RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Analyzing Data with Map-Reduce

Prof. Dr. Artur Andrzejak
http://pvs.ifi.uni-heidelberg.de/
artur@uni-hd.de

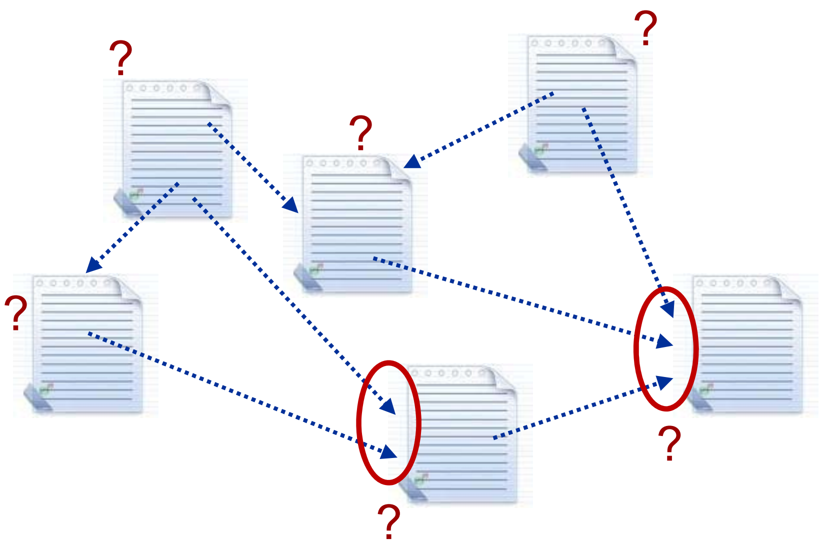
INTRODUCTION TO MAPREDUCE

MapReduce Programming Model

- Re-discovered by Google with goals:
  - "Reliability has to come from the software"
  - "How can we make it easy to write distributed programs?"

- A major tool at Google
  - 2.2 million jobs in September 2007 (http://goo.gl/dsnDl)
  - In 2008 about 100k MapReduce jobs **per day**
    - over 20 petabytes of data processed per day
    - each job occupies about 400 servers

Example: Reverse Web-Link Graph



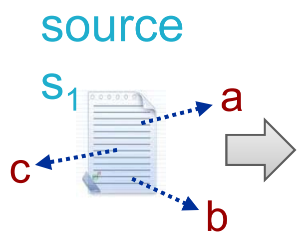
A Problem suitable for MapReduce:

In a set of html-documents, find out which other documents point (via links) to it (for each document)

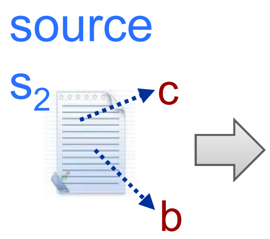
Reverse Web-Link Graph: Solution

parallel MAP

1. For each link to target t found in document source emit (t, source)



(a, s_1)
(b, s_1)
(c, s_1)



(c, s_2)
(b, s_2)

2. Sort all pairs by same t's

a: (a, s_1)
b: (b, s_1) (b, s_2)
c: (c, s_1) (c, s_2)

parallel SORT

3. Results are lists:

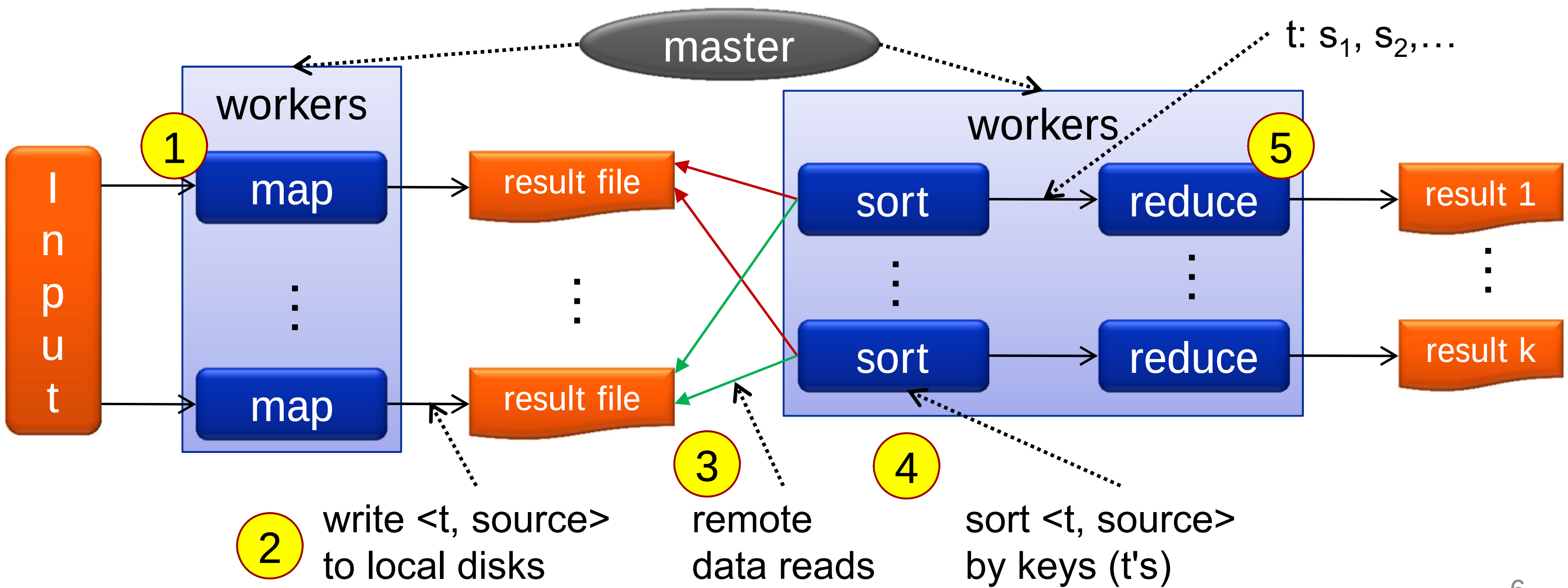
   $list_t(source)$ for each t

$list_a(s_1)$
$list_b(s_1, s_2)$
$list_c(s_2, s_2)$

parallel REDUCE

Distributed MapReduce Frameworks

- Large data sets require **distribution**
  - e.g. 1000s of map / reduce tasks in parallel
- Not only Google: **Hadoop** is an open-source implementation



What Can Go Wrong?

1. Master failure
2. (Some) workers fail (each with probability p)
   - Case 1: Complete failure
   - Case 2: Unusual slow execution: **stragglers**

- Failures "1" are very rare
- **Failures 2 are frequent**
  - Workers are commodity (low cost) machines
  - Recall: given n workers, probability of at least one "problem" is

$$1 - (1 - p)^n$$

Worker Fault Tolerance in MapReduce

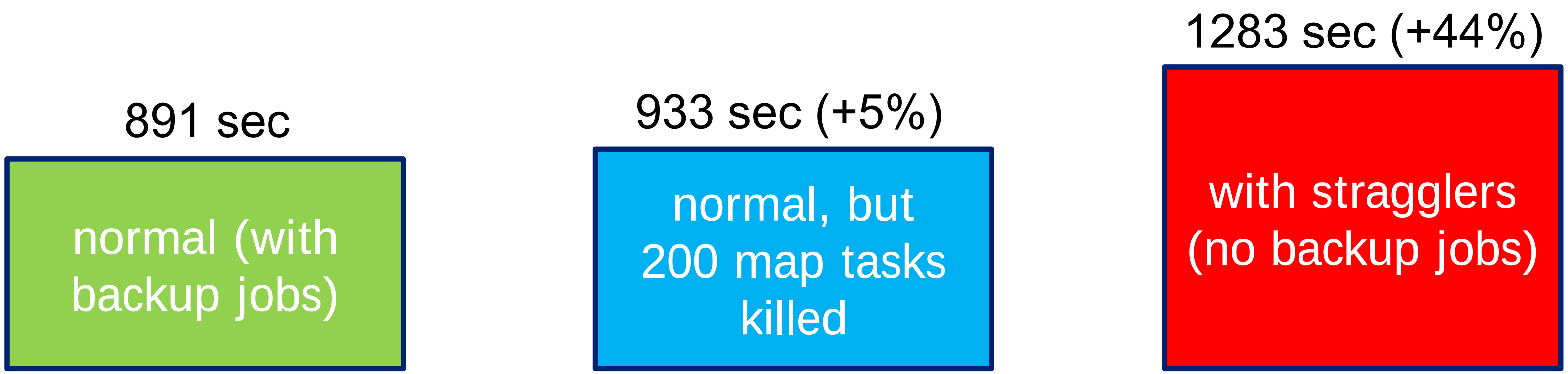
Worker fails completely

- Master pings regularly each worker
- If not responding, worker is marked as "dead"
- All his (map/red.) jobs *in-progress* are re-executed

Slow execution: stragglers

- Close to completion, masters schedules redundant execution of the remaining *in-progress* jobs
- Overhead is few % ...
- ... but can reduce time-to-solution by up to 1/3

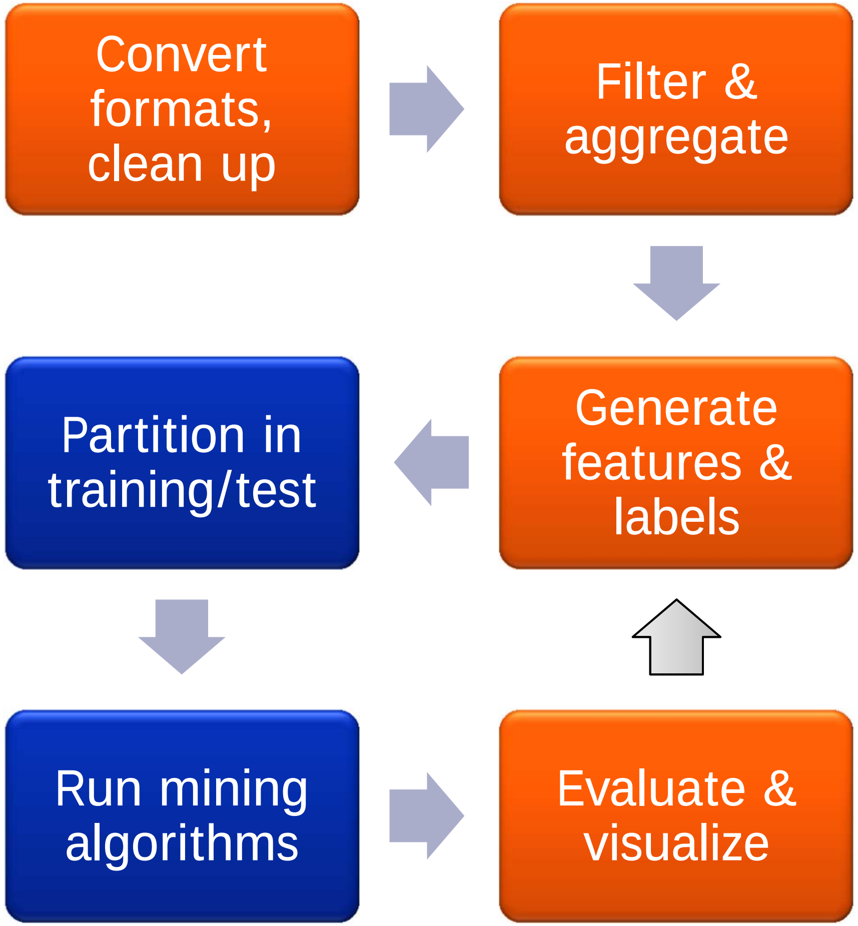
Example: sorting 10^{10} 100-byte records (1 TB) with ~1700 workers



ONLINE COMPUTING WITH MAPREDUCE

Explorative Data Analysis

- Our research at PVS requires a lot of data analysis
- Usually interactive work:
  1. Change parameters /code
  2. Re-run
  3. Evaluate the results
  4. Adjust and repeat
- One of the bottlenecks is **time for the mining algorithms to finish**
- => Shortening it reduces the total time of exploration



Orange: Lots of scripting!

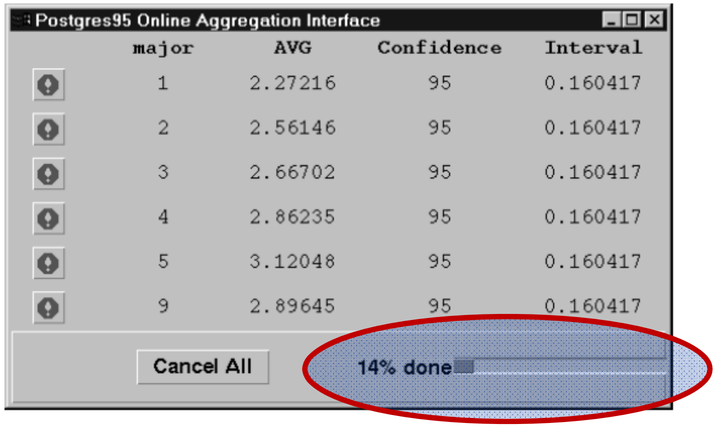
Time-to-Solution is Important

- Researchers work to a large extend in an "exploratory" way
  - They evaluate last results and decide *then* on the next job(s)
  - Waiting time for a batch computation to finish is called Time-to-Solution
- **Shortening the Time-to-Solution reduces significantly the total time of exploration**

Online Aggregation

- J. M. Hellerstein, P. Haas and H. Wang introduced in 1997 the concept of Online Aggregation
  - Report online preliminary results (and confidence intervals) for very large queries

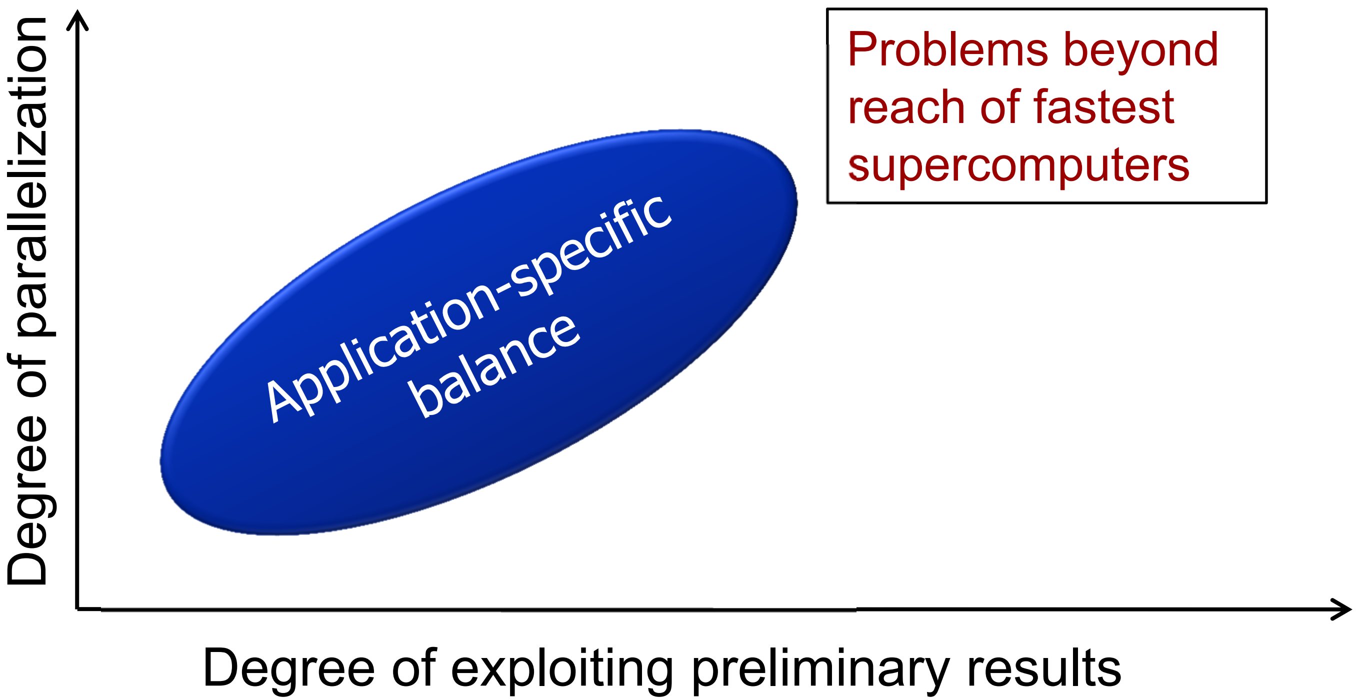
```
SELECT AVG(final_grade) from grades
WHERE course_name = `CS186`
GROUP BY major;
```



- **Shortens „Time-to-Decision"** in an exploratory data study
  - Allows to cancel a futile query prematurely
  - ... Or stop fast if results are precise enough
  - Helps to identify early how to drill down data

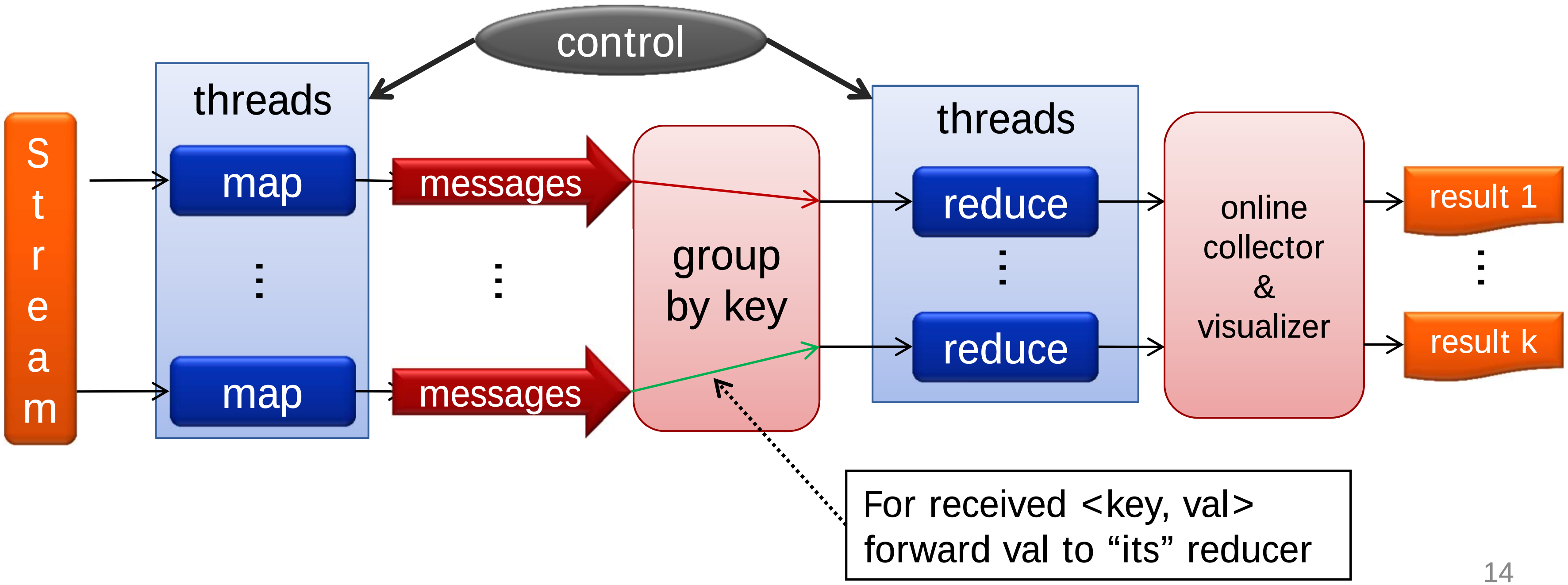
Incremental-Parallel Data Analysis

- By combining preliminary results <u>with</u> parallelization we get two dimensions of scalability
- We use **Map-Reduce** to combine both approaches

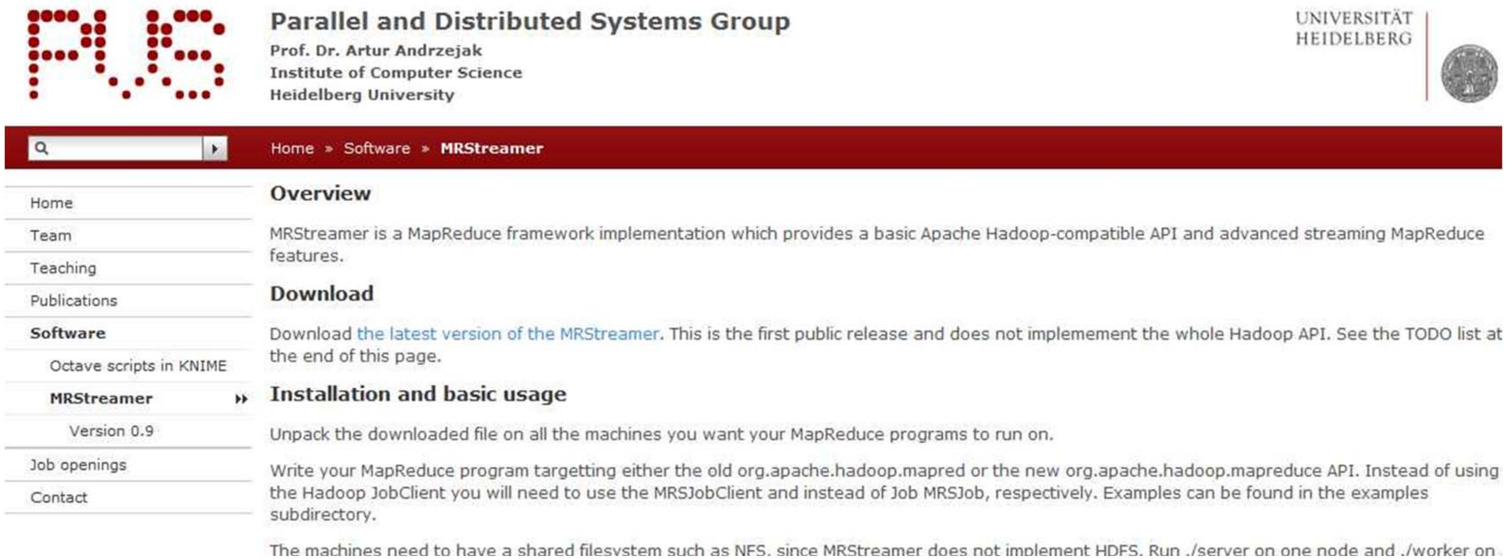


MRStreamer

- We implemented **MRStreamer** - an “enhanced” version of Hadoop
- It can process data online (“Streaming”)
- ... OR in batch mode



MRStreamer



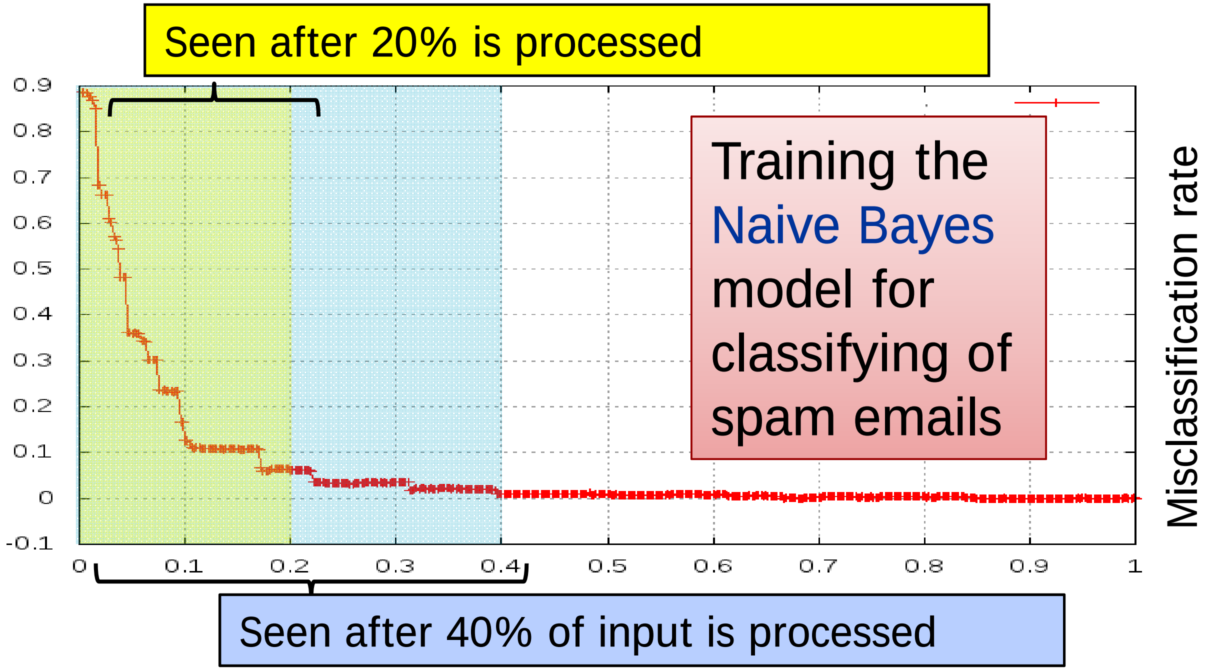
Features:

- Batch-mode and incremental (online) processing
- Efficient shared-memory processing and “flip-a-switch” cluster processing
- Hadoop-compatible APIs

http://pvs.ifi.uni-heidelberg.de/software/mrstreamer/

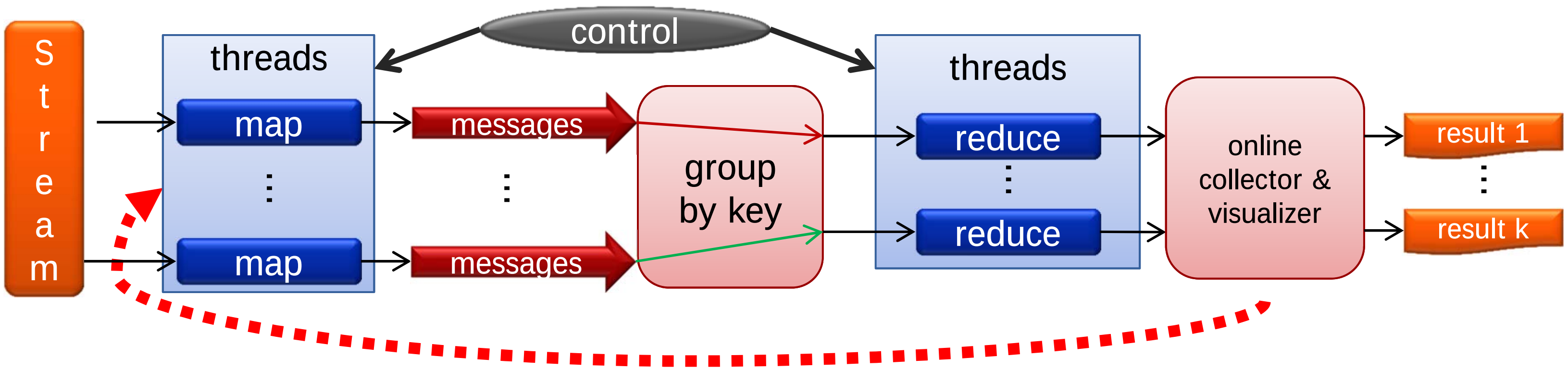
Applications to Data Analysis

- Goal: **faster deciding in data analysis studies**
  - Estimating whether preliminary solution is stable enough
  - Detecting changes in data profile
- Example: **online convergence graph**
  - Updates periodically the history of preliminary results



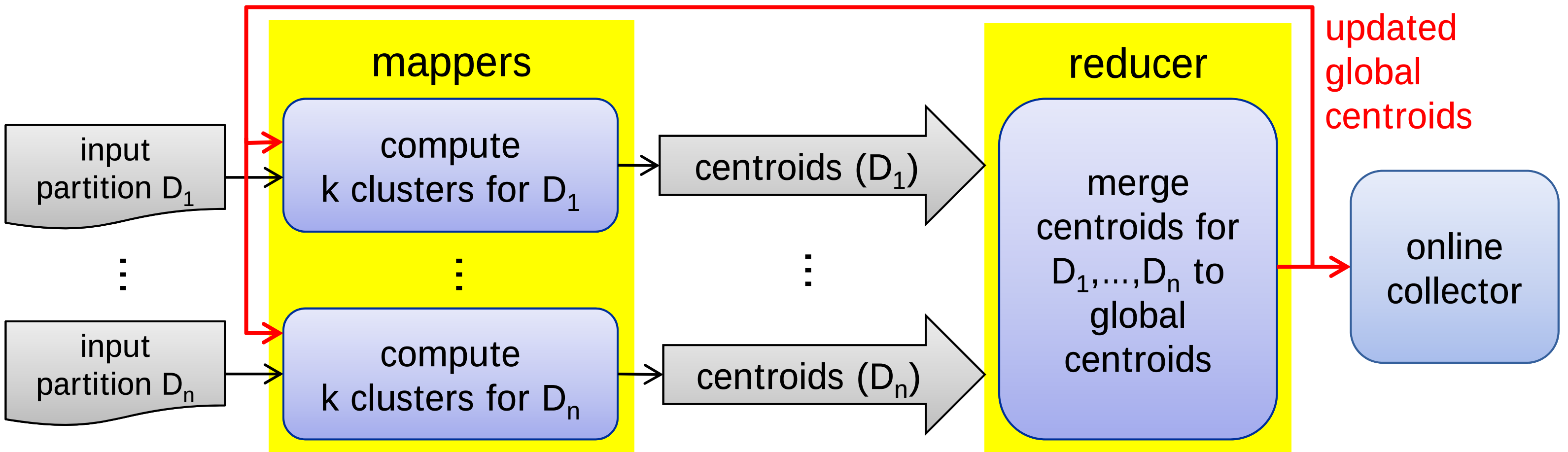
Algorithms

- Simpler algorithms require **only *one* MapReduce pass**
  - Aggregation (AVG, SUM, ...), Linear regression, PCA, Classification (Naïve Bayes), ...
- **Challenging are multi-pass algorithms**
  - For iterative approaches, e.g. clustering via k-means
- Efficiency dictates changes in algorithms / framework



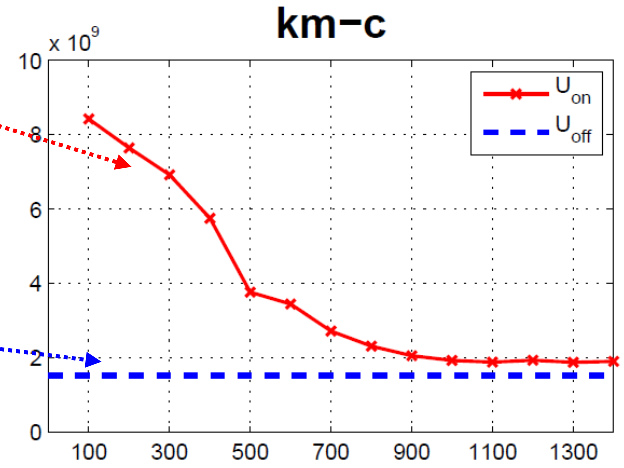
Feeding back preliminary results to avoid multiple passes

K-Means Clustering Algorithm

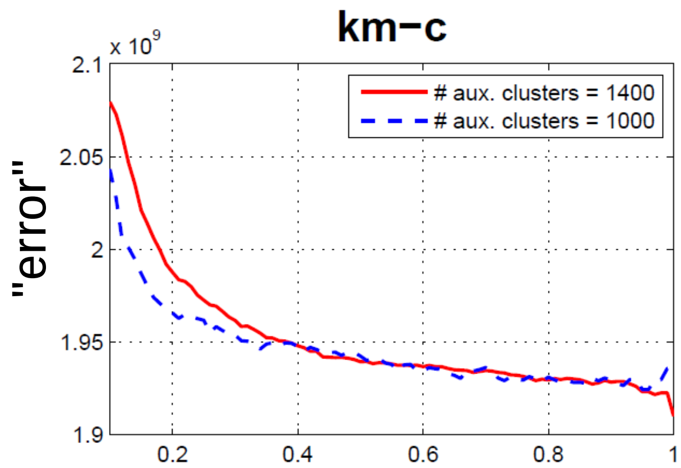


"error" of new algorithm

"error" of off-line (standard) algorithm



~ larger memory of mappers



processed input fraction

EXTENSIONS AND ALGORITHMS FOR ONLINE MAPREDUCE

More Research Problems

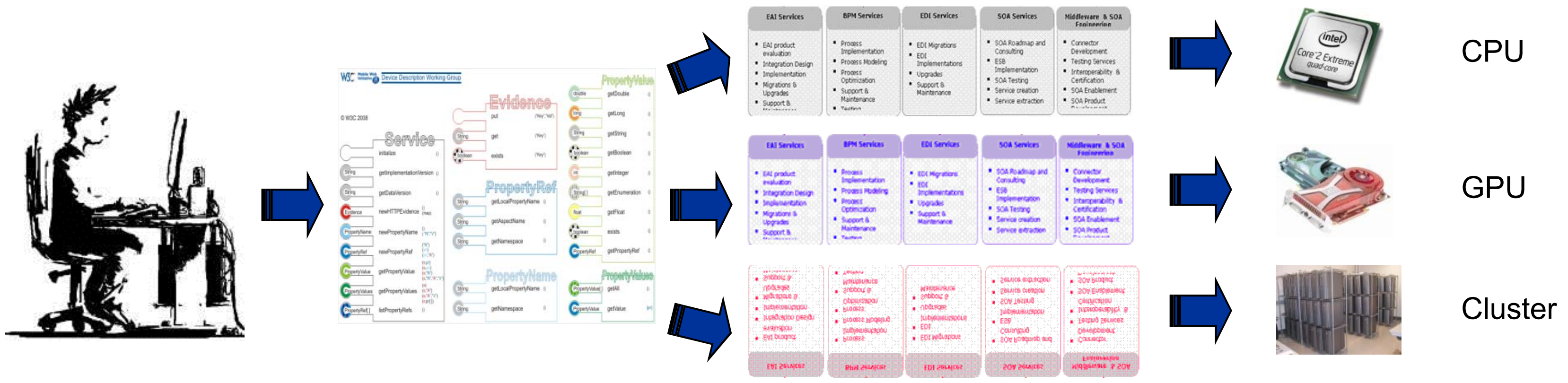
- How to enable machine learning algorithms to work **incrementally** (online) and **in parallel**?
- How to help programmers to access / **integrate MapReduce processing in only few lines of code**?
- How to **reduce the inefficiencies** of the frameworks **for smaller data sets**?

Framework Inefficiences

- Popular Map-Reduce frameworks like Hadoop are very inefficient for small to medium data sets
  - A job with 5 MB (linear regression) needs on Hadoop 30x longer than on a simple “ad-hoc” MapReduce simulator with 2 threads
  - Hadoop startup time is in the range of 10-20 Seconds
- Key Problem: the code + libraries required for distributed processing introduce overhead not necessary for smaller data sets

Efficient Map-Reduce

- Idea: **one API but resource-specific framework implementations**
  - Dynamic selection of the right infrastructure depending on the input size
- Challenges: coherent APIs and „semantics“
- First step: MRStreamer
  - Both shared-memory and distributed architecture



Developer Single API Infrastructure-Specific Middleware Infrastructure Type

Refining our Motivation

Why incremental-parallel processing?

Reduced time-to-decision

- Hadoop is inefficient compared to custom shared-memory code
- Programming in M-R style takes significantly longer

Processing big data sets

- Big data sets have specific properties
- Huge data sets are very rare (or inaccessible)

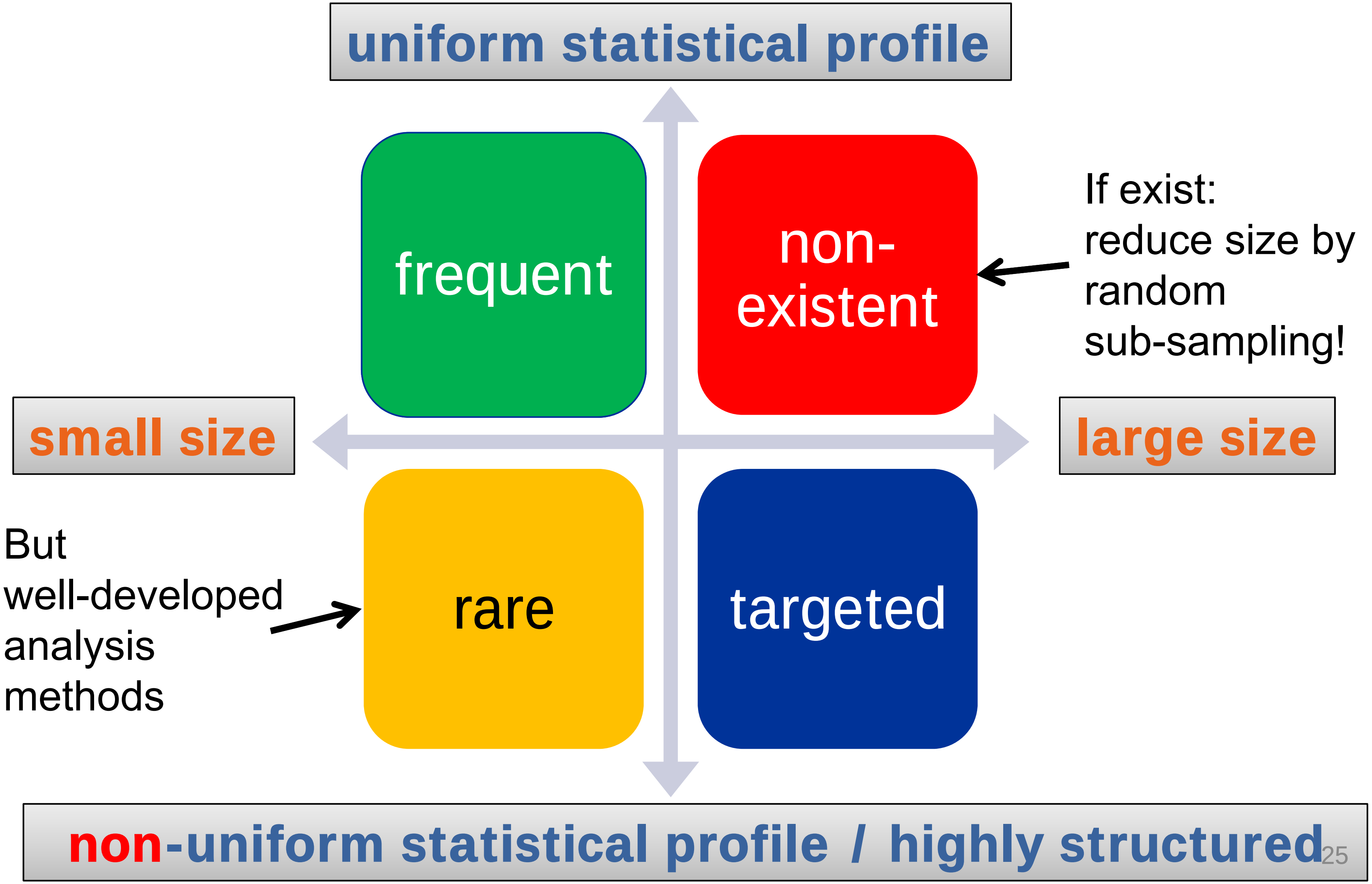
Which Algorithms Should We Adapt?

- There is a big collection of machine learning alg‘s running on M-R
  - E.g. **Apache Mahout**
- Adapting them to incremental-parallel processing could be fun ...
- But which ones to choose?
- ... Which are really needed?

Mahout currently has

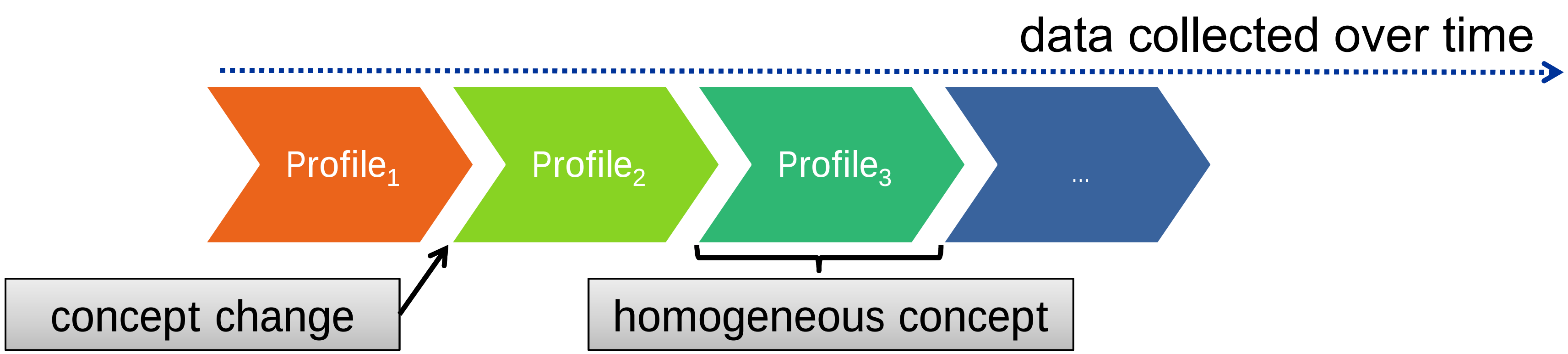
- Collaborative Filtering
- User and Item based recommenders
- K-Means, Fuzzy K-Means clustering
- Mean Shift clustering
- Dirichlet process clustering
- Latent Dirichlet Allocation
- Singular value decomposition
- Parallel Frequent Pattern mining
- Complementary Naive Bayes classifier
- Random forest decision tree based classifier
- High performance java collections (previously colt collections)
- A vibrant community
- and many more cool stuff to come by this summer thanks to Google summer of code

Data Sets: Scale vs. Homogeneity



Consequences for Analyzing Large Data Sets

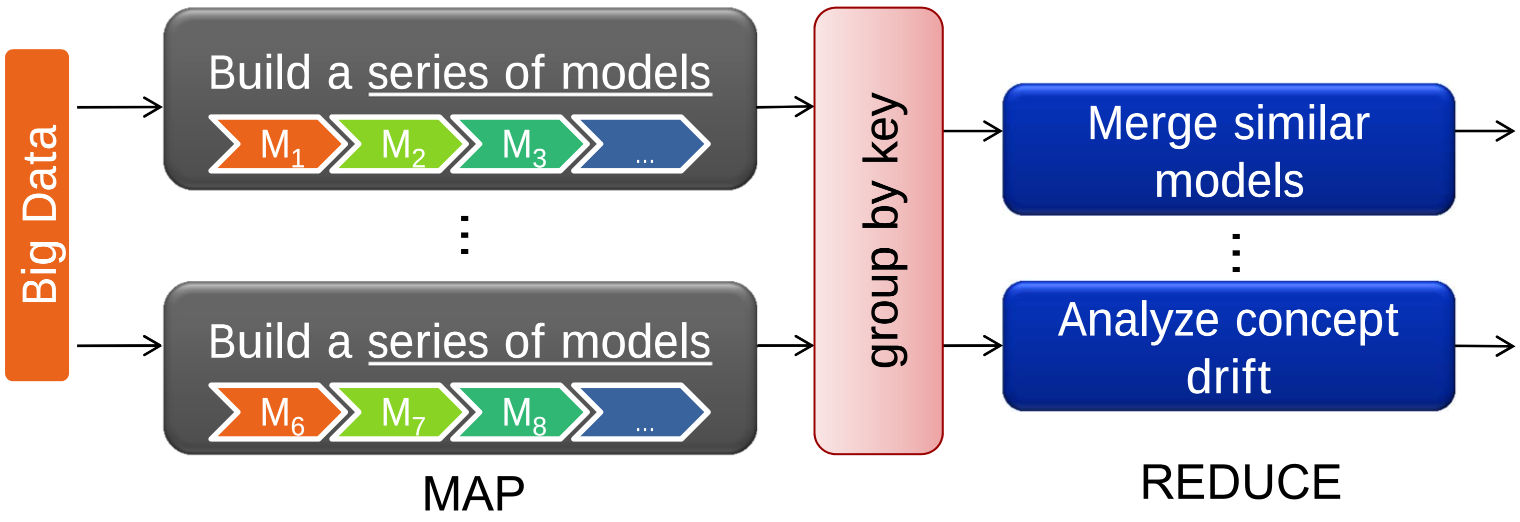
- Some „small-data“ algorithms become useless
  - They assume a uniform profile in over all data set
- Other challenges than in „small-data“ machine learning
  - Identify and recognize rare patterns
  - Split data into sets with homogeneous profile
  - Understand and visualize concept drift
  - Unify models built on parts of the data



Identifying Concept Drift in Parallel

- Profile of a modeled phenomenon changes over time
  - E.g. Profiles of spam emails evolve within days
- We need a series of models instead of a single one
- **Concept drift detection** tells us when to switch model
- Algorithms exist for serial case, what about parallelism?

A joint project with I2R, Singapore



SEQUENTIAL CONCEPT DRIFT DETECTION

Classification in a Nutshell

- In **classification** we want to learn from examples model f = a function from samples (vectors) to elements of a finite set (labels)
- Phase 1: **Training**: we fit/optimize f so that it maps most accurately training samples to their labels
- Phase 2: **Prediction**: f is applied to an unknown sample s to predict its most likely label f(s)

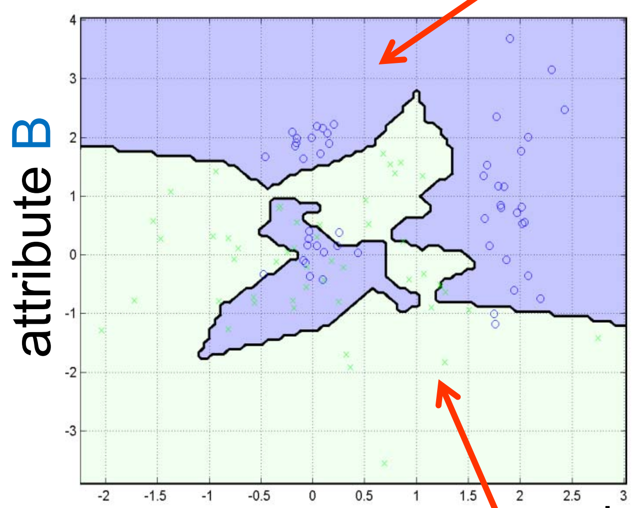
	Attribute 1	**...**	**Attribute k**	**Labels**
Training Sample 1	Thursday		1000	ok
...	...	...	...	...
Training Sample N	Sunday		10^6	defect
Unknown Sample	Monday		50000	?

1. training

2. prediction

Classifiers explained visually

- If we have only two attributes, we can interpret each sample as <u>a point in R^2</u>
- Labels are encoded as colors
- **Training**: finding a suitable subdivision of the plane given the (colored) training points
  - **Model f** = a compact representation of the subdivision of R^2
- **Prediction**: given a new sample, find its color = label
- More metrics => R^d

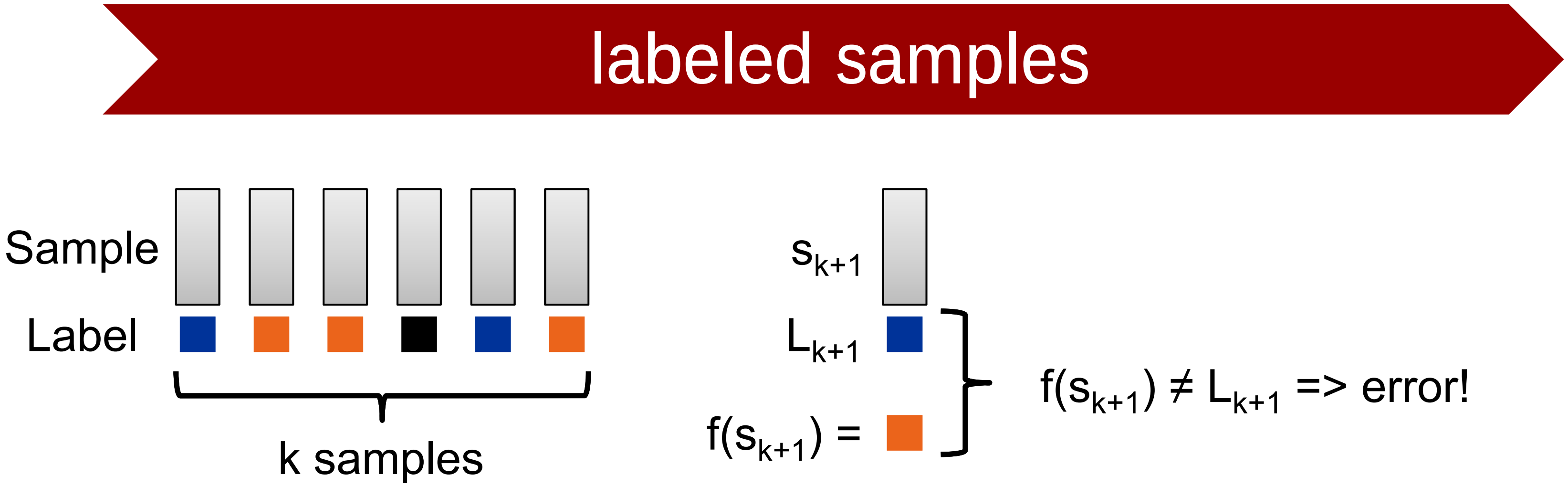


Incremental Training

f learned in order of new training samples

- Assume that we have a long table with labeled samples
- We learn a model f incrementally – in order of „new“ labeled samples
- How could we detect a concept shift in this scenario?

Incremental Training with Testing

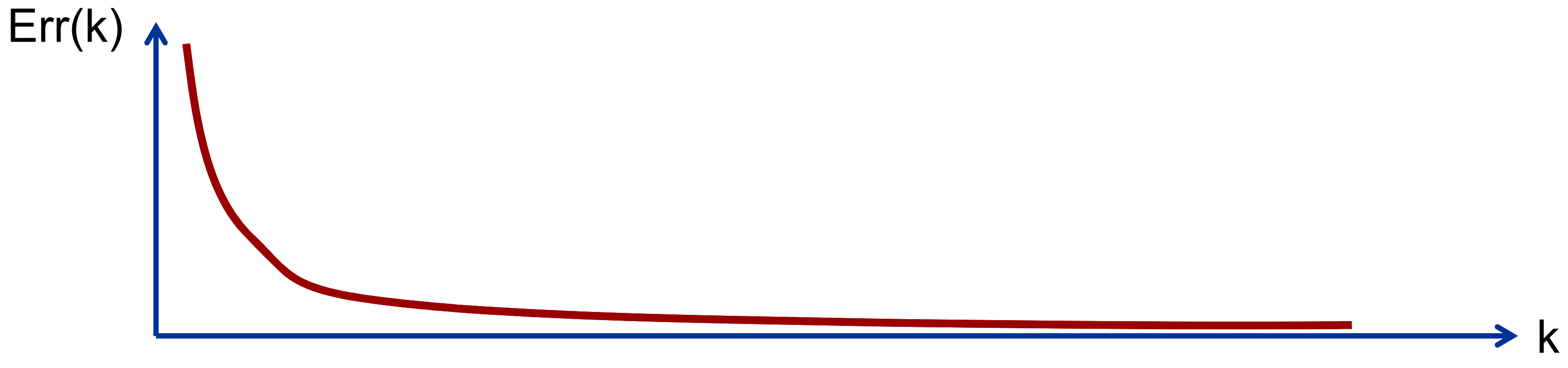


- After learning f on k first samples, we predict on sample s_{k+1} and compare prediction $f(s_{k+1})$ against true label L_{k+1}
- Then we use labeled s_{k+1} to further do training of f
- => So we learn f as before but now also evaluate its accuracy on each new sample (before learning on it)

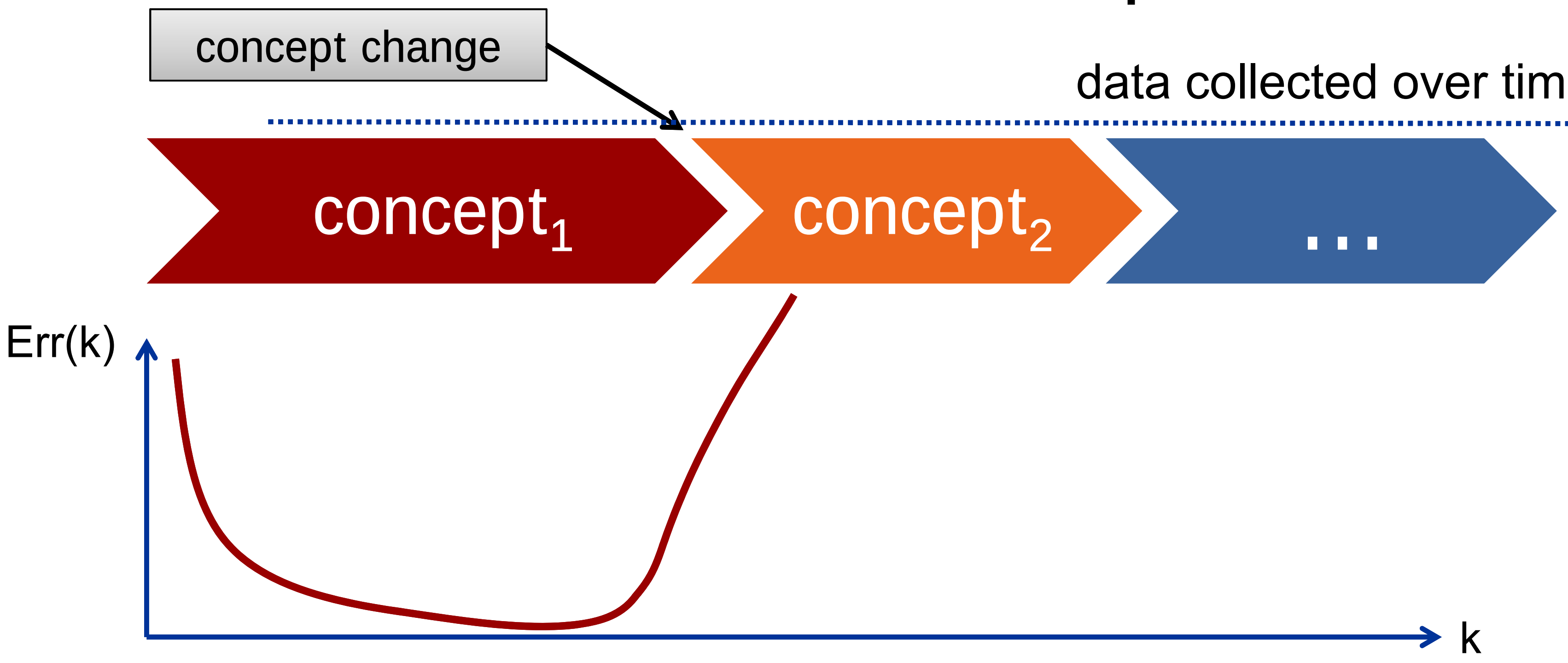
Error Rate

- For each k we can compute the **error rate Err(k)** as:
  - Err(k) = (# errors of f until now)/k
- No concept drift: with each new sample f becomes more accurate => Err(k) drops

labeled samples

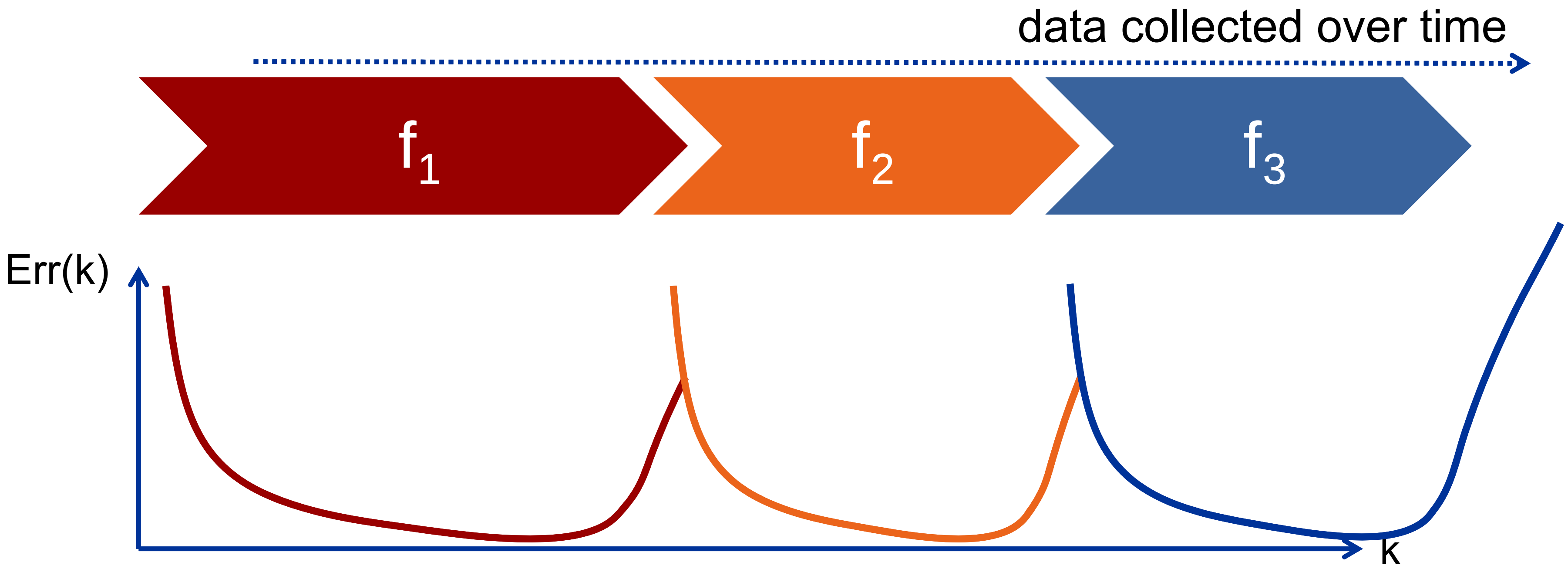


Error Rate under Concept Drift



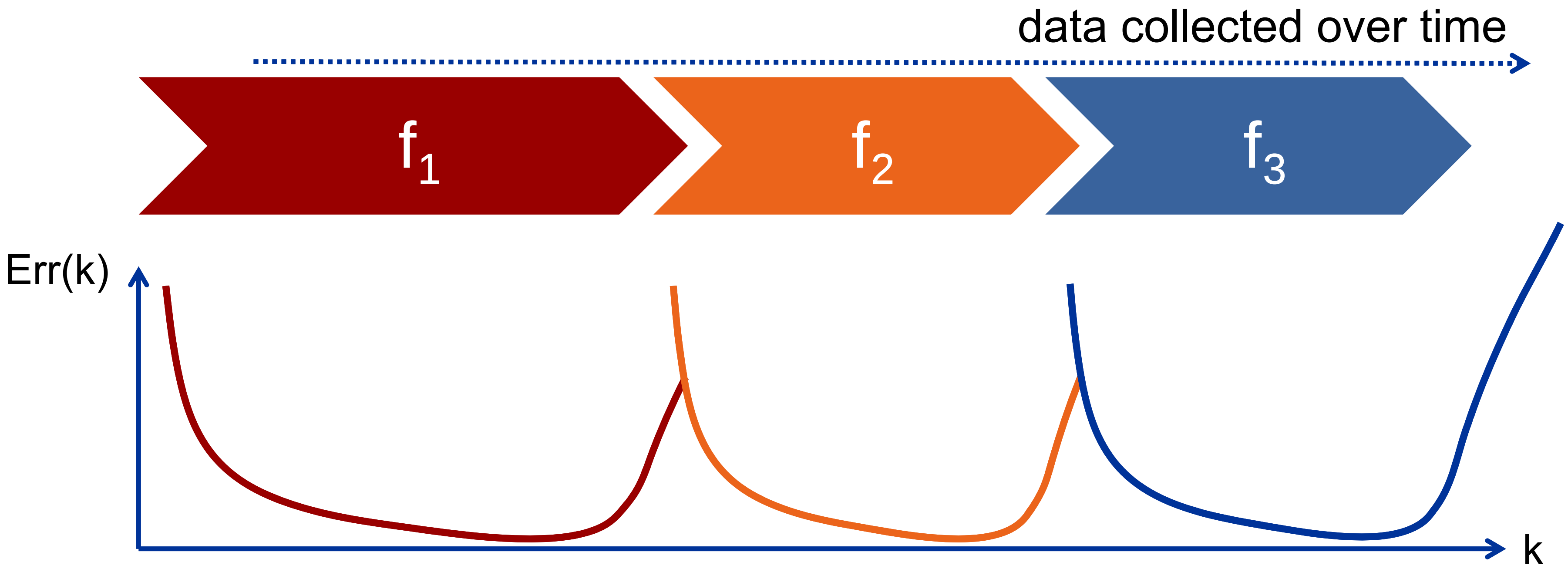
- If the relationship „sample – label“ changes over time (i.e. we have concept drift), the Err(k) starts to increase after some time!
- => By monitoring Err(k) we can detect concept drift

Reset after a Concept Drift



- We need a new model after a concept drift!
- If Err(k) reaches a critical level:
  - Drop the old model (f_1)
  - Start learning a new model (f_2)
  - Reset Err(k)

Error Rate under Concept Drift



- We need a new model after a concept drift!
- If Err(k) reaches a critical level:
  - Drop the old model
  - Start learning a new one
  - Reset Err(k)

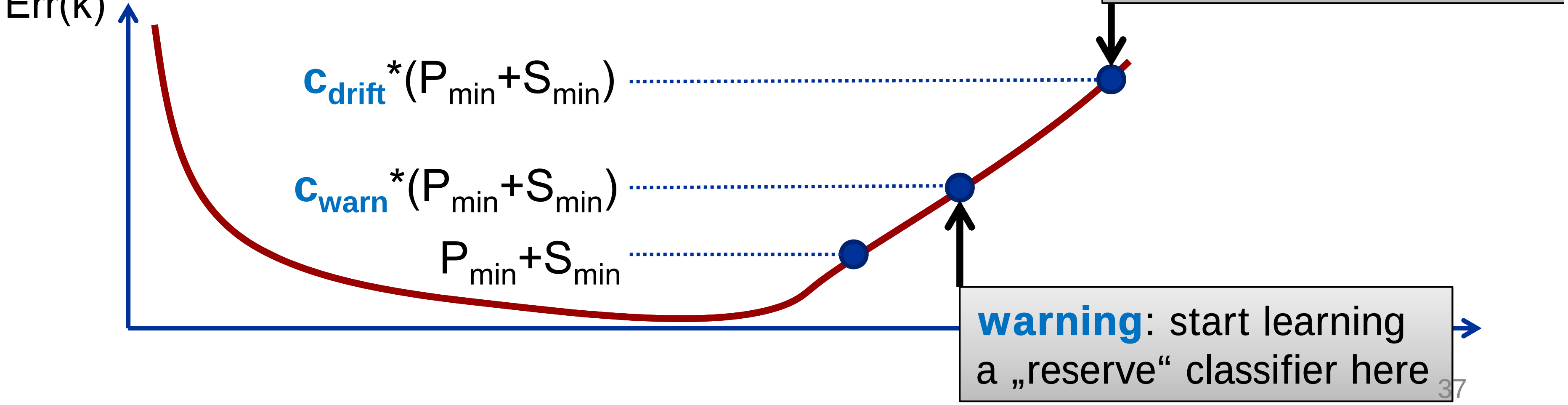
Sequential Concept Drift Detection (CDD)

- The complete algorithm is more complex (link)
- In addition to the error rate Err(k) we also monitor:
  - minimum P_{min} of Err(k)
  - minimum S_{min} of the std. deviation of Err(k)

We have now two critical levels

- If Err(k) $> c_{warn} * (P_{min} + S_{min})$ then **warning signal** is issued
- If Err(k) $> c_{drift} * (P_{min} + S_{min})$ then a **drift signal** is issued

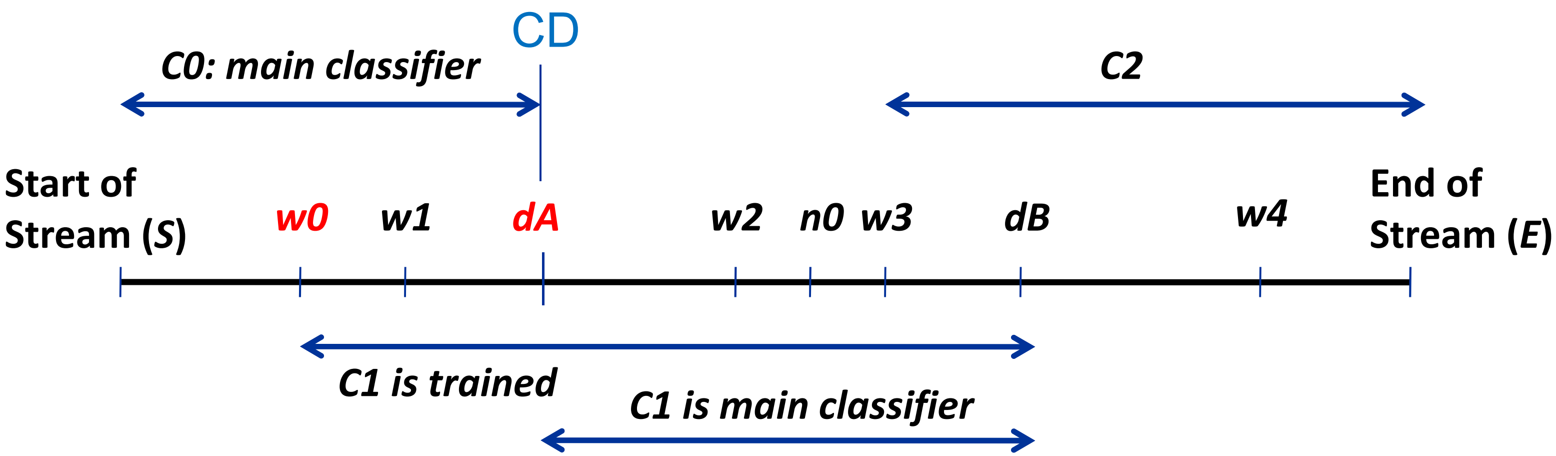
drift: switch to the „reserve“ classifier here; reset Err(k), P_{min}, S_{min}



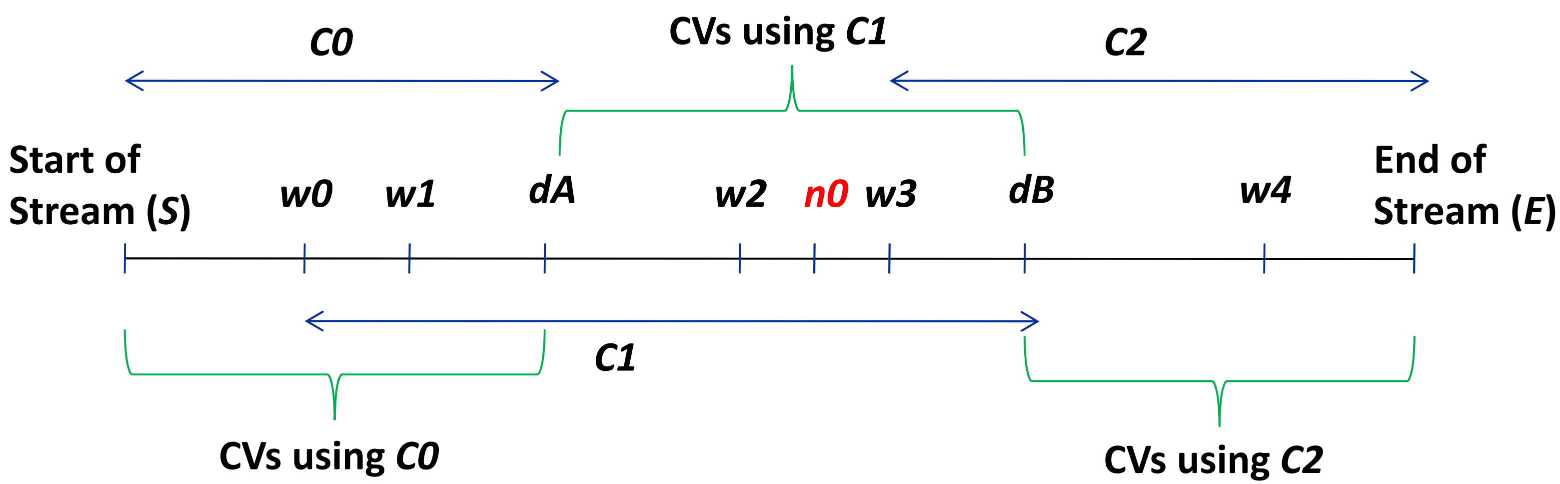
warning: start learning a „reserve“ classifier here

Sequential CDD /2

- When a warning signal is issued at position **w0**, a **“reserve”** classifier **C1** is created
  - C1 is trained since w0 but not used (yet); the current classifier C0 remains the “main” classifier
- When afterwards a drift signal is issued (at position **dA**):
  - We report a concept drift (CD) at d0 and we save or discard C0
  - C1 replaces C0: C1 becomes the main classifier
  - Minimum P_{min} of the error and of variance S_{min} are resetted



A More Complex Example

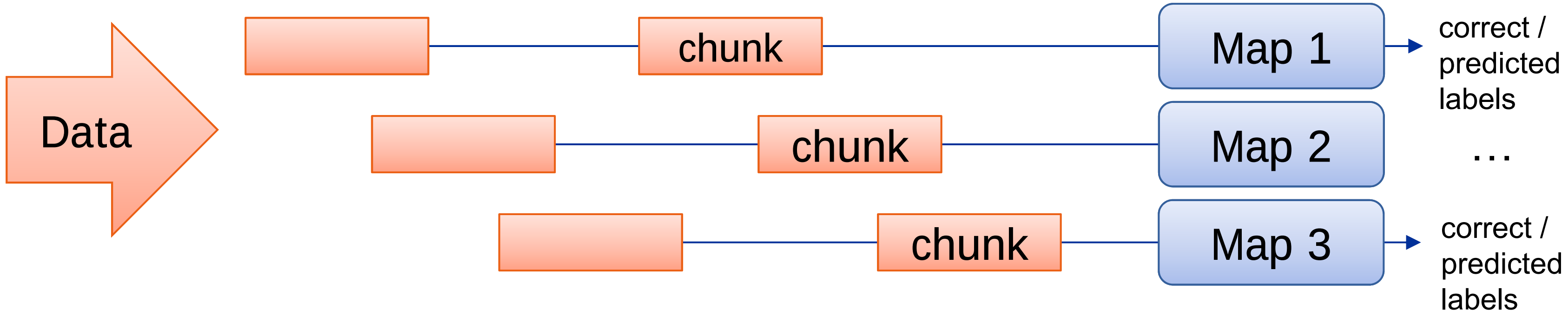


- For a distributed version we use also **correctness vectors** (**CV**s) which are sequences of false/true's depending on whether the main classifier was correct or not on a particular sample
  - CVs have associated information about their stream position
- If there are several warnings before a drift, we use the oldest one (not sure: is this also the case in seq. version?)
- If error(C1) goes below warning level (see **n0** above), we discard a reserve model and create a new (reserve, C2) at next warning (w3)

PARALLEL CONCEPT DRIFT DETECTION

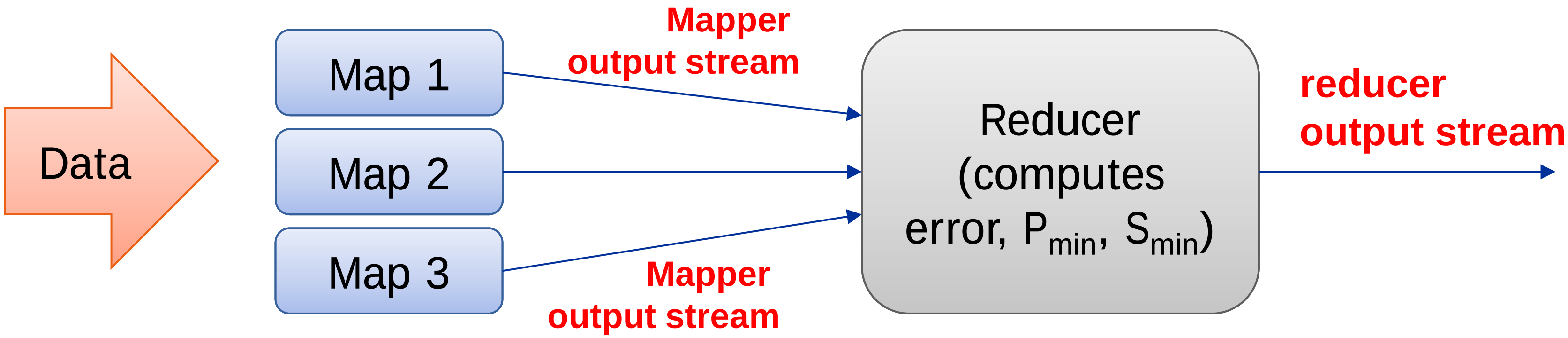
Overall Parallel Architecture /1

- We assume that the incoming stream is multiplexed into **chunks**, each is sent to a mapper / processor



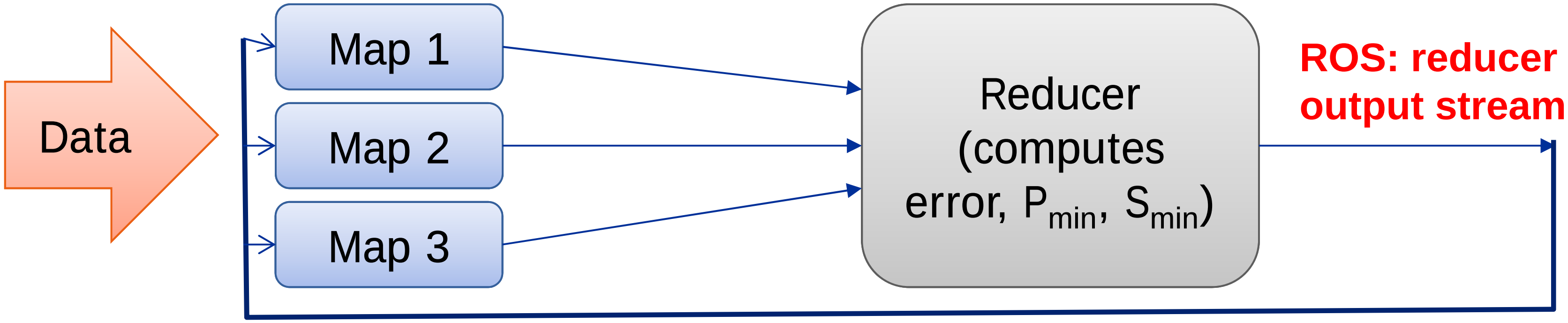
- Each mapper learns the main (and possibly a reserve) classifier on its chunks (considered as a single stream)
- Their output (**correctness vector** (**CV**)) contains
  - Correct labels and
  - Labels predicted by the mapper's current classifier

Overall Parallel Architecture /2



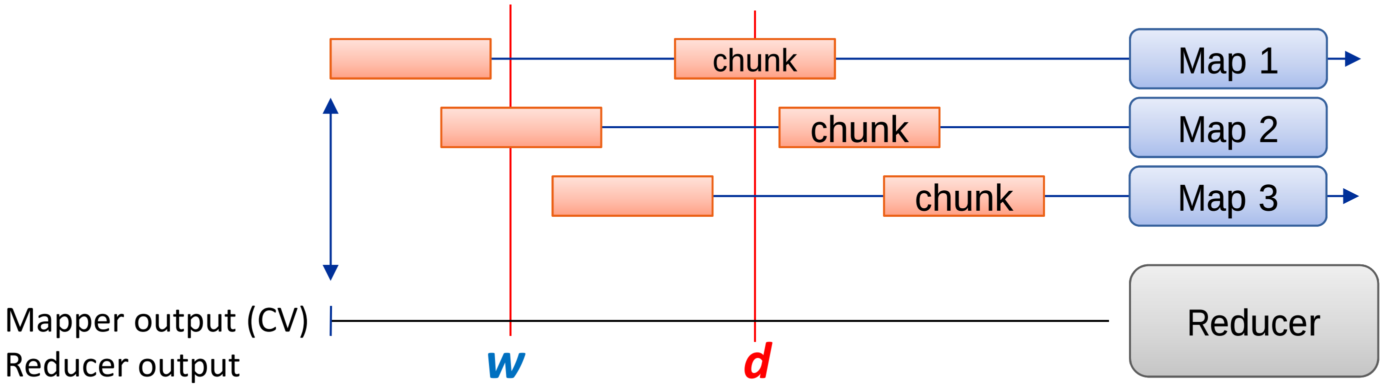
- Reducer collects the input from all mappers
- It computes the global error rate Err(k) over all mappers
- From Err(k) it computes the minima P_{min}, S_{min} and outputs the **levels** normal / warning / drift
  - This is **reducer output stream** (**ROS**)
  - Technicality: The levels are labeled with stream positions

Overall Parallel Architecture /3



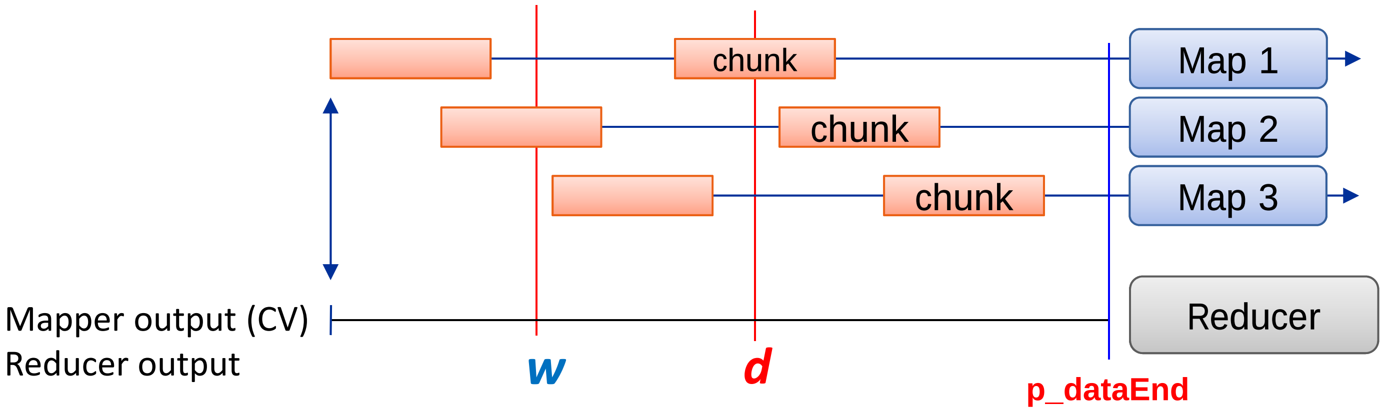
- Reducer output stream is "feeded back" to the mappers
- Each mapper must react to changes in levels of the reducer output:
  - Normal -> warning: start learning reserve classifier
  - Warning -> drift: switch to the reserve classifier

Reactions to Warning / Drift Events /1



- Here mappers have sent their outputs to reducer which detected warning at **w** and drift at **d**
- What should reducer do (at d) to mimic the sequential algorithm?
- Obviously part of the CV after d is useless because it comes from (main) classifiers which should have been replaced at d
- Reducer shall discard the CV after d and wait for recomputed and "correct" CV (coming from a new classifier)
- => When reducer receives chunks of the new & correct mapper outputs, it assembles them and continues since d

Reactions to Warning / Drift Events /2



- *Here mappers have sent their outputs to reducer which detected warning at **w** and drift at **d***
- So upon receiving new input from reducer a mapper does:
  - For warning at position w: starts learning a reserve classifier at w
  - For drift event at d:
    - It switches to the reserve classifier at d
    - It re-computes own output from d to p_dataEnd and sends it to the reducer

Summary of Behavior

- Reducer:
  - On drift event at d it discards mapper input after d, resets P_{min}, S_{min} and waits for re-computed, correct mapper inputs
- Mapper:
  - On warning event at w: it starts training a reserved classifier since (historical) w
  - On drift event at d: it switches to the reserve classifier, re-computes and re-sends all output to reducer
- Note: drift event at d is like a “sync barrier”, it causes all to stop and re-compute everything since d

References

- Joos-Hendrik Böse, Artur Andrzejak, Mikael Högqvist: ***Beyond Online Aggregation: Parallel and Incremental Data Mining with Online MapReduce***, ACM MDAC 2010, Raleigh, NC, 2010.

- Artur Andrzejak, Joao Bartolo Gomes: ***Parallel Concept Drift Detection with Online Map-Reduce***, International Workshop on Knowledge Discovery (KDCloud-2012), at IEEE ICDM 2012, Brussels, December 2012.

THANK YOU.
QUESTIONS?