Big Data Processing, 2014/15

Lecture 7: MapReduce design patterns

Claudia Hauff (Web Information Systems)
ti2736b-ewi@tudelft.nl

Course content

- Introduction
- Data streams 1 & 2
- The MapReduce paradigm
- Looking behind the scenes of MapReduce: HDFS & Scheduling
- **Algorithm design for MapReduce**
- A high-level language for MapReduce: Pig 1 & 2
- MapReduce is not a database, but HBase nearly is
- Lets iterate a bit: Graph algorithms & Giraph
- How does all of this work together? ZooKeeper/Yarn

Learning objectives

- **Implement** the four introduced design patterns and choose the correct one according to the usage scenario

- **Express** common database operations as MapReduce jobs and argue about the pros & cons of the implementations
  - Joins
  - Union, Selection, Projection, Intersection (covered in the assignment)

MapReduce design patterns

Design patterns

“Arrangement of components and specific techniques designed to handle frequently encountered situations across a variety of problem domains.”

- Programmer’s tasks (Hadoop does the rest):
  - Prepare data
  - **Write mapper code**
  - **Write reducer code**
  - Write combiner code
  - Write partitioner code

But: every task needs to be converted into the Mapper/ Reducer schema

Design patterns

- In parallel/distributed systems, synchronisation of intermediate results is difficult
- MapReduce paradigm offers **one opportunity** for cluster-wide synchronisation: shuffle & sort phase
- Programmers have little control over:
    - **Where** a Mapper/Reducer runs
    - **When** a Mapper/Reducer starts & ends
    - **Which** key/value pairs are processed by which Mapper or Reducer

Controlling the data flow

- **Complex data structures** as keys and values
- Execution of user-specific **initialisation/ termination** code at each Mapper/Reducer
- **State preservation** in Mapper/Reducer across multiple input or intermediate keys (Java objects)
- User-controlled **partitioning of the key space** and thus the set of keys that a particular Reducer encounters

Local aggregation

Local aggregation

- Moving data from Mappers to Reducers
  - **Data transfer** over the network
  - **Local disk writes**

- Local aggregation: reduces amount of **intermediate data** & increases algorithm efficiency

- Exploited concepts:
  - Combiners
  - State preservation

Most popular: in-mapper combining

Our WordCount example

Question 1: what is the number of emitted key/value pairs?
Question 2: to what extent can that number be reduced?

```
map(docid a, doc d):
    foreach term t in doc:
        EmitIntermediate(t, count 1);

reduce(term t, counts[c1, c2, ..])
    sum = 0;
    foreach count c in counts:
        sum += c;
    Emit(term t, count sum)
```

Local aggregation on two levels

```
map(docid a, doc d):
    H = associative_array;
    foreach term t in doc:
        H{t}=H{t}+1;
    foreach term t in H:
        EmitIntermediate(t,count H{t});
```

Question 1: are we running into memory problems here?
Question 2: for which type of documents does this help the most?

Local aggregation on two levels

```
setup():
    H = associative_array;

map(docid a, doc d):
    foreach term t in doc:
        H{t}=H{t}+1;

clean():
    foreach term t in H:
        EmitIntermediate(t,count H{t});
```

Question 1: are we running into memory problems here?
Question 2: what happens if we run a Combiner now?

Local aggregation: pros and cons (vs. Combiners)

- **Advantages**
  - **Controllable** when aggregation occurs and how it takes place
  - **More efficient** (no disk spills, no object creation/destruction overhead)

- Disadvantages
  - **Breaks functional programming paradigm** (state preservation between `map()` calls)
  - Algorithmic behaviour might depend on the **order** of `map()` input key/values (hard to debug)
  - **Scalability bottleneck** (extra programming effort to avoid it)

Local aggregation: always useful?

- **Example**: input is a list of towns/cities across the world and their size (number of citizens)

- Efficiency improvements **dependent** on
  - **Size of intermediate key space**
  - Distribution of keys
  - Number of key/value pairs emitted by individual map tasks

- WordCount
  - Scalability limited by vocabulary size?
  - A problem? (Heap's law)

Correctness of local aggregation: the mean

Identity Mapper

```
map(string t, int r):
    EmitIntermediate(string t, int r)

reduce(string t, ints [r1, r2, r3, ..])
    sum = 0;
    count = 0;
    foreach int r in ints:
        sum += r;
        count++;
    avg=sum/count;
    Emit(string t, int avg);
```

mean != mean of means

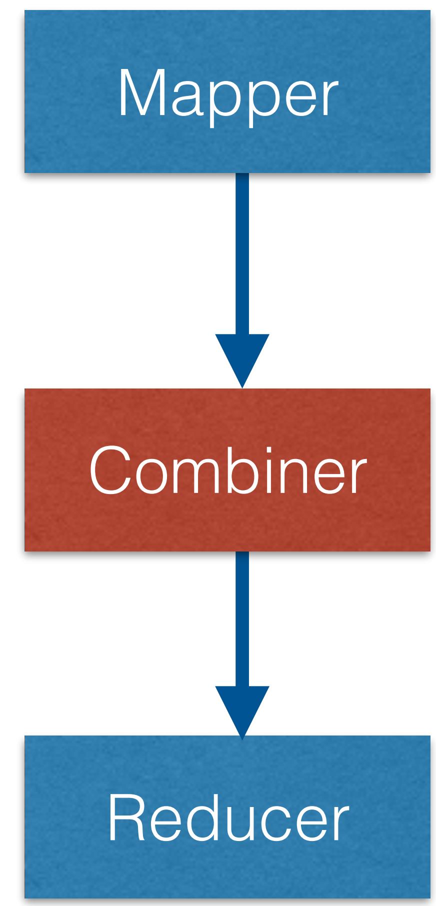
Correctness of local aggregation: the mean

```
map(string t, int r):
     EmitIntermediate(string t, int r)

combine(string t, ints [r1, r2, ..])
     sum = 0; count = 0;
     foreach int r in ints:
          sum += r;
          count += 1;
     EmitIntermediate(string t, pair(sum,count))

reduce(string t, pairs [(s1,c1), (s2,c2),..])
     sum = 0; count = 0;
     foreach pair (s,c) in pairs:
          sum += s;
          count += c;
     avg=sum/count;
     Emit(string t, int avg);
```

mapper output grouped by key



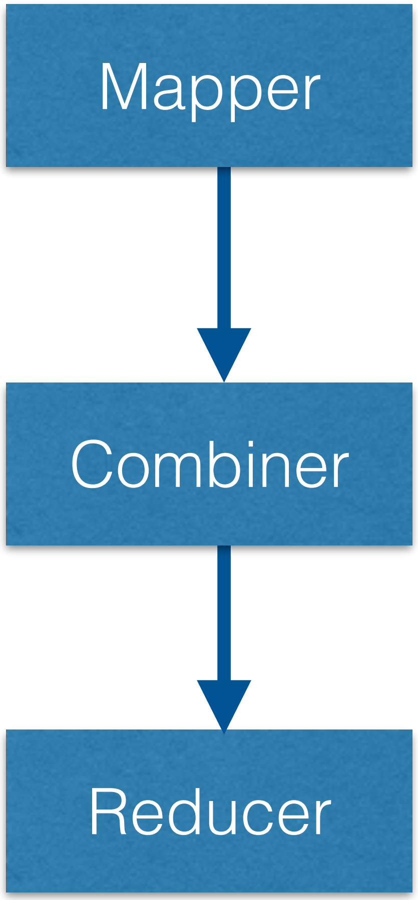
incorrect

Correctness of local aggregation: the mean

```
map(string t, int r):
     EmitIntermediate(string t, pair (r,1))

combine(string t, pairs [(s1,c1), (s2,c2),..])
     sum = 0; count = 0;
     foreach int r in ints:
          sum += r;
          count += 1;
     EmitIntermediate(string t, pair(sum,count))

reduce(string t, pairs [(s1,c1), (s2,c2),..])
     sum = 0; count = 0;
     foreach pair (s,c) in pairs:
          sum += s;
          count += c;
     avg=sum/count;
     Emit(string t, int avg);
```



correct

Correctness of local aggregation: the mean

- Reducer input key/value type must match mapper key/value type
- Combiner input & output type must match mapper output key/value type
- Combiners are optimisations, they must not change the correctness of the program

In-mapper combiner pattern: calculating the means

Best option for data-aware programs!

```
setup():
    S = associative_array;
    C = associative_array;

    map(string t, int r):
    S{t} = S{t} + r;
    C{t} = C{t} + 1;

cleanup():
    foreach string t in S:
        EmitIntermediate(string t,
                                    pair(S{t},C{t}));
```

Pairs & Stripes

To motivate the next design pattern .. co-occurrence matrices

Corpus: 3 documents
Delft is a city.
Amsterdam **is a** city.
Hamlet **is a** dog.

Co-occurrence matrix
(on the document level)

Applications:
clustering, retrieval, stemming, text mining, …

stripe

	delft	is	a	city	the	dog	amsterdam	hamlet
delft	**X**	**1**	**1**	**1**	0	0	0	0
is		X	**3**	2	0	1	1	1
a			X	2	0	1	1	1
city				X	1	1	2	0
the					X	1	1	0
dog						X	0	1
amsterdam							X	0
hamlet								X

pair

- Square matrix of size nxn (n: vocabulary size)
- Unit can be document, sentence, paragraph, …

To motivate the next design pattern .. co-occurrence matrices

Corpus: 3 documents
Delft is a city.
Amsterdam **is a** city.
Hamlet **is a** dog.

Co-occurrence matrix
(on the document level)

Applications:
clustering, retrieval,
stemming, text mining, …

stripe

	delft	is	a	city	the	dog	amsterdam	hamlet
delft	**X**	**1**	**1**	**1**	0	0	0	0
is		X	3	2	0	1	1	1
a			X	2	0	1	1	1
city				X	1	1	2	0
the					X	1	1	0
dog						X	0	1
amsterdam							X	0
hamlet								X

pair

More general: estimating distributions of discrete joint events from a large number of observations.
Not just NLP/IR: think sales analyses (*people who buy X also buy Y)*

Pairs

each pair is one cell in the matrix (complex key)

emit co-occurrence count

```
map(docid a, doc d):
      foreach term w in d:
         foreach term u in d:
            EmitIntermediate(pair(w,u),1)

reduce(pair p, counts [c1, c2, ..]
   s = 0;
   foreach c in counts:
      s += c;
   Emit(pair p, count s);
```

sum co-occurrence count

Question 1: which approach benefits more from a Combiner?
Question 2: which approach generates more intermediate key/value pairs?
Question 3: Which approach scales seamlessly?

emit terms co-occurring with term w

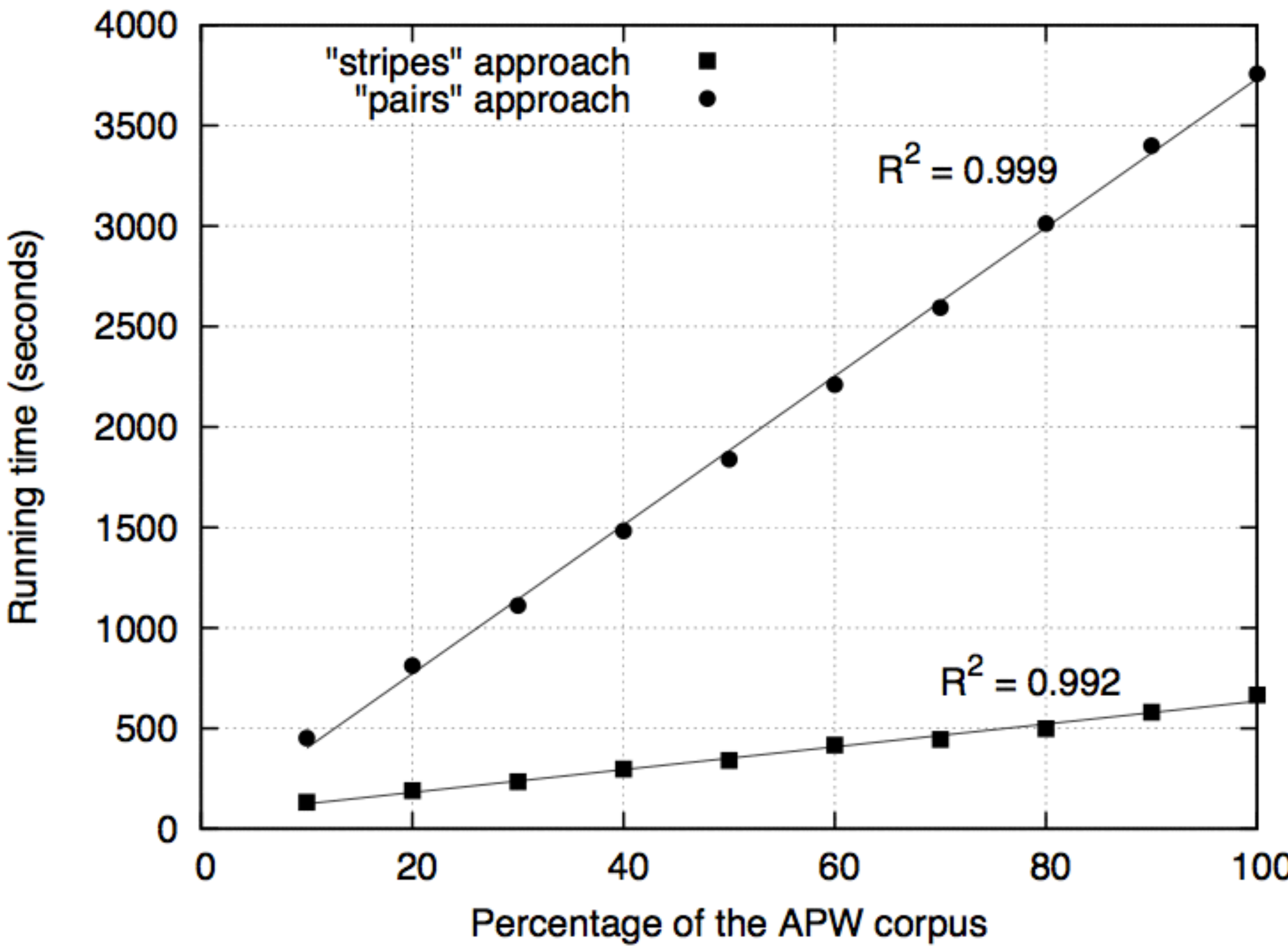
```
map(docid a, doc d):
        foreach term w in d:
            H = associative_array;
            foreach term u in d:
                H{u}=H{u}+1;
            EmitIntermediate(term w, Stripe H);

reduce(term w, stripes [H1,H2,..])
    F = associative_array;
    foreach H in stripes:
        sum(F,H);
    Emit(term w, stripe F)
```

one row in the co-occurrence matrix

Pairs & Stripes

2.3M documents
Co-occurrence window: 2
19 nodes in the cluster



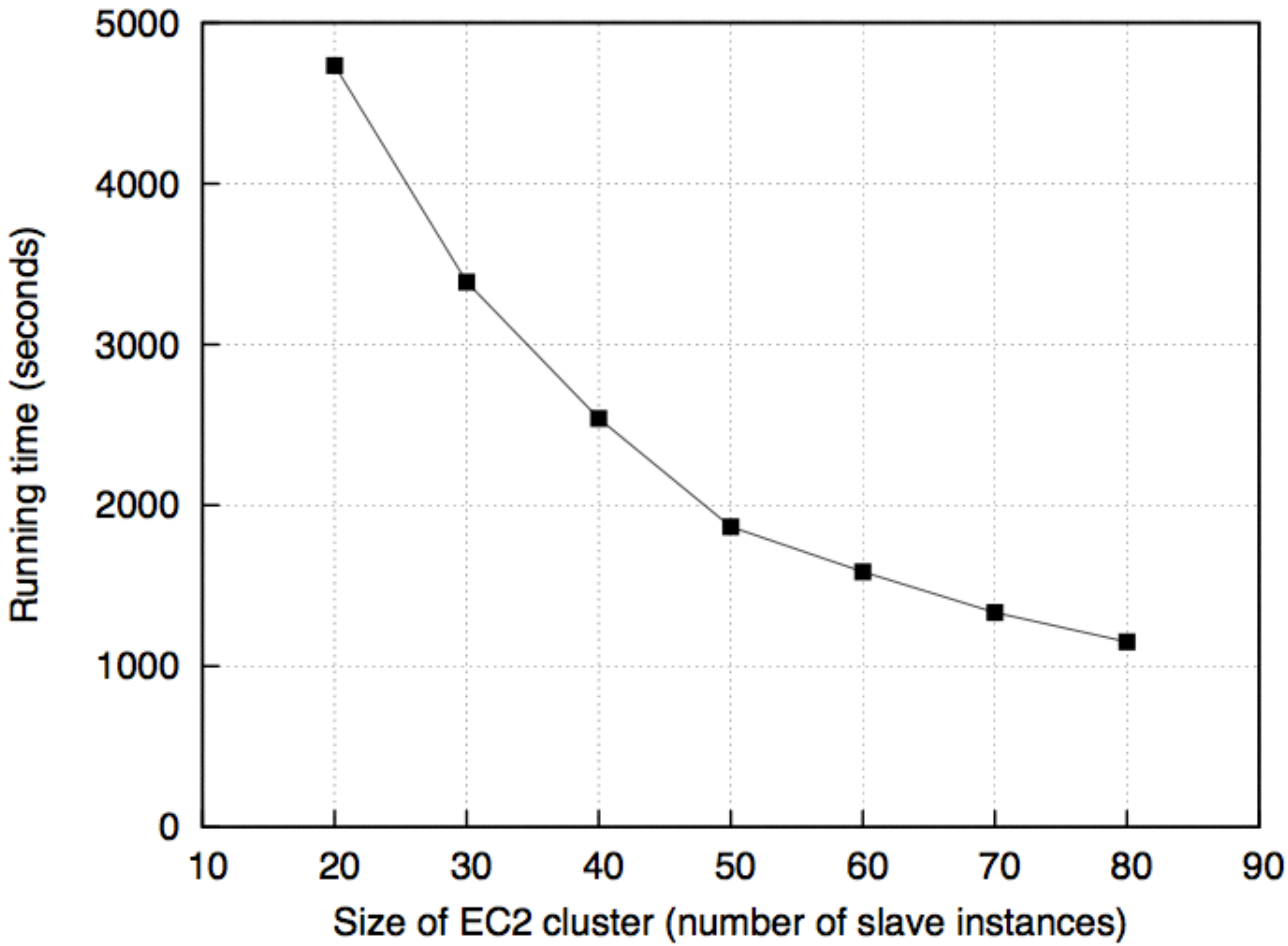
1 hour
2.6 billion intermediate key/value pairs;
after combiner run: 1.1B
final key/value pairs: 142M

11 minutes
653M intermediate key/value pairs;
after combiner run: 29M
final: 1.69M rows

Source: Jimmy Lin's MapReduce Design Patterns book

Stripes

2.3M documents
Co-occurrence window: 2
Scaling up (on EC2)



Source: Jimmy Lin's MapReduce Design Patterns book

Pairs & Stripes: two ends of a spectrum

Pairs

each co-occurring event is recorded

Stripes

all co-occurring events wrt. the conditioning event are recorded

Middle ground: divide key space into buckets and treat each as a “sub-stripe”.
If one bucket in total: stripes, if buckets=vocab.: pairs

Order inversion

From absolute counts to relative frequencies

number of times
a co-occurring
word pair is observed

$$f(w_j \mid w_i) = \frac{\overbrace{N(w_i, w_j)}}{\sum_{w'} N(w_i, w')}$$

The marginal

$$f(\mathrm{a} \mid \mathrm{is}) = \frac{3}{3+2+4\times 1} = \frac{1}{3}$$

$$f(\mathrm{city} \mid \mathrm{amsterdam}) = \frac{2}{2+3\times 1} = \frac{2}{5}$$

$$f(\mathrm{city} \mid \mathrm{is}) = \frac{2}{3+2+4\times 1} = \frac{2}{9}$$

Corpus: 3 documents

Delft is a city.
Amsterdam is a city.
Hamlet is a dog.

Co-occurrence matrix
(on the document level)

	delft	is	a	city	the	dog	amsterdam	hamlet
delft	X	1	1	1	0	0	0	0
is		X	3	2	0	1	1	1
a			X	2	0	1	1	1
city				X	1	1	2	0
the					X	1	1	0
dog						X	0	1
amsterdam							X	0
hamlet								X

From absolute counts to relative frequencies: stripes

$$f(w_j \mid \boxed{w_i}) = \frac{\overbrace{N(w_i, w_j)}}{\sum_{w'} \boxed{N(w_i, w')}}$$

Marginal can be computed easily in one job.
Second Hadoop job to compute the relative frequencies.

```
map(docid a, doc d):
        foreach term w in d:
                H = associative_array;
                foreach term u in d:
                        H{u}=H{u}+1;
                EmitIntermediate(term w, Stripe H);

reduce(term w, stripes [H1,H2,..])
        F = associative_array;
        foreach H in stripes:
                sum(F,H);
        Emit(term w, stripe F)
```

From absolute counts to relative frequencies: pairs

$$f(w_j \mid w_i) = \frac{N(w_i, w_j)}{\sum_{w'} N(w_i, w')}$$

2 options to make Pairs work:

- build in-memory associative array; but advantage of pairs approach (no memory bottleneck) is removed
- properly sequence the data: (1) compute marginal, (2) for each joint count, compute relative frequency

```
map(docid a, doc d):
        foreach term w in d:
                foreach term u in d:
                        EmitIntermediate(pair(w,u),1)

reduce(pair p, counts [c1, c2, ..])
        s = 0;
        foreach c in counts:
                s += c;
        Emi(pair p, count s);
```

From absolute counts to relative frequencies: pairs

$$f(w_j \mid w_i) = \frac{N(w_i, w_j)}{\sum_{w'} N(w_i, w')}$$

```
map(docid a, doc d):
        foreach term w in d:
                foreach term u in d:
                        EmitIntermediate(pair(w,u),1)
                        EmitIntermediate(pair(w,*),1)

reduce(pair p, counts [c1, c2, ..])
        s = 0;
        foreach c in counts:
                s += c;
        if(p.right==*)
                marginal=s;//keep marginal across reduce calls
        else
                Emit(pair p, s/marginal);
```

extra key/value pair for marginal

assumes a specific key ordering (* before the rest)

From absolute counts to relative frequencies: pairs

$$f(w_j \mid w_i) = \frac{N(w_i, w_j)}{\sum_{w'} N(w_i, w')}$$

```
map(docid a, doc d):
        foreach term w in d:
                foreach term u in d
                        EmitInterme
                        EmitInterme

reduce(pair p, counts [c1, c2, ..])
        s = 0;
        foreach c in counts:
                s += c;
        if(p.right==*)
                marginal=s;//keep ma
        else
                Emit(pair p, s/margi
```

Properly sequence the data:

- Custom **partitioner**: partition based on left part of pair
- Custom **key sorting**: * before anything else
- Combiner usage (or in-memory mapper) required for (w,*)
- Preserving state of marginal

extra key/value pair for marginal

assumes a specific key ordering (* before the rest)

Design pattern: order inversion

From absolute counts to relative frequencies: pairs

$$f(w_j \mid w_i) = \frac{N(w_i, w_j)}{\sum_{w'} N(w_i, w')}$$

Example data flow for pairs approach:

key	values	
(dog, $*$)	[6327, 8514, ...]	compute marginal: $\sum_{w'} N(\text{dog}, w') = 42908$
(dog, aardvark)	[2,1]	$f(\text{aardvark} \mid \text{dog}) = 3/42908$
(dog, aardwolf)	[1]	$f(\text{aardwolf} \mid \text{dog}) = 1/42908$
...		
(dog, zebra)	[2,1,1,1]	$f(\text{zebra} \mid \text{dog}) = 5/42908$
(doge, $*$)	[682, ...]	compute marginal: $\sum_{w'} N(\text{doge}, w') = 1267$
...		

Order inversion

- **Goal**: compute the result of a computation (marginal) in the reducer before the data that requires it is processed (relative frequencies)

- **Key insight**: convert sequencing of computation into a **sorting** problem

- Ordering of key/value pairs and key partitioning controlled by the programmer
  - Create a notion of “**before**” and “**after**”

- **Major benefit**: reduced memory footprint

Secondary sorting

Secondary sorting

- **Order inversion: sorting by key**
- What about **sorting by value** (a "secondary" sort)?
  - **Hadoop does not allow it**

(t_1, m_1, r_{80521}) time, sensor, reading
(t_1, m_2, r_{14209})
(t_1, m_3, r_{76042}) Goal: activity of each sensor over time
$\dots$ Idea: sensor id as intermediate key,
(t_2, m_1, r_{21823}) the rest as value $m_1 \rightarrow (t_1, r_{1234})$
(t_2, m_2, r_{66508})
(t_2, m_3, r_{98347}) Wanted: secondary sort by timestamp

Secondary sorting

- **Solution**: move part of the value into the intermediate key and **let Hadoop do the sorting**

$$(m_1, t_1) \rightarrow r_{1234}$$

- Also called "value-to-key" conversion

$$(m_1, t_1) \rightarrow [(r_{80521})]$$
$$(m_1, t_2) \rightarrow [(r_{21823})]$$
$$(m_1, t_3) \rightarrow [(r_{146925}]$$

Requires:
- **Custom key sorting**: first by left element (sensor id), and then by right element (timestamp)
- **Custom partitioner**: partition based on sensor id only
- **State across `reduce()` calls tracked**

Database operations

Databases….

Relation *Hyperlinks*

attributes

tuples

FROM	TO
url1	url2
url2	url3
url3	url5

- **Scenario**:
  - Database tables can be written out to file, one tuple per line
  - MapReduce jobs can perform standard database operations
  - Useful for operations that pass over most (all) tuples
- **Example:**
  - Find all paths of length 2 in Hyperlinks
  - Result should be tuples `(u,v,w)` where a link exists between `(u,v)` and `(v,w)`

join Hyperlinks with itself

FROM	TO
url1	url2
url2	url3
url3	url5

FROM	TO
url1	url2
url2	url3
url3	url5

(url1,url2,url3)
(url2,url3,url5)

Database operations: relational joins

Relational joins

- Popular application: data-warehousing
  - Often data is relational (sales transactions, product inventories,..)

- 3 different strategies for performing relational joins on two datasets (tables in a database) S and T:

  e.g. S are user profiles, T logs of online activity

$$
\begin{array}{lll}
(k_1, s_1, \mathbf{S}_1) & \text{key to join on} & (k_1, t_1, \mathbf{T}_1) \\
(k_2, s_2, \mathbf{S}_2) & \text{tuple id} & (k_3, t_2, \mathbf{T}_2) \\
(k_3, s_3, \mathbf{S}_3) & \text{tuple attributes} & (k_8, t_3, \mathbf{T}_3)
\end{array}
$$

Relational joins: reduce side

database tables exported to file

- **Idea**: map over both **datasets** and emit the join key as intermediate key and the tuple as value

- **One-to-one join**: *at most* one tuple from S and T share the same join key

$$k_{23} \rightarrow [(s_{64}, S_{64}), (t_{84}, T_{84})]$$
$$k_{37} \rightarrow [(s_{68}, S_{68})]$$
$$k_{59} \rightarrow [(t_{97}, T_{97}), (s_{81}, S_{81})]$$
$$\ldots$$

Four possibilities for the values in `reduce()`:

- a tuple from S
- a tuple from T
- (1) a tuple from S, (2) a tuple from T
- (1) a tuple from T, (2) a tuple from S

reducer emits key/value if the value list contains 2 elements

Relational joins: reduce side

- **Idea**: map over both **datasets** and emit the join key as intermediate key and the tuple as value

- **One-to-many join**: the primary key in S can join to *many keys* in T

$$k_{23} \rightarrow [(t_{55}, T_{55}), (t_{44}, T_{44}), (s_{64}, S_{64}), (t_{84}, T_{84})]$$
$$k_{37} \rightarrow [(s_{68}, S_{68})]$$
$$\ldots$$

We do not know when S will be encountered

Possible solution: buffer all tuples in memory, find S, and cross S with all T

Relational joins: reduce side

- **Idea**: map over both **datasets** and emit the join key as intermediate key and the tuple as value
- **One-to-many join**: the primary key in S can join to *many keys* in T

$$
\begin{aligned}
&k_{23} \rightarrow [(t_{55}, T_{55}), (t_{44}, T_{44}), (s_{64}, S_{64}), (t_{84}, T_{84})] \\
&k_{37} \rightarrow [(s_{68}, S_{68})] \\
&\ldots
\end{aligned}
$$

Better (less memory intensive): value-to-key conversion to create a composite key (join key, tuple id)

$$
\begin{aligned}
&(k_{82}, s_{105}) \rightarrow [(S_{105})] \\
&(k_{82}, t_{98}) \rightarrow [(T_{98})] \\
&\ldots
\end{aligned}
$$

Requires:
(1) Sort order by keys
(2) Custom partitioner

Relational joins: reduce side

- **Idea**: map over both **datasets** and emit the join key as intermediate key and the tuple as value

- **Many-to-many join**: many tuples in S can join to many tuples in T

Possible solution: **employ the one-to-many approach**.

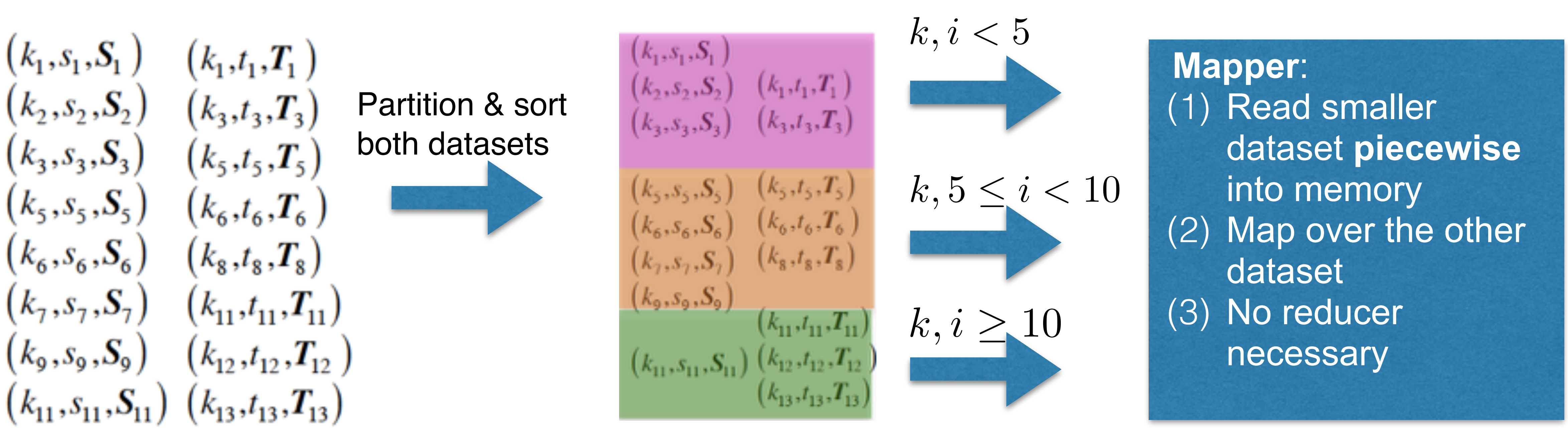
Works well if S has only a few tuples per join (requires data knowledge).

$$(k_{82}, s_{105}) \rightarrow [(S_{105})]$$
$$(k_{82}, s_{145}) \rightarrow [(S_{145})]$$
$$\ldots$$
$$(k_{82}, t_{98}) \rightarrow [(T_{98})]$$
$$(k_{82}, t_{101}) \rightarrow [(T_{101})]$$
$$(k_{82}, t_{137}) \rightarrow [(T_{137})]$$
$$\ldots$$

Relational joins: map side

- **Problem of reduce-side joins**: both datasets are shuffled across the network

- **Assumption**: both datasets sorted by join key, they can be joined by “scanning” both datasets simultaneously



no shuffling across the network!

Relational joins: comparison

- Problem of **reduce-side joins**: both datasets are shuffled across the network
- **Map-side join**: no data is shuffled through the network, very efficient
- Preprocessing steps take up more time in map-side join (partitioning files, sorting by join key)
- **Usage scenarios**:
  - Reduce-side: adhoc queries
  - Map-side: queries as part of a longer workflow; preprocessing steps are part of the workflow (can also be Hadoop jobs)

Recommended reading

- **Mining of Massive Datasets** by Rajaraman & Ullman. Available online. Chapter 2.
  - The last part of this lecture (database operations) has been drawn from this chapter.
- **Data-Intensive Text Processing with MapReduce** by Lin et al. Available online. Chapter 3.
  - The lecture is mostly based on the content of this chapter.

THE END