Part I

Text Evolution and Author Contributions

Luca de Alfaro
UC Santa Cruz
(on leave, visiting Google)

Text tracking across revisions

Wikis evolve due to people editing.

Text evolves from one revision to the next.

There are two fundamental problems:

- **How do we track text across revisions?** How can we attribute text to its original author?
- **How do we measure author contribution?**

We explore here some basic ideas.

Text tracking

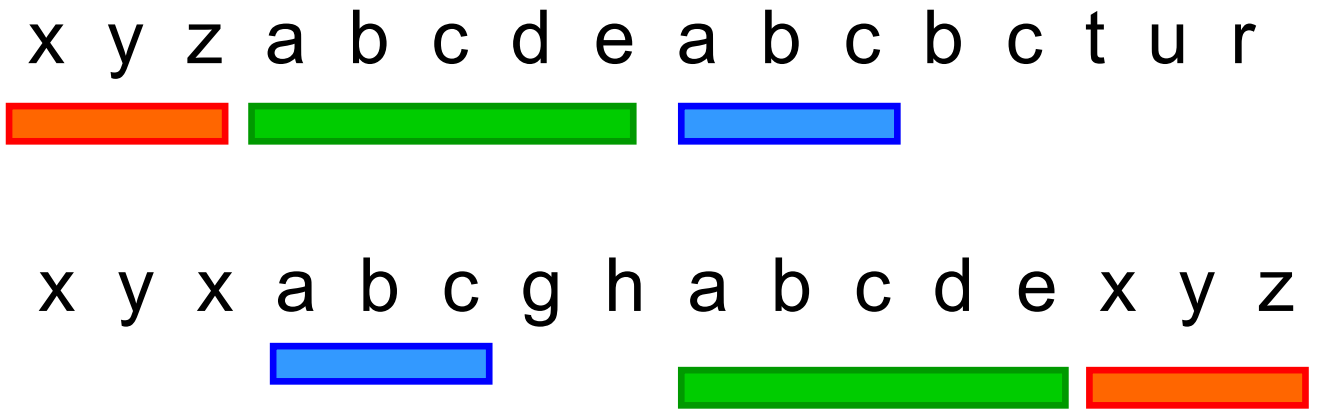
There are many tools that track text across revisions:

- diff (line-based)
- wdiff (word-based)
- git, svn (revision systems based on the above, fast analysis of multiple revisions)

What is the problem?

- **Block moves:** if the order of two paragraphs is swapped, they consider one removed in a place, then reinserted in the other.
- **Deleted and re-inserted text:** these tools just compare consecutive revisions: if text is removed, then re-inserted, they attribute it to the person who reinserted it last.
  - Very open to attacks!

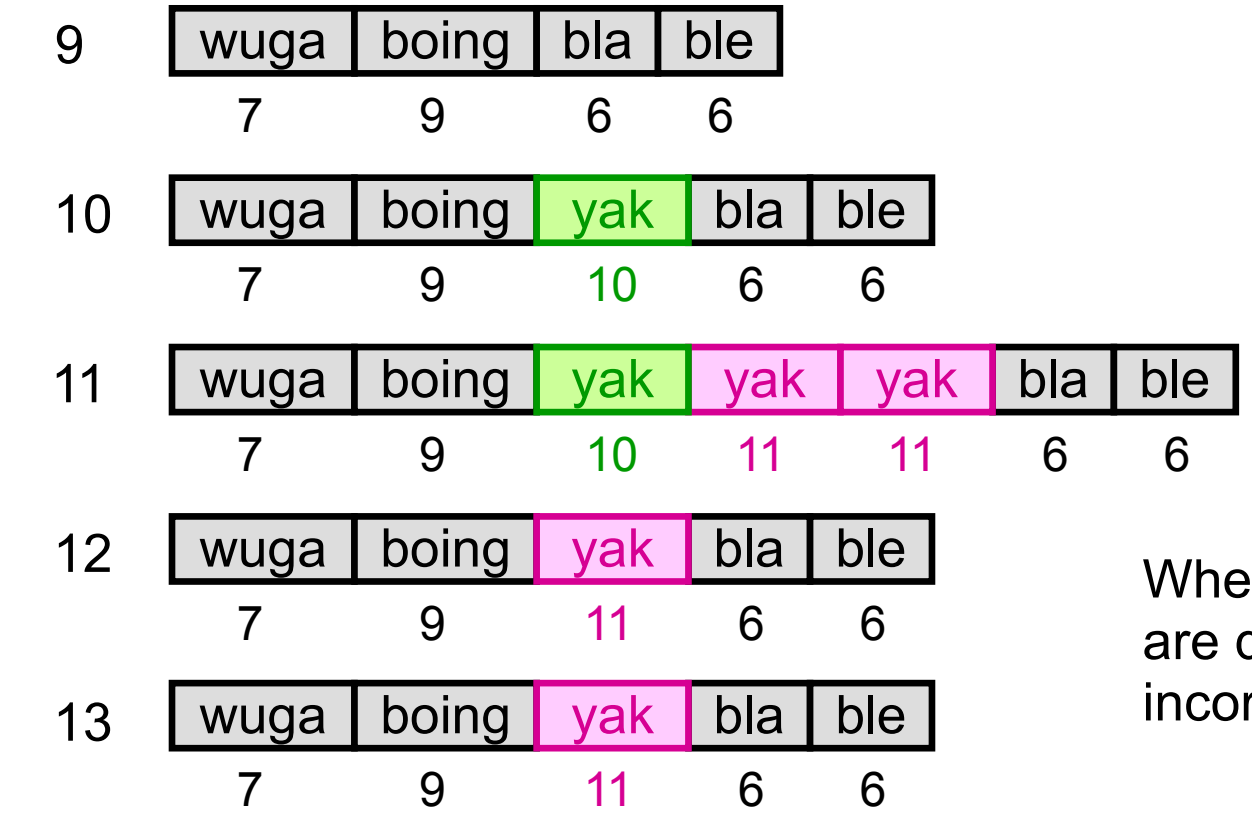
Text matching: Greedy longest-match first



We match in greedy fasion, longest subsequences first.

We are not sure this is optimal, and we are intrigued by computational biology algorithms, which try to compute first a match that overlooks small errors, and gets the big picture right, then refines the match. (anyone interested in writing this?)

Text tracking: Duplication

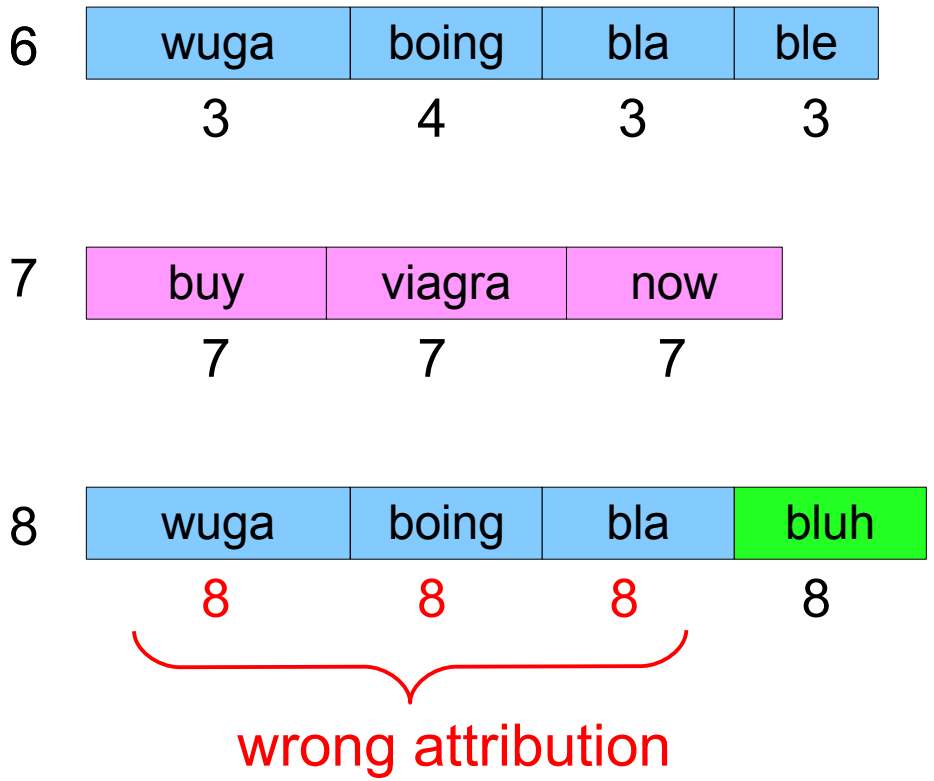


Contribution

Duplication

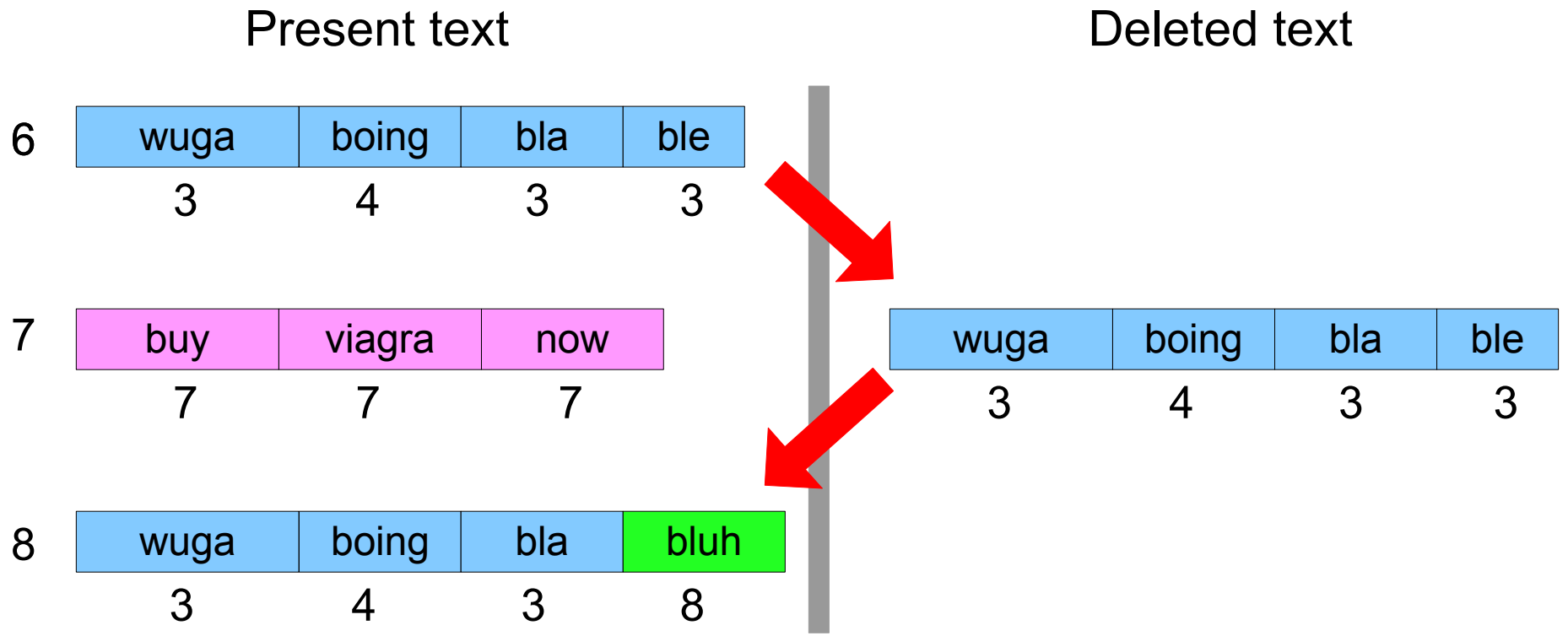
When the duplications are deleted, text may be incorrectly attributed.

Tracking text: Deletions and Re-insertions



Normal text difference tools compare one revision with the next only. This can create attribution problems for text that is deleted and re-inserted.

Tracking text: Deletions and Re-insertions



Deleted text should be tracked, so that it can be correctly attributed if it is reinserted.

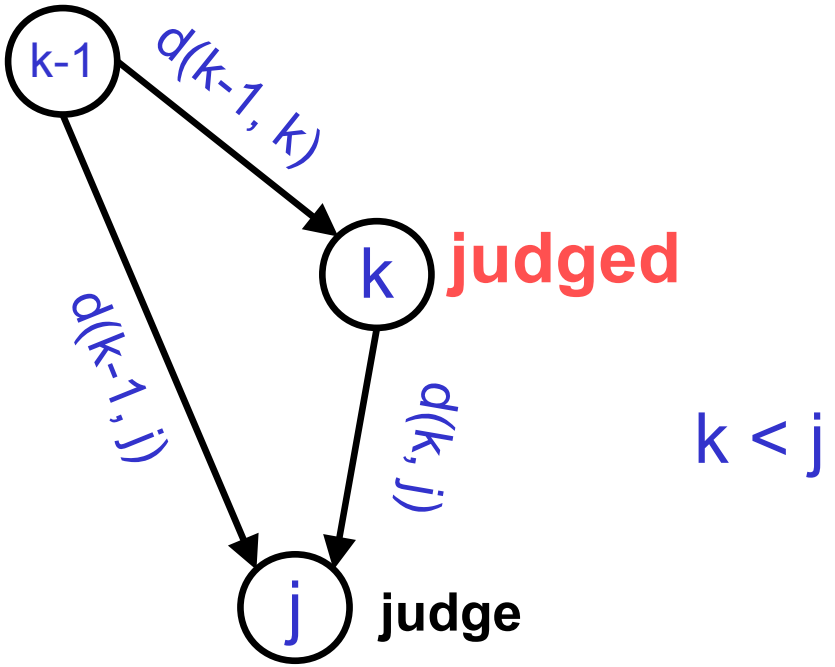
Measuring author contribution

Several methods have been proposed; we have a survey paper in WikiSym 2008 which also contains the approach we favor.

Some of the approaches proposed:

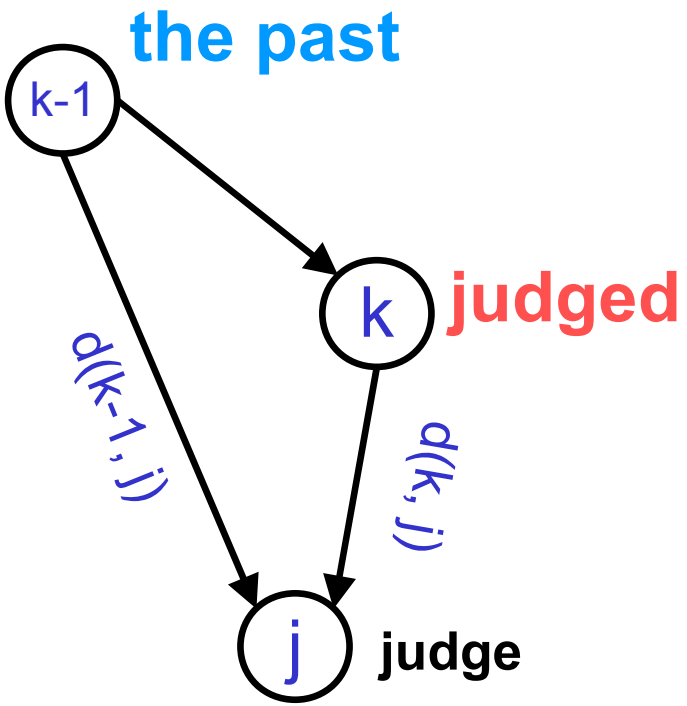
- **Text present in current revision.** Problem: it disregards the contributions given earlier on in the history of a page.
- **Total text added.** Problem: to spam it, let me just add a revision consisting of 1,000,000 times the word “spam”.
- **Word days.** This is a measure invented for this tutorial: if you contribute n words, which are displayed for m days, then you get mn word-days. Problem: it does not take into account the work of the people who revise / refine / rearrange the text. (Is this really bad??)

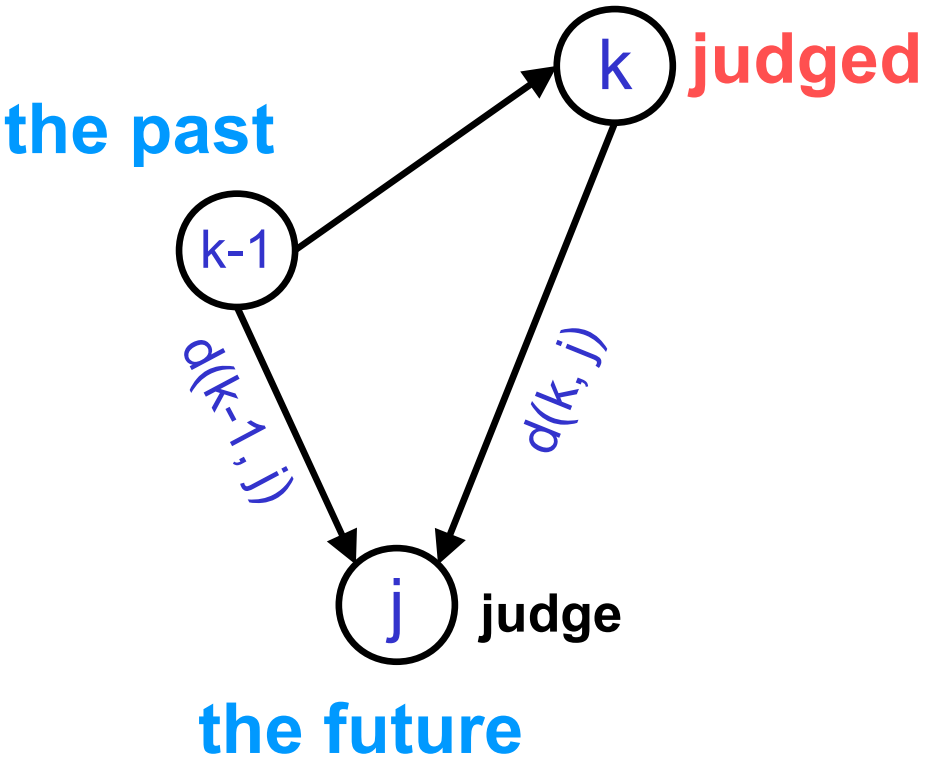
Measuring author contribution



We compute the edit distance between versions k-1, k, and j, with k < j (see later for details on the distance).

Good or bad contribution?





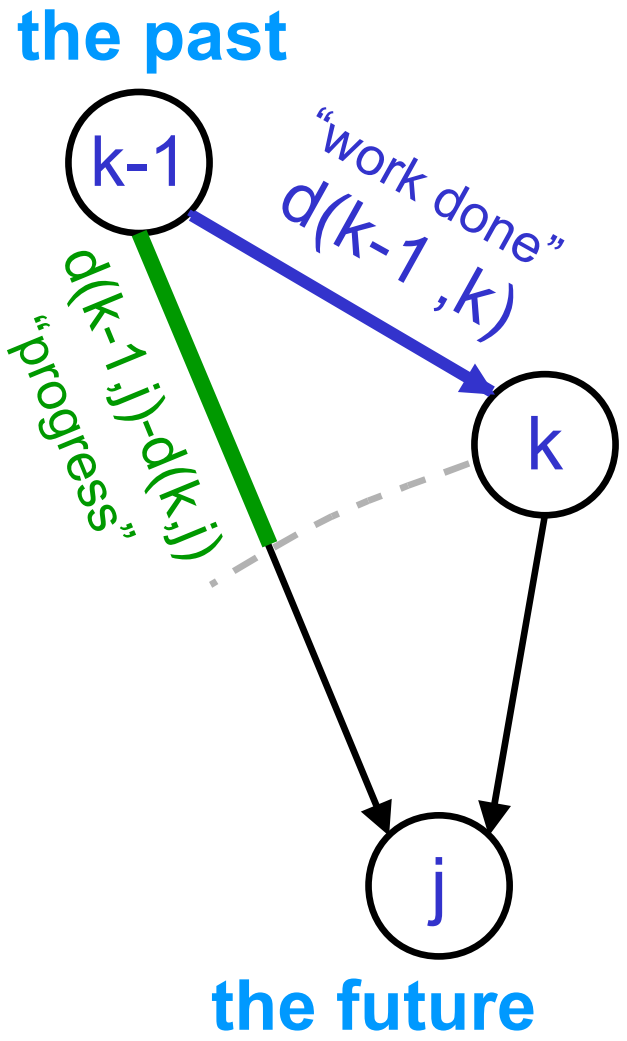
k is **good**: d(k-1, j) > d(k, j)

"k went towards the future"

k is **bad**: d(k-1, j) < d(k, j)

"k went against the future"

Edit quality



Edit quality:

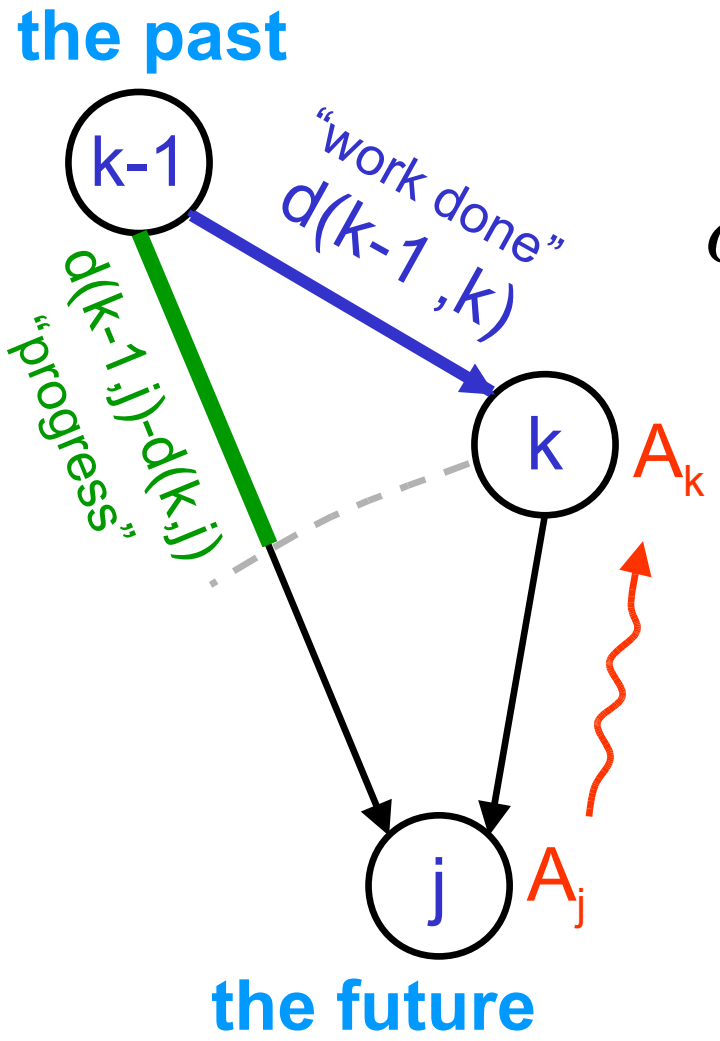
$$\alpha_{edit} = \frac{d(k-1,j) - d(k,j)}{d(k-1,k)}$$

The fraction of change that is in the same direction of the future.

- α_{edit} 1: k is a good edit
- α_{edit} -1: k is reverted

Corollary: we can detect reversions automatically!!

User contribution: our proposal



Edit quality:

$$\alpha_{edit} = \frac{d(k-1,j) - d(k,j)}{d(k-1,k)}$$

User contribution:

$\alpha_{edit}\ d(k\text{-}1,\ k)$

The quality of the edit, multiplied by the size of the edit.

Computing edit distance

We compare an old version u and a new version v, and we compute:

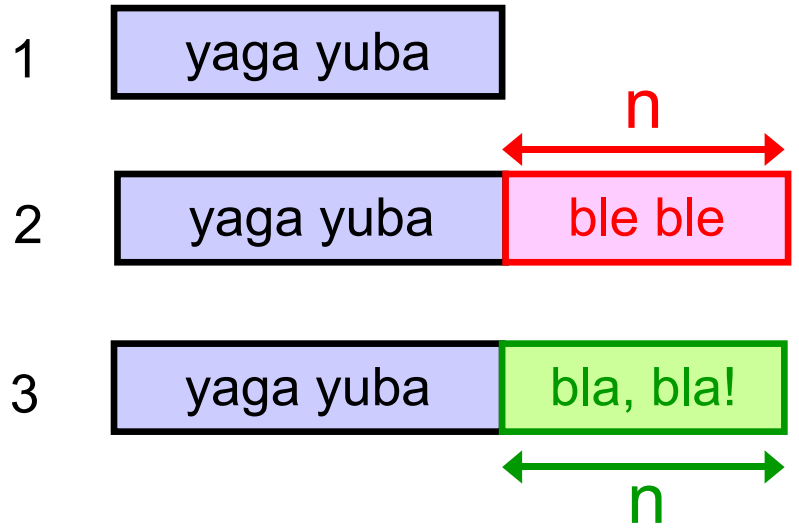
- I: Total inserted text
- D: Total deleted text
- M: Total amount of relative exchange of position (a measure of text reordering).

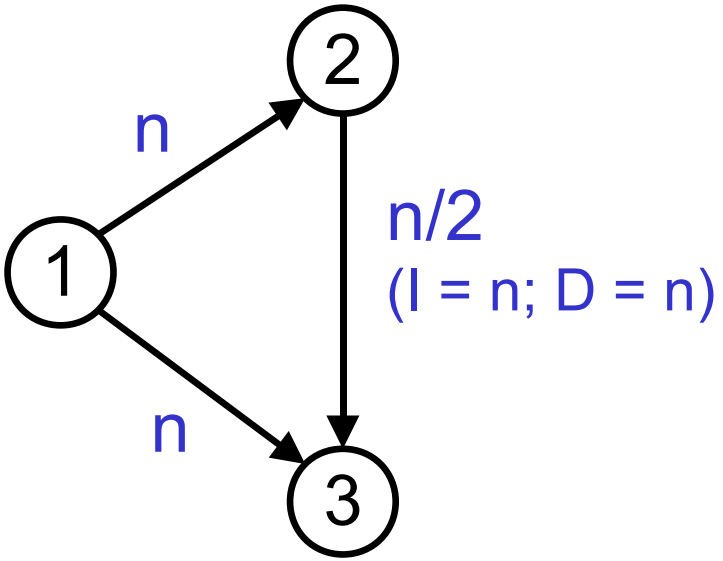
$$d(u,v) = \max(I, D) - \tfrac{1}{2} \min(I, D) + M$$

Computing edit distance

$$d(u,v) = \max(I, D) - \tfrac{1}{2} \min(I, D) + M$$

Dealing with rewrites:





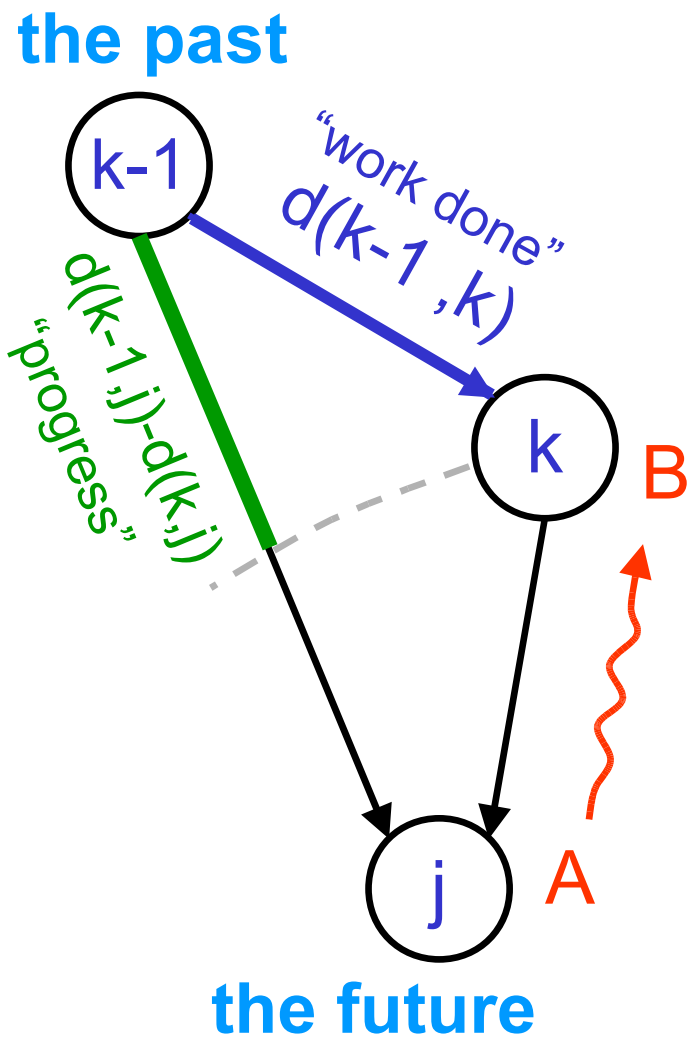
For the author of version 2:

$$\alpha = \frac{n - n/2}{n} = \frac{1}{2}$$

Detecting instabilities and reversion wars

- We use a sliding window over the revision history, and we look for windows where many edits are of negative quality.
- Very effective! Among the results for italy:
  - Partito della libertà
  - Barolo
  - ...
- Can this be used as a community alert tool?
- Should one look into which topics, in each culture, generate the most quarrels? “comparative quarrelletics”?

Extracting a social graph of collaboration



- The edit quality enables us to compute a feedback from user A to user B.
- A graph of user-to-user feedbacks can then easily be extracted.
- Groups that collaborate, or groups that are quarrelsome, can easily be spotted.

Part II

Working with large Wikipedias

… a few lessons we learned

Organizing the work

It's a big monster, and you will have to tame it.

Cut it into pieces.
Parallelism, decomposition will be your friends.

Split!

The dumps come in single compressed files. The English Wikipedia would expand to 1.5 TB? The Italian one to 500GB?

A single large file is useless:

- You cannot parallelize the computation
- Even if you do it serially, it takes too long: how do you restart if something goes wrong?

Recommendations:

1. **Split the file immediately into small chunks.** How small? Balance:
   - Large chunks: waste more time if something breaks
   - Small chunks: too many chunks, heavy on filesystem
2. **Make a point to keep it always compressed.** A decompressed format does not make sense, not even from the point of view of raw speed, given the CPU/disk speed differences.

If something can go wrong, it will

It's a lot of data, and it will contain everything possible.

- You think of taking the text, and splitting it along whitespace to separate in words?
  - Make sure you can do that... somewhere in there lurks a million-word revision... some string functions blow up!
- You think of making a hash table, that associates to each 3-gram (triple of words) the places where it occurs in a revision?
  - Check the implementation of hash tables. There are revisions in which the same trigram occurs 100,000+ times.

Design algorithms that are resistant to the worst case.

- **Getting things wrong on the very rare worst case may be ok.**
- **Breaking is not.**

Corollary: make sure you can restart

You will encounters lots of errors. So:

- Make sure that, in a run, you have a way of pinpointing in which revision of which page things break.
- Make sure you can re-run just on that (see before on splitting).
- Make sure that you are using an environment that makes debugging possible and pleasant.
- Make sure that the smallest possible fraction of work is wasted, if things go wrong (see before on splitting).

It took us months before we had code that could, glitch-lessly, process the whole English Wikipedia.

Don't be lazy! Devise scalable approaches

Case study 1: WikiTrust text analysis

- The “online” system, whenever a revision arrives, updates user reputation, and analyzes text.
- **Problem**: not parallelizable: if you work on blocks of pages in isolation, you don't know the reputation that users gain working on other pages.
- **Solution**: A three-pass process:
  1. Analyze blocks of pages to discover how much work each user has done (parallelizable).
  2. Put together work summaries, and compute user reputation (single-thread, but fast).
  3. Using the user reputations, analyze the blocks of pages (parallelizable).
- **Big effort, but big win!** A whole Wikipedia analysis in one day! x10 speedup!

Don't be lazy! Devise scalable approaches

Case study 2: WikiTrust revision storage

- **Problem**: The revisions cannot be stored in a DB, and need to be stored in the filesystem.
- **Lazy solution**: Store individually compressed revisions in the filesystem, using a tree with ~100 branching.
- **Disaster!** on multiple levels:
  - Too many files (200M); du -sh takes days
  - Too much disk traffic, fragmentation
- **The outcome:** We had to re-do it from scratch, moving to a more sophisticated scheme where we store compressed blobs containing consecutive revisions of pages, and we use the DB to keep an index of the blobs.
  - We wasted a lot of time by being lazy.

If you can do something once only, then you cannot do it at all

You might come up with a way of getting a job done that you can just barely do, on a once-only basis (e.g., a job that lasts a week). You think this is a viable approach.

But it is not!

- (Almost) nothing works the first time. Something will break, and you will need to restart.
- Once it finishes, you will realize that you really needed to do it in a slightly different way.

Corollaries:

- Nothing is done “only once”.
- Only things that can be done in routine fashion, can be done at all.

You will hit limits you don't know...

Be aware of filesystem, data base, networking limitations.

Example: you are now convinced you need to split the input file into smaller chunks.

- One file per revision? 200 million files?? (fragmentation, file system will get you)
- One file per page? 20 million files?
- One file per 100 pages? 200,000 files?
    - A directory with 200,000 files? Don't try that!

Once, we unpacked with much sweat all the information, only to realize that the command “du -sh” to measure the total size of all files would have taken *4 days!!*

You cannot store revisions in the database

You cannot. It will blow up. It takes too much space, and does not allow you to have good control on compression. Not even the Wikimedia Foundation, with their hundreds of servers, does it.

- The best, if possible, is to work on the dumps, avoiding storing individual revisions.
- If you do have to store individual revisions, you will have to implement (or reuse) a compressed revision store:
  - Group (some number of) revisions of a page, and compress them together.
  - Keep the revision index in a DB or similar, to know where to find them.

The language, oh the language!

Working with the Wikipedia markup language is a nightmare:

- There is no grammar for it, not even a complete description in any single place.
- Parsing is *not* grammar based, and is very contrieved: try """***this is bold and italic'''*** *and this only italic*".
- The rule on the Wikipedia: “If it looks fine, it is fine”.
- Computer science grammars are good at encoding the *correct* sentences. Most of the complexity of parsing markup language consists in knowing how to recover from errors.
  - “Syntax error on line 213” is not an accepted output!
  - You must recover from errors in the same way as Mediawiki!

The language, oh the language!

It is really more similar to a natural language than to a computer language:

- Irregular.
- Guided by tradition and use: if it looks fine, it is fine.
- Very similar to situation for HTML on the Web.

Solutions:

- If you can, avoid parsing!
- If you must, can you get Mediawiki to do it for you? (but php is slow)
- If acceptable, give up on goal of being 100% correct.
- Otherwise, be prepared to suffer.

Text.ml, where the parsing is, is the most regular source of problems in WikiTrust, in spite of being some of the oldest and conceptually simplest components.

On data

Computer scientists focus on code.

Scientists focus on data.

You will have to focus on both.

What is the problem? Scenario #1

You have finished implementing and debugging your super-fancy algorithm that computes the conceptual correlation between the French and the German Wikipedias.

You turn the handle, and out pops the number: 0.7364
(or worse, tables chock full of numbers).

And you suddenly realize...

- How on Earth are you going to decide whether 0.7364 is due to:
  - A bug
  - An artifact of how you do the analysis
  - The correct result
- Which one of the above is the *least* likely for the first run?

Solution #1: test sets, alternative algorithms

- Can you compute something in two different ways? Can you approximate the results in another way? This can help validate a computation.
- If not, can you inject a change in the input, and validate that the difference in the output is the correct one?
- Invest in tools that help you compare results.
- Create small test cases, and test obsessively.
- Freeze small test cases, and version them with your code.

What is the problem? Scenario #2

After *much* sweat, you believe you understand what your algorithm is computing.

Now you make a change in the algorithm – a change that affects a bit how things are computed.

You run it, and get some results.

How can you check that the change is correct?

- Do you still have the results before the change?
- Can you run the code before / after the change on similar input data?
- If your code outputs millions of stats / numbers / ... , how are you going to compare the output before and after?

What is the problem? Scenario #3

You did great work in April, stored a file called simil_measure.txt containing all your treasured results.

Now, you have what you think is a better idea, and you implement it. Somehow, the results are ... different.

You would like to investigate, but you cannot reproduce your earlier results.

Can you still tell:

- What version of the code produced simil_measure.txt?
- What input did you give to that code?
- Were there any command-line parameters? Which ones?
- Do you still have the input data around?

Solution #2:
Version the data, not only the code

- Code versioning is an accepted practice.
- But data versioning is equally important!
- In classical sciences (biology, physics, ...) lab notebooks are used – there is a whole discipline for keeping track of “data”. In computer science, this is a skill which is not taught.
- Data, due to the size, is also difficult to version.

Our solution: We wrote an infrastructure that annotates every data file with:

- The version of the code used to produce it.
- The version, and details, of all the input data used to produce it.
- Any parameters of the process that generated the data.

The notes are stored in XML. See the module Fileinfo.

On languages and code

Programming languages are something on which everybody disagrees.

They are also one of the big factors in getting things to work while preserving your sanity.

Facts, and consequences

- Wikipedias are very large
  - It pays off to use a very efficient language; interpreted languages are suitable for small tasks only.
- Text analysis involves working with lists (of words, revisions), strings, trees (of xml), variable-sized text, ...
  - You need to use a language with good datatype support and excellent memory management.
- The datasets are so large, you cannot afford running for 2 hours to have the code break. You need to have some confidence that, when you make a change, it is going to work.
  - You need to use languages with strong typing, and excellent compile-time checking.

Our choice: Ocaml

We started in Python: this made it easy to experiment, is simple and concise, very pleasant.

But after a while, we discovered two problems.

Problem 1:

- If you don't test it, you don't know whether it will work.
- In Python, errors propagate, and you need to test the whole thing at every change. So, if your system is of size M, and you do a change of size n, you need to debug M + n code.

Consequence: development time is quadratic in the size of the system.

Problem 2:

- Memory leaks. Python's memory management was not perfect.

Our choice: Ocaml

(hey, let me digress, this is at OOPSLA after all)

We came up with a wish list of features:

- Strongly typed, to avoid having to re-test everything every time.
- Excellent memory management
- Compiled
- Concise
- Great support for advanced data structures
- Libraries for mysql, xml, http, marshaling data to disk, ...

There aren't so many programming languages that fulfill these all. Java is out (slower, not strongly typed), C (no memory management), C++ (yuck! not concise, not strongly typed, baroque memory management, baroque STL), Python, Ruby...

- People tell me Haskell is also a great choice; I don't know.

On code organization

- Divide computation into small, individual tasks, which can be executed multiple times if needed.
- Make sure each tasks fails gracefully in presence of errors.
- Make sure each tasks improves the situation.

Goal: contain and reduce errors.

Part III

Building your analysis on top of WikiTrust

WikiTrust

WikiTrust processes edits in batch or in real-time, and computes author reputation, text trust, text author, and text origin.

WikiTrust -- structure

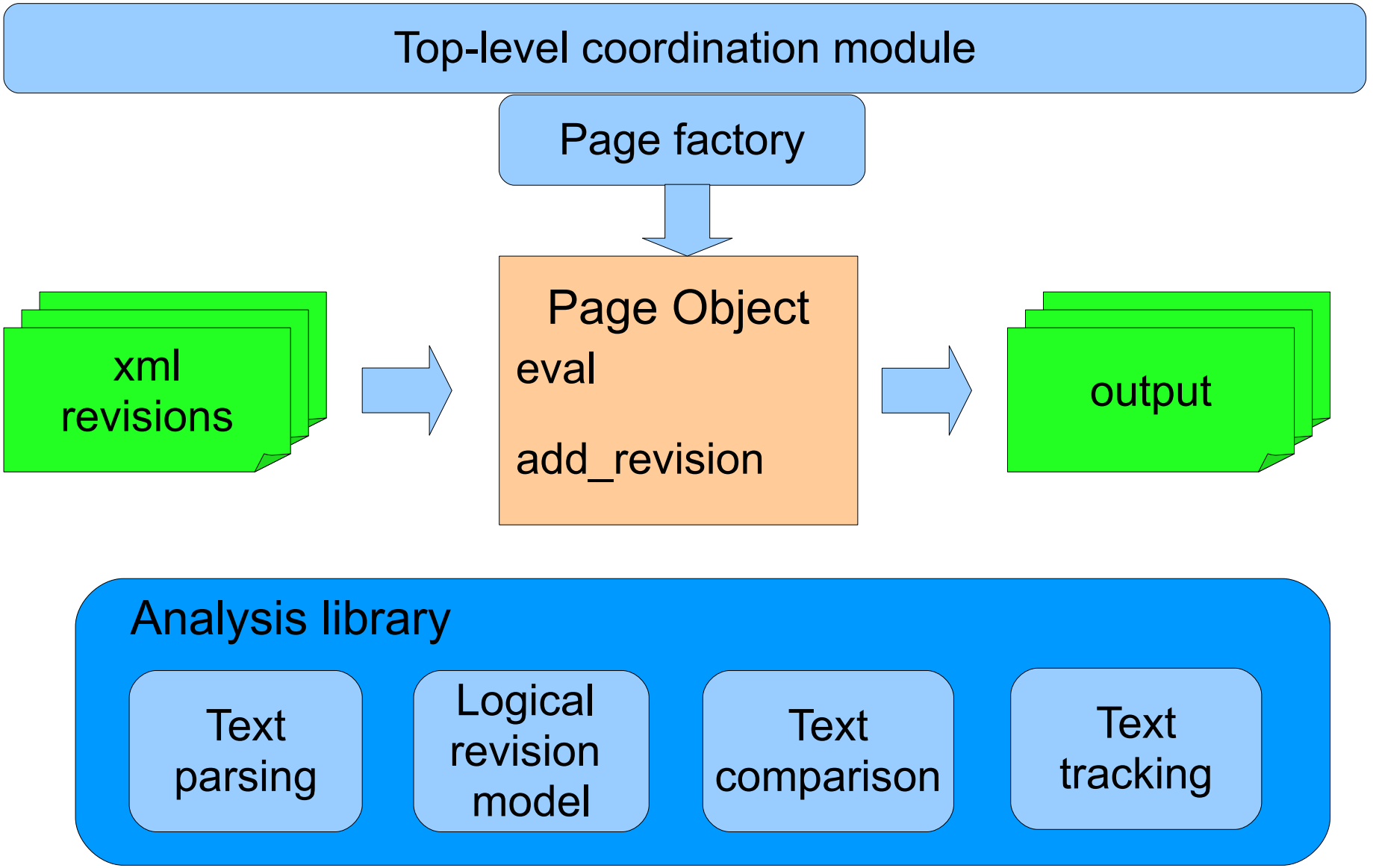
WikiTrust has two modes of working:

- **Online**, as a MediaWiki extension. Whenever a user makes an edit, it updates text trust, origin, and authorship, as well as user reputations. It can then color the text, etc.
- **Batch**, as a series of tools that perform various text tracking and analysis tasks.

General structure of batch mode:

- Input: a compressed xml file containing some pages.
- Output: one, or if you really want, a few, files for each input file.
- Perfectly suited for parallel analysis.

WikiTrust – batch mode logical structure



WikiTrust – Text parsing

Text parsing produces two types of output:

- A list of words, renormalized (lowercased, no punctuation)
- A list of *seps,* which include both words, and syntactic markers (title beginning, title end, bullet, paragraph separator, ...)

The list of seps is a superset of the list of words.

The two lists are cross-linked; you can always go from a word to the corresponding sep, or from the sep to the word (if any).

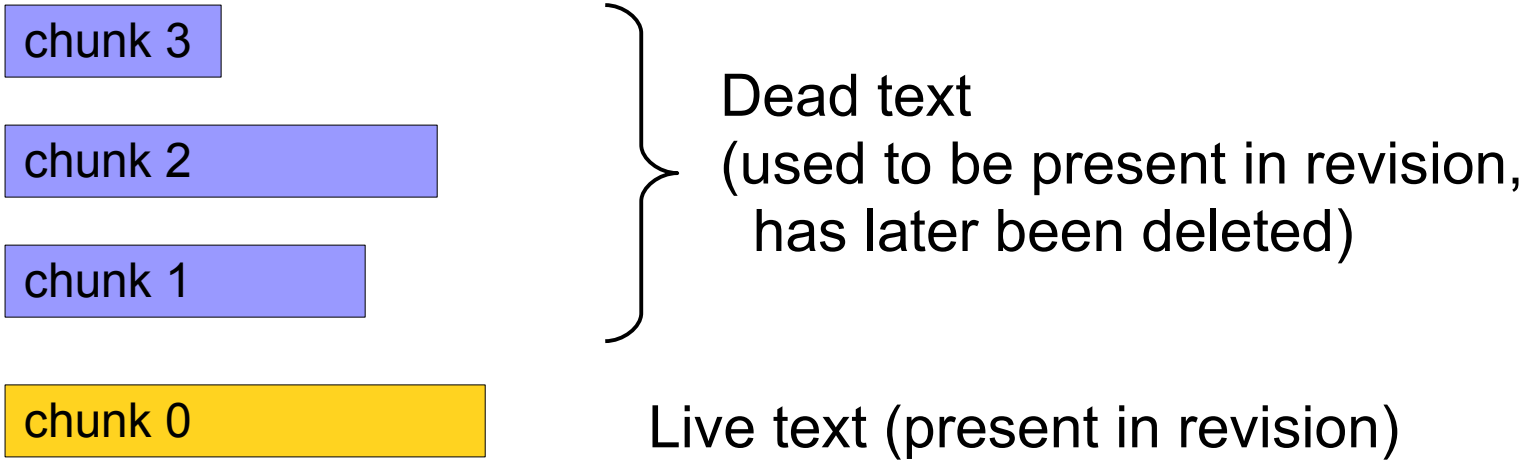
WikiTrust – Text comparison

Chdiff.edit_diff word_array_1 word_array_2 describes the difference between word_array_1 and word_array_2 in terms of a list of:

- Del (k, n) Deleted n words at position k in word_array_1
- Ins (k, n) Inserted n words at position k in word_array_2
- Mov (k, m, n) Moved n words from position k in word_array_1 to position m in word_array_2

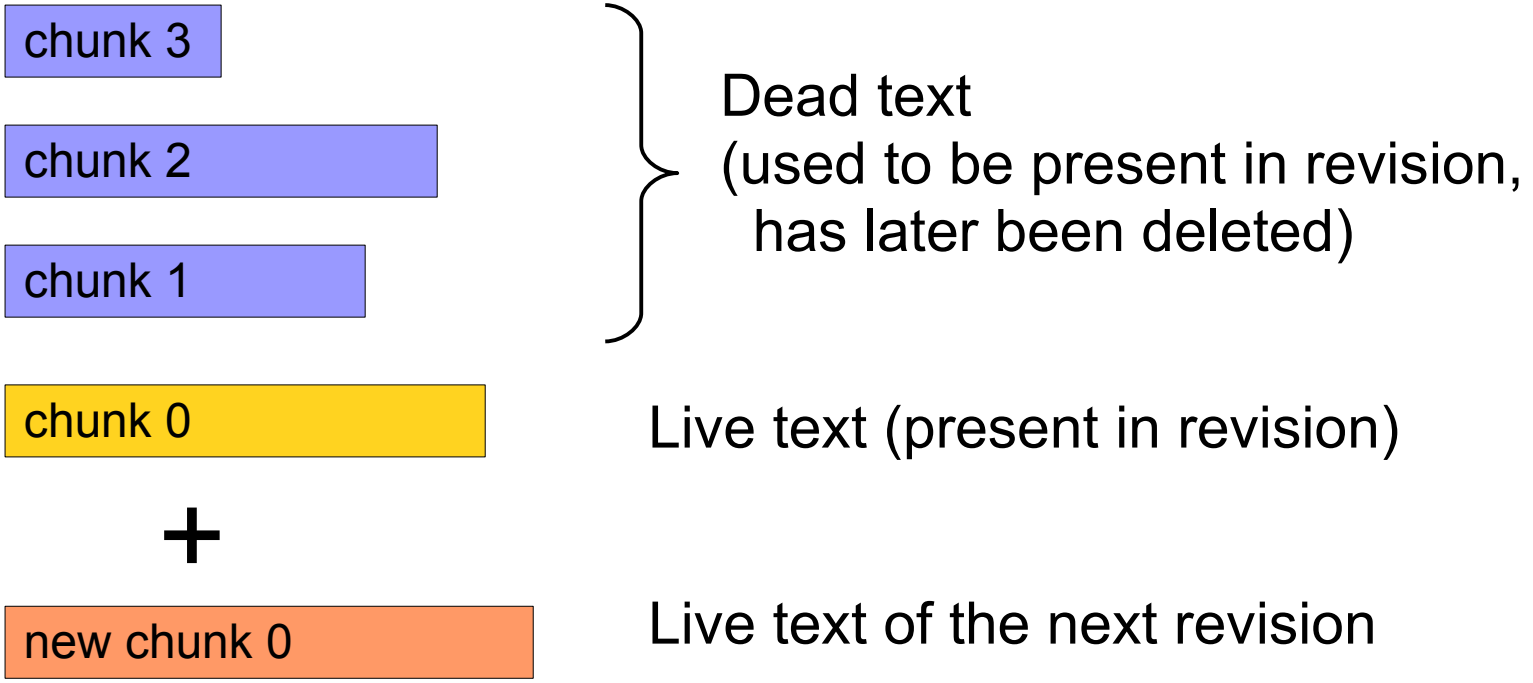
This information enables the computation of edit distances, and also, to find the previous revision most similar to the current one.

WikiTrust – Text tracking

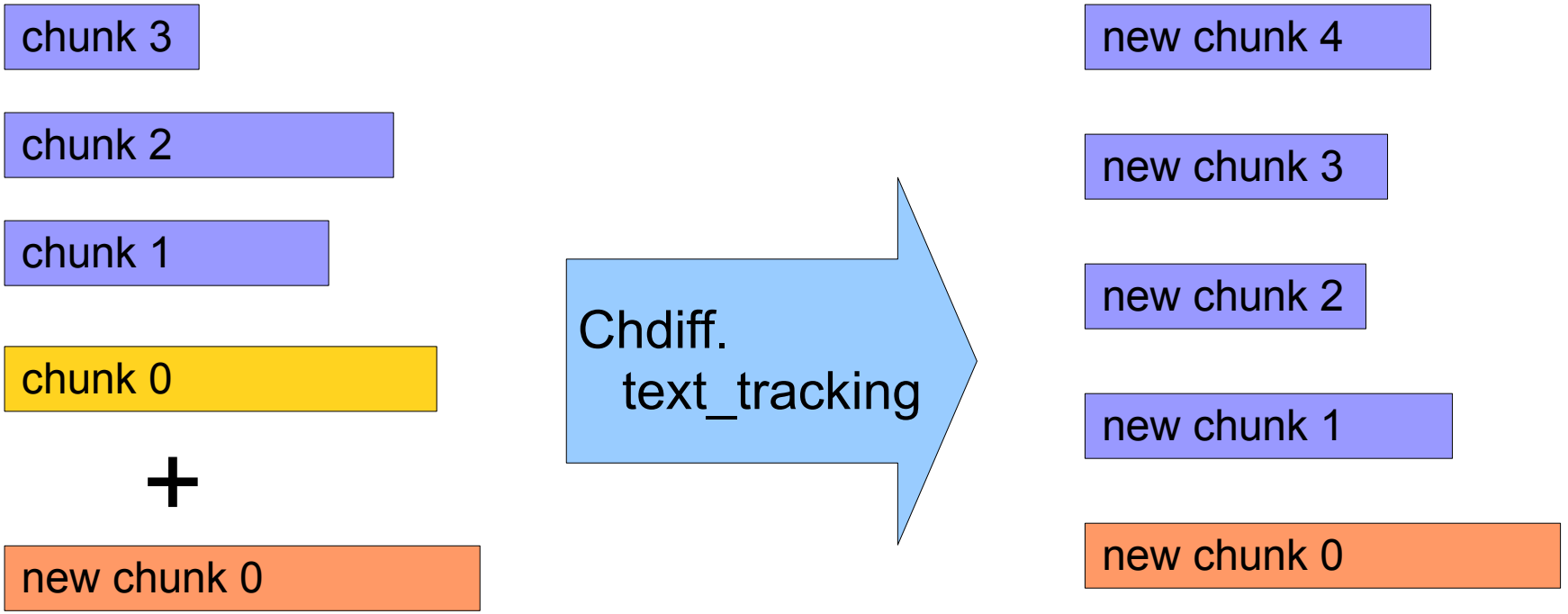


Wikitrust tracks both the text present in a revision, and the text that used to be present in the revision history, but has been subsequently deleted.

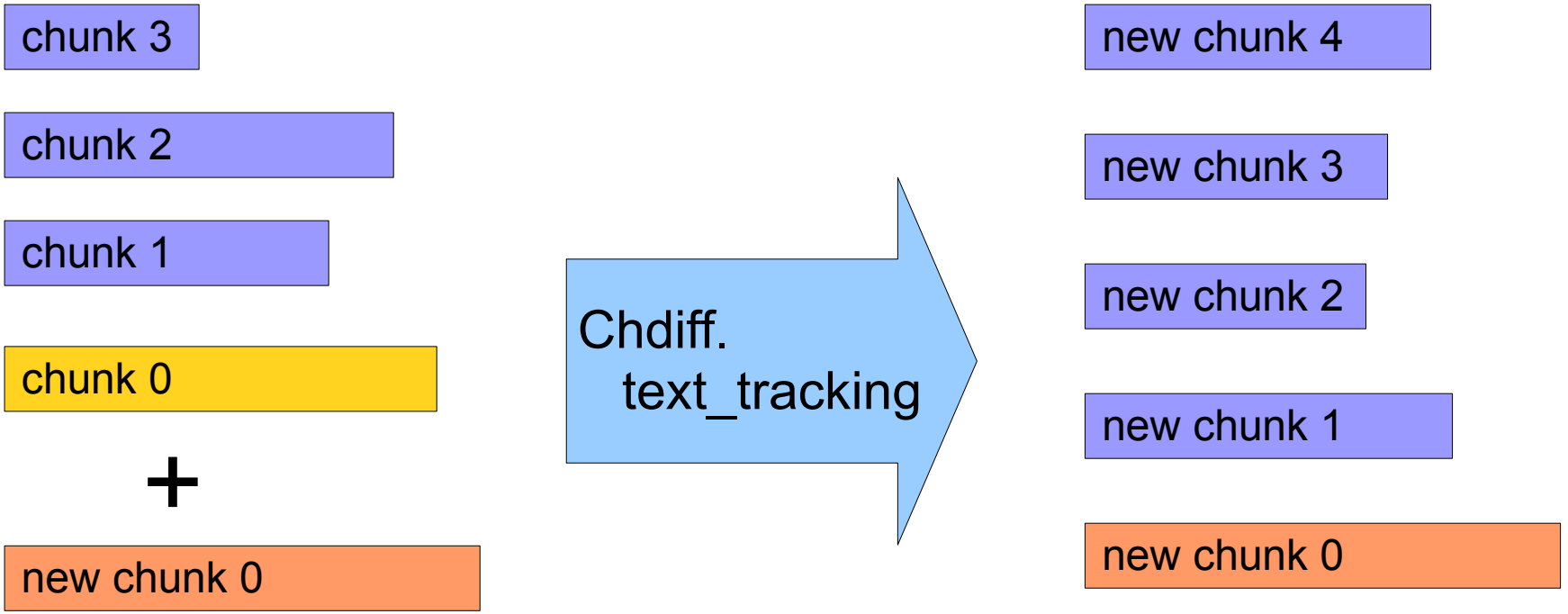
WikiTrust – Text tracking



WikiTrust – Text tracking



WikiTrust – Text tracking



- Mins (k, n) Insert n words at position k of chunk_0
- Mdel (k, c, n) Deletes n words from position k of chunk_n
- Mmov (k, c, m, d, n) Moves n words from position k of chunk_c to position m of chunk_d

Recipe to add a new analysis

To make a new type of analysis, you need to do only two things:

- Create a new subclass of Page, where:
  - method add_revision adds a revision
  - method eval signifies there are no more revisions, and does any last processing.
- Modify page_factory.ml, adding an option to produce objects of the new subclass, whenever a page is encountered.

Example: Let's see how to add an analysis that computes, for each user, the sum of the “live time” of all the words introduced by the user. (The “live time” is the time for which a word is shown).