple g
      Emit(tuple g, null)

class Reducer
   method Reduce(tuple t, array n)   // n is an array of nulls
      Emit(tuple t, null)
```

Union in MapReduce

- Suppose relations *R* and *S* have the same schema
  - Map tasks will be assigned chunks from either *R* or *S*; it doesn't matter which. The Map tasks don't really do anything except pass their input tuples as key-value pairs to the Reduce tasks.
  - Reducer is used to eliminate duplicates.
- Implementation of Union

```
class Mapper
   method Map(rowkey key, tuple t)
      Emit(tuple t, null)

class Reducer
   method Reduce(tuple t, array n)  // n is an array of
                                    // one or two nulls
      Emit(tuple t, null)
```

Intersection in MapReduce

- Suppose relations *R* and *S* have the same schema
  - Mappers are fed by all records of two sets to be intersected.
  - Reducer emits only records that occurred twice. It is possible only if both sets contain this record because record includes primary key and can occur in one set only once.
- Implementation of Union

```
class Mapper
   method Map(rowkey key, tuple t)
      Emit(tuple t, null)

class Reducer
   method Reduce(tuple t, array n)  // n is an array of
                                    // one or two nulls
      if n.size() = 2
          Emit(tuple t, null)
```

Difference in MapReduce

- The Difference *R* – *S* requires a bit more thought.
  - The only way a tuple *t* can appear in the output is if it is in *R* but not in *S*.
  - The Map function can pass tuples from *R* and *S* through, but must inform the Reduce function whether the tuple came from R or S.

- Exercise: propose a MapReduce implementation for *R* – *S.*

GroupBy and Aggregation in MapReduce

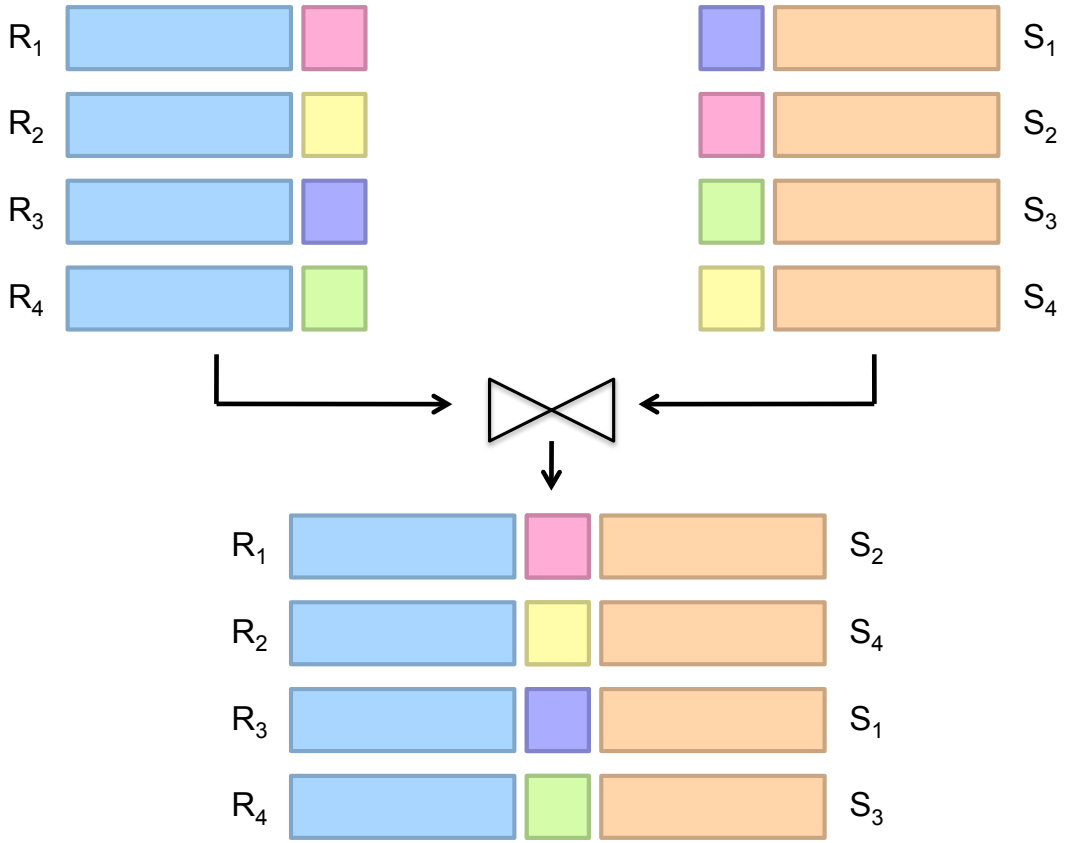
- Grouping and aggregation can be performed in one MapReduce job.
  - Mapper extracts from each tuple values to group by and aggregate and emits them.
  - Reducer receives values to be aggregated already grouped and calculates an aggregation function.

```
class Mapper
   method Map(rowkey key,
              tuple [value GroupBy, value AggregateBy, value ...])
      Emit(value GroupBy, value AggregateBy)

class Reducer
   method Reduce(value GroupBy, [v1, v2,...])
      Emit(value GroupBy, aggregate( [v1, v2,...] ) )
                        // aggregate(): sum(), count(), max(), etc.
```

- **Exercice**: Given the table R(a, b, c) write a MapReduce schema that calculates the following query:
  SELECT b, COUNT (DISTINCT c)
  FROM R
  GROUP BY b

Relational Joins



Join Algorithms in MapReduce

- Let's suppose that we want to perform relational join on two data sets named *R*, and *S*:

R	*S*
(k_1, r_1, R_1)	(k_1, s_1, S_1)
(k_2, r_2, R_2)	(k_2, s_2, S_2)
(k_3, r_3, R_3)	(k_3, s_3, S_3)
...	...

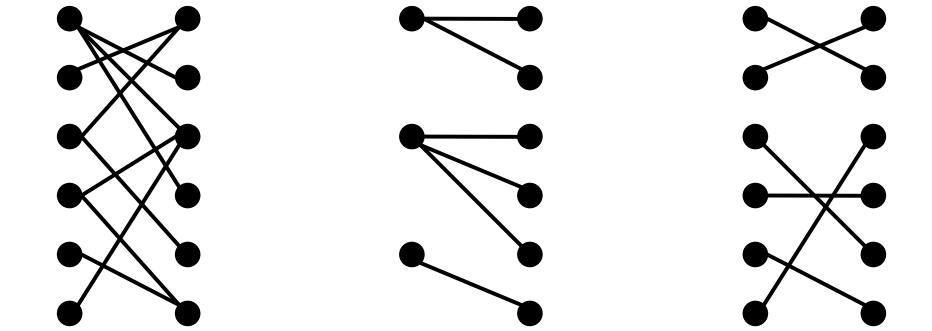
k: the key we want to join on
r_n and s_n : the unique ids for each table
R_n and S_n : other attributes in the tables

- We introduce three implementations of Join
  - Reduce-side join
  - Map-side join
  - In-memory join

Reduce-side Join

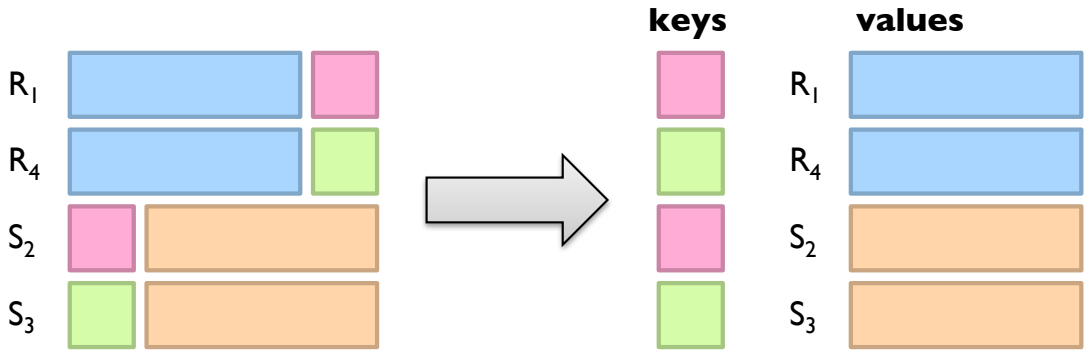
- Basic idea: group by join key
  - Map over both sets of tuples
  - Emit join key as the intermediate key and the tuple itself as the intermediate value
  - Execution framework groups together tuples sharing the same key
    which is exactly what we need to perform the join operation!
  - Perform actual join in reducer
  - This approach is know as parallel “sort-merge join” in database community

- Three different cases to consider



Reduce-side Join
One-to-One example

Map



Reduce

Note: no guarantee if R is going to come first or S

Reduce-side Join
One-to-One case

- The reducer will be presented keys and lists of values, as for example:

$k_{23} \rightarrow [(r_{64}, R_{64}), (s_{84}, S_{84})]$
$k_{37} \rightarrow [(r_{68}, R_{68})]$
$k_{59} \rightarrow [(s_{97}, S_{97}), (r_{81}, R_{81})]$
$k_{61} \rightarrow [(s_{99}, S_{99})]$

At most one tuple from S and one tuple from T share the same join key. (it may be the case that no tuple from S shares the join key with a tuple from T, or vice vers)

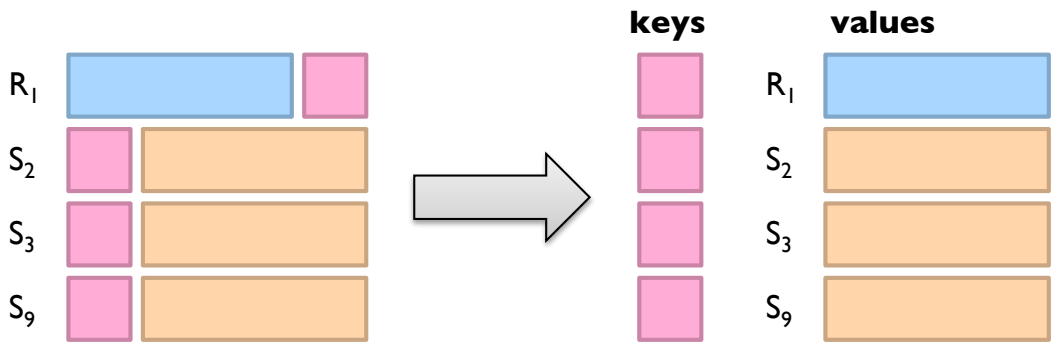
- If there are two values associated with a key, then we know that one must be from *R* and the other must be from *S*.
  - Recall that in the basic MapReduce programming model, no guarantees are made about value ordering, so the first value might be from *R* or from *S*.
- We can proceed to join the two tuples.
  - If there is only one value associated with a key, this means that no tuple in the other dataset shares the join key, so the reducer does nothing.

Reduce-side Join
One-to-Many problem

- The above algorithm will still work, but when processing each key in the reducer, we have no idea when the value corresponding to the tuple from *R* will be encountered, since values are arbitrarily ordered.
- The easiest solution is to **buffer all values in memory**, pick out the tuple from *R*, and then cross it with every tuple from *S* to perform the join.
  - This creates a **scalability bottleneck** since we may not have sufficient memory to hold all the tuples with the same join key!
- What would be a better solution?
  - We have already seen this before...

Reduce-side Join
One-to-Many example

Map



Reduce

What's the problem?

Reduce-side Join
One-to-Many solution

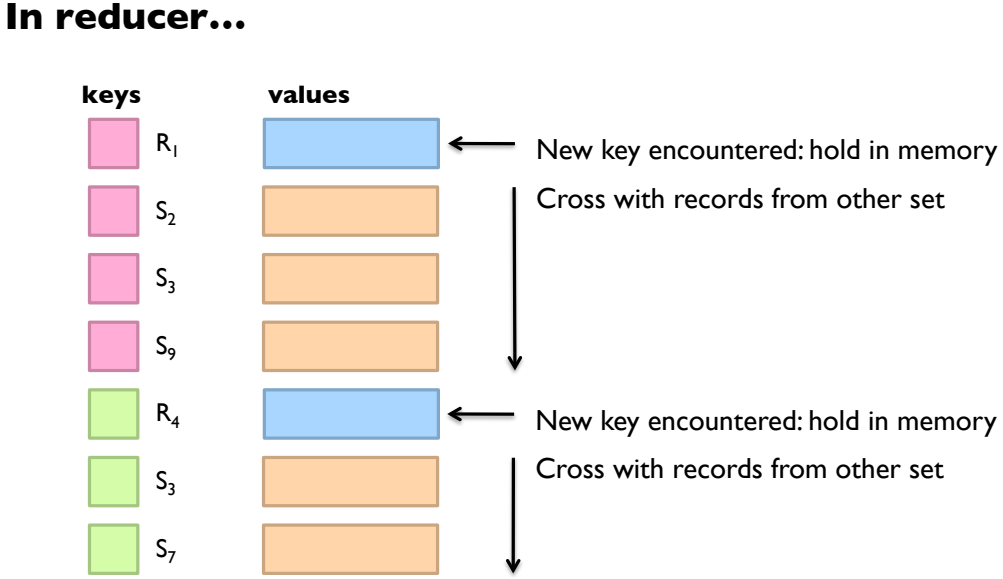
- This is a problem that requires a **secondary sort**
  => Use **value-to-key conversion** design pattern!

- In the mapper, instead of simply emitting the join key as the intermediate key, we create a composite key consisting of the join key and the tuple id (from either *R* or *S*).
  - Two additional changes are required
    1. We must define the sort order of the keys to first sort by the join key, and then sort all tuple ids from *R* before all tuple ids from *S*.
    2. We must define the partitioner to pay attention to only the join key, so that all composite keys with the same join key arrive at the same reducer.
  - After applying the value-to-key conversion design pattern, the reducer will be presented with keys and values such as the following:

    $(k_{82}, r_{105}) \rightarrow [(R_{105})]$
    $(k_{82}, s_{98}) \rightarrow [(S_{98})]$
    $(k_{82}, s_{101}) \rightarrow [(S_{101})]$
    $(k_{82}, s_{137}) \rightarrow [(S_{137})]$
    $(k_{89}, r_{50}) \rightarrow [(R_{50})]$
    ...

Reduce-side Join
One-to-Many solution

- Whenever the reducer encounters a new join key, it is guaranteed that the associated value will be the relevant tuple from *R*.
- The reducer can hold this tuple in memory and then proceed to cross it with tuples from *S* in subsequent steps (until a new join key is encountered).



- Since the MapReduce execution framework performs the sorting, there is no need to buffer tuples (other than the single one from *R*).
  - Thus, we have eliminated the scalability bottleneck.

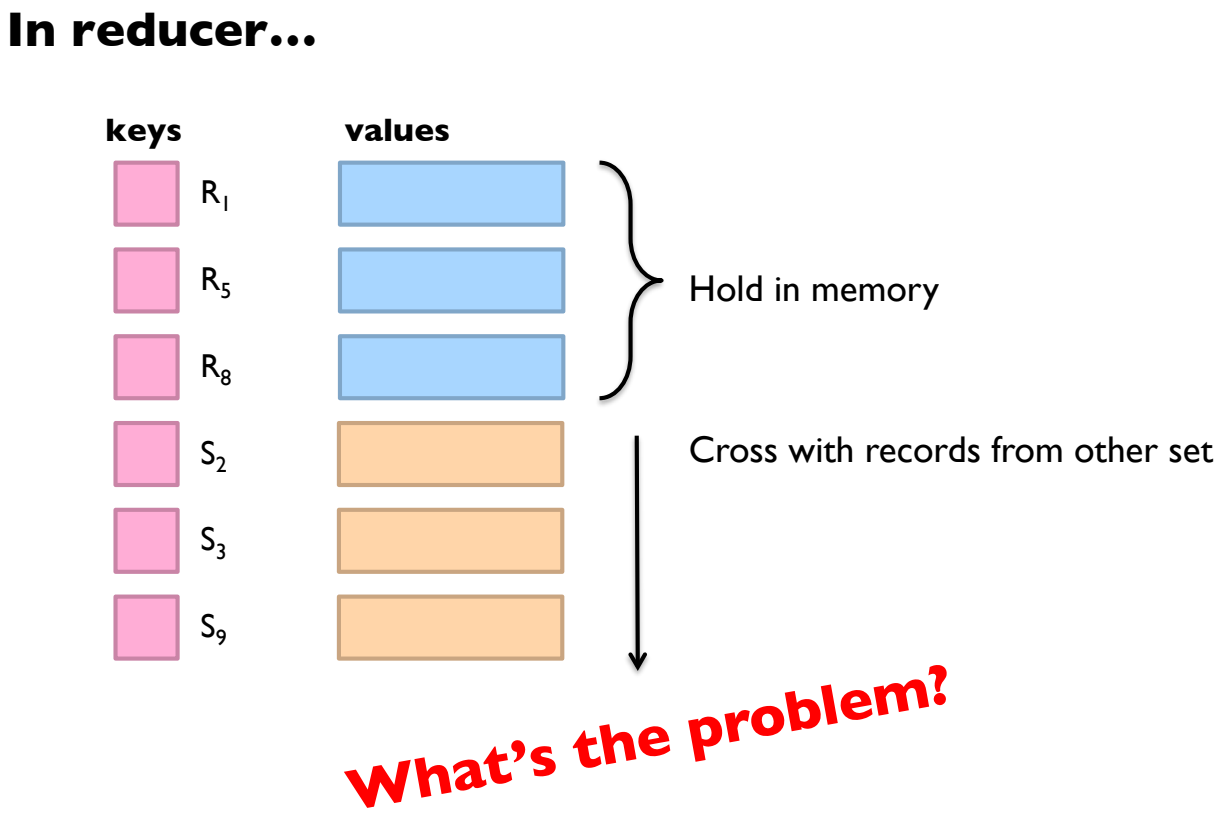
Reduce-side Join
Many-to-Many case

- Assuming that *R* is the smaller dataset, the above algorithm works as well.
  Consider what happens at the reducer:

$(k_{82}, r_{105}) \rightarrow [(R_{105})]$
$(k_{82}, r_{124}) \rightarrow [(R_{124})]$
...
$(k_{82}, s_{98}) \rightarrow [(S_{98})]$
$(k_{82}, s_{101}) \rightarrow [(S_{101})]$
$(k_{89}, s_{137}) \rightarrow [(S_{137})]$
...

- All the tuples from *R* with the same join key will be encountered first, which the reducer can buffer in memory.
- As the reducer processes each tuple from *S*, it is crossed with all the tuples from *R*.
  - Attention: We are assuming that the tuples from *R* (with the same join key) will fit into memory, which is a limitation of this algorithm (and why we want to control the sort order so that the smaller dataset comes first).

Reduce-side Join
Many-to-Many example

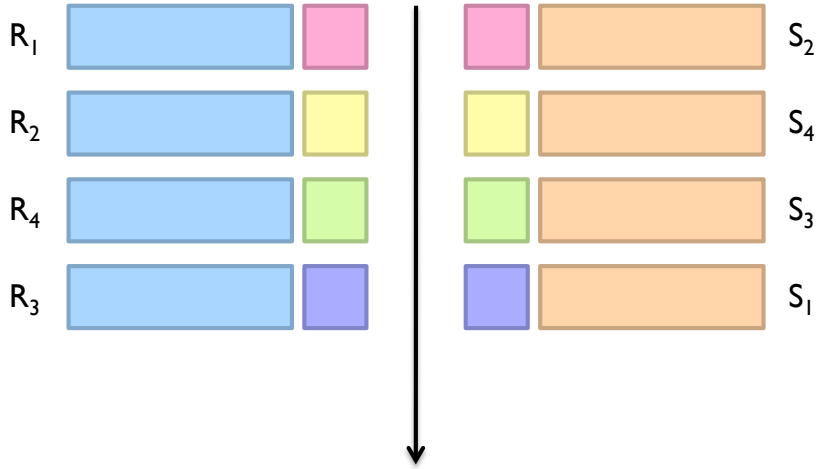


Reduce-side Join
Disadvantages

- This approach has the following disadvantages:
  - Mapper emits absolutely all data, even for keys that occur only in one set and have no pair in the other.
  - Reducer should hold all data for one key in the memory. If data doesn't fit the memory, its Reducer's responsibility to handle this by some kind of swap.

Map-side Join
Basic Idea

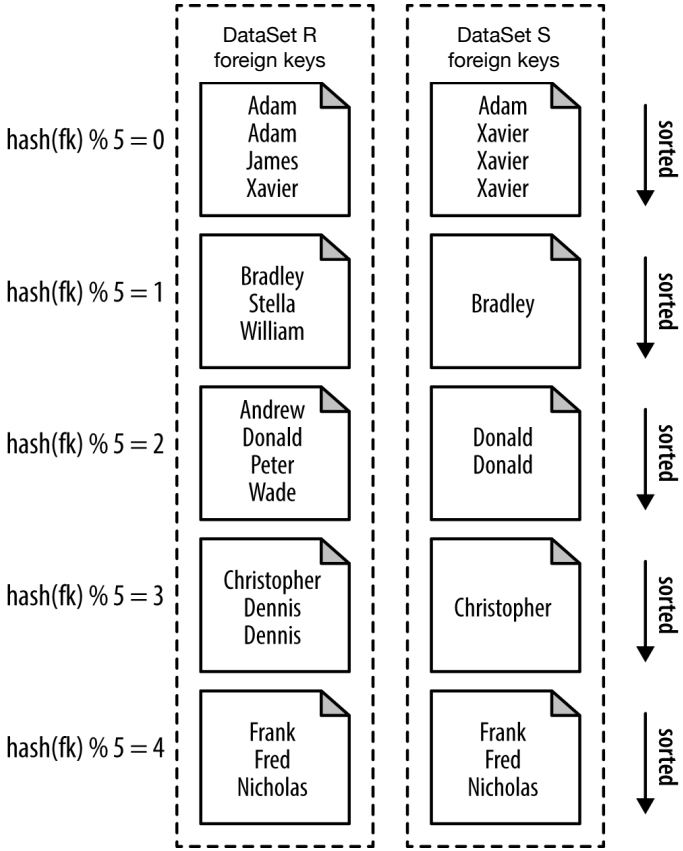
Assume two datasets are sorted by the join key:



A sequential scan through both datasets to join
(called a “merge join” in database terminology)

Map-Side Join
Parallel Scans

- If datasets are sorted by join key, join can be accomplished by a scan over both datasets
- How can we accomplish this in parallel?
  - Partition and sort both datasets in the same manner



Map-Side Join
Parallel Scans

- In MapReduce:
  - Map over one dataset, read from other corresponding partition
  - No reducers necessary (unless to repartition or resort)
- Example:
  - suppose *R* and *S* are both divided into five files, partitioned in the same manner by the join key.
  - suppose that in each file, the tuples were sorted by the join key.
  - In this case, we simply need to merge join the first file of *R* with the first file of *S*, the second file with *R* with the second file of *S*, etc.
- No reducer is required, unless the programmer wishes to repartition the output or perform further processing.

Map-Side Join
Discussion

- A map-side join is far more efficient than a reduce-side join
  - There is no need to shuffle the datasets over the network.
- But is the condition of “consistently partitioned datasets” realistic?
  - In many cases, yes
    - Relational joins happen within the broader context of a workflow, which may include multiple steps
    - The datasets that are to be joined may be the output of previous processes.
    - If the workflow is known in advance and relatively static (both reasonable assumptions in a mature workflow), we can engineer the previous processes to generate output sorted and partitioned in a way that makes efficient map-side joins possible (in MapReduce, by using a custom partitioner and controlling the sort order of key-value pairs).
  - Less efficient for ad hoc data analysis
    - If we wish to join datasets on multiple keys
    - Map-side joins will require repartitioning of the data
    - It is always possible to repartition a dataset using an identity mapper and reducer, but of course, this incurs the cost of shuffling data over the network

In-Memory Join

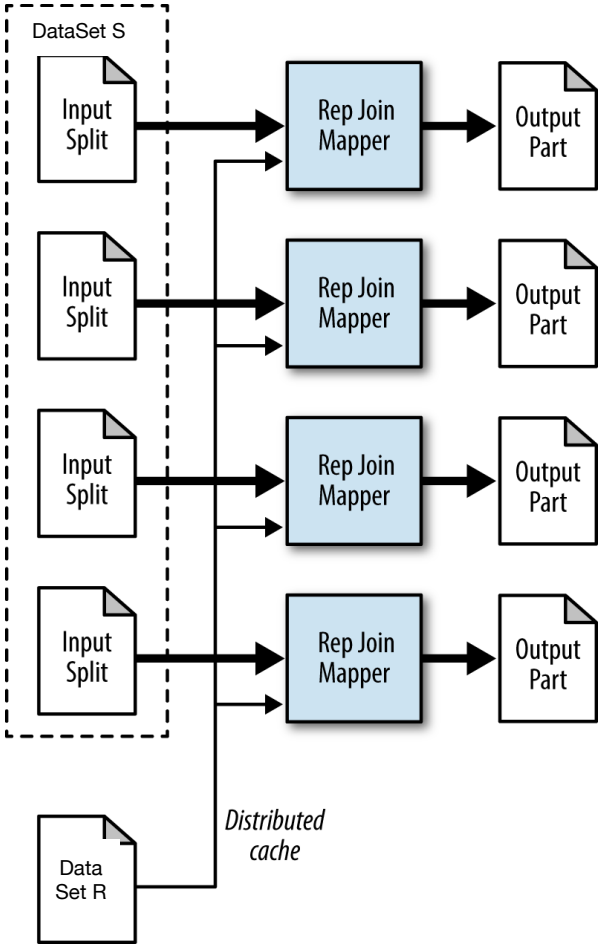
- Basic idea: load one dataset into memory, stream over other dataset
  - Works if *R* << *S* and *R* fits into memory
  - Called a “hash join” in database terminology
- MapReduce implementation
  - Distribute *R* to all nodes
  - Map over *S*, each mapper loads R in memory, hashed by join key
  - For every tuple in *S*, look up join key in *R*
  - No reducers, unless for regrouping or resorting tuples

In-Memory Join

- What if neither dataset fits in memory?
  - The simplest solution is to divide the smaller dataset, let's say S, into n partitions: $S = S_1 \cup S_2 \cup \ldots \cup S_n$. We can choose n so that each partition is small enough to fit in memory.
  - We can then run n memory-backed hash joins. This, of course, requires streaming through the other dataset n times.

In-Memory Join

- There is an alternative approach to memory-backed joins for cases where neither datasets fit into memory.
  - A distributed key-value store can be used to hold one dataset in memory across multiple machines while mapping over the other .
  - The mappers would then query this distributed key-value store in parallel and perform joins if the join keys match.
  - The open-source caching system memcached can be used for exactly this purpose, and therefore we've dubbed this approach memcached join.



Memcached Join

- Memcached join
  - Load R into memcached
  - Replace in-memory hash lookup with memcached lookup
- Capacity and scalability?
  - Memcached capacity >> RAM of individual node
  - Memcached scales out with cluster
- Latency?
  - Memcached is fast (basically, speed of network)
  - Batch requests to amortize latency costs
  - *For implementation see “Hadoop the definitive Guide”, Tom White, Distributed cache, P. 289.*

Which join to use?

- In-memory join > map-side join > reduce-side join
  - Why?
- Limitations of each?
  - In-memory join: memory
  - Map-side join: sort order and partitioning
  - Reduce-side join: general purpose

Processing Relational Data: Summary

- MapReduce algorithms for processing relational data
  - Group by, sorting, partitioning are handled automatically by shuffle/sort in MapReduce
  - Selection, projection, and other computations (e.g., aggregation), are performed either in mapper or reducer
  - Multiple strategies for relational joins
- Complex operations require multiple MapReduce jobs
  - Opportunities for automatic optimization

- Hadoop is great for large-data processing!
  - But writing Java programs for everything is verbose and slow
- Need for higher-level data processing languages
    - Hive: HQL is like SQL
  - Pig: Pig Latin is a bit like Perl

Exercice 1

- Take the relation R(A,B) and S(A,B) and the query :

```
SELCT a, max(b) as topb
FROM R
WHERE a>0
GROUP BY a
```

- Explain how the query would be executed in MapReduce. Make sure to specify the computation performed in the map and the reduce functions.

Exercice 2

- Take the relations R(a,b). ands the query :

```
SELCT R.A, max(S.B) as topb
FROM R,S
WHERE R.A = S.A AND R.B <20
GROUP BY R.A
```

- Explain how MapReduce can be used **most efficiently** to process this query. That is, explain how many MapReduce jobs are needed, and what the respective Map phase(s) and Reduce phase(s).