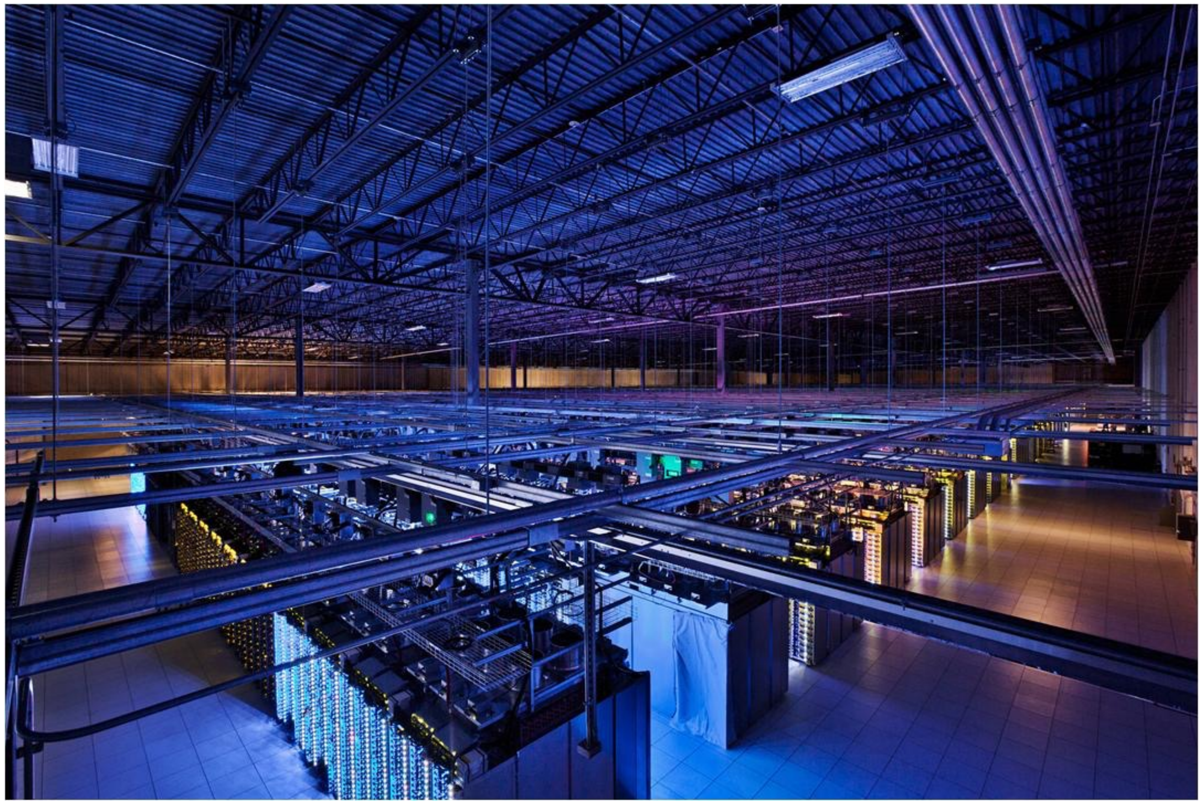
CWI

Centrum Wiskunde & Informatica

Large-Scale Data Engineering

Some notes on Access Patterns, Latency, Bandwidth



+ Tips for practical

event.cwi.nl/lsde

CWI
Centrum Wiskunde & Informatica

Assignment 1: Querying a Social Graph

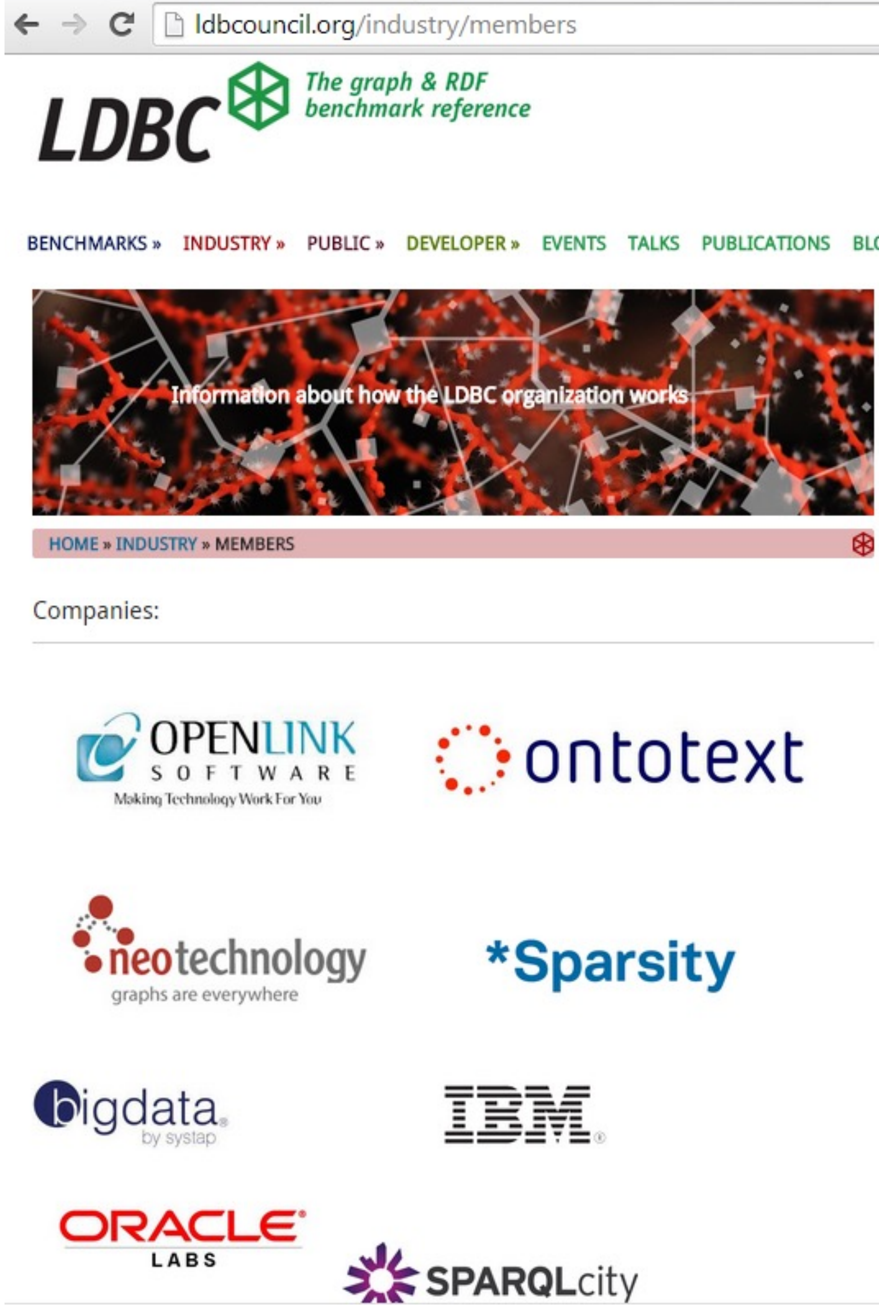
event.cwi.nl/lsde

LDBC Data generator

- Synthetic dataset available in different scale factors
  - SF100 🡐 for quick testing
  - SF3000 🡐 the real deal
- Very complex graph
  - Power laws (e.g. degree)
  - Huge Connected Component
  - Small diameter
  - Data correlations

    Chinese have more Chinese names
  - Structure correlations

    Chinese have more Chinese friends


ldbcouncil.org/industry/members

LDBC The graph & RDF benchmark reference

BENCHMARKS » INDUSTRY » PUBLIC » DEVELOPER » EVENTS TALKS PUBLICATIONS BL

Information about how the LDBC organization works

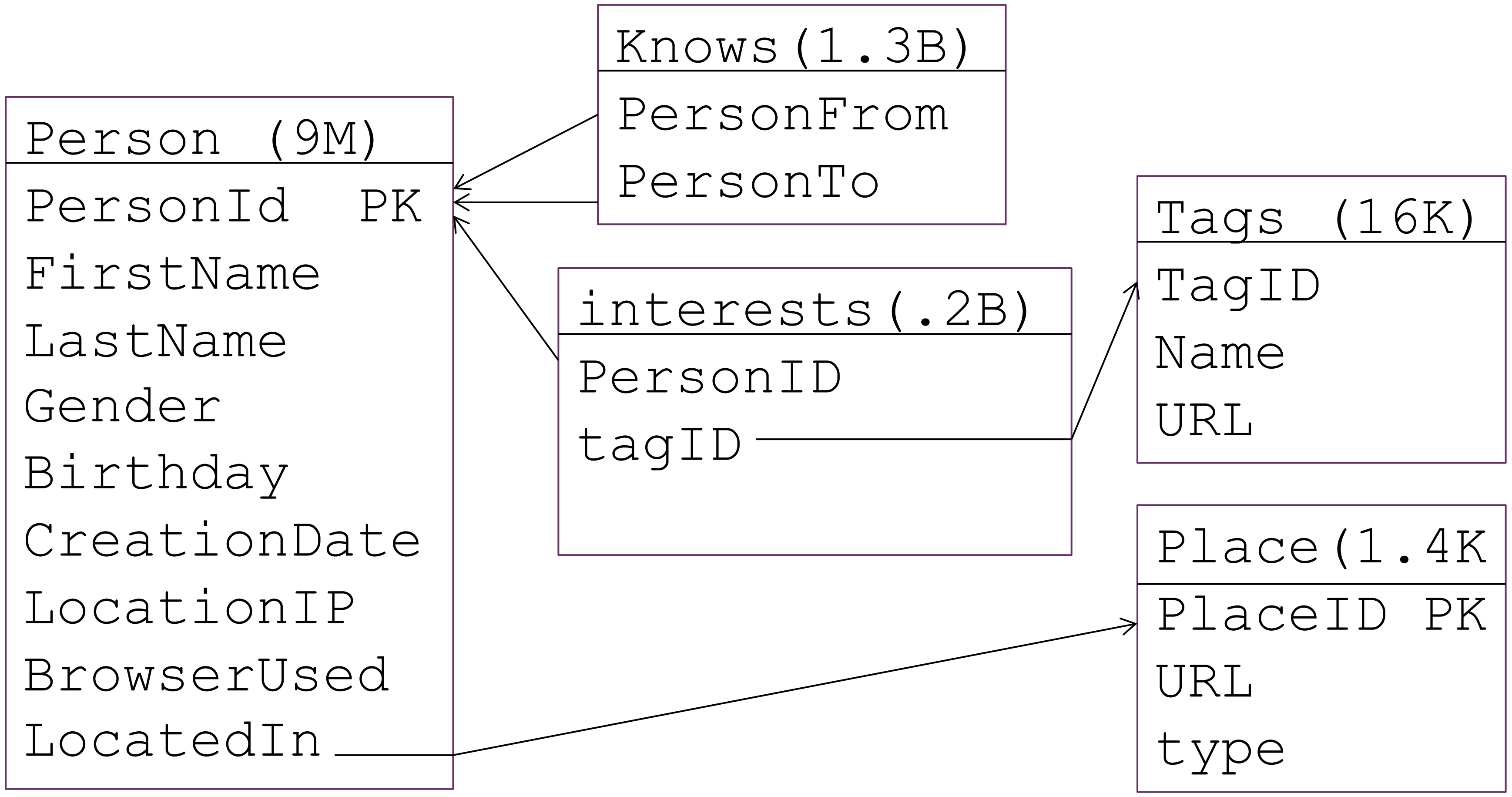
HOME » INDUSTRY » MEMBERS

Companies:

OPENLINK SOFTWARE Making Technology Work For You · ontotext · neotechnology graphs are everywhere · *Sparsity · bigdata by systap · IBM · ORACLE LABS · SPARQLcity

CSV file schema

- See: https://event.cwi.nl/lsde/data (sf100 only)
- Counts for sf3000 (total size: 37GB CSV, 7GB bz2 compressed)



The Query

- The marketeers of a social network have been data mining the musical preferences of their users. They have built statistical models which predict given an interest in say artists A2 and A3, that the person would also like A1 (i.e. rules of the form: A2 and A3 ➔ A1). Now, they are commercially exploiting this knowledge by selling targeted ads to the management of artists who, in turn, want to sell concert tickets to the public but in the process also want to expand their artists' fanbase.
- The ad is a suggestion for people who already are interested in A1 to buy concert tickets of artist A1 (3 for the price of 2!) for your birthday celebration birthday to invite two of your friends ("who we know will love it" - the social network says), who are also friends themselves, who live in the same city, who are not yet interested in A1 yet, because they are interested in other artists A2, A3 and A4 that the data mining model predicts to be correlated with A1.

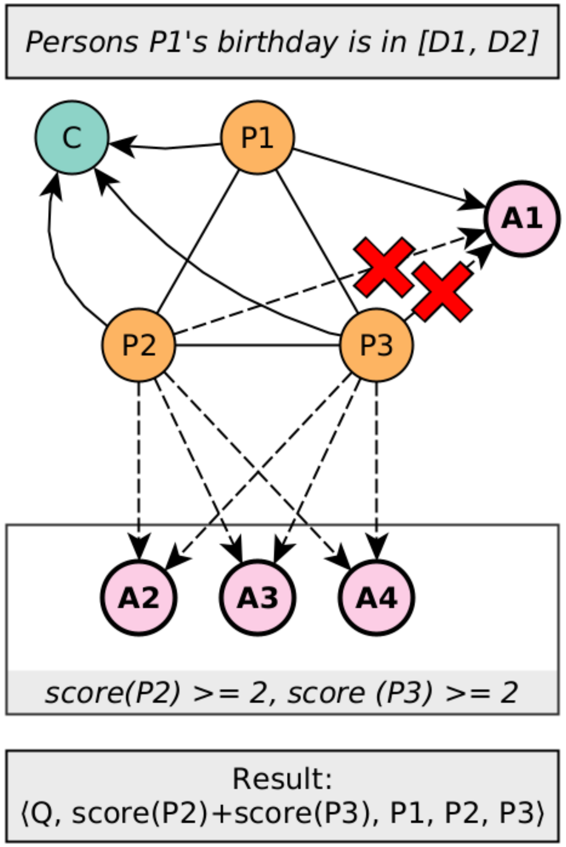
The Query

For all persons P1:

- *who have their birthday on or in between D1..D2*
- *who like A1 already*
- *who have friends P2 and P3, who live in the same city as P1*
  - *where P2 and P3 are themselves also friends*
  - *who do not like A1 yet, but have a score >=2, where we give a score of*
    - *1 for liking any of the artists A2, A3 and A4*
    - *0 if not*

    the final score, the sum, hence is a number between 0 and 3.

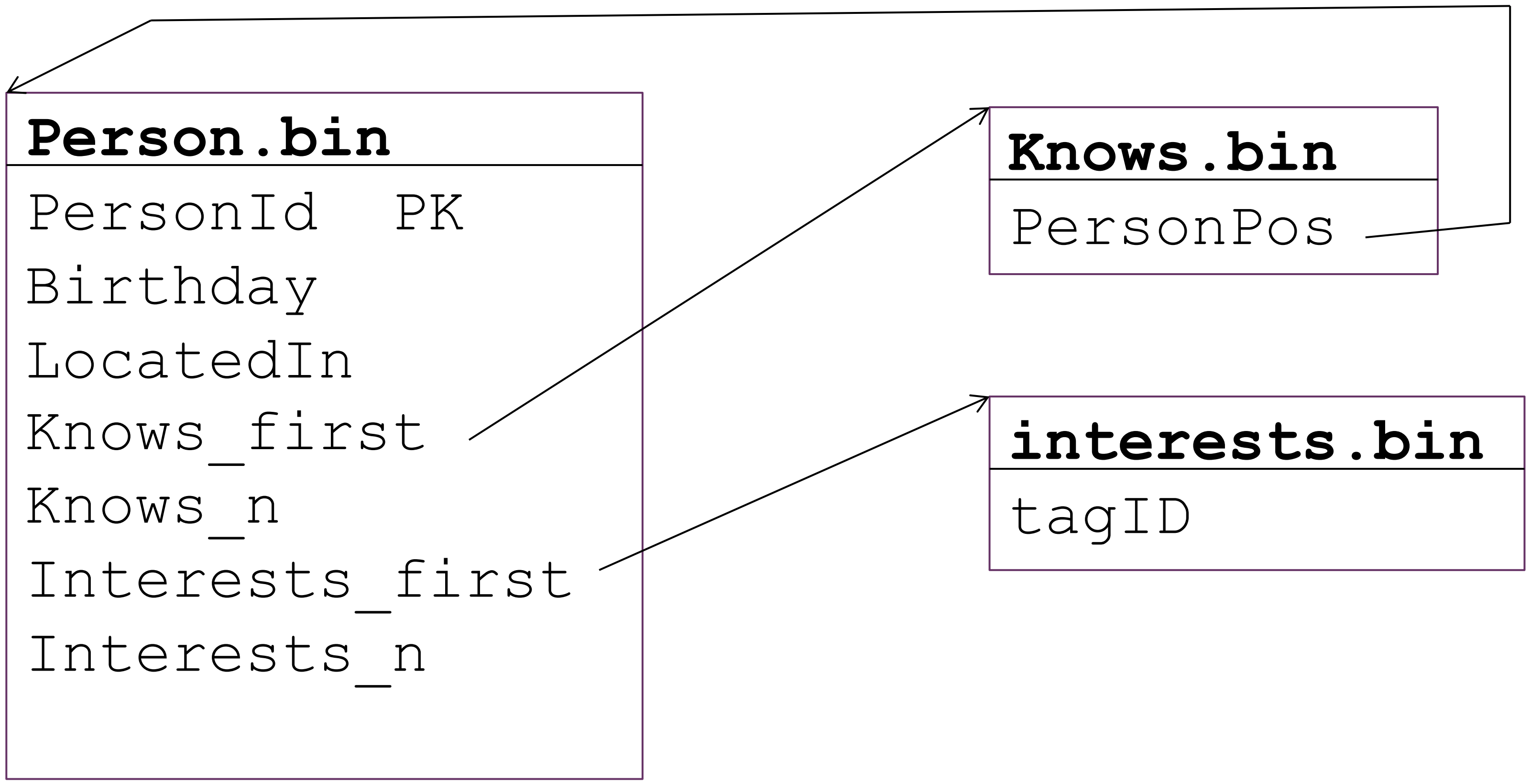
The answer of the query is a table (score, P1, P2, P3) where score is the sum of the scores for P2 and P3; ordered on all columns


Persons P1's birthday is in [D1, D2]
C
P1
A1
P2
P3
A2
A3
A4
score(P2) >= 2, score (P3) >= 2
Result:
⟨Q, score(P2)+score(P3), P1, P2, P3⟩

CWI
Centrum Wiskunde & Informatica

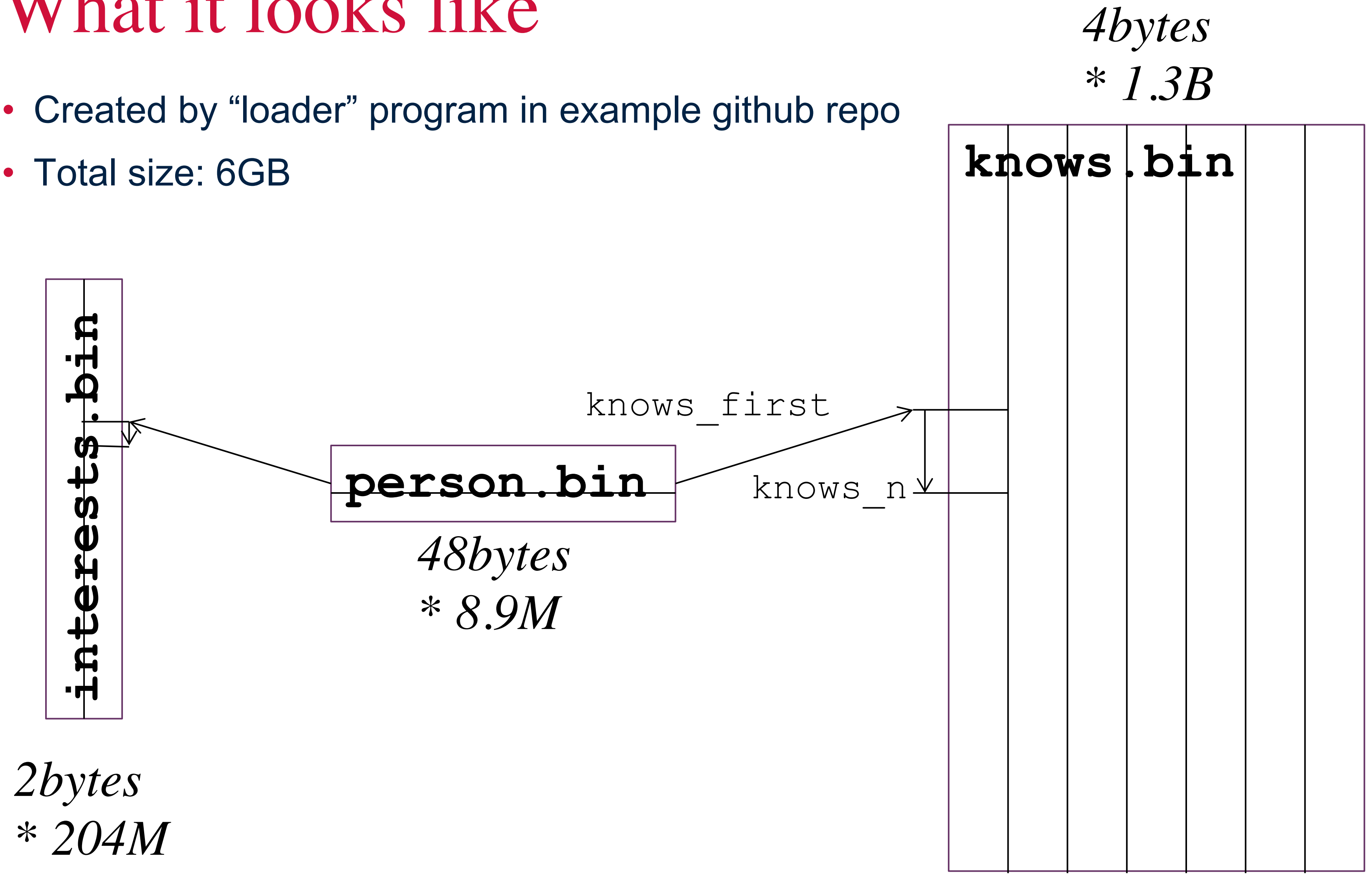
Binary files

- Created by “loader” program in example github repo
- Total size: 6GB



What it looks like

- Created by “loader” program in example github repo
- Total size: 6GB



The Naïve Implementation

The "cruncher" program

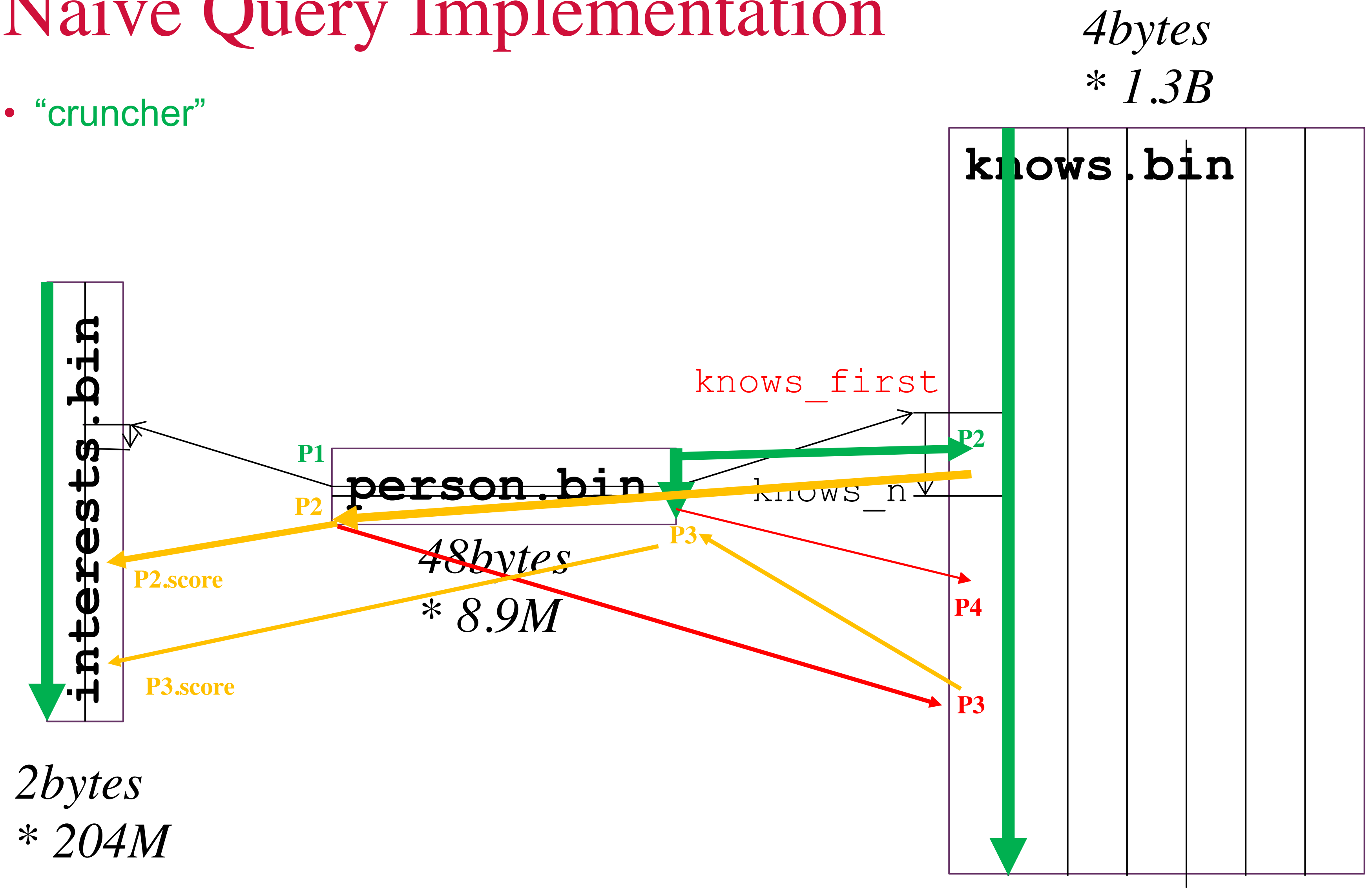
Go through the persons P1 sequentially

- *check whether P1's birthdate is in range D1..D2*
- *check in interests whether this person likes A1, if so*
- *visit all friends P2 of P1, for each:*
    - *Check in the person data that P2 lives in the same places as P1*
    - *Compute in interests the score for P2 (likes A2,A3,A4?)*
    - *If the P2.score >= 2, visit all friends P3 of P2, for each:*
        - *Check in the person data that P3 lives in the same places as P1*
        - *Compute in interests the score for P3 (likes A2,A3,A4?)*
        - *If the P3.score >= 2, see if P1 is among the friends of P3, if so*
            - *We have a result (P2.score+P3.score,P1,P2,P3)*

CWI Centrum Wiskunde & Informatica

Naïve Query Implementation

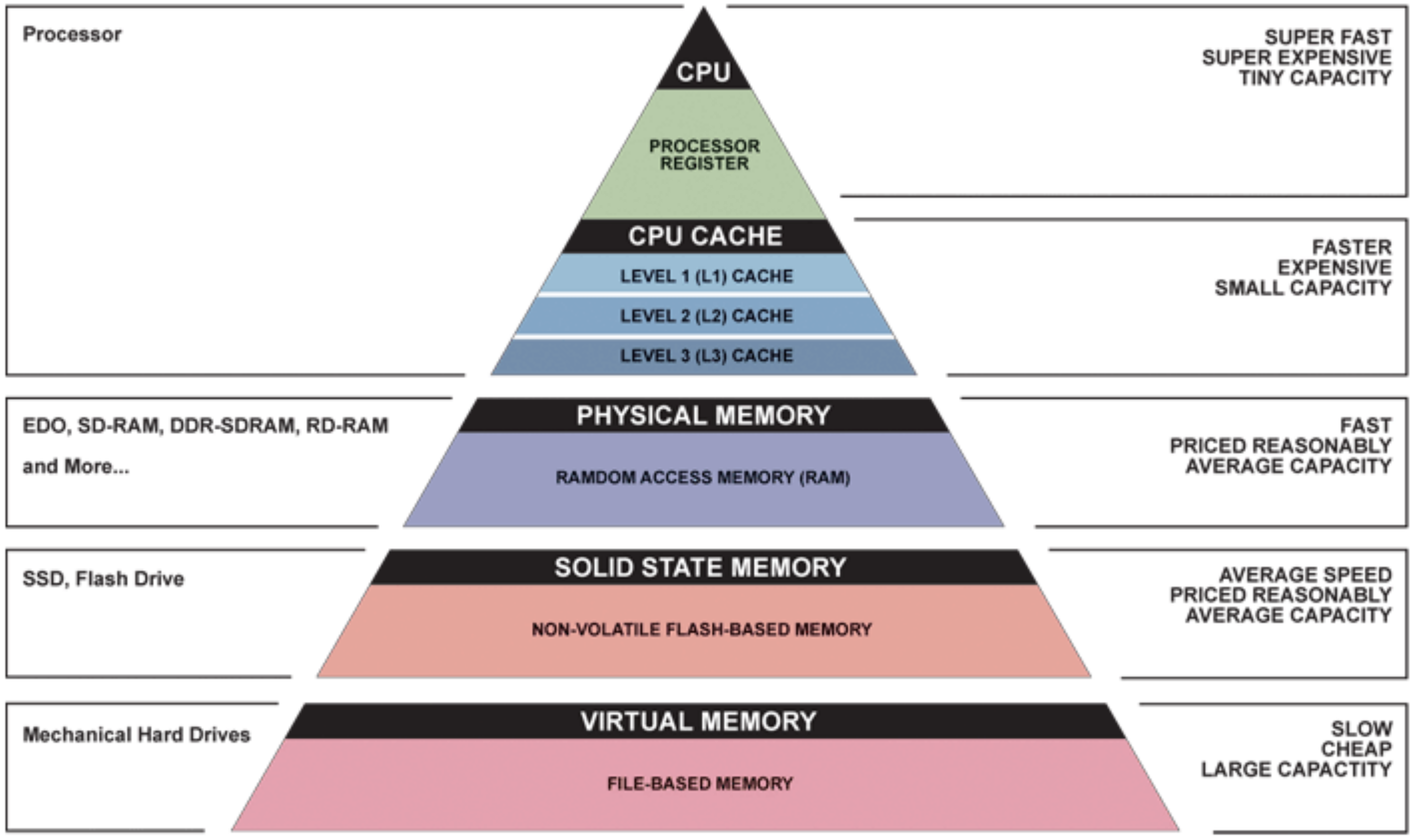
- "cruncher"



results

Centrum Wiskunde & Informatica

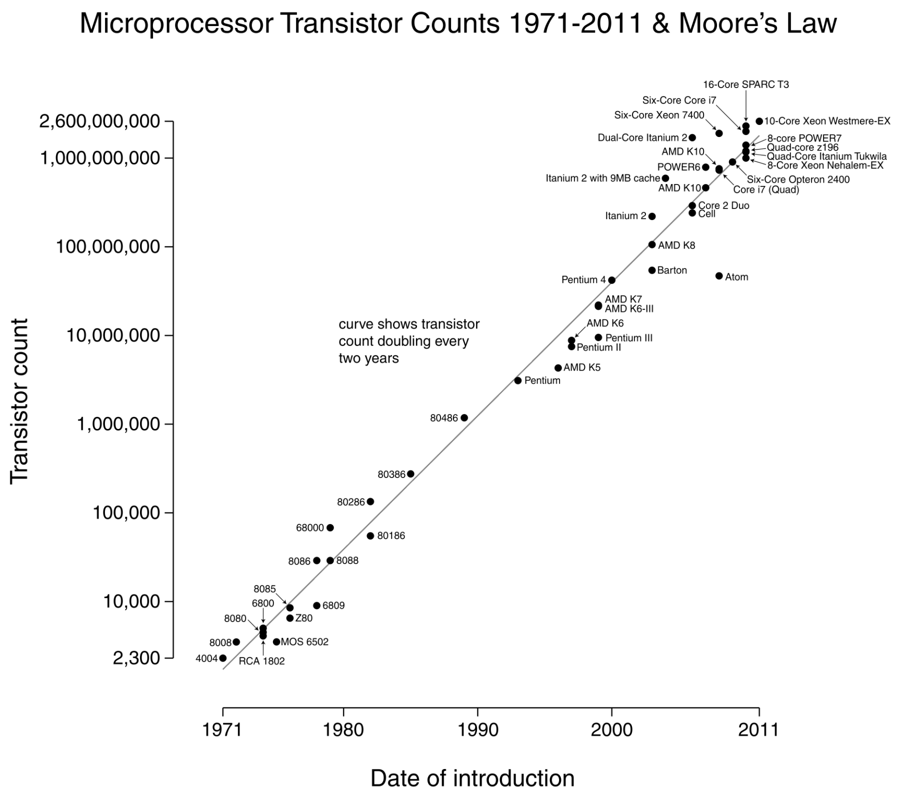
Memory Hierarchy



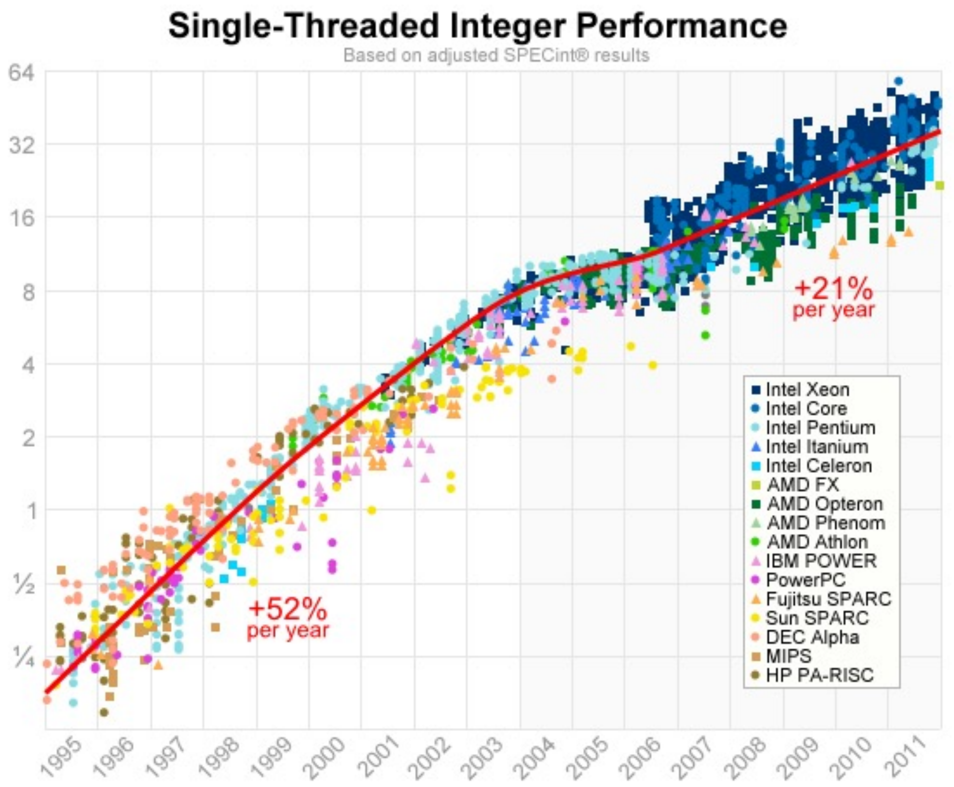
▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

Centrum Wiskunde & Informatica

Hardware Progress



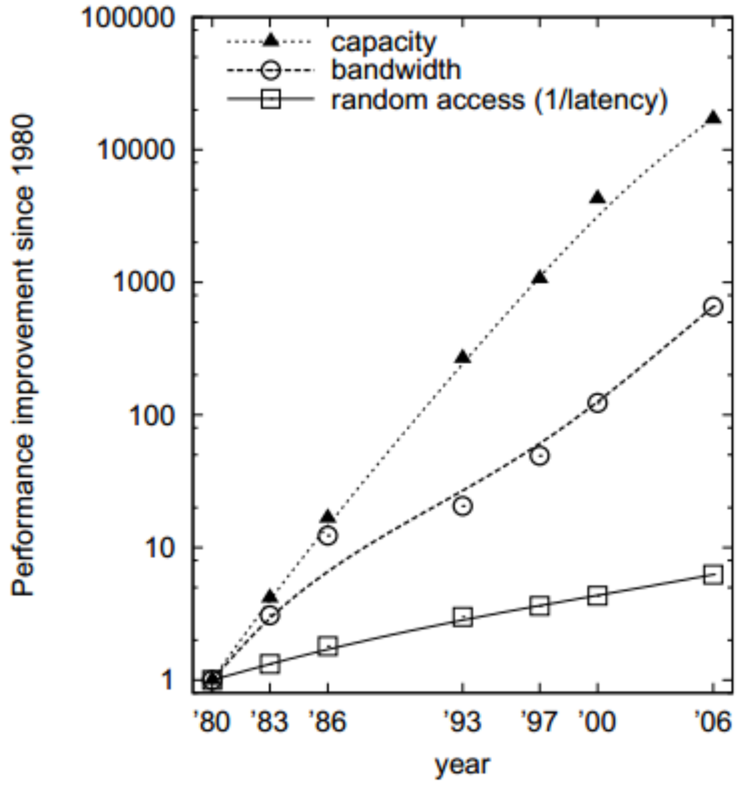
Transistors



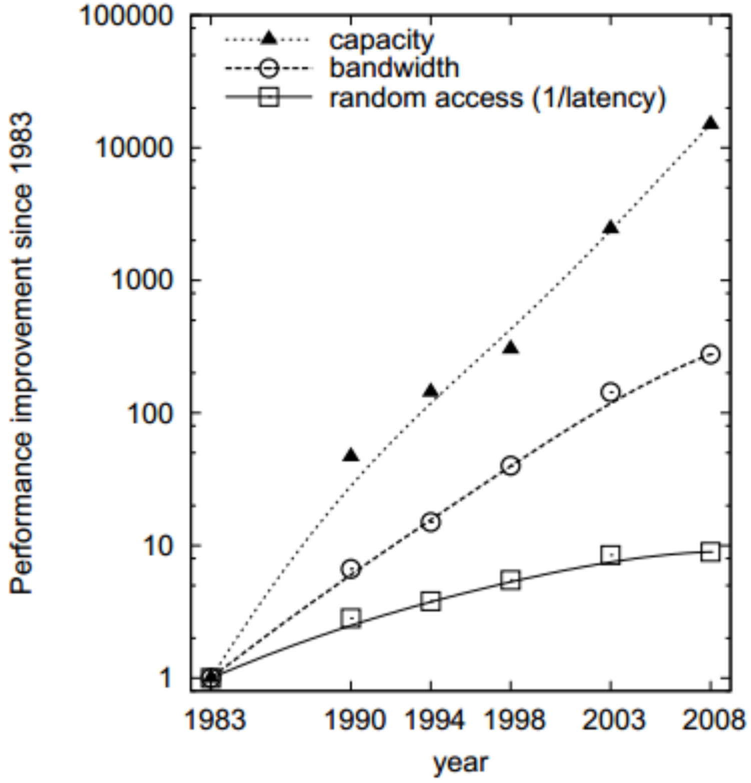
CPU performance

event.cwi.nl/lsde

RAM,Disk Improvement Over the Years



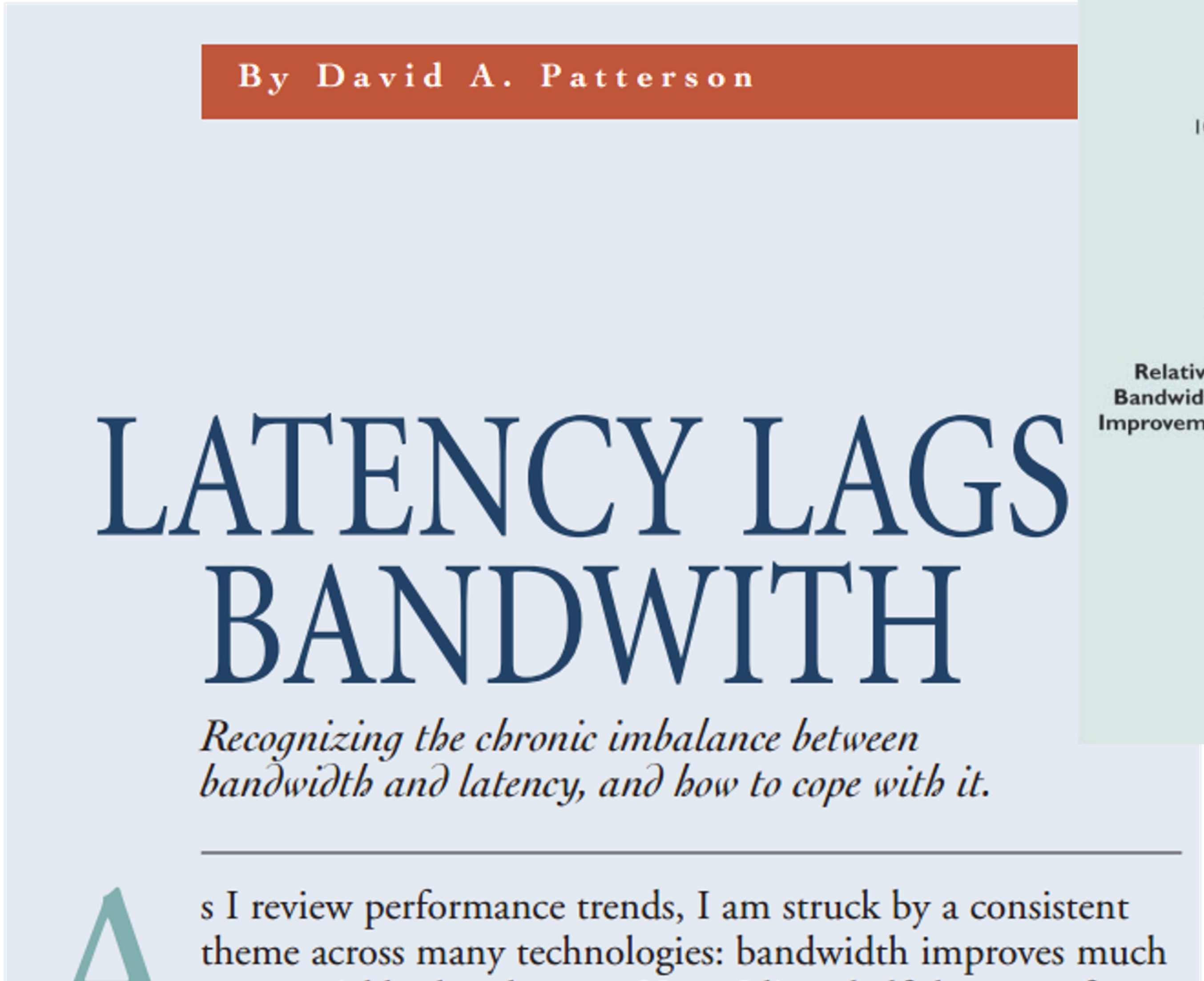
RAM



Magnetic Disk

Latency Lags Bandwidth

- Communications of the ACM, 2004

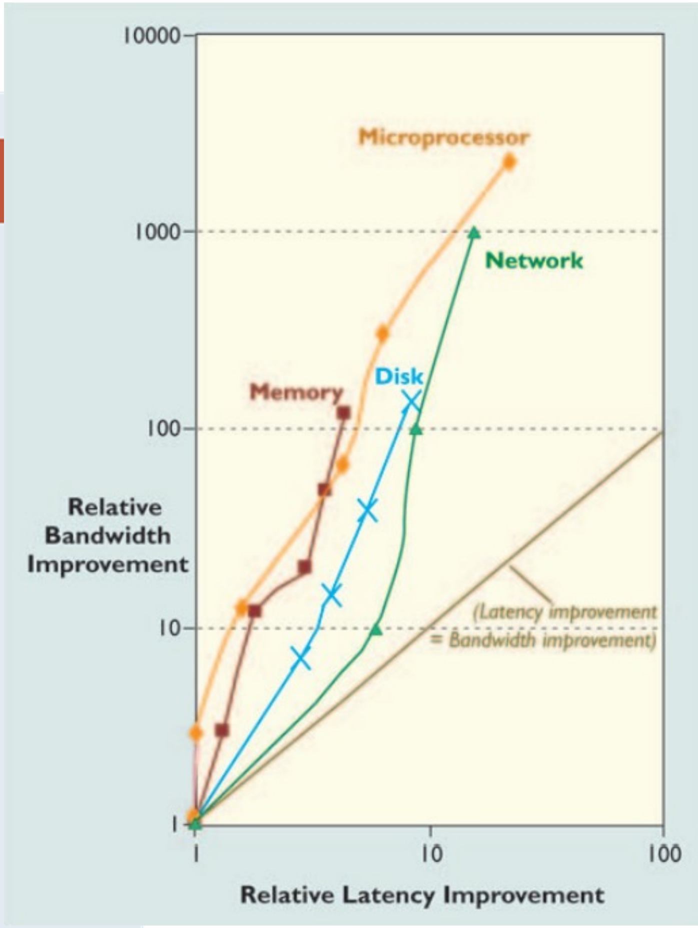


By David A. Patterson

LATENCY LAGS BANDWITH

Recognizing the chronic imbalance between bandwidth and latency, and how to cope with it.

As I review performance trends, I am struck by a consistent theme across many technologies: bandwidth improves much



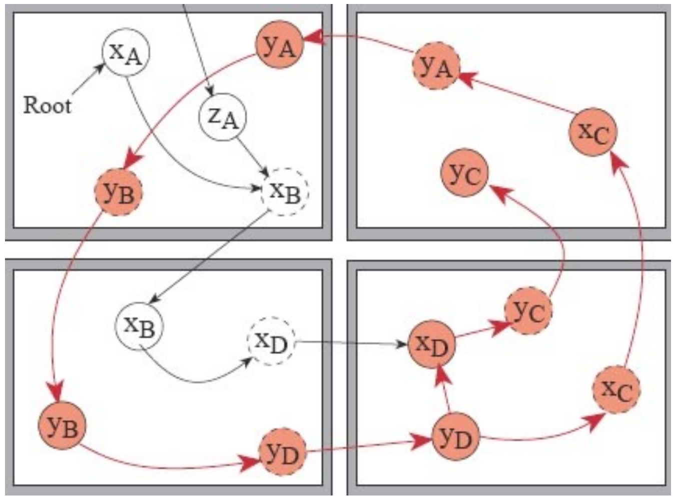
event.cwi.nl/lsde

Sequential Access Hides Latency

- Sequential RAM access
  - CPU prefetching: multiple consecutive cache lines being requested concurrently
- Sequential Magnetic Disk Access
  - Disk head moved once
  - Data is streamed as the disk spins under the head
- Sequential Network Access
  - Full network packets
  - Multiple packets in transit concurrently

Consequences For Algorithms

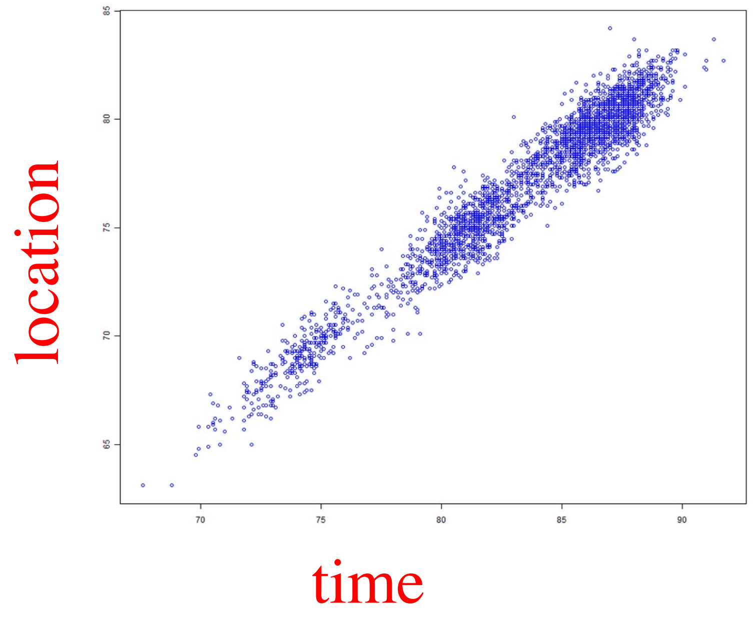
- Analyze the main data structures
  - How big are they?
    - Are they bigger than RAM?
    - Are they bigger than CPU cache (a few MB)?
  - How are they laid out in memory or on disk?
    - One area, multiple areas?

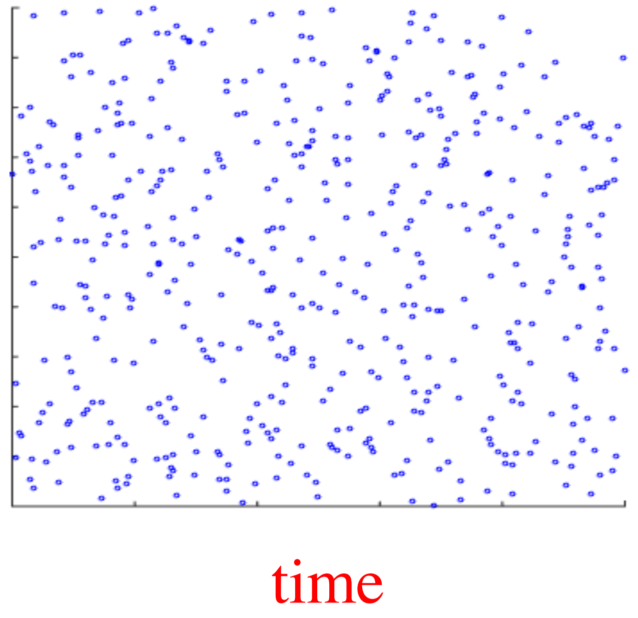


Java Object Data Structure
vs
memory pages (or cache lines)

Consequences For Algorithms

- Analyze your access patterns
  - Sequential: you're OK
  - Random: it better fit in cache!
    - What is the access granularity?
    - Is there temporal locality?
    - Is there spatial locality?





Improving Data Access Patterns

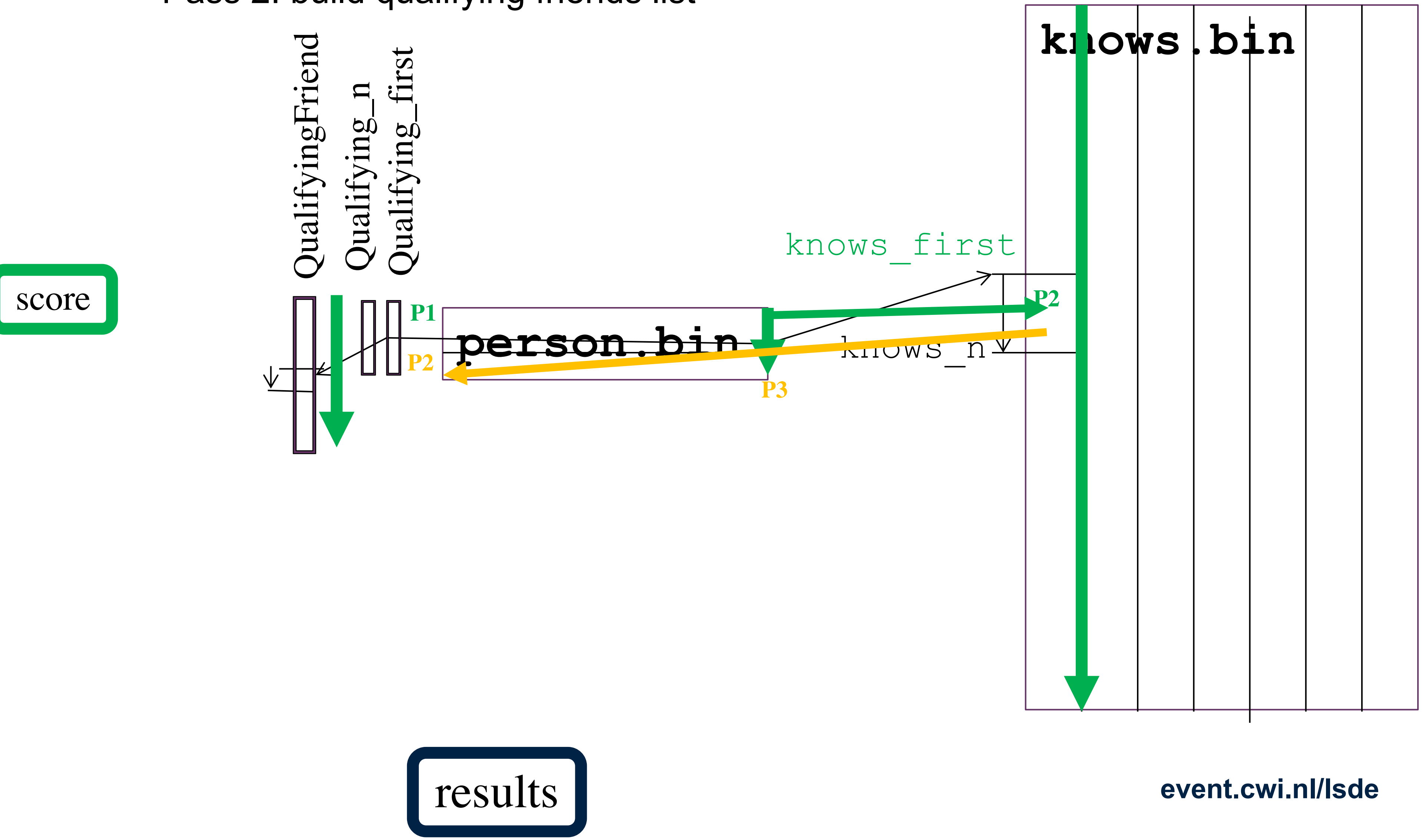
- **Make the data smaller**
  - Remove unused data from the structure
  - Apply data compression (of some kind)
    - If random access is needed, gzip does not work
    - zero surpression ➔ use the smallest datatype possible
- **Do Not Access All Data**
  - Apply filters as soon as possible
  - Cluster or Partition the data
    - Only access data in particular clusters/partitions
  - Build an index
    - Avoid full access to the main table by identifying useful regions using an index
- **Trade Random Access For Sequential Access**
  - Make more passes over the data. Separate access to different regions into different phases.
- **Try Denormalizing the Schema**
  - Remove joins/lookups, add looked up stuff to the table
    - Does not help if the join explodes the size (this is the case with friends!)

Improving Data Access Patterns

- **Make the data smaller**
  - Remove unused data from the structure
  - Apply data compression (of some kind)
    - If random access is needed, gzip does not work
    - zero surpression ➔ use the smallest datatype possible
- **Do Not Access All Data**
  - Apply filters as soon as possible
  - Cluster or Partition the data
    - Only access data in particular clusters/partitions
  - Build an index
    - Avoid full access to the main table by identifying useful regions using an index
- **Trade Random Access For Sequential Access**
  - Make more passes over the data. Separate access to different regions into different phases.
- **Try Denormalizing the Schema**
  - Remove joins/lookups, add looked up stuff to the table
    - Does not help if the join explodes the size (this is the case with friends!)

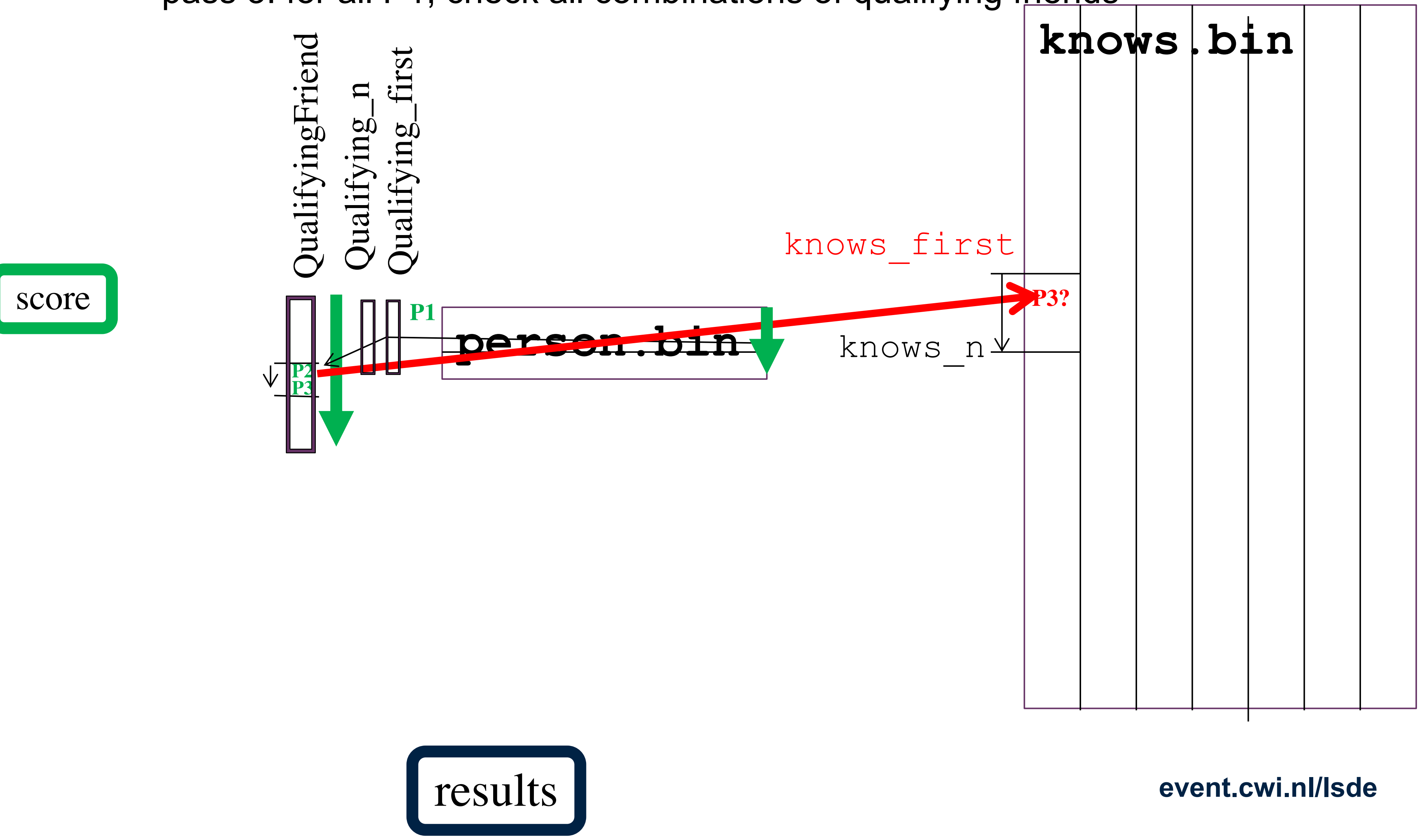
3-pass Cruncher

- Pass 2: build qualifying friends list



3-pass Cruncher

- pass 3: for all P1, check all combinations of qualifying friends



Challenges for your Reorg program

Questions for re-org:

- Can we throw way unneeded data?
- Can we store the data more efficiently?
- Can we build indexes?

Questions for cruncher:

- Can we move some of the work to the re-org phase?
- Can we exploit indexes?

We will meet on the leaderboard!

Improving Data Access Patterns

- **Make the data smaller**
  - Remove unused data from the structure
  - Apply data compression (of some kind)
    - If random access is needed, gzip does not work
    - zero surpression ➔ use the smallest datatype possible
- **Do Not Access All Data**
  - Apply filters as soon as possible
  - Cluster or Partition the data
    - Only access data in particular clusters/partitions
  - Build an index
    - Avoid full access to the main table by identifying useful regions using an index
- **Trade Random Access For Sequential Access**
  - Make more passes over the data. Separate access to different regions into different phases.
- **Try Denormalizing the Schema**
  - Remove joins/lookups, add looked up stuff to the table
    - Does not help if the join explodes the size (this is the case with friends!)

Improving Data Access Patterns

- **Make the data smaller**
  - Remove unused data from the structure
  - Apply data compression (of some kind)
    - If random access is needed, gzip does not work
    - zero surpression ➔ use the smallest datatype possible
- **Do Not Access All Data**
  - Apply filters as soon as possible
  - Cluster or Partition the data
    - Only access data in particular clusters/partitions
  - Build an index
    - Avoid full access to the main table by identifying useful regions using an index
- **Trade Random Access For Sequential Access**
  - Make more passes over the data. Separate access to different regions into different phases.
- **Try Denormalizing the Schema**
  - Remove joins/lookups, add looked up stuff to the table
    - Does not help if the join explodes the size (this is the case with friends!)

CWI
Centrum Wiskunde & Informatica

The Query and Its Filter Conditions

The "cruncher" program

Go through the persons P sequentially, and for those ***in birthday range***

- *count how many of the artists* ***A2,A3,A4*** *are liked as the* ***score***

  for those with ***score****>0 and who do not like* ***A1****:*

  – *visit all persons F known to P.*

    For each F:

    - *checks on* ***equal location***
    - *check whether F already likes* ***A1***
    - *check whether* ***F also knows P***

    if all this succeeds (score,P,F) is added to a result table.

The Query and its Filter Conditions

The "cruncher" program

Go through the persons P1 sequentially

- *check whether P1's birthdate is in range **D1..D2***
- *check in interests whether this person likes **A1**, if so*
- *visit all friends P2 of P1, for each:*
    - *Check in the person data that P2 lives in the **same location** as P1*
    - *Compute in interests the score for P2 (likes **A2,A3,A4**?)*
    - *If the P2.score >= 2, visit all friends P3 of P2, for each:*
        - *Check in the person data that P3 lives in the **same location** as P1*
        - *Compute in interests the score for P3 (likes **A2,A3,A4**?)*
        - *If the P3.score >= 2, see if P1 is among the friends of P3, if so*
            - *We have a result (P2.score+P3.score,P1,P2,P3)*

Reducing The Problem

- knows.bin
  - is big (larger than RAM)
  - is accessed randomly
    - random access unavoidable (denormalization too costly)

Ideas:

- Only keep **mutual-knows**
  - Idea: remove non-mutual knows in reorg
    - Advantage: queries do not need to check (only reorg), **queries get faster**
    - Problem: 99% of knows in this dataset is mutual (**no reduction**)
    - Problem: finding non-mutual knows is costly (**requires full sort on person-id**)

CWI
Centrum Wiskunde & Informatica

Reducing The Problem

- knows.bin
  - is big (larger than RAM)
  - is accessed randomly
    - random access unavoidable (denormalization too costly)

Ideas:

- Only keep **mutual-knows**
- Only keep **local-knows**
  - Idea: remove knows where persons live in different cities (**30x less:** 150 ➔ 5 friends)
    - Reorg: one pass with random access in a ‘location’ array (2b * 8.9M)
  - Idea: remove persons with zero friends left-over (**halves it**)
    - 8.9M ➔ 5M persons, 8.9*23M ➔ 5*23M interests
  - Idea: remove non-mutual local friends *after* removing the above (smaller knows!)
    - Can be done with random access
      - Reorg: write a localknows.tmp file, mmap it, use it i.s.o. knows.bin to filter
      - localknows.tmp = 5*10M=50M knows = 200MB random access

CWI
Centrum Wiskunde & Informatica

Reducing The Problem

- knows.bin
  - is big (larger than RAM), and is accessed randomly
    - random access unavoidable (denormalization too costly)

Ideas:

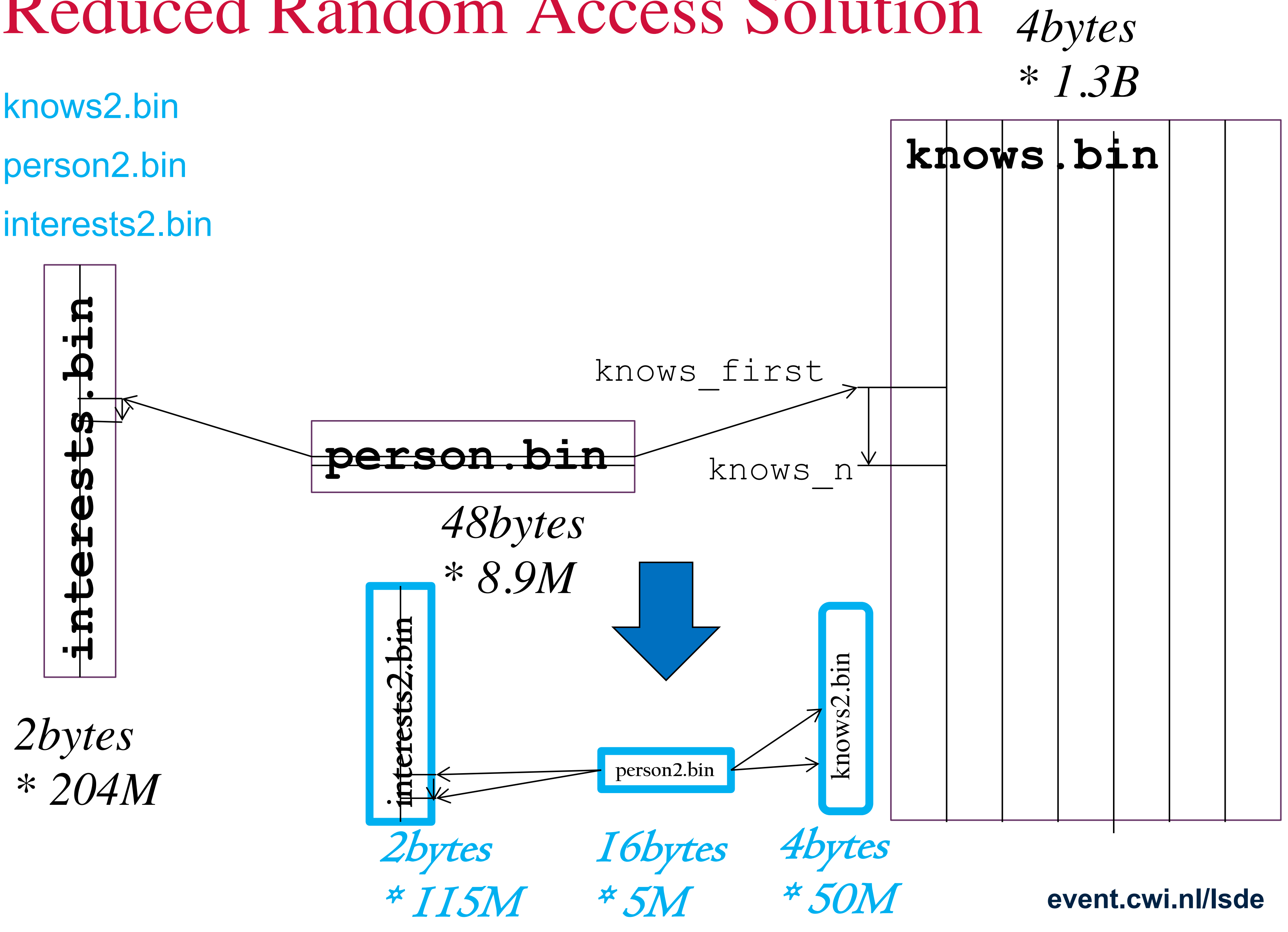
- Only keep **local-knows**
  - Idea: only keep knows when both live in the same location (**30x less:** 150➔5 friends)
    - reorg: one pass with random access in a 'location' array (2b * 8.9M)
  - Idea: remove persons with **0** friends left-over (**more than halves it**)
    - 8.9M ➔ 5M persons,
    - 8.9*23M ➔ 5*23M interests (only keep interests of surviving persons)
  - More aggressive idea (harder to implement):
    - check whether each remaining friend of p has another friend of p as friend
      - ..otherwise, a triangle is impossible
    - challenge: if you remove a persons, they will still refer to you (inconsistency)
      - need to remove the removed person from the friends list
      - this could lead to more persons to prune (recursive!!)

Reduced Random Access Solution

knows2.bin

person2.bin

interests2.bin



CWI
Centrum Wiskunde & Informatica

Improving Data Access Patterns

- **Make the data smaller**
  - Remove unused data from the structure
  - Apply data compression (of some kind)
    - If random access is needed, gzip does not work
    - zero surpression ➔ use the smallest datatype possible
- **Do Not Access All Data**
  - Apply filters as soon as possible
  - Cluster or Partition the data
    - Only access data in particular clusters/partitions
  - Build an index
    - Avoid full access to the main table by identifying useful regions using an index
- **Trade Random Access For Sequential Access**
  - Make more passes over the data. Separate access to different regions into different phases.
- **Try Denormalizing the Schema**
  - Remove joins/lookups, add looked up stuff to the table
    - Does not help if the join explodes the size (this is the case with friends!)

Improving Data Access Patterns

- **Make the data smaller**
  - Remove unused data from the structure
  - Apply data compression (of some kind)
    - If random access is needed, gzip does not work
    - zero surpression ➔ use the smallest datatype possible
- **Do Not Access All Data**
  - Apply filters as soon as possible
  - Cluster or Partition the data
    - Only access data in particular clusters/partitions
  - Build an index
    - Avoid full access to the main table by identifying useful regions using an index
- **Trade Random Access For Sequential Access**
  - Make more passes over the data. Separate access to different regions into different phases.
- **Try Denormalizing the Schema**
  - Remove joins/lookups, add looked up stuff to the table
    - Does not help if the join explodes the size (this is the case with friends!)

The Query and Its Filter Conditions

The "cruncher" program

Go through the persons P sequentially, and for those ***in birthday range***

- *count how many of the artists* ***A2,A3,A4*** *are liked as the* ***score***

  for those with ***score****>0 and who do not like* ***A1****:*

  – *visit all persons F known to P.*

    For each F:

    - *checks on* ***equal location***
    - *check whether F already likes* ***A1***
    - *check whether* ***F also knows P***

    if all this succeeds (score,P,F) is added to a result table.

The Query and Its Filter Conditions

The "cruncher" program

Go through the persons P sequentially, and for those ***in birthday range***

- *count how many of the artists* ***A2,A3,A4*** *are liked as the* ***score***

  for those with ***score****>0 and who do not like* ***A1****:*

  - *visit all persons F known to P.*

    For each F:

    - *checks on* ***equal location***
    - *check whether F already likes* ***A1***
    - *check whether* ***F also knows P***

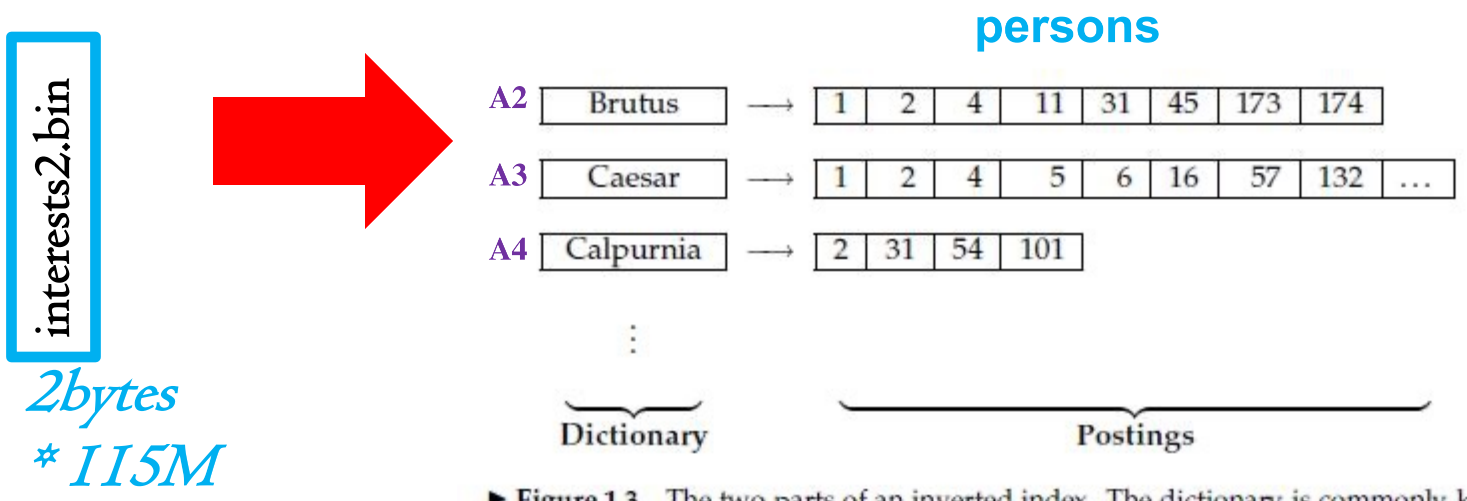
    if all this succeeds (score,P,F) is added to a result table.

Idea: using Inverted Files

The search engine data structure

- For each term (keyword), a list of document IDs

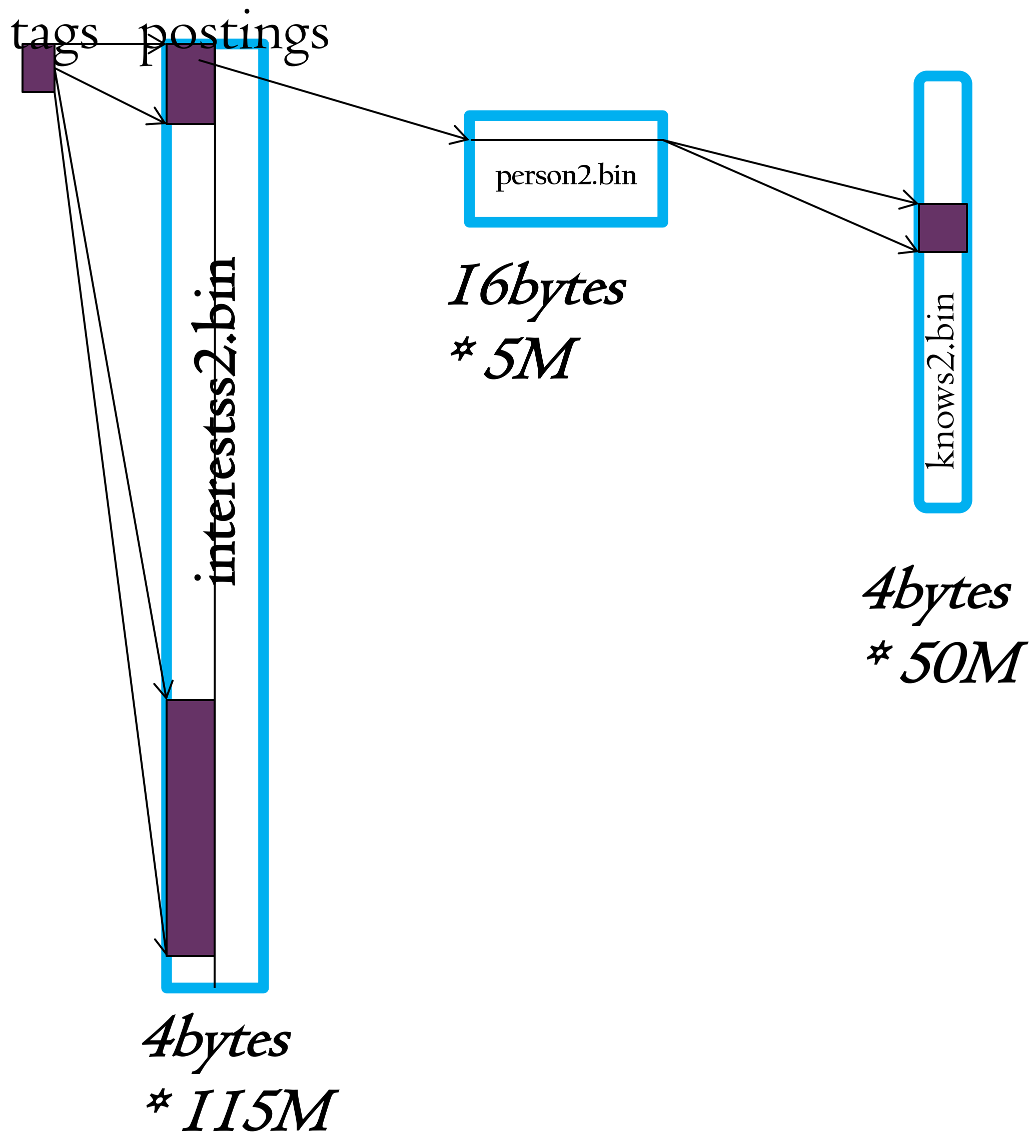
Here: for each **Tag (e.g. A1,A2,A3,A4)** a list of


interests2.bin
2bytes
* 115M
persons
A2 Brutus → 1 2 4 11 31 45 173 174
A3 Caesar → 1 2 4 5 6 16 57 132 ...
A4 Calpurnia → 2 31 54 101
⋮
Dictionary
Postings

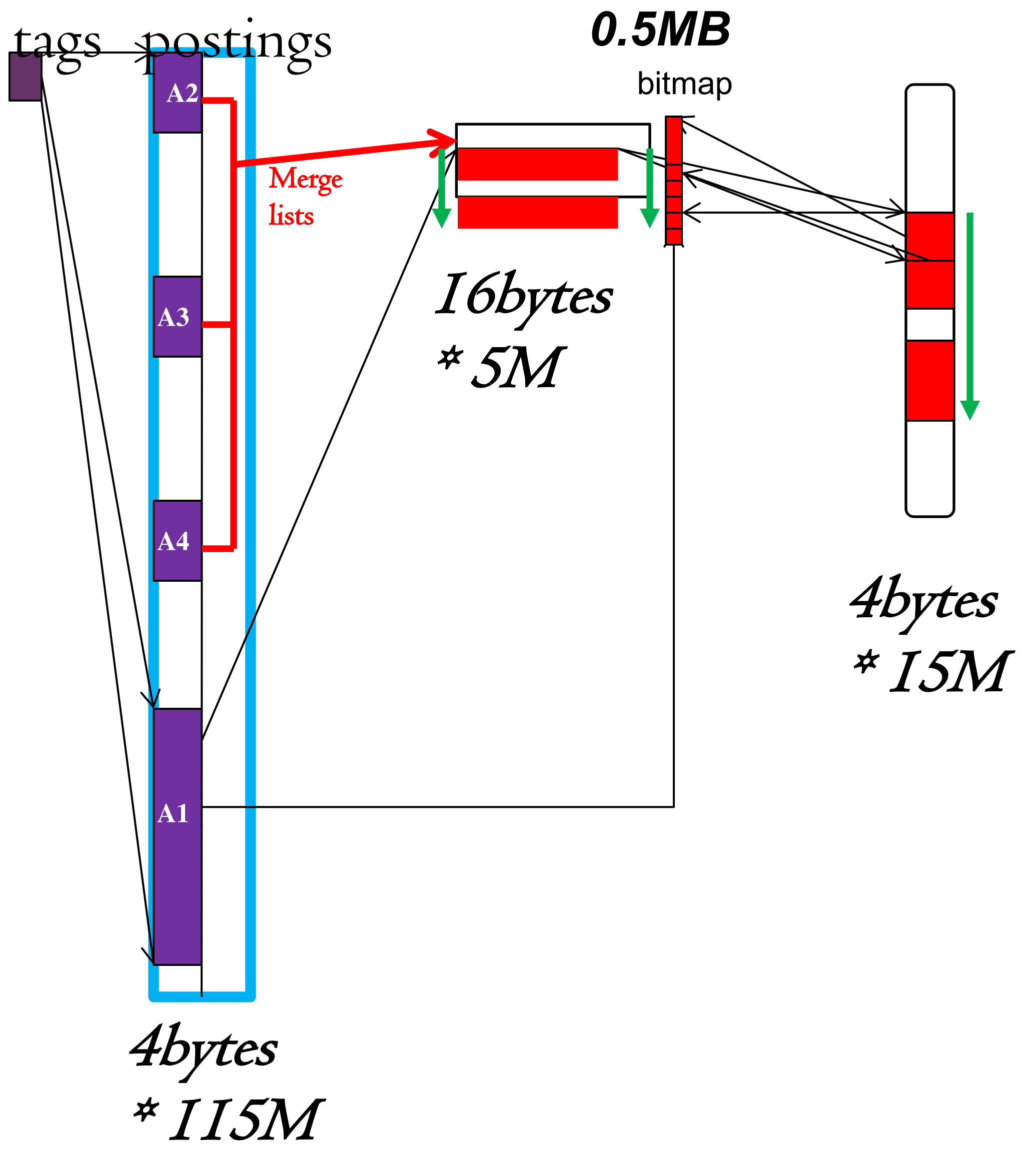
▶ **Figure 1.3** The two parts of an inverted index. The dictionary is commonly kept in memory, with pointers to each postings list, which is stored on disk.

Centrum Wiskunde & Informatica

Inverted File on Tags (=Artists)



Inverted File on Tags (=Artists)



Algorithm:

- Read A1 invlist. Populate a boolean array (bitmap) all false, but put all persons in A1 to true
- Read invlists A2,A3,A4 and union-sum them in a merge to get person candidates + scores
- Visit the candidates, check bitmap to see they don't like A1. If so, visit their friends and confirm that they do like A1

The Query and Its Filter Conditions

The "cruncher" program

Go through the persons P sequentially, and for those ***in birthday range***

- *count how many of the artists* ***A2,A3,A4*** *are liked as the* ***score***

  for those with ***score****>0 and who do not like* ***A1****:*

  – *visit all persons F known to P.*

    For each F:

    - *checks on* ***equal location***
    - *check whether F already likes* ***A1***
    - *check whether* ***F also knows P***

    if all this succeeds (score,P,F) is added to a result table.

The Query and Its Filter Conditions

The "cruncher" program

Go through the persons P sequentially, and for those ***in birthday range***

- *count how many of the artists* ***A2,A3,A4*** *are liked as the* ***score***

  for those with ***score****>0 and who do not like* ***A1****:*

  – *visit all persons F known to P.*

    For each F:

    - *checks on* ***equal location***
    - *check whether F already likes* ***A1***
    - *check whether* ***F also knows P***

    if all this succeeds (score,P,F) is added to a result table.

The Query and its Filter Conditions

The "cruncher" program

Go through the persons P1 sequentially

- *check whether P1's birthdate is in range **D1..D2***
- *check in interests whether this person likes **A1**, if so*
- *visit all friends P2 of P1, for each:*
    - *Check in the person data that P2 lives in the **same location** as P1*
    - *Compute in interests the score for P2 (likes **A2,A3,A4**?)*
    - *If the P2.score >= 2, visit all friends P3 of P2, for each:*
        - *Check in the person data that P3 lives in the same places as P1*
        - *Compute in interests the score for P3 (likes **A2,A3,A4**?)*
        - *If the P3.score >= 2, see if P1 is among the friends of P3, if so*
            - *We have a result (P2.score+P3.score,P1,P2,P3)*

The Query and its Filter Conditions

The "cruncher" program

Go through the persons P1 sequentially

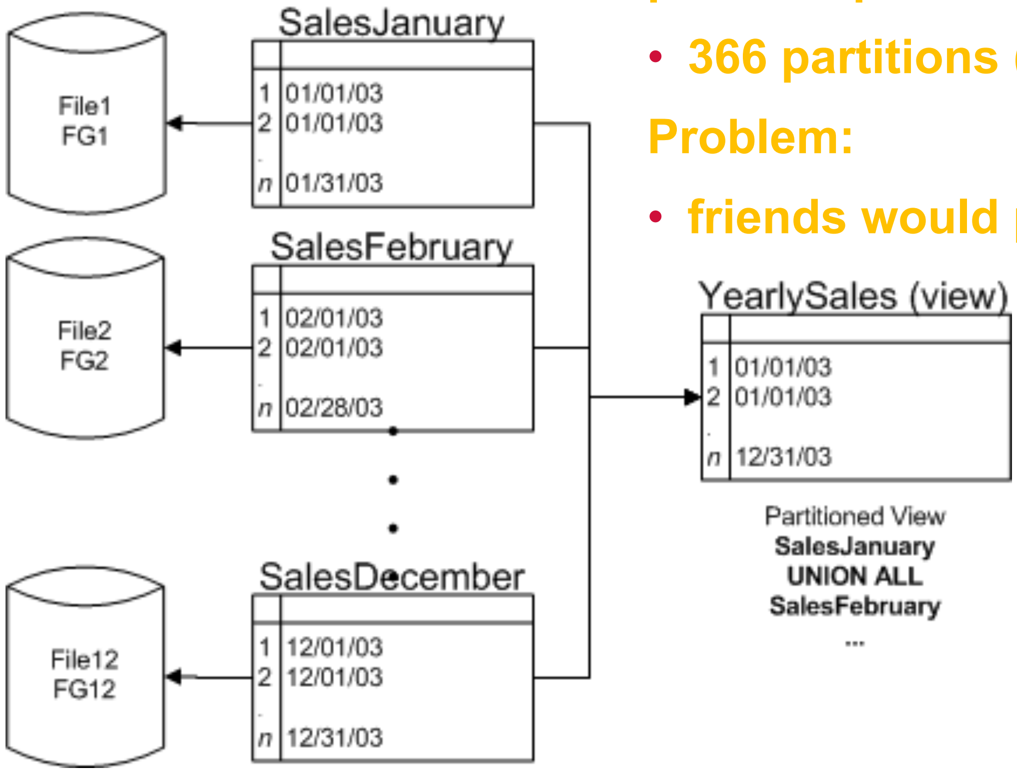
- *check whether P1's birthdate is in range* ***D1..D2***
- *check in interests whether this person likes* ***A1****, if so*
- *visit all friends P2 of P1, for each:*
    - *Check in the person data that P2 lives in the* ***same location*** *as P1*
    - *Compute in interests the score for P2 (likes* ***A2,A3,A4****?)*
    - *If the P2.score >= 2, visit all friends P3 of P2, for each:*
        - *Check in the person data that P3 lives in the* ***same location*** *as P1*
        - *Compute in interests the score for P3 (likes* ***A2,A3,A4****?)*
        - *If the P3.score >= 2, see if P1 is among the friends of P3, if so*
            - *We have a result (P2.score+P3.score,P1,P2,P3)*

CWI
Centrum Wiskunde & Informatica

Idea: use Table Partitioning

Goals:

- make birthdate comparisons faster
- remove birthdate column (no longer needed, implicit)
- **Increase locality in person.bin and knows.bin**!



partition person.bin by birthdate

- **366 partitions (one for each day)**

Problem:

- **friends would point across all 366 tables**

CWI
Centrum Wiskunde & Informatica

Partitioning → Sorting

Range-partitioning is similar to sorting

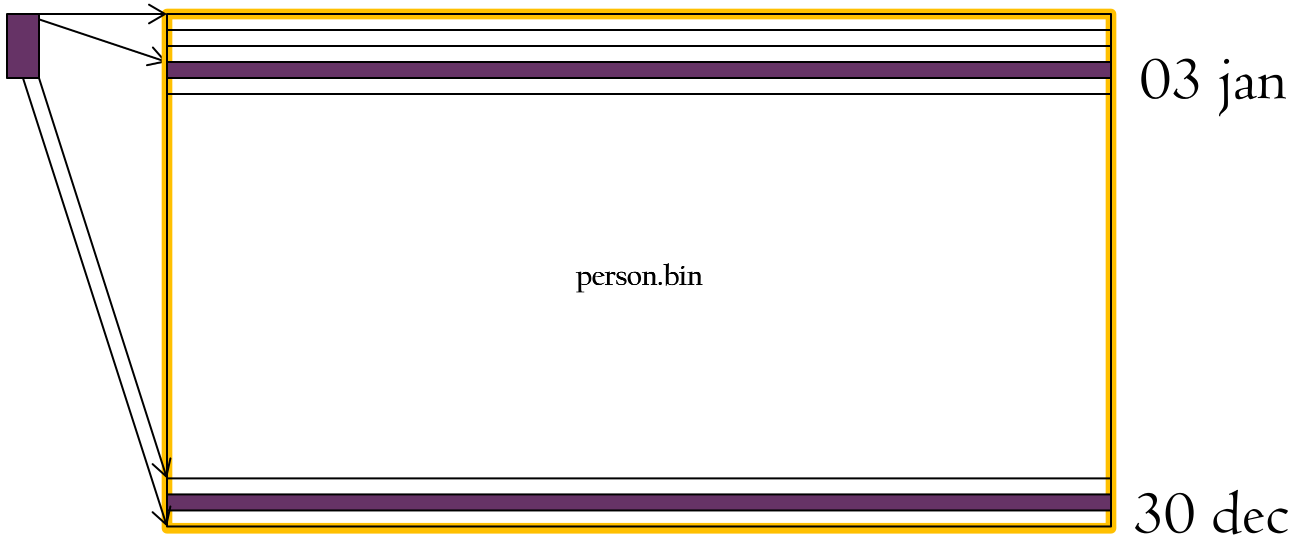
Idea:

sort person.bin on birthdate

- every person gets a new number (=offset)
- change all numbers in knows2.bin!

have a small index on date

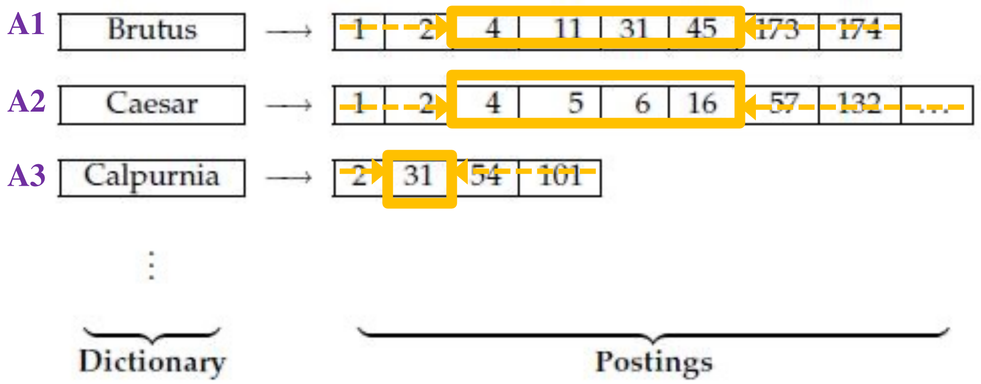
- offset of the first person with that birthdate



Inverted Files Revisited

The birthdate clustering gives us for a **birthdate range** a **person range**

- Say people with bday in **February** are at positions between **[4,50]**
- **Idea:** binary search in the postings lists for artists (**A2,A3,A4**)

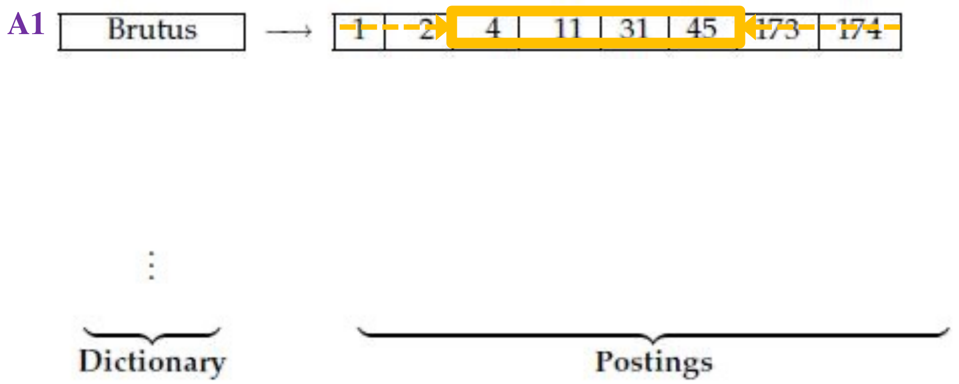


► **Figure 1.3** The two parts of an inverted index. The dictionary is commonly kept in memory, with pointers to each postings list, which is stored on disk.

Inverted Files Revisited

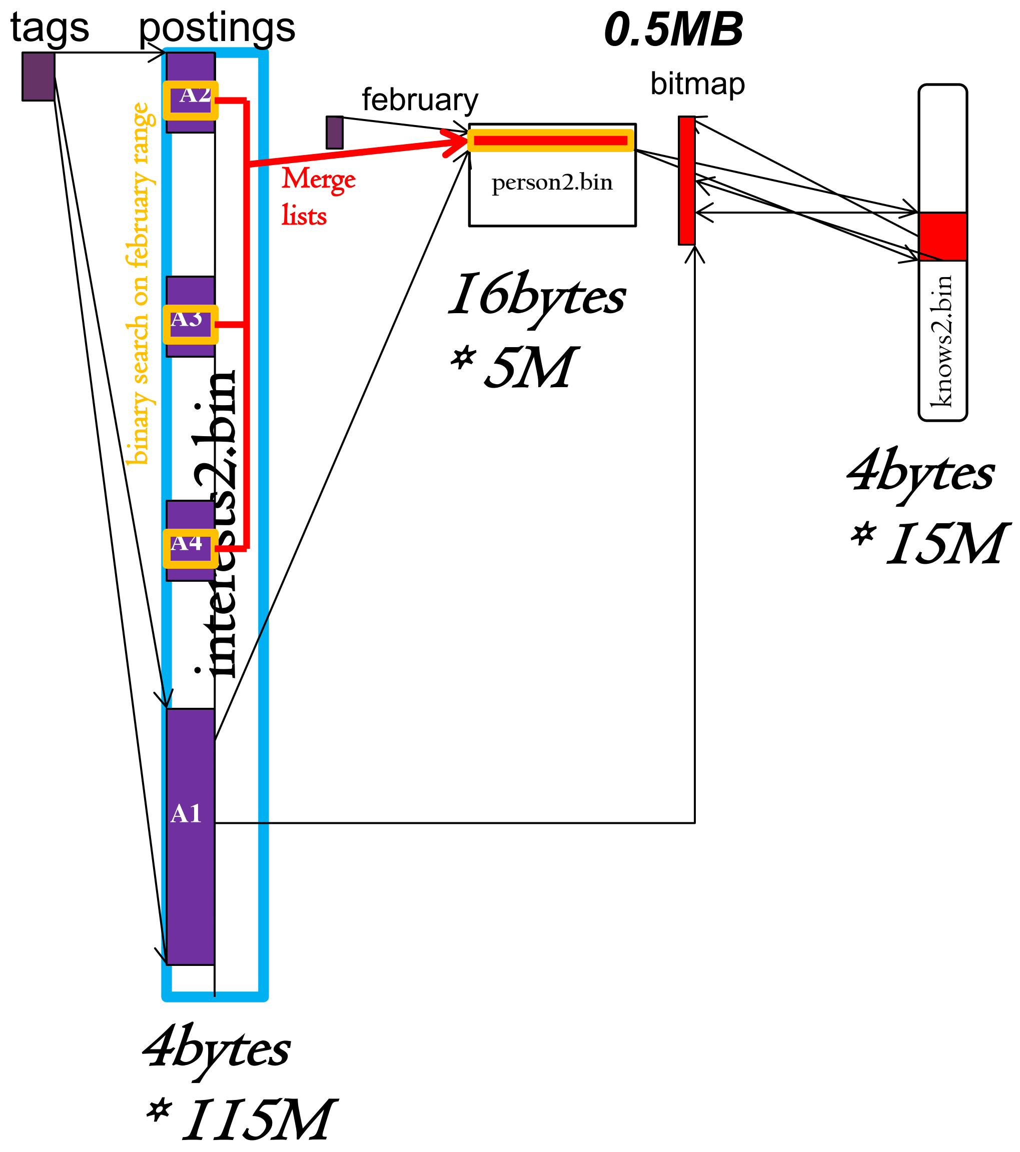
The birthdate clustering gives us for a **birthdate range** a **person range**

- Say people with bday in **February** are at positions between **[4,50]**
- **Idea:** binary search in the postings lists for artists **A1**



▶ **Figure 1.3** The two parts of an inverted index. The dictionary is commonly kept in memory, with pointers to each postings list, which is stored on disk.

Inverted File on Tags (=Artists)



Algorithm:
Same as before, but..
- Instead of fetching the *whole* invlists for A2, A3, A4, do a binary search in them to limit them to a person ramge. The number of candidates will be more than 4x smaller.

(the person range is given by the birthdate offset array – see two slides back)

Centrum Wiskunde & Informatica

Improving Data Access Patterns

- **Make the data smaller**
  - Remove unused data from the structure
  - Apply data compression (of some kind)
    - If random access is needed, gzip does not work
    - zero surpression ➔ use the smallest datatype possible
- **Do Not Access All Data**
  - Apply filters as soon as possible
  - Cluster or Partition the data
    - Only access data in particular clusters/partitions
  - Build an index
    - Avoid full access to the main table by identifying useful regions using an index
- **Trade Random Access For Sequential Access**
  - Make more passes over the data. Separate access to different regions into different phases.
- **Try Denormalizing the Schema**
  - Remove joins/lookups, add looked up stuff to the table
    - Does not help if the join explodes the size (this is the case with friends!)