Outline

1 Motivation
2 Overview of MR Design Patterns
3 Summarization Patterns
4 Filtering Patterns
   - Filtering
   - Bloom Filtering
   - Top Ten
   - Distinct
5 Join Patterns
   - Reduce Side Join
   - Replicated Join
   - Composite Join
   - Cartesian Product
Outline

1 Motivation

2 Overview of MR Design Patterns

3 Summarization Patterns

4 Filtering Patterns
   - Filtering
   - Bloom Filtering
   - Top Ten
   - Distinct

5 Join Patterns
   - Reduce Side Join
   - Replicated Join
   - Composite Join
   - Cartesian Product
MapReduce Recap

- Programmers must specify:
  - **map**: \((k, v) \rightarrow (k', v')^*\)
  - **reduce**: \((k', v'[]) \rightarrow (v'')^*\)
    All values with the same key are reduced together

- Optionally, also:
  - **partition** \((k', \text{number of partitions}) \rightarrow \text{partition for } k'\)
    - Often a simple hash of the key, e.g., \(\text{hash}(k') \mod n\)
    - Divides up key space for parallel reduce operations.
  - **combine**: \((k', v'[]) \rightarrow (k', v'')^*\)
    - Mini-reducers that run in memory after the map phase
    - Used as an optimization to reduce network traffic

- The execution framework handles everything else
- But **what** should be done by these modules is not always easy
Task: Compute average income in each city in 2007

Input data (sorted by SSN)

SSTable 1

<table>
<thead>
<tr>
<th>SSN</th>
<th>Personal Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>123456</td>
<td>(John Smith; Sunnyvale, CA)</td>
</tr>
<tr>
<td>123457</td>
<td>(Jane Brown; Mountain View, CA)</td>
</tr>
<tr>
<td>123458</td>
<td>(Tom Little; Mountain View, CA)</td>
</tr>
</tbody>
</table>

SSTable 2

<table>
<thead>
<tr>
<th>SSN</th>
<th>year, income</th>
</tr>
</thead>
<tbody>
<tr>
<td>123456</td>
<td>(2007, $70000), (2006, $65000), (2005, $6000), ...</td>
</tr>
<tr>
<td>123457</td>
<td>(2007, $72000), (2006, $70000), (2005, $6000), ...</td>
</tr>
<tr>
<td>123458</td>
<td>(2007, $80000), (2006, $85000), (2005, $7500), ...</td>
</tr>
</tbody>
</table>

The two tables need to be “joined” (mimic join in MR)
Average Income Example/2

Mapper 1a:
Input: SSN → Personal Information
Output: (SSN, City)

Mapper 1b:
Input: SSN → Annual Incomes
Output: (SSN, 2007 Income)

Reducer 1:
Input: SSN → {City, 2007 Income}
Output: (SSN, [City, 2007 Income])

Mapper 2:
Input: SSN → [City, 2007 Income]
Output: (City, 2007 Income)

Reducer 2:
Input: City → 2007 Incomes
Output: (City, AVG(2007 Incomes))
Common Mistakes to Avoid/1

- Mapper and reducer should be **stateless**
- Don’t use static variables
- After `map` and `reduce` return, they should remember nothing about the processed data!
- Reason: No guarantees about which key-value pairs will be processed by which workers!

```java
HashMap h = new HashMap();
map(key, value) {
    if (h.contains(key)) {
        h.add(key, value);
    }
    emit(key, "Y");
}
```

Wrong!
Don’t try to do your own I/O!
Don’t try to read from, or write to, files in the file system
The MapReduce framework does all the I/O for you
  - All the incoming data will be fed as arguments to map and reduce.
  - Any data your functions produce should be output via emit.

```java
map(key, value) {
    File foo =
        new File("xyz.txt");
    while (true) {
        s = foo.readLine();
        ...
    }
}
```
Wrong!
Common Mistakes to Avoid/3

- Mapper must not map **too much data to the same key**
- In particular, don’t map everything to the same key!
- Otherwise the reduce worker will be overwhelmed.
- It’s okay if some reduce workers have more work than others.
- Example: In WordCount, the reduce worker that works on the key ’and’ has a lot more work than the reduce worker that works on ’syzygy’.

```java
map(key, value) {
    emit("FOO", key + " " + value);
}
```

Wrong!
Designing MapReduce Algorithms

- **Key decision:** What should be done by *map* and what by *reduce*?
  - **map**
    - Can do something to each individual key-value pair, but it cannot look at other key-value pairs
      - Example: Filtering out key-value pairs we don’t need
    - Can emit more than one intermediate key-value pair for each incoming key-value pair
      - Example: Incoming data is text, map produces (word, 1) for each word
    - Can emit data with specific keys to all reducers, e.g., EmitToAllReducers()
  - **reduce**
    - Can aggregate data
    - Can look at multiple values, as long as map has mapped them to the same (intermediate) key
      - Example: Count the number of words, add up the total cost, ...

- **Important to get the intermediate form right!**
- **Design pattern** help to develop algorithms.
Outline

1 Motivation

2 Overview of MR Design Patterns

3 Summarization Patterns

4 Filtering Patterns
   - Filtering
   - Bloom Filtering
   - Top Ten
   - Distinct

5 Join Patterns
   - Reduce Side Join
   - Replicated Join
   - Composite Join
   - Cartesian Product
What are Design Patterns?

- Reusable solutions to problems
- Domain independent
- Not a cookbook, but a guide
- Not a finished solution
- Makes the intent of code easier to understand
- Provides a common language for solutions
- Be able to reuse code
- Known performance profiles and limitations of solutions
Why MapReduce Design Patterns?

- Recurring patterns in data-related problem solving
- Groups are building patterns independently
- Lots of new users every day
- MapReduce is a new way of thinking
- Foundation for higher-level tools (Pig, Hive, ...)
- Community is reaching the right level of maturity
MapReduce Design Patterns

- Book by Donald Miner & Adam Shook
- Building effective algorithms and analytics for Hadoop and other systems.
- 23 pattern grouped into six categories
  - Summarization
  - Filtering
  - Data Organization
  - Joins
  - Metapatterns
  - Input and output
Pattern Categories/1

- **Filtering patterns**: Extract interesting subsets of the data
  - Filtering
  - Bloom filtering
  - Top ten
  - Distinct
- **Summarization patterns**: Top-down summaries to get a top-level view
  - Numerical summarizations
  - Inverted index
  - Counting with counters
- **Data organization patterns**: Reorganize and restructure data to work with other systems or to make MapReduce analysis easier
  - Structured to hierarchical
  - Partitioning
  - Binning
  - Total order sorting
  - Shuffling
Pattern Categories/2

- **Join patterns**: Bringing and analyze different data sets together to discover interesting relationships.
  - Reduce-side join
  - Replicated join
  - Composite join
  - Cartesian product

- **Metapatterns**: Piece together several patterns to solve a complex problem or to perform several analytics in the same job.
  - Job chaining
  - Chain folding
  - Job merging

- **Input and output patterns**: Custom the way to use Hadoop to input and output data.
  - Generating data
  - External source output
  - External source input
  - Partition pruning
Outline

1 Motivation

2 Overview of MR Design Patterns

3 Summarization Patterns

4 Filtering Patterns
   - Filtering
   - Bloom Filtering
   - Top Ten
   - Distinct

5 Join Patterns
   - Reduce Side Join
   - Replicated Join
   - Composite Join
   - Cartesian Product
**Numerical Summarizations**

- **Numerical Summarizations**
  - A general pattern for calculating *aggregate statistical values* over your data, e.g., minimum, maximum, average, median, and standard deviation.
  - Group records together by a key field and calculate a numerical aggregate per group to get a top-level view of a large data set.

- **Applications**
  - Word count, record count
  - Min, max, count of a particular event
  - Average, median, standard deviation

- **SQL resemblance**
  ```sql
  SELECT MIN(numericalcol1), MAX(numericalcol1), COUNT(*)
  FROM table
  GROUP BY groupcol2;
  ```
**Numerical Summarizations Structure**

- **Mapper**: outputs keys that consist of each field to group by, and values consisting of any pertinent numerical items.
- **Reducer**: receives a set of numerical values \((v_1, v_2, v_3, \ldots, v_n)\) associated with a group-by key and performs the (aggregate) function \(\theta(v_1, \ldots, v_n)\).
Performance Analysis

- Aggregations typically **perform well** when the **combiner** is properly used.
  - These types of operations are what MR was built for
- **Data skew** of reduce groups is problematic
  - many more intermediate key/value pairs with a specific key than other keys;
  - one reducer is going to have a lot more work to do than others.

<table>
<thead>
<tr>
<th>Input Key</th>
<th>Input Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>Minimum</td>
</tr>
<tr>
<td>12345</td>
<td>10</td>
</tr>
<tr>
<td>12345</td>
<td>8</td>
</tr>
<tr>
<td>12345</td>
<td>21</td>
</tr>
<tr>
<td>54321</td>
<td>1</td>
</tr>
<tr>
<td>54321</td>
<td>47</td>
</tr>
<tr>
<td>99999</td>
<td>7</td>
</tr>
<tr>
<td>99999</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output Key</th>
<th>Output Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Minimum</td>
</tr>
<tr>
<td>12345</td>
<td>8</td>
</tr>
<tr>
<td>54321</td>
<td>1</td>
</tr>
<tr>
<td>99999</td>
<td>7</td>
</tr>
<tr>
<td>99999</td>
<td>12</td>
</tr>
</tbody>
</table>
Given a list of user comments in a mailing list, determine the first and last time a user commented and the total number of comments from that user.

User comment

- <row Id="8189677" PostId="6881722" Text="Have you looked at Hadoop?" CreationDate="2011-07-30T07:29:33.343" UserId="831878"/>

After a grouping operation, the reducer has to iterate over all values associated with a group and to compute the aggregate functions.
Create Writable object MinMaxCountTuple to store the mapper output (instead of using a Text object)

```java
public class MinMaxCountTuple implements Writable {
    private Date min = new Date(), max = new Date();
    private long count = 0;

    public Date getMin() { return min; }
    public void setMin(Date min) { this.min = min; }
    public Date getMax() { return max; }
    public void setMax(Date max) { this.max = max; }
    public long getCount() { return count; }
    public void setCount(long count) { this.count = count; }

    public void readFields(DataInput in) throws IOException {
        // Read the data out in the order it is written
        min = new Date(in.readLong());
        max = new Date(in.readLong());
        count = in.readLong();
    }

    public void write(DataOutput out) throws IOException {
        // Write the data out in the order it is read
        out.writeLong(min.getTime());
        out.writeLong(max.getTime());
        out.writeLong(count);
    }
}
```
Summarization Patterns

Numerical Summarizations Example/3

```java
class MinMaxCountMapper extends Mapper<Object, Text, Text, MinMaxCountTuple> {
    private Text outUserId = new Text();
    private MinMaxCountTuple outTuple = new MinMaxCountTuple();

    public void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
        Map<String, String> parsed = transformXmlToMap(value.toString());
        String strDate = parsed.get("CreationDate");
        String userId = parsed.get("UserId");
        Date creationDate = frmt.parse(strDate);
        outTuple.setMin(creationDate);
        outTuple.setMax(creationDate);
        outTuple.setCount(1);
        outUserId.set(userId);
        context.write(outUserId, outTuple);
    }
}
```
public static class MinMaxCountReducer extends 
Reducer<Text, MinMaxCountTuple, Text, MinMaxCountTuple> {
    // Our output value Writable
    private MinMaxCountTuple result = new MinMaxCountTuple();

    public void reduce(Text key, Iterable<MinMaxCountTuple> values, Context context) {
        // Initialize result
        result.setMin(null);
        result.setMax(null);
        result.setCount(0);
        int sum = 0;

        // Iterate through all input values for this key
        for (MinMaxCountTuple val : values) {
            if (result.getMin() == null || val.getMin().compareTo(result.getMin()) < 0) {
                result.setMin(val.getMin());
            }
            if (result.getMax() == null || val.getMax().compareTo(result.getMax()) > 0) {
                result.setMax(val.getMax());
            }
            sum += val.getCount();
        }
        result.setCount(sum);
        context.write(key, result);
    }
}
Outline

1 Motivation

2 Overview of MR Design Patterns

3 Summarization Patterns

4 Filtering Patterns
   - Filtering
   - Bloom Filtering
   - Top Ten
   - Distinct

5 Join Patterns
   - Reduce Side Join
   - Replicated Join
   - Composite Join
   - Cartesian Product
Filtering Patterns

- To understand a **smaller piece/subset** of data
  - e.g., a top-ten listing, the results of a de-duplication
- **Sampling** as a special form of filtering
  - Get a small representative sample of a data set.
- Filtering does **not change** the actual records
- Filtering Patterns:
  - Filtering
  - Bloom Filtering
  - Top Ten
  - Distinct
Filtering is the most basic pattern and evaluates each record separately based on some condition.

Intent
- Filter out records that are not of interest.
- You want to focus your analysis on a subset of a large data set.

Applications
- Closer view of data
- Tracking a thread of events
- Distributed grep
- Data cleansing
- Removing low scoring data (if you can score your data)

SQL resemblance
- SELECT * FROM table WHERE value < x
Filtering Patterns

Filtering Structure

- No reducer needed, i.e., no further processing/aggregation of the data
- `map(key, record) {  
  if we want to keep record then  
  emit key, value  
}

![Filtering Structure Diagram]
Performance Analysis

- No reducers
  - Both the sort phase and the reduce phase are cut out
  - Data never has to be transmitted between the map and reduce phase.
- With one single reducer, all data would be collected into a single file.
- Most of the map tasks pull data off of their locally attached disks and then write back out to that node.
Random Sampling Example

- Task: Grab a random subset of a dataset
- Random number generator produces a number: if the value is below a threshold, keep the record, otherwise skip it
- Hadoop provides a **setup** method that is called once for each mapper prior to the many calls to map.

```java
public static class SRS Mapper extends Mapper<Object, Text, NullWritable, Text> {
    private Random rands = new Random();
    private Double percentage;

    protected void setup(Context context) throws ...
    {
        // Retrieve the percentage that is passed in via the configuration
        // like this: conf.set("filter_percentage", .5) for .5%
        String strPercentage = context.getConfiguration().get("filter_percentage");
        percentage = Double.parseDouble(strPercentage) / 100.0;
    }

    public void map(Object key, Text value, Context context) throws ...
    {
        if (rands.nextDouble() < percentage)
        {
            context.write(NullWritable.get(), value);
        }
    }
}
```
Bloom Filtering

- **Intent**
  - Keep records that are member of a predefined set of hot values
  - Some false positives are acceptable, i.e., some records will get through the filter although they are not in the hot values

- **Applications**
  - Removing most of the non-watched values
  - Prefiltering a data set for an expensive set membership check
Bloom Filtering

- A Bloom filter (B.H. Bloom, 1970) is a space-efficient probabilistic data structure that is used to test set membership.
- Filter returns either possibly in set or definitely not in set
  - i.e., false positive matches are possible, but not false negatives
- The filter is represented by a bit vector of size $m$ and $k$ different hash functions that map each element (hot value) of the set to one of the $m$ bits.
- Training phase: for all elements in the set, the $k$ hash functions are computed and the corresponding bits are set to 1.
- Check membership: compute the $k$ hash functions for the element
  - if all hash functions map to 1 $\rightarrow$ possibly in the set
  - if at least one hash function maps to 0 $\rightarrow$ not in the set

\[
\begin{array}{cccccccccccc}
0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0
\end{array}
\]

- $M = 18$, $k = 3$
- $w$ is not in the set $\{x, y, z\}$
Bloom Filtering Structure

- **Structure**
  - The Bloom filter is first trained and stored in the HDFS.
  - The mapper then calls the setup method to load the Bloom filter before processing the input data.
  - The `DistributedCache` is a Hadoop utility that ensures that a file in the HDFS is present on the local file system of each task that requires it.
Performance Analysis

- Loading up the Bloom filter is not that expensive since the file is relatively small.
- Checking a value against the Bloom filter is also a relatively cheap operation by $O(1)$ hashing.
**Bloom Filtering Example**

- **Task:** Given a list of user comments, filter out a majority of the comments that do not contain any of a set of predefined keywords.

```java
public static class BloomFilteringMapper extends Mapper<Object, Text, Text, NullWritable> {
    private BloomFilter filter = new BloomFilter();

    protected void setup(Context context) throws ... {
        // Get Bloom filter from the DistributedCache
        URI[] files = DistributedCache.getCacheFiles(context.getConfiguration());
        DataInputStream strm = new DataInputStream(new FileInputStream( files[0].getPath()));
        filter.readFields(strm);
        strm.close();
    }

    public void map(Object key, Text value, Context context) throws ... {
        Map<String, String> parsed = transformXmlToMap(value.toString());
        // Get the value for the comment
        String comment = parsed.get("Text");
        StringTokenizer tokenizer = new StringTokenizer(comment);
        // For each word: if the word is in the filter, output the record and break
        while (tokenizer.hasMoreTokens()) {
            String word = tokenizer.nextToken();
            if (filter.membershipTest(new Key(word.getBytes()))) {
                context.write(value, NullWritable.get());
                break;
            }
        }
    }
}
```
Top Ten

- **Intent**
  - Retrieve a small number of top $K$ records, according to some ranking/criterion
  - The number of output records should be significantly lower than the number of input records.
  - It must be possible to determine an ranking.

- **Applications**
  - Outlier analysis
  - Select interesting data (most valuable data)

- **SQL Resemblance**
  - SQL: SELECT * FROM table WHERE col4 DESC LIMIT 10
**Top Ten Structure**

- **Mapper**: find local top $K$
- **(only one) Reducer**: $K \cdot M$ records $\rightarrow$ the final top $K$

![Diagram of Top Ten Structure](image-url)
Performance Analysis with one Reducer

- Reducer gets $K \times M$ records
- The sort can become expensive if the reducer gets too many records and sorting needs to be done on local disk instead of in memory
- The reducer host will receive a lot of data over the network $\Rightarrow$ might create a network resource hot spot
- Scanning through all the data in the reduce will take a long time if there are many records to look through.
- Writes to the output file are not parallelized
Top Ten Example

- Hadoop provides a cleanup method that is called once after all key/value pairs have been through map (just like setup which is called before)

class mapper:
    setup():
        initialize top ten sorted list

    map(key, record):
        insert record into top ten sorted list
        if length of array > 10 then
            truncate list to a length of 10

    cleanup():
        for record in top sorted ten list:
            emit null, record

class reducer:
    setup():
        initialize top ten sorted list

    reduce(key, records):
        sort records
        truncate records to top 10
        for record in records:
            emit record
Distinct

- **Intent**
  - Find a unique set of values from similar records with potential duplicates

- **Applications**
  - Deduplicate data
  - Getting distinct values
  - Protecting from an inner join explosion

- **SQL Resemblance**
  - SQL: `SELECT DISTINCT * FROM table;`
Distinct Structure

- Exploits MapReduce’s ability to group keys together to remove duplicates
- The mapper outputs the input value as intermediate key
- Reducer groups all duplicates together and simply outputs the key
- Duplicate records are often located close to each other in a data set, so a combiner will deduplicate most of them in the map phase

```
map(key, record):
    emit (record, null)

reduce(key, records):
    emit (key)
```
Performance Analysis

- Finding the **right number of reducers** is crucial
- If duplicates are very rare within an input split, almost all of the data is sent to the reduce phase, hence use many reducers
- If there are many duplicates, many reducers might produce very small output files, and therefore unnecessary overhead
Outline

1 Motivation

2 Overview of MR Design Patterns

3 Summarization Patterns

4 Filtering Patterns
   - Filtering
   - Bloom Filtering
   - Top Ten
   - Distinct

5 Join Patterns
   - Reduce Side Join
   - Replicated Join
   - Composite Join
   - Cartesian Product
An SQL query walks into a bar, sees two tables and asks them “May I join you?”

- Joins are very important in RDBMS, but among the most complex operations in MapReduce
  - MR is good in processing datasets by looking at each record in isolation
  - Joining/combining datasets does not fit gracefully into the MR paradigm
- Refresh of RDMS equality joins
  - Inner Join
  - Outer Join
  - Cartesian Product
  - Anti Join = full outer join – inner join
- Join patterns in MR
  - Reduce Side Join
  - Replicated Join
  - Composite Join
  - Cartesian Product
Reduce Side Join

- **Reduce Side Join**: Reducer executes the actual join
  - Join large multiple data sets by a **foreign key**
  - **Simple** to implement in Reducers
  - Supports all different join operations
  - **No limitation** on the size of the data sets

- **SQL resemblance**

```sql
SELECT users.ID, users.Location, comments.upVotes
FROM users [INNER|LEFT|RIGHT] JOIN comments
ON users.ID = comments.UserID
```
Reduce Side Join Structure

- Mapper prepares \((key, record)\)
  - key is the join attribute, the (data) record is flagged with ID of data set
- Reducer performs join operation on identical keys
  - Creates for each key a list for each data set and joins them

```
Data Set A
Input Split → Join Mapper → Shuffle and Sort → Join Reducer → Output Part

Data Set B
Input Split → Join Mapper → Shuffle and Sort → Join Reducer → Output Part
```

(map, record)
Performance Analysis

- Cluster’s **network bandwidth** is bottleneck!!!
  - Pretty much all of the data is sent to the shuffle and sort step
- Utilize relatively **more reducers** than for other analytic tasks
Reduce Side Join Example/1

- Task: Enrich comments with user information
  - Table A contains user information, table B contains user comments
  - Connected by user ID

- Mapper
  - The UserJoinMapper adds “A” in front of each value/record
    - Thus, the reducer knows from which relation the value comes
  - Similar, the CommentJoinMapper prepends “B”

```java
public static class UserJoinMapper extends Mapper<Object, Text, Text, Text> {
    private Text outkey = new Text();
    private Text outvalue = new Text();

    public void map(Object key, Text value, Context context) throws ...{
        // Parse the input string and extract the user ID
        String userId = value.toString().get("Id");

        // The foreign join key is the user ID
        outkey.set(userId);

        // Flag this record for the reducer and then output
        outvalue.set("A" + value.toString());
        context.write(outkey, outvalue);
    }
}
```
Reduce Side Join Example/2

- R educer iterates through all values of each group and separates the values
- Join logic is applied then on these lists (and differs depending on the join)

```java
public static class UserJoinReducer extends Reducer<Text, Text, Text, Text> {
    private ArrayList<Text> listA = new ArrayList<Text>();
    private ArrayList<Text> listB = new ArrayList<Text>();
    private String joinType = null;

    public void setup(Context context) {
        joinType = context.getConfiguration().get("join.type");
    }

    public void reduce(Text key, Iterable<Text> values, Context context) throws ...{
        listA.clear(); listB.clear();
        // Iterate through all values and separate them into an A-list and B-list
        while (values.hasNext()) {
            tmp = values.next();
            if (tmp.charAt(0) == 'A') {
                listA.add(new Text(tmp.toString().substring(1)));
            } else if (tmp.charAt('0') == 'B') {
                listB.add(new Text(tmp.toString().substring(1)));
            }
        }
        // Execute our join logic now that the lists are filled
        executeJoinLogic(context);
    }
}...
```
executeJoinLogic computes the actual join as part of the reducer task

Inner join: if both lists are not empty, simply perform two nested loops and join each of the values.

```java
private void executeJoinLogic(Context context) throws ...
{
...
    if (joinType.equalsIgnoreCase("inner")) {
        // If both lists are not empty, join A with B
        if (!listA.isEmpty() && !listB.isEmpty()) {
            for (Text A : listA) {
                for (Text B : listB) {
                    context.write(A, B);
                }
            }
        }
    }
...
```
Reduce Side Join Example/4

- **Left outer join:**
  - if the right list is not empty, join $A$ with $B$;
  - otherwise, output each record of $A$ with an empty string.
- **Right outer join** is similar.

... else if (joinType.equalsIgnoreCase("leftouter")) {
    // For each entry in A
    for (Text A : listA) {
      if (!listB.isEmpty()) {
        // Join A and B
        for (Text B : listB) {
          context.write(A, B);
        }
      } else {
        // Output A with empty string
        context.write(A, "");
      }
    }
  }
...
Full outer join: all records need to be kept
- if list A is not empty, for every element in A:
  - join with B if B is not empty;
  - otherwise, output A;
- if A is empty, just output B.

```java
else if (joinType.equalsIgnoreCase("fullouter")) {
    if (!listA.isEmpty()) {
        for (Text A : listA) {
            if (!listB.isEmpty()) {
                // Join A with B
                for (Text B : listB) {
                    context.write(A, B);
                }
            } else {
                // Output A with empty string
                context.write(A, "");
            }
        }
    } else {
        // list A is empty: just output B
        for (Text B : listB) {
            context.write("", B);
        }
    }
}
```
Anti join: if exactly one of the lists is empty, output the records from the non-empty list with an empty text.

Recall that the anti-join contains only those tuples from both relations that do not have a match in the other relation.

else if (joinType.equalsIgnoreCase("anti")) {
    // If list A is empty and B is not empty or vice versa
    if (listA.isEmpty() XOR listB.isEmpty()) {
        // Iterate over both A and B
        // The previous XOR check will make sure exactly one of
        // these lists is empty and therefore the list will be skipped
        for (Text A : listA) {
            context.write(A, EMPTY_TEXT);
        }
        for (Text B : listB) {
            context.write(EMPTY_TEXT, B);
        }
    }
}

...
Replicated Join

- **Replicated Join**: Mapper implements the actual join, **no** reducer is used.
- All data sets, except a large one, are read into **memory** during the setup phase of each map task.
- The large data set is the “left” part of the join.
Replicated Join Structure

- Map-only pattern, i.e., no combiner, partitioner or reducer is used.
- Read all files from the distributed cache during the setup of the mapper method and store them into in-memory lookup tables.
- Mapper processes each record and joins it with all the data stored in memory.
Performance Analysis

- Eliminates the need to shuffle any data to the reduce phase.
- A replicated join can be the **fastest type of join** because no reducer is required.
- Limited by the amount of data that can be stored safely inside JVM.
Replicated Join Example/1

- Task: Enrich comments (large relation) with user information (small relation)

```java
public static class ReplicatedJoinMapper extends Mapper<Object, Text, Text, Text> {
    private HashMap<String, String> userIdToInfo = new HashMap<String, String>();
    private Text outvalue = new Text();
    private String joinType = null;

    public void setup(Context context) throws ...
    {
        Path[] files = DistributedCache.getLocalCacheFiles(context.getConfiguration());
        // Read all files in the DistributedCache
        for (Path p : files) {
            BufferedReader rdr = new BufferedReader(... new File(p.toString())...);
            String line = null;
            while ((line = rdr.readLine()) != null) {
                // Get the user ID for this record
                String userId = line.get("Id");
                // Map the user ID to the record
                userIdToInfo.put(userId, line);
            }
        }
        // Get the join type from the configuration
        joinType = context.getConfiguration().get("join.type");
    }
}
```

public void map(Object key, Text value, Context context) throws ...
{
    String userId = value.toString().get("UserId");
    String userInformation = userIdToInfo.get(userId);
    if (userInformation != null) {
        // If the user information is not null, then output
        outvalue.set(userInformation);
        context.write(value, outvalue);
    } else if (joinType.equalsIgnoreCase("leftouter")) {
        // For a left outer join output the record with an empty value
        context.write(value, "");
    }
}"
Composite Join

- **Composite join** is performed on the map-side with many very large inputs.
- Completely eliminates the need to shuffle and sort all the data to the reduce phase.
- Data need to be already organized or prepared in a very specific way:
  - Sorted by foreign key, partitioned by foreign key, and read in a very particular manner.
- Particularly useful if you want to join very large data sets together.
Composite Join Applicability

- All data sets can be read with the foreign key as the input key to the mapper.
- All data sets have the same number of partitions.
- Each partition is sorted by foreign key, and all the foreign keys reside in the associated partition of each data set.
- The data sets do not change often (if they have to be prepared).
Composite Join Structure

- Map-only
- Mapper is very trivial.
- Two values are retrieved from the input tuple and output to file system, e.g., (key, value1, value2)
- Most of the work is done by the driver code `CompositeInputFormat`
  - parses all the input files and outputs records to the mapper.
Can be executed relatively quickly over large data sets.

- Data Preparation = sorting cost
- The cost of preparing the data is averaged out over all of the runs.
Join Patterns

Cartesian Product

- **Intent**
  - Pair up and compare *every single record with every other record* in one or more data sets
  - A Cartesian product does not fit nicely into the MapReduce paradigm
    - The operation is not intuitively splittable and cannot be parallelized very well
- **Applications**
  - You want to analyze relationships between all pairs of individual records.
- **SQL Resemblance**
  - `SELECT * FROM tableA, tableB;`
Map-only
Essentially a RecordReader job
Cross product of input splits is determined during job setup.
Each record reader is responsible for generating the cross product of records from both input splits.
Cartesian Product Performance Analysis

- A massive explosion in data size $O(n^2)$
- If a single input split contains a thousand records → the right input split needs to be read a thousand times before the task can finish.
- If a single task fails for an odd reason, the whole thing needs to be restarted.
Summary

- MapReduce requires a **new way of thinking** and problem solving.
- Common pitfalls:
  - Mappers and reducers should be **stateless**.
  - Avoid your own IO and too much data to the same key.
- **Design patterns** are helpful for designing MapReduce algorithms.
  - Provide templates for solving common data manipulation problems.
  - Different categories of patterns.
- **Filtering patterns** are used to extract a small subset of the data.
  - Filter analyse each record **individually**, and data is not modified.
  - **Sampling** as a special form of filtering
  - Different filtering patterns: Filtering, Bloom filtering, Top Ten, Distinct.
- **Numerical summarization** patterns for calculating aggregate values.
- **Join patterns** combine data from different sources
  - Among the most complex patterns in MR
  - Combining data does not fit gracefully into the MR paradigm (which considers tuples individually)
  - Different join patterns: Reduce Side Join, Replicated Join, Composite Join, Cartesian Product