Large-Scale Data Engineering

Some notes on Access Patterns, Latency, Bandwidth

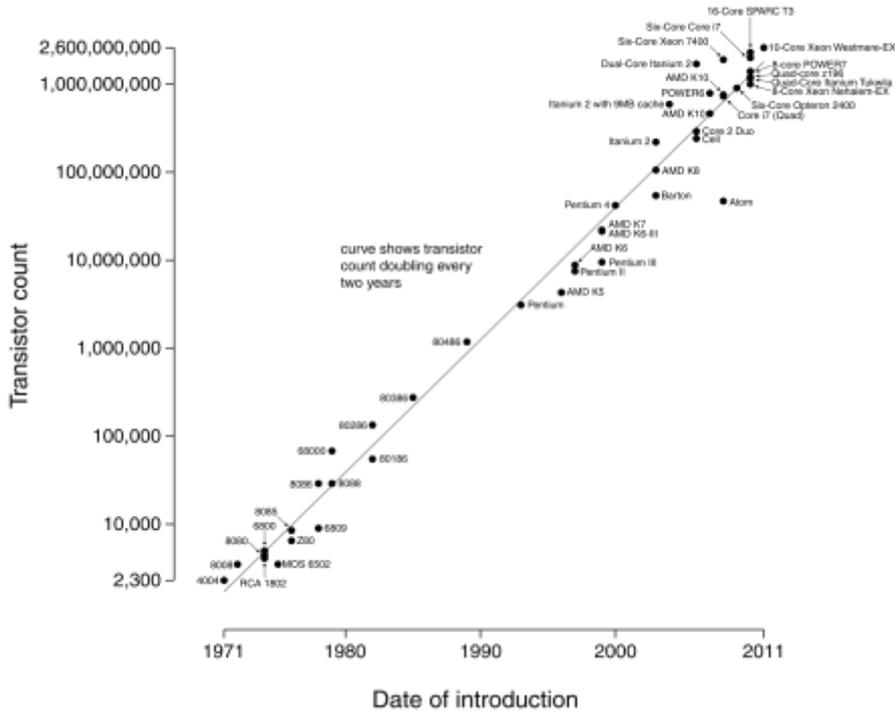### + Tips for practical

event.cwi.nl/lsde/2016
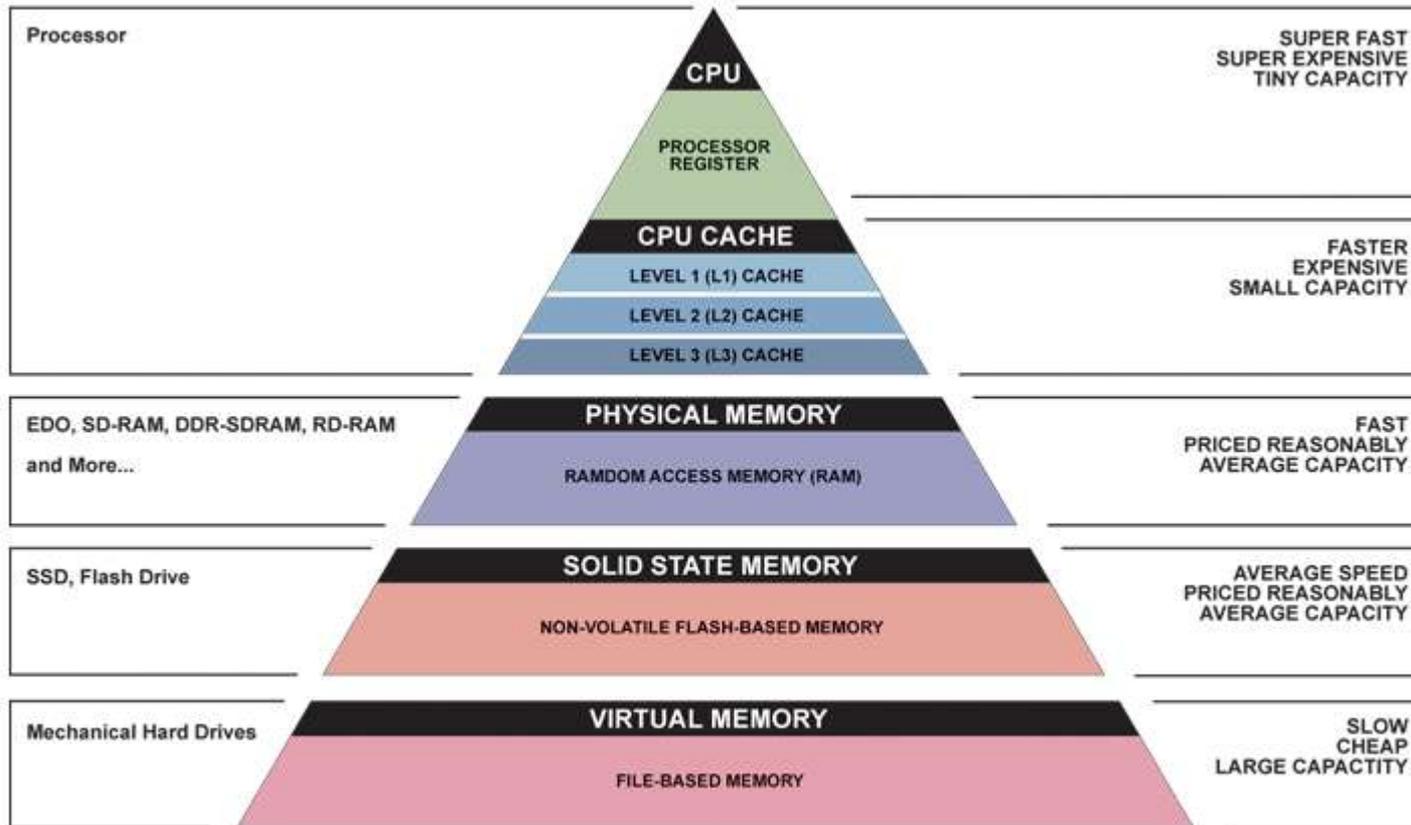
Memory Hierarchy

▲ Simplified Computer Memory Hierarchy
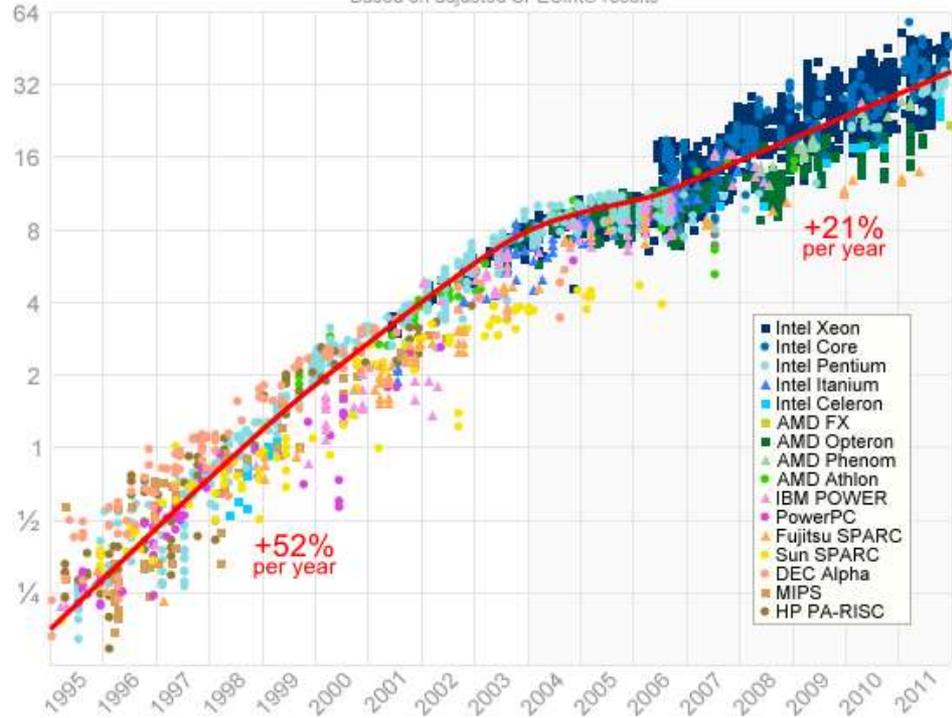Illustration: Ryan J. Leng

Hardware Progress

Microprocessor Transistor Counts 1971-2011 & Moore's Law

Transistors

Single-Threaded Integer Performance
Based on adjusted SPECint® results

+21% per year

+52% per year

- Intel Xeon
- Intel Core
- Intel Pentium
- Intel Itanium
- Intel Celeron
- AMD FX
- AMD Opteron
- AMD Phenom
- AMD Athlon
- IBM POWER
- PowerPC
- Fujitsu SPARC
- Sun SPARC
- DEC Alpha
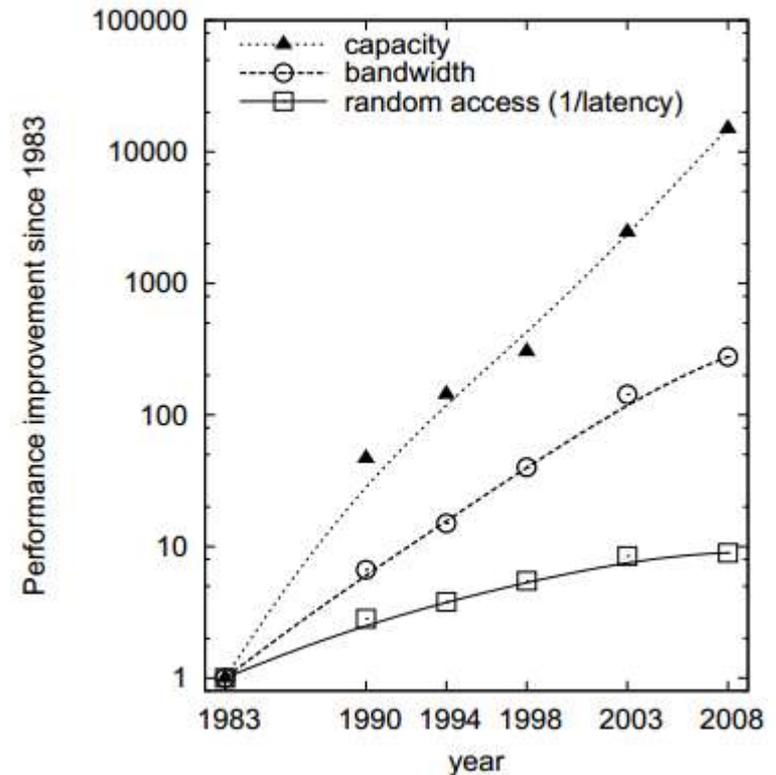- MIPS
- HP PA-RISC

CPU performance

RAM,Disk Improvement Over the Years
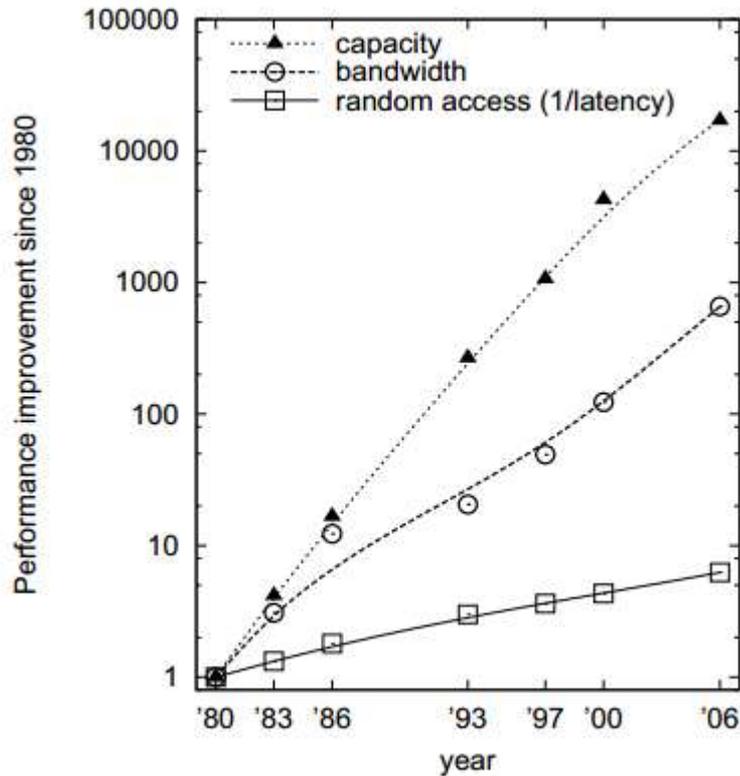
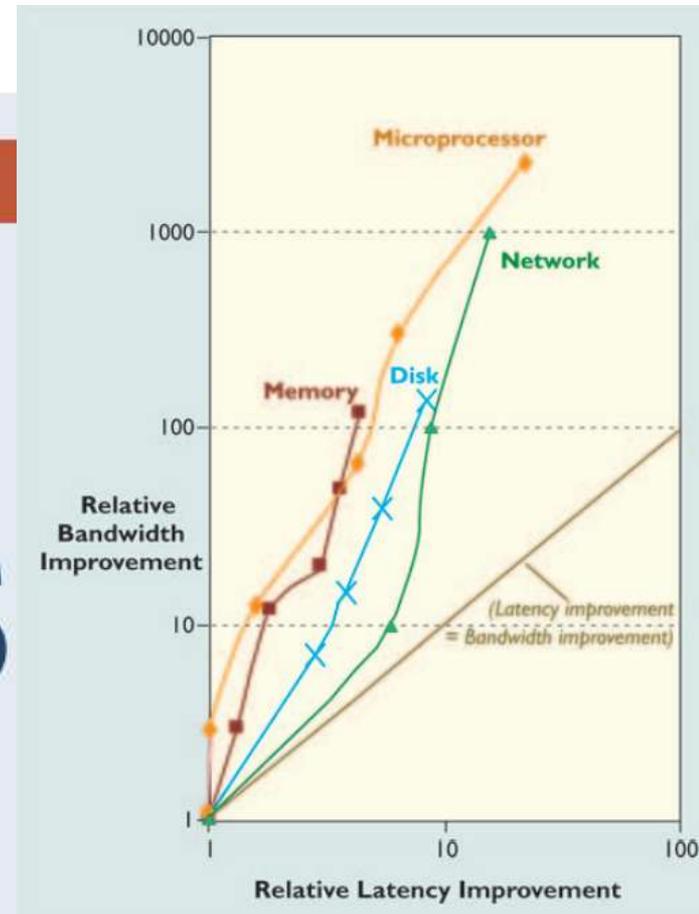
RAM

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.

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

Geeks on Latency

jboner / latency.txt
Created on 31 May 2012

Latency Numbers Every Programmer Should Know

◁▷ latency.txt

```
 1   Latency Comparison Numbers
 2   --------------------------
 3   L1 cache reference                      0.5 ns
 4   Branch mispredict                         5  ns
 5   L2 cache reference                        7  ns                 14x L1 cache
 6   Mutex lock/unlock                        25  ns
 7   Main memory reference                   100  ns                 20x L2 cache, 200x L1 cache
 8   Compress 1K bytes with Zippy          3,000  ns
 9   Send 1K bytes over 1 Gbps network    10,000  ns    0.01 ms
10   Read 4K randomly from SSD*          150,000  ns    0.15 ms
11   Read 1 MB sequentially from memory  250,000  ns    0.25 ms
12   Round trip within same datacenter   500,000  ns    0.5  ms
13   Read 1 MB sequentially from SSD*  1,000,000  ns    1    ms  4X memory
14   Disk seek                        10,000,000  ns    10   ms  20x datacenter roundtrip
15   Read 1 MB sequentially from disk 20,000,000  ns    20   ms  80x memory, 20X SSD
16   Send packet CA->Netherlands->CA 150,000,000  ns   150   ms
17
18   Notes
19   -----
20   1 ns = 10-9 seconds
21   1 ms = 10-3 seconds
22   * Assuming ~1GB/sec SSD
```

event.cwi.nl/lsde/2016

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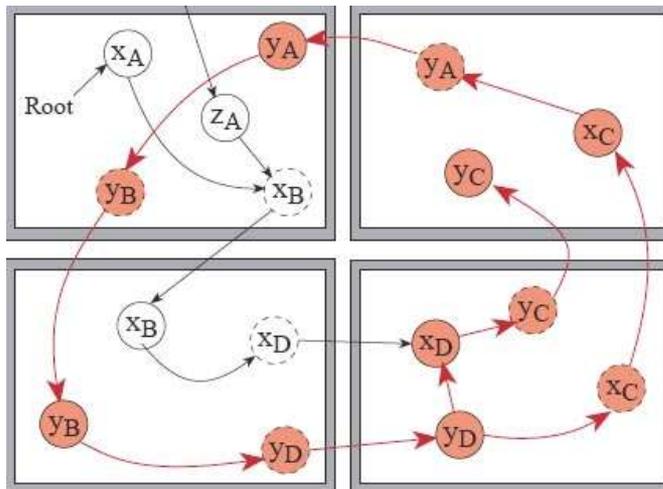
Java Object Data Structure
vs
memory pages (or cache lines)

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?

location

time

time
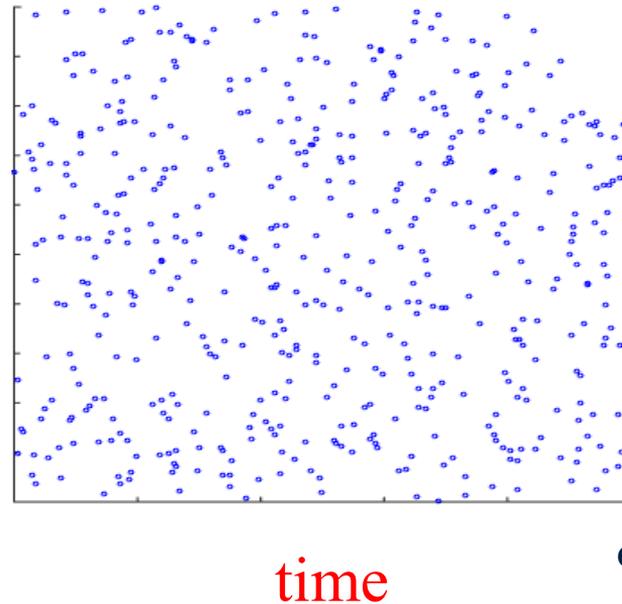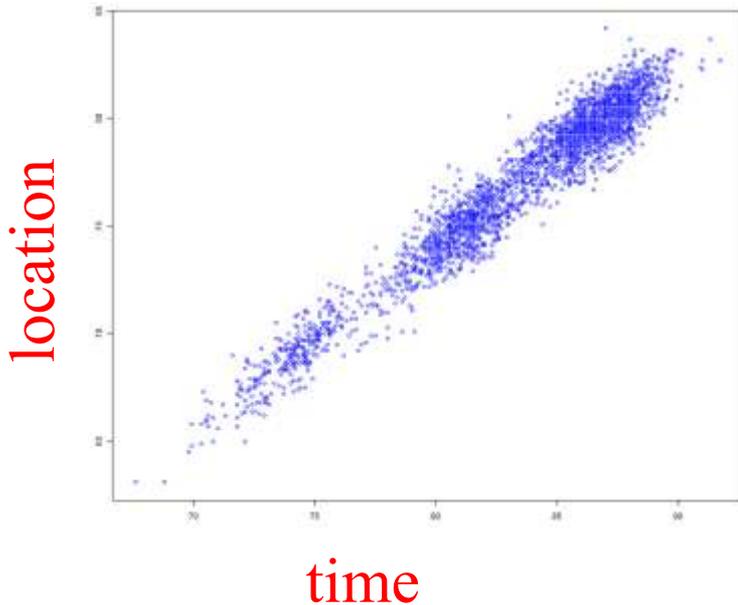
Storage Layout of a Table

Basics - Row vs. Column-Stores

Row-Store Storage

First name	Email	Phone #	Street Address

➔ Multiple rows are stored per page
➔ Traditional way for storage

☺ Easy to add a new record

☹ Might read in unnecessary data

Column Store Storage

First name	Email	Phone #	Street Address

➔ Stores each column in separate set of disk pages

☺ Only need to read relevant data

☺ Data compression

☹ Tuple writes might require multiple seeks

Improving Bad Access Patterns

- Minimize Random Memory Access
  - Apply filters first. Less accesses is better.

- Denormalize the Schema
  - Remove joins/lookups, add looked up stuff to the table (but.. makes it bigger)

- Trade Random Access For Sequential Access
  - perform a 100K random key lookups in a large table
    - ➔ put 100K keys in a hash table, then
      scan table and lookup keys in hash table

- Try to make the randomly accessed region smaller
  - Remove unused data from the structure
  - Apply data compression
  - Cluster or Partition the data (improve locality) …hard for social graphs

- If the random lookups often fail to find a result
  - Use a Bloom Filter

Assignment 1: Querying a Social Graph

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

event.cwi.nl/lsde/2016
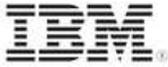
CSV file schema

- See: http://wikistats.ins.cwi.nl/lsde-data/practical_1
- Counts for sf3000 (total 37GB)

```
Knows(1.3B)
PersonFrom
PersonTo
```

```
Person (9M)
PersonId    PK
FirstName
LastName
Gender
Birthday
CreationDate
LocationIP
BrowserUsed
LocatedIn
```

```
interests(.2B)
PersonID
tagID
```

```
Tags (16K)
TagID
Name
URL
```

```
Place(1.4K
PlaceID PK
URL
type
```
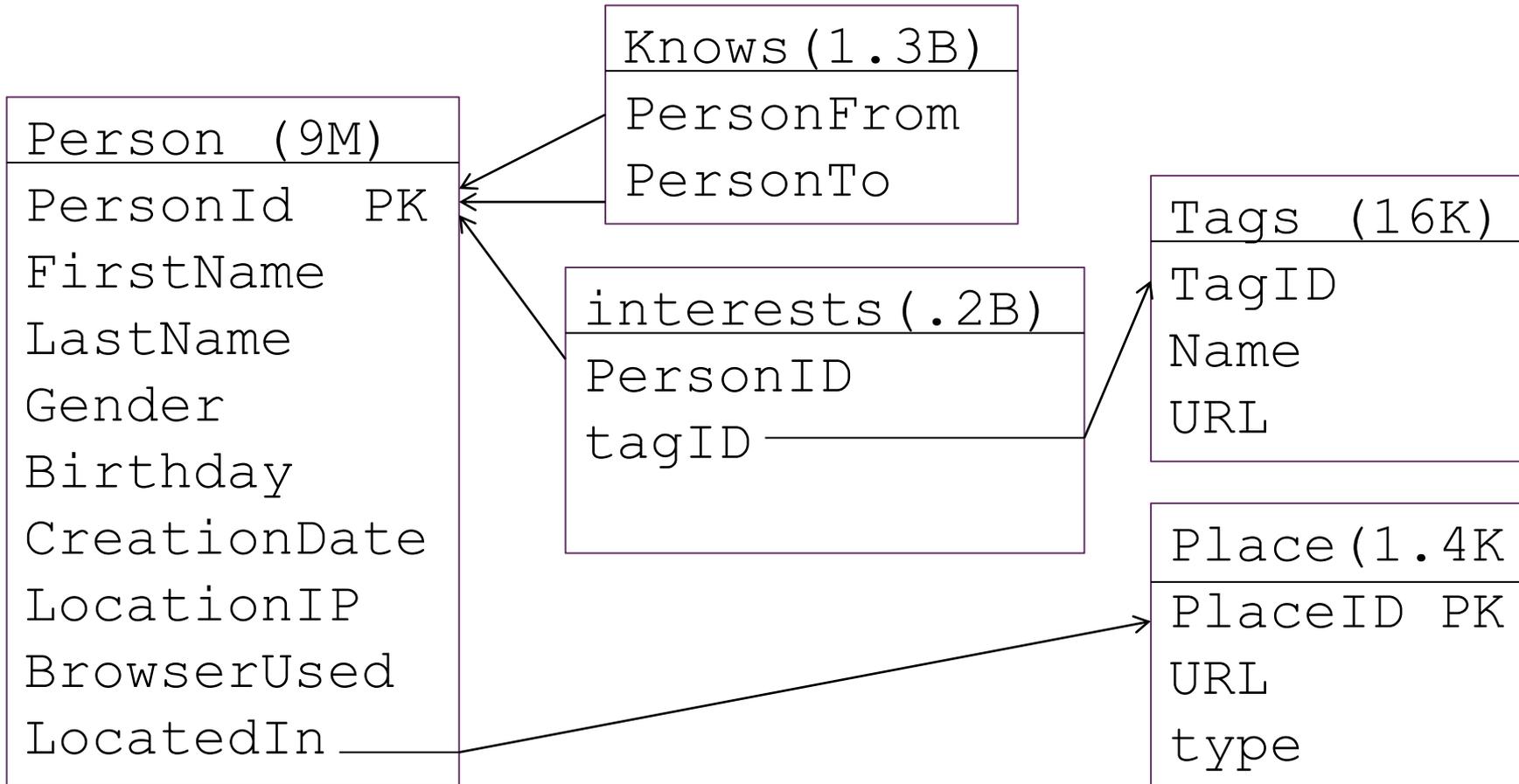
The Query

- 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 (with a discount!) as a birthday present for a friend ("who we know will love it" - the social network says) who lives in the same city, who is not yet interested in A1 yet, but is interested in other artists A2, A3 and A4 that the data mining model predicts to be correlated with A1.

The Query

For all persons P :

- *who have their birthday on or in between D1..D2*

- *who do not like A1 yet*

  we give a score of

  - *1 for liking any of the artists A2, A3 and A4 and*

  - *0 if not*

  the final score, the sum, hence is a number between 0 and 3.
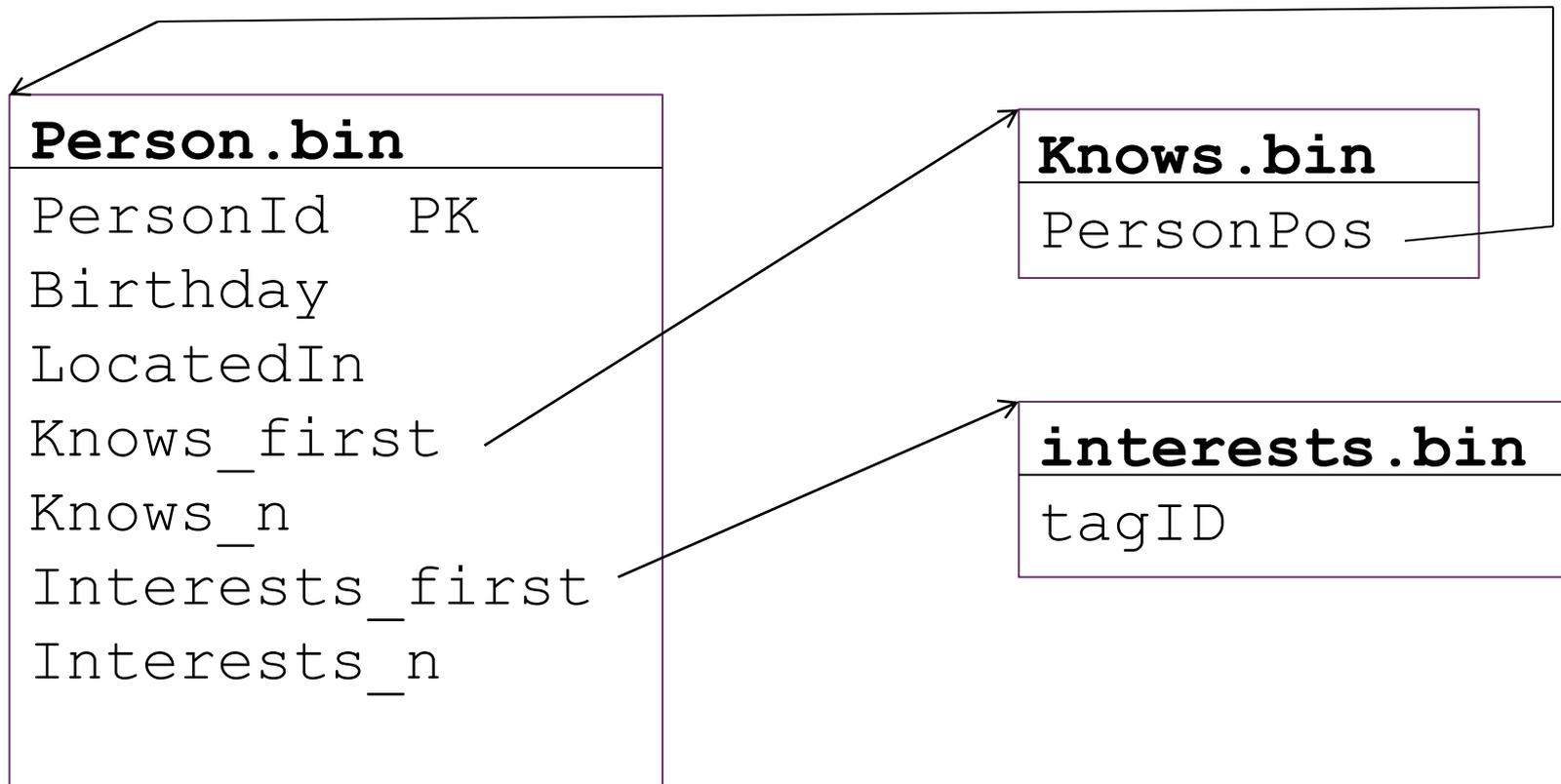
  Further, we look for friends F:

  - *Where P and F who know each other mutually*

  - *Where P and F live in the same city and*

  - *Where F already likes A1*

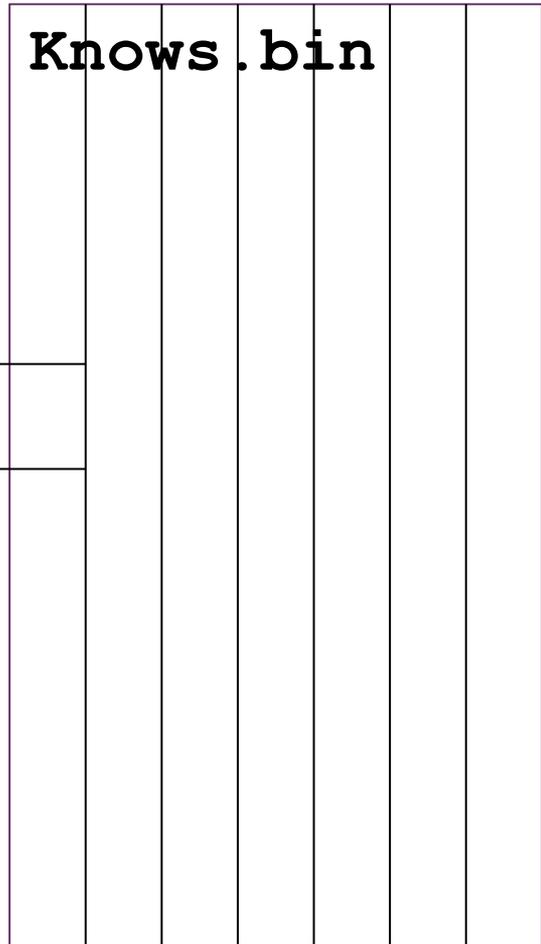 *The answer of the query is a table (score, P, F) with only scores > 0*

Binary files

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

Person.bin

PersonId   PK
Birthday
LocatedIn
Knows_first
Knows_n
Interests_first
Interests_n

Knows.bin

PersonPos

interests.bin

tagID

What it looks like

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

4bytes
* 1.3B

Knows.bin

interests.bin

knows_first

Person.bin

knows_n

48bytes
* 8.9M

2bytes
* 204M

The Naïve Implementation

The "cruncher" program

Go through the persons P sequentially

- *counting how many of the artists A2,A3,A4 are liked as the score for those with score>0:*
  - *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.
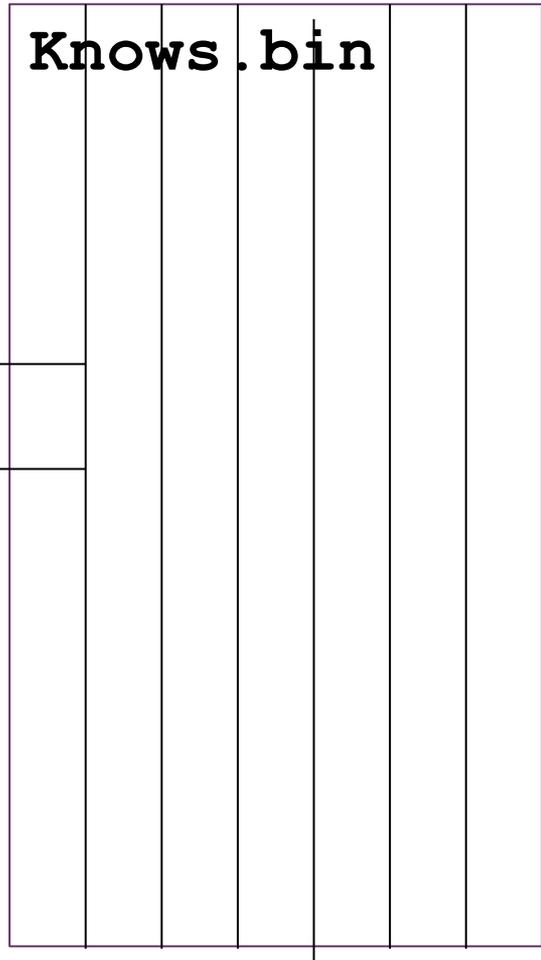
Naïve Query Implementation

- "cruncher"

4bytes
* 1.3B

Knows.bin

interests.bin

knows_first

Person.bin

knows_n

48bytes
* 8.9M

results

2bytes
* 204M

Naïve Query Implementation

- "cruncher"

4bytes
* 1.3B

Knows.bin

interests.bin

knows_first

Person.bin

knows_n

48bytes
* 8.9M

results

2bytes
* 204M

Naïve Query Implementation

- "cruncher"

4bytes
* 1.3B

Knows.bin

interests.bin

knows_first

Person.bin

knows_n

48bytes
* 8.9M

2bytes
* 204M

results

Naïve Query Implementation

- "cruncher"
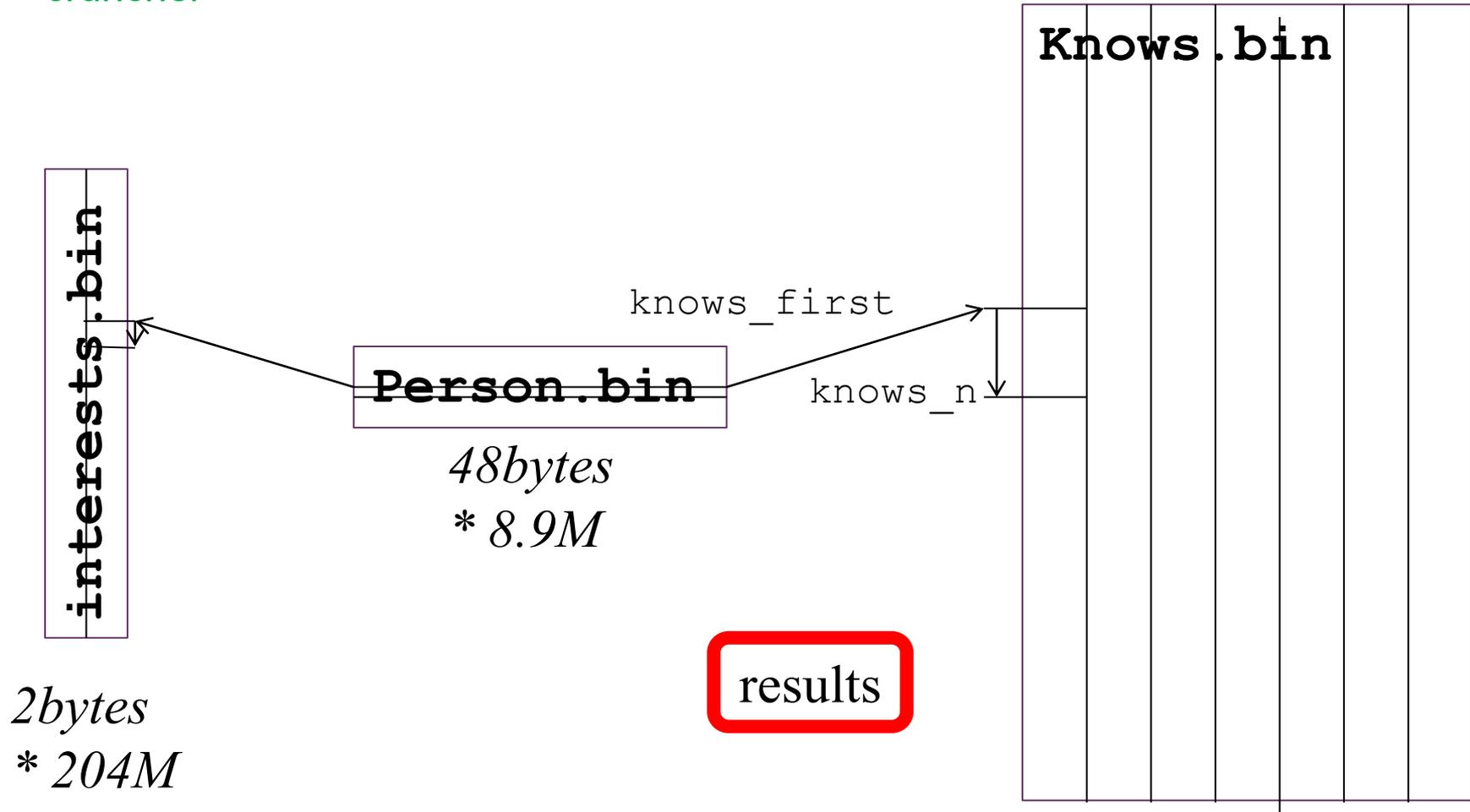
4bytes
* 1.3B

Knows.bin

knows_first

knows_n

Person.bin

48bytes
* 8.9M

interests.bin

2bytes
* 204M

results

Naïve Query Implementation

- "cruncher"

4bytes
* 1.3B

Knows.bin

interests.bin

knows_first

Person.bin

knows_n

48bytes
* 8.9M

results
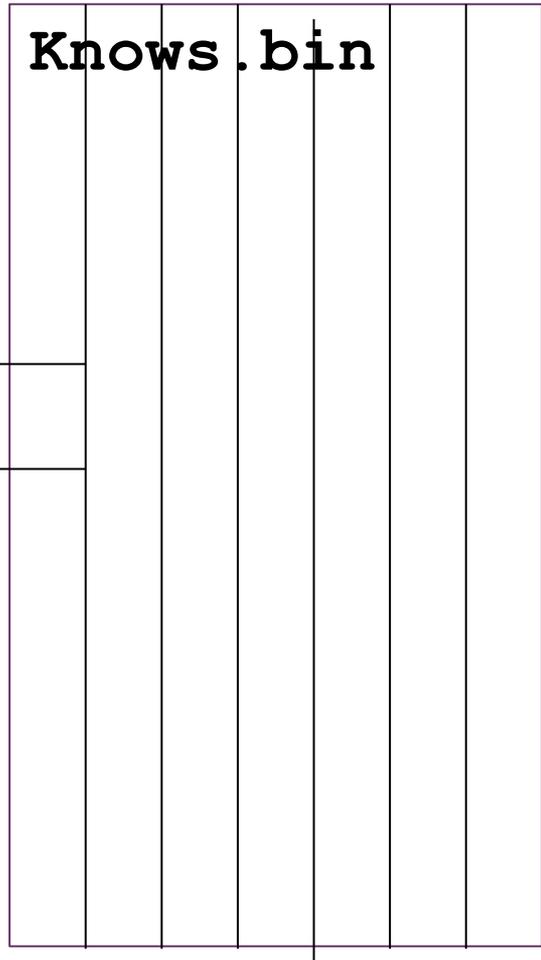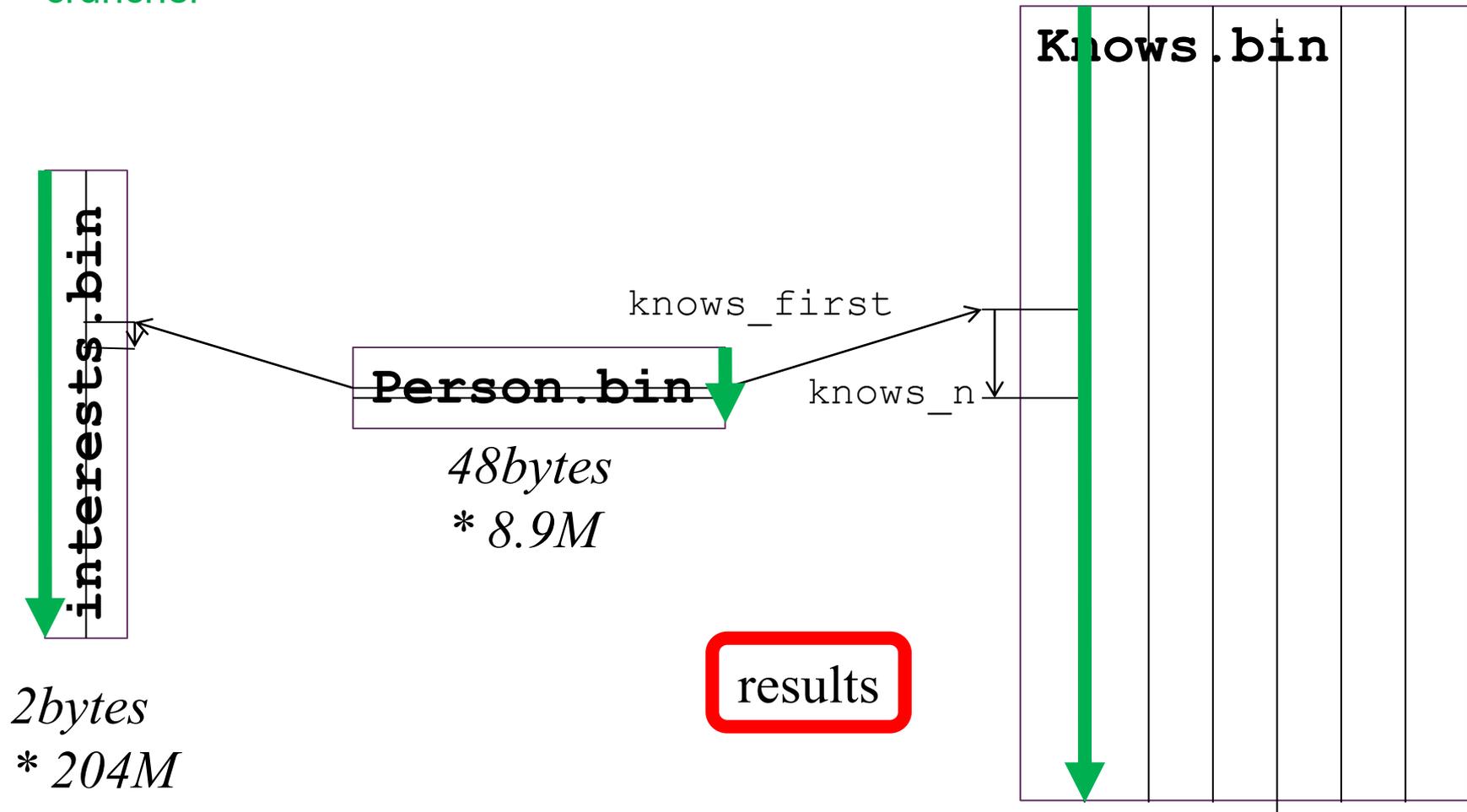
2bytes
* 204M
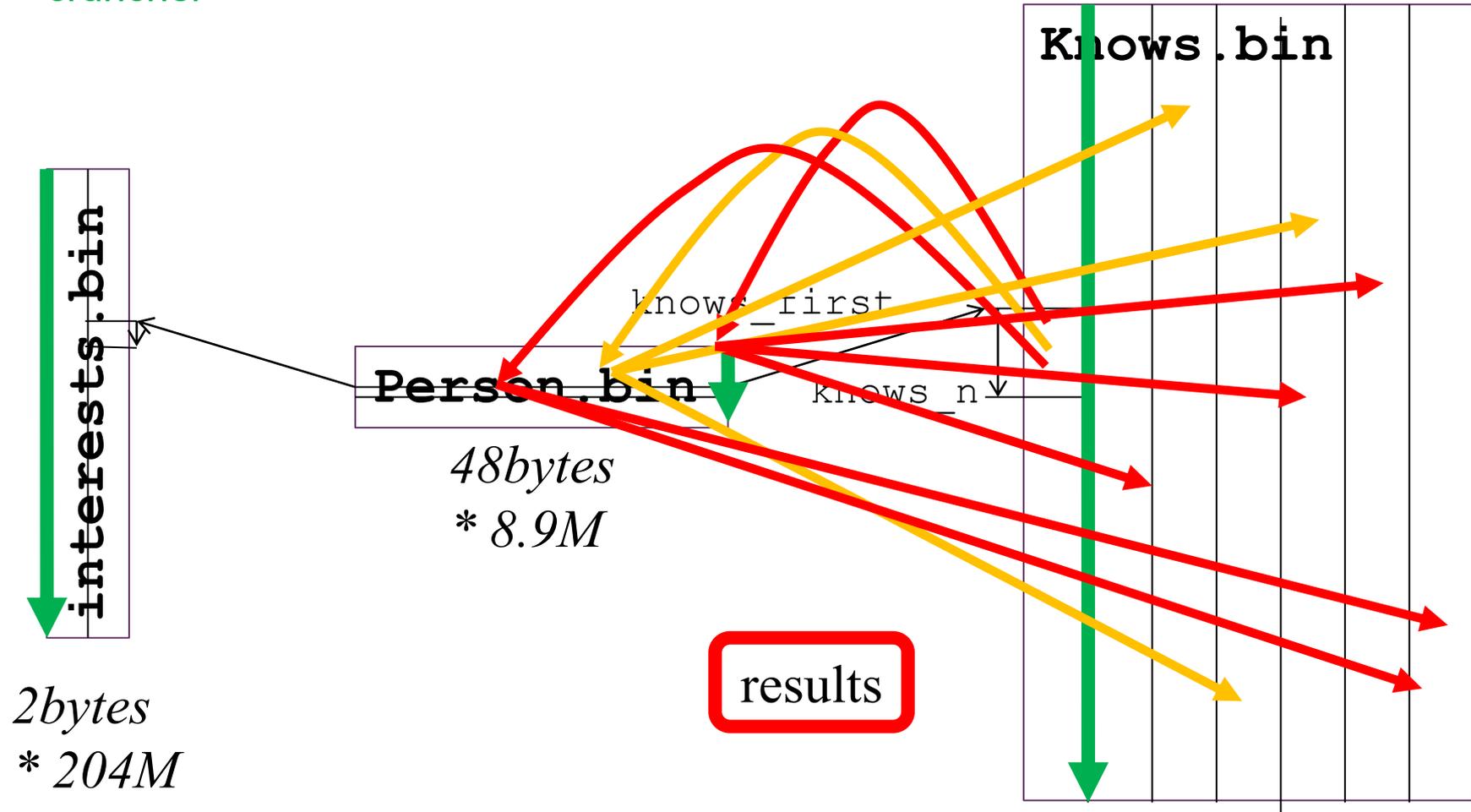
Challenges, questions

For the "reorg" program:

- Can we throw way unneeded data?

- Can we store the data more efficiently?

- Can we put the data in some order to improve access patterns?

For the "query" program:

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

- Can we improve the access pattern?

    – we trade random access for sequential access?

        - Multiple passes, instead of one?

We will meet on the leaderboard!