Centrum Wiskunde & Informatica

Benchmarking Graph Data Management Systems

EDBT Summer School 2015

Peter Boncz

boncz@cwi.nl

1. ***LDBC Social Network Benchmark***

   Tuesday: *LDBC & SNB introduction*

   Friday: *SNB in depth*

2. **SNB Programming Challenge** www.cwi.nl/~boncz/snb-challenge

   Tuesday: *what it is about & hardware properties & tips*

   Friday: *the solution space & winners*

CWI
Centrum Wiskunde & Informatica

The LDBC Social Network Benchmark

Interactive Workload

LDBC The graph & RDF benchmark reference

http://www.ldbcouncil.org

Orri Erling
OpenLink Software, UK
oerling@openlinksw.com

Alex Averbuch
Neo Technology, Sweden
alex.averbuch@neotechnology.com

Josep Larriba-Pey
Sparsity Technologies, Spain
larri@sparsity-technologies.com
Norbert Martinez

Hassan Chafi
Oracle Labs, USA
hassan.chafi@oracle.com

Andrey Gubichev
TU Munich, Germany
gubichev@in.tum.de
Thomas Neumann, Linnea Passing

Arnau Prat
Universitat Politècnica de Catalunya, Spain
aprat@ac.upc.edu
David Dominguez
Xavier Sanchez

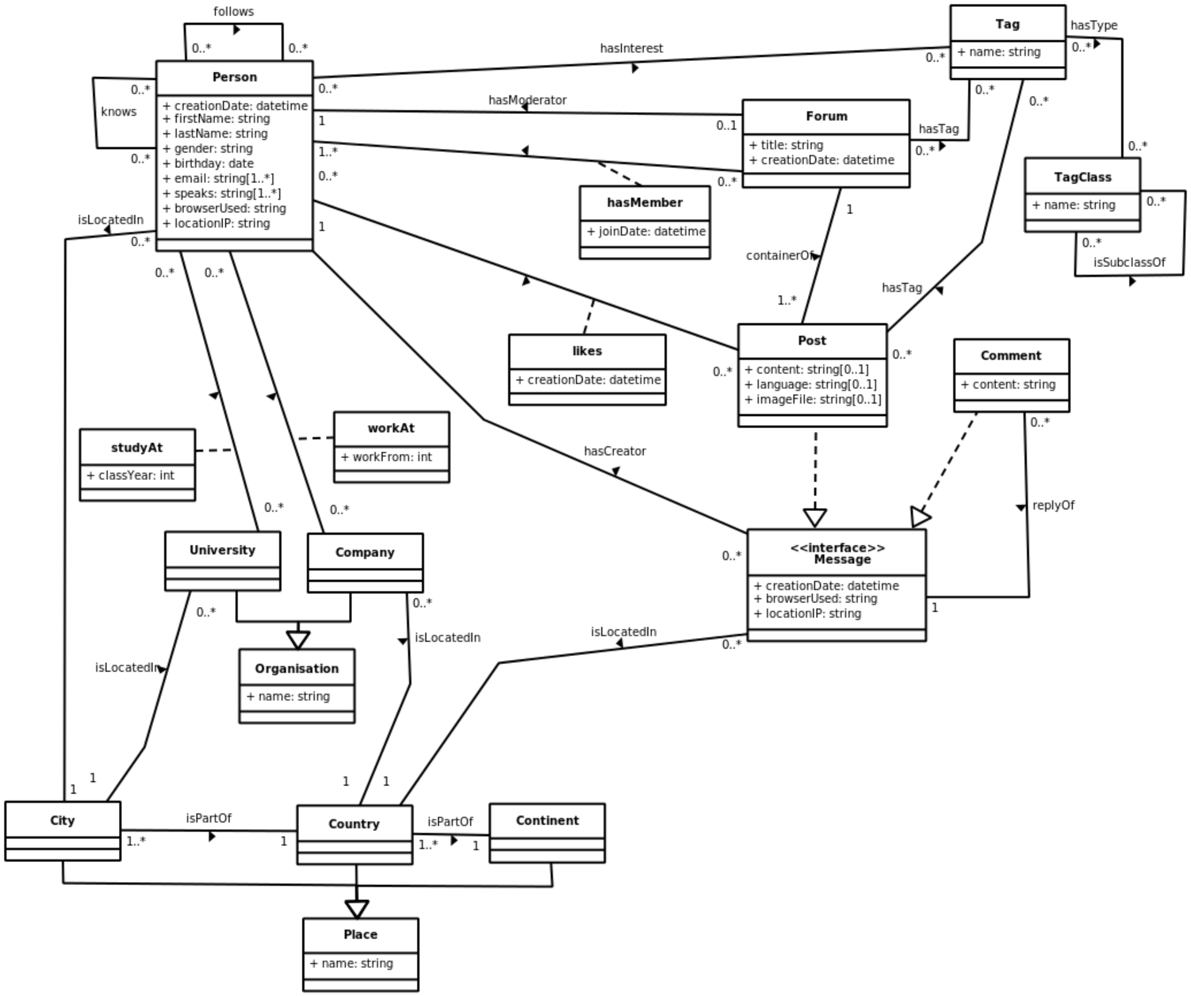
Minh-Duc Pham
VU University Amsterdam, The Netherlands
m.d.pham@vu.nl
Renzo Angles (U. Talca)

Peter Boncz
CWI, Amsterdam, The Netherlands
boncz@cwi.nl

SNB "Task Force" acknowledgements

SEVENTH FRAMEWORK PROGRAMME

Social Network Benchmark: schema



Database Benchmark Design

Desirable properties:

- Relevant. → "Choke Points"
- Representative.
- Understandable.
- Economical.
- Accepted.
- Scalable.
- Portable.
- Fair.
- Evolvable.
- Public.

Jim Gray (1991) *The Benchmark Handbook for Database and Transaction Processing Systems*

Dina Bitton, David J. DeWitt, Carolyn Turbyfill (1993) *Benchmarking Database Systems: A Systematic Approach*

Multiple TPCTC papers, e.g.
Karl Huppler (2009) *The Art of Building a Good Benchmark*

Stimulating Technical Progress

- An aspect of ‘Relevant’
- The benchmark metric
  - depends on,
  - or, rewards:

  solving certain technical challenges

(not commonly solved by technology at benchmark design time)

Benchmark Design with Choke Points

Choke-Point = well-chosen difficulty in the workload

- "difficulties in the workloads"
  - arise from Data (distribs)+Query+Workload
  - there may be different technical solutions to address the choke point
    - or, there may not yet exist optimizations

      → lot's of research opportunities!

Example: TPC-H choke points

- Even though it was designed without specific choke point analysis
- TPC-H contained a lot of interesting challenges
  – many more than Star Schema Benchmark
  – considerably more than Xmark (XML DB benchmark)
  – not sure about TPC-DS (yet)

CP1.4 Dependent GroupBy Keys

Q10

```
SELECT c_custkey,  c_name, c_acctbal,
   sum(l_extendedprice * (1 - l_discount)) as revenue,
    n_name,  c_address,  c_phone, c_comment
FROM  customer, orders,  lineitem,  nation
WHERE  c_custkey = o_custkey and l_orderkey =
   o_orderkey
   and o_orderdate >= date '[DATE]'
   and o_orderdate < date '[DATE]' + interval '3'
   month
   and l_returnflag = 'R‘ and c_nationkey =
   n_nationkey
GROUP BY
   c_custkey, c_name,   c_acctbal,  c_phone,  n_name,
   c_address, c_comment
ORDER BY revenue DESC
```

CP1.4 Dependent GroupBy Keys

Q10

```
SELECT c_custkey,  c_name, c_acctbal,
   sum(l_extendedprice * (1 - l_discount)) as revenue,
    n_name,  c_address,  c_phone, c_comment
FROM  customer, orders,  lineitem,  nation
WHERE  c_custkey = o_custkey and l_orderkey =
   o_orderkey
   and o_orderdate >= date '[DATE]'
   and o_orderdate < date '[DATE]' + interval '3'
   month
   and l_returnflag = 'R‘ and c_nationkey =
   n_nationkey
GROUP BY
   c_custkey, c_name,   c_acctbal,  c_phone,
   c_address, c_comment, n_name
ORDER BY revenue DESC
```

CP1.4 Dependent GroupBy Keys

- Functional dependencies:

  c_custkey ➔ c_name, c_acctbal, c_phone, c_address, c_comment, c_nationkey ➔ n_name

- Group-by hash table should exclude the colored attrs ➔ less CPU+ mem footprint
- in TPC-H, one can choose to declare primary and foreign keys (all or nothing)
  - this optimization requires declared keys
  - Key checking slows down RF (insert/delete)

CP2.2 Sparse Joins

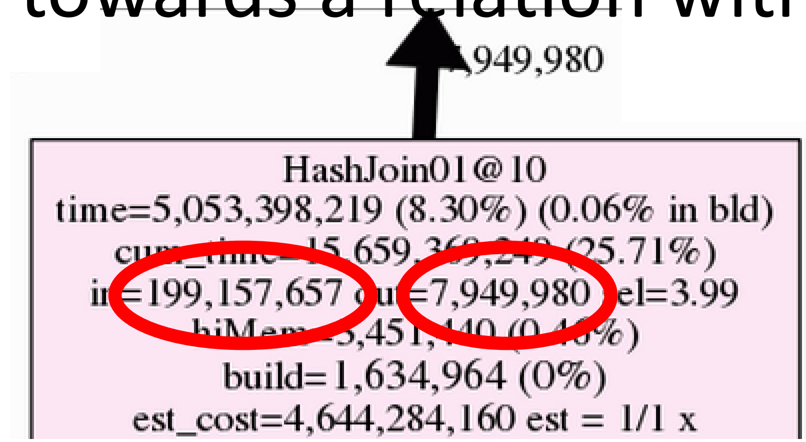
- Foreign key (N:1) joins towards a relation with a selection condition
  - Most tuples will *not* find a match
  - Probing (index, hash) is the most expensive activity in TPC-H
- Can we do better?
  - Bloom filters!

CP2.2 Sparse Joins

- Foreign key (N:1) joins towards a relation with a selection condition

Q21

probed: 200M tuples
result: 8M tuples
➔ 1:25 join hit ratio



Vectorwise:
TPC-H joins typically accelerate 4x
Queries accelerate 2x

2G cycles 29M probes ➔ cost would have been 14G cycles ~= 7 sec

```
#PROF 2021162220   OWN 28950172  9.8avg rdtsc  307565 calls vht_lookup_keys() "vht_lookup_keys" in con
#PROF 1575739535   OWN 199097581  7.9avg rdtsc  307534 calls sel_bitfiltercheck_uchr_col_slng_val_sint_
```

1.5G cycles 200M probes ➔ 85% eliminated

CP4.1 Raw Expression Arithmetic

How fast is a query processor in computing, e.g.

- Numerical Arithmetic
- Aggregates
- String Matching

Q1

```
SELECT
   l_returnflag, l_linestatus, count(*),
   sum(l_quantity),sum(l_extendedprice),
   sum(l_extendedprice*(1-l_discount)),
   sum(l_extendedprice*(1-l_discount)*(1+l_tax)),
   avg(l_quantity),avg(l_extendedprice),avg(l_discount),
FROM lineitem
```

SIMD? Interpreter Overhead?
Vectorwise, Virtuoso, SQLserver cstore → vectorized execution
Hyper, Netteza, ParAccel → JIT query compilation
Kickfire, ParStream → hardware compilation (FPGA/GPU)

TPCTC 2013: “TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark”

CP5.2 Subquery Rewrite

Q17

```
SELECT sum(l_extendedprice) / 7.0 as avg_yearly
FROM lineitem,  part
WHERE p_partkey = l_partkey
   and p_brand = '[BRAND]'
   and p_container = '[CONTAINER]'
   and l_quantity <( SELECT 0.2 * avg(l_quantity)
          FROM lineitem
          WHERE l_partkey = p_partkey)
```

This subquery can be extended with restrictions from the outer query.

```
          SELECT 0.2 * avg(l_quantity)
            ineitem
            partkey = p_partkey
            brand = '[BRAND]'
            p_container = '[CONTAINER]'
```

Hyper:
CP5.1+CP5.2+CP5.3
results in 500x faster
Q17

+ CP5.3 Overlap between Outer- and Subquery.

Choke Point Wrap up

Choke-point based benchmark design

- What are Choke-points?
  - examples from good-old TPC-H
- Graph benchmark Choke-Point, in-depth:
  - **Structural Correlation in Graphs**
  - and what we do about it in LDBC

Graphalytics Choke Points

- Excessive network utilization
- Large graph memory footprint
- Poor Access Locality
- Skewed Execution Intensity

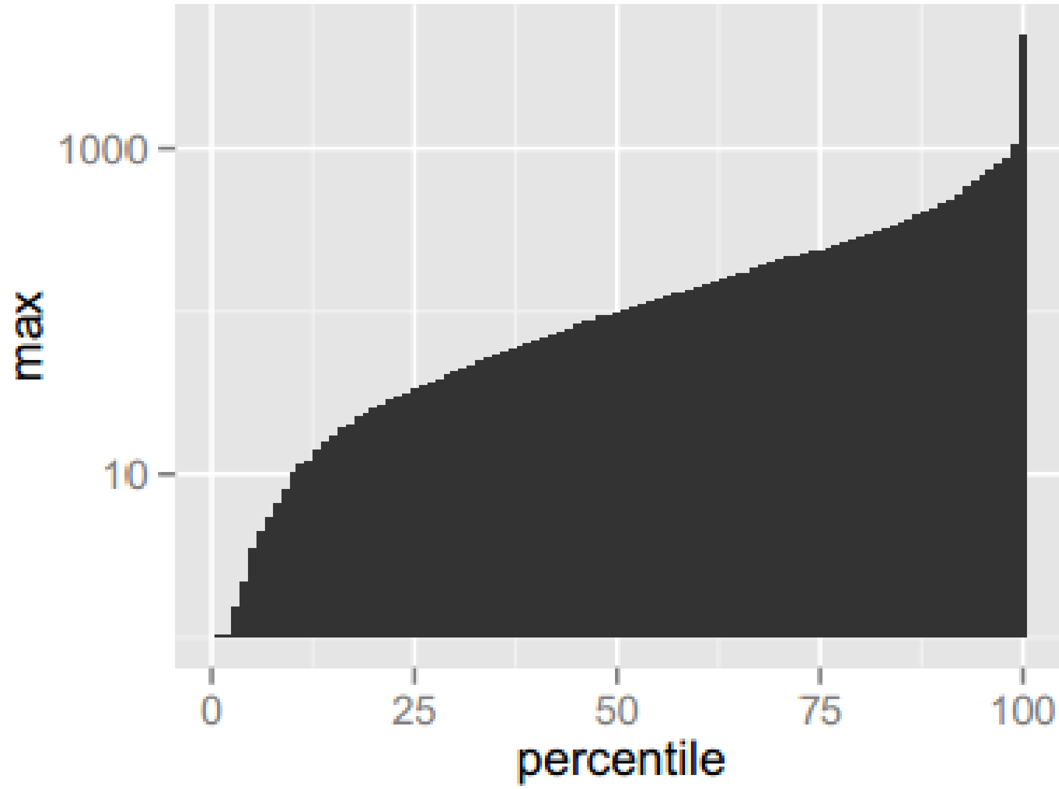
DATAGEN: social network generator

advanced generation of:

- network structure
  - **Power law** distributions, small diameter

Friendship Degree Distribution

- Based on “Anatomy of Facebook” blogpost (2013)
- Diameter increases logarithmically with scale factor
  - New:
    function has been
    made pluggable



DATAGEN: social network generator

advanced generation of:

- network structure
  - Power law distributions, small diameter
- property values
  - **realistic**, **correlated** value distributions

Data correlations between attributes

```
SELECT personID from person
WHERE firstName = 'Joachim' AND addressCountry = 'Germany'
```

Anti-Correlation

```
SELECT personID from person
WHERE firstName = 'Cesare'  AND addressCountry = 'Italy'
```

- Query optimizers may underestimate or overestimate the result size of conjunctive predicates

Joachim Loew
Cesare

Cesare Prandelli
Joachim

Data correlations **between attributes**

```
SELECT COUNT(*)
FROM paper pa1 JOIN conferences cn1 ON pa1.journal = jn1.ID
     paper pa2 JOIN conferences cn2 ON pa2.journal = jn2.ID
WHERE  pa1.author = pa2.author   AND
             cn1.name = 'VLDB'  AND  cn2.name = 'SIGMOD'
```

Data correlations **over joins**

```
SELECT COUNT(*)
FROM paper pa1 JOIN conferences cn1 ON pa1.journal = cn1.ID
     paper pa2 JOIN conferences cn2 ON pa2.journal = cn2.ID
WHERE  pa1.author = pa2.author    AND
               cn1.name = 'VLDB'   AND  cn2.name = 'SIGMOD' 'Nature'
```

- A challenge to the optimizers to adjust estimated join hit ratio

  `pa1.author = pa2.author`

  depending on other predicates

Correlated predicates are still a frontier area in database research

Realistic Correlated Value Distributions

- Person.firstname correlates with Person.location
  - Values taken from **DBpedia**
- Many other correlations and dependencies..
  - e.g. university depends on location

Person.location =<Germany>

Name	Number
Karl	215
Hans	190
Wolfgang	174
Fritz	159
Rudolf	159
Walter	150
Franz	115
Paul	109
Otto	99
Wilhelm	74

Person.location =<China>

Name	Number
Yang	961
Chen	929
Wei	887
Lei	789
Jun	779
Jie	778
Li	562
Hao	533
Lin	456
Peng	448

- In forum discussions, people read DBpedia articles to each other (= correlation between message text and discussion topic)
  - Topic = DBpedia article title
  - Text = one sentence of the article

Generating Property Values

- How do data generators generate values? E.g. `FirstName`

- **Value** Dictionary **D()**
  - a fixed set of values, e.g.,

    {“Andrea”,“Anna”,“Cesare”,“Camilla”,“Duc”,“Joachim”, .. }

- **Probability** density function **F()**
  - steers how the generator chooses values
    - cumulative distribution over dictionary entries determines which value to pick
  - could be anything: uniform, binomial, geometric, etc…
    - geometric (discrete exponential) seems to explain many natural phenomena

Generating **Correlated** Property Values

- How do data generators generate values? E.g. `FirstName`
- **Value** Dictionary **D()**
- **Probability** density function **F()**
- **Ranking** Function **R()**
  - Gives each value a unique rank between one and **|D|**
    – determines which value gets which probability
  - Depends on some parameters (parameterized function)
    – value frequency distribution becomes correlated by the parameters or **R()**

Generating **Correlated** Property Values

- How do data generators generate values? E.g. `FirstName`
- **Value** Dictionar
  {"Andrea"," ... **'O**
- **Probability** d
  geometric d
- **Ranking** Function **R(gender,country,birthyear)**
  - **gender, country, birthyear → correlation parameters**

How to implement R()?

We need a table storing

limited #combinations

|Gender| X |Country| X |BirthYear| X |D|

Potentially Many! ☹

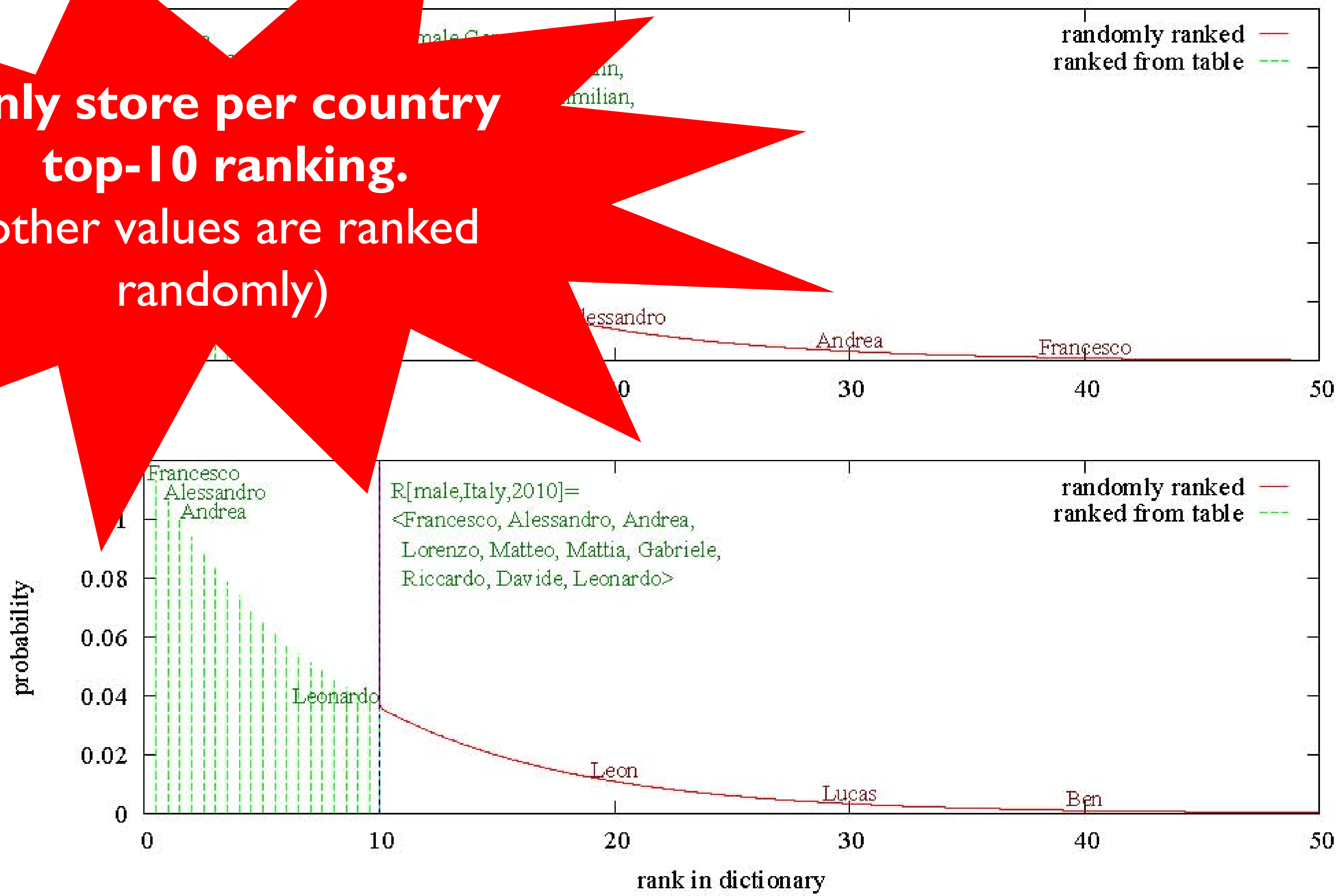
Solution:

- Just store the rank of the **top-N** values, not all **|D|**
- Assign the rank of the other dictionary values randomly

Compact Correlated Property Value Generation

Using geometric distribution for function F()

Only store per country top-10 ranking.
(other values are ranked randomly)



Correlated Value Property in LDBC SNB

- Main source of dictionary values from DBpedia (http://dbpedia.org)

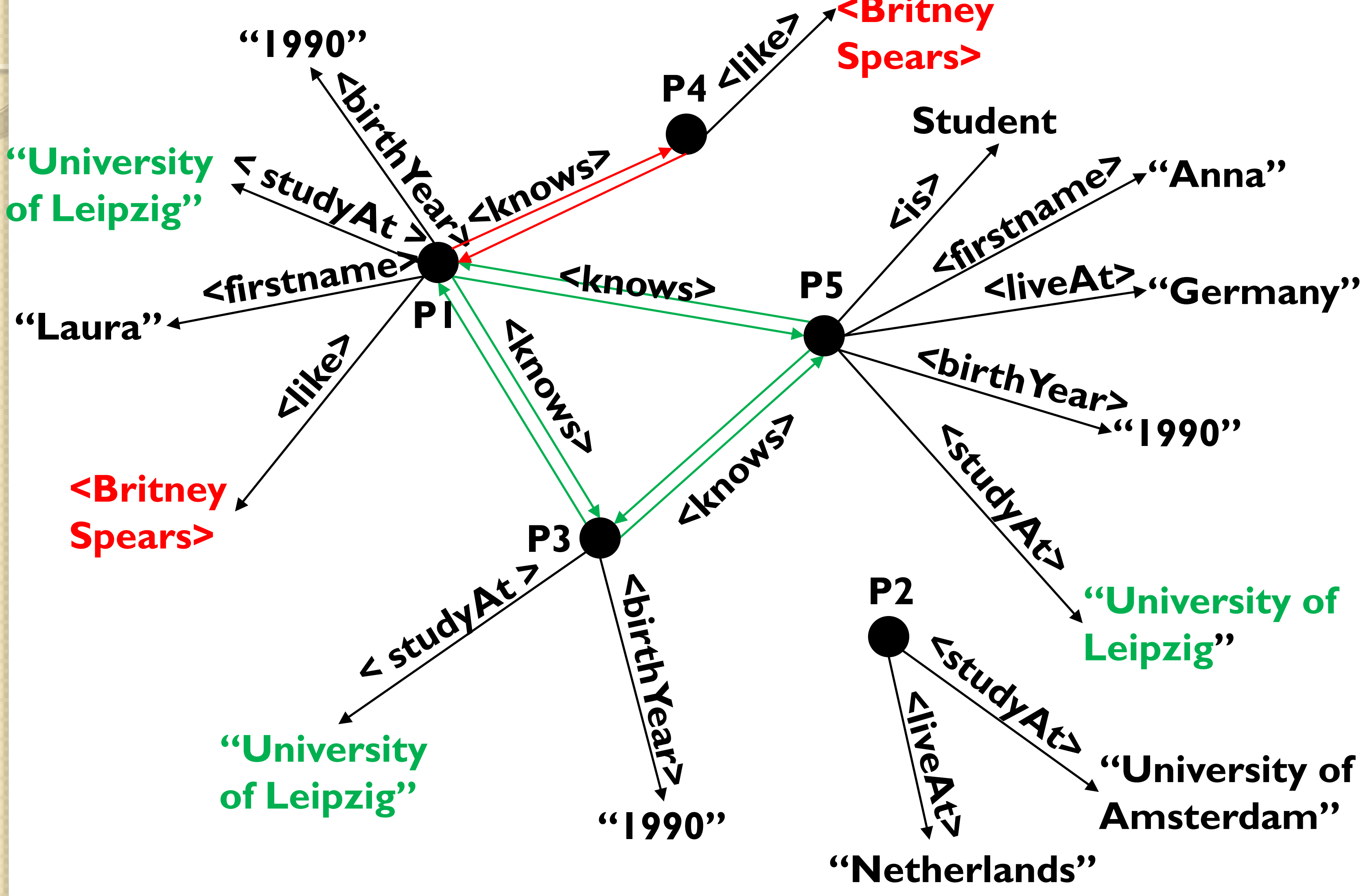
(person.location, person.gender)	person.firstName	*(typical names)*
	person.interests	*(popular artist)*
person.location	person.lastName	*(typical names)*
	person.university	*(nearby universities)*
	person.company	*(in country)*
	person.languages	*(spoken in country)*
person.language	person.forum.message.language	*(speaks)*
person.interests	person.forum.post.topic	*(in)*
post.topic	post.text	*(DBpedia article lines)*
	post.comment.text	*(DBpedia article lines)*
person.employer	person.email	*(@company, @university)*
(friendship.userId1, friendship.userId2)	friendship.terminator	*(=one of the two)*
message.photoLocation	message.latitude	*(matches location)*
	message.longitude	*(matches location)*
friendship.requestDate	friendship.approveDate	*(>)*
	friendship.deniedDate	*(>)*
person.birthDate	person.createdDate	*(>)*
person.createdDate	person.forum.message.createdDate	*(>)*
	person.forum.createdDate	*(>)*
forum.createdDate	message.photoTime	*(>)*
	forum.post.createdDate	*(>)*
	forum.groupmembership.joinedDate	*(>)*
message.createdDate	message.comment.createdDate	*(>)*

DATAGEN: social network generator

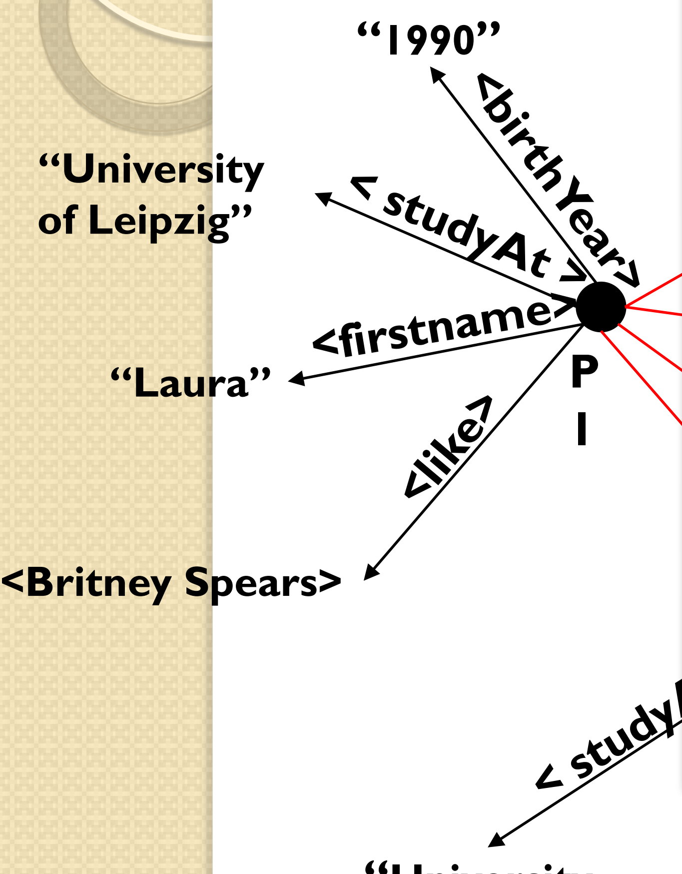
advanced generation of:

- network structure
  - Power law distributions, small diameter
- property values
  - realistic, correlated value distributions
  - temporal correlations / "flash mobs"
- **correlations between values and structure**
  - 2 correlation "dimensions": location & interests

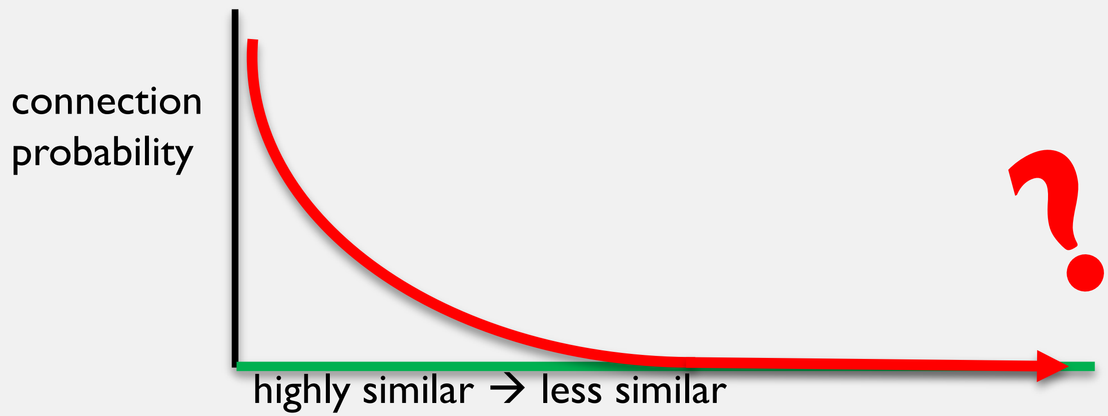
Correlated Edge Generation



Simple approach

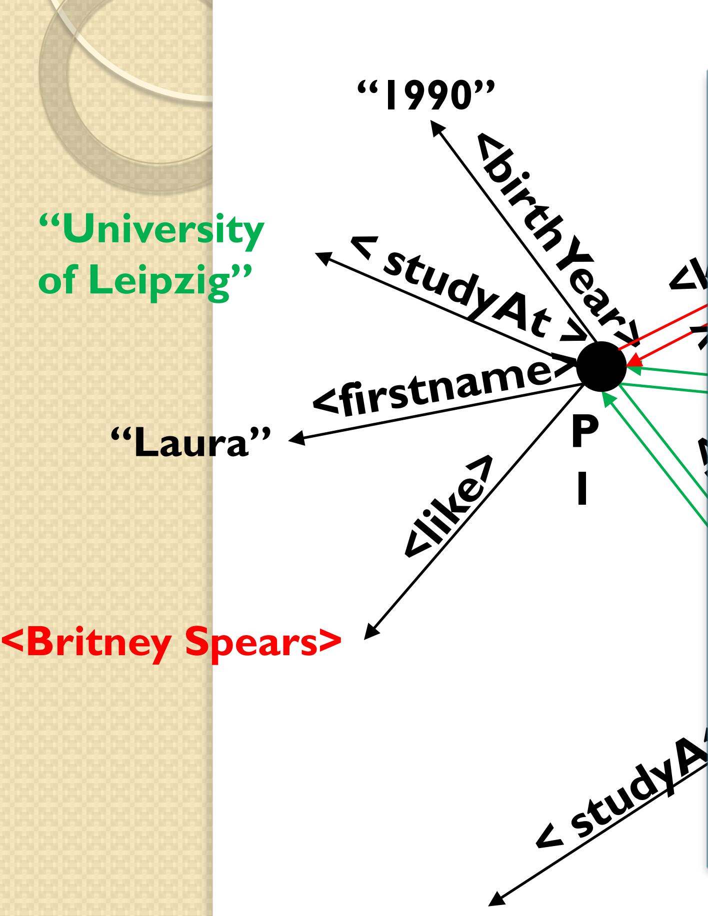


- Compute **similarity** of two nodes based on their (correlated) **properties**.
- Use a **probability density function** wrt to this similarity for connecting nodes

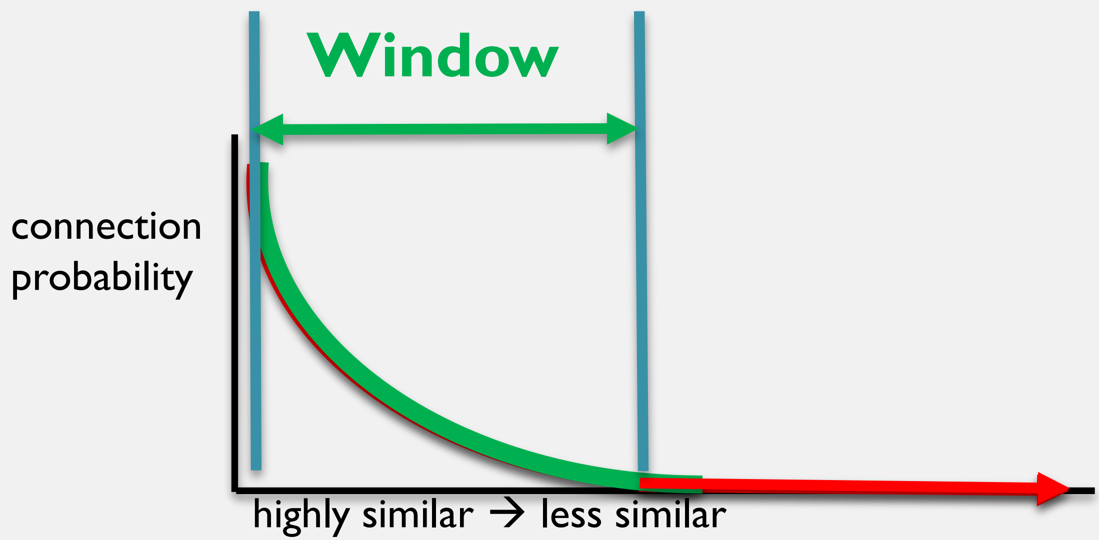


Danger: this is very expensive to compute on a large graph! (quadratic, random access)

Our observation

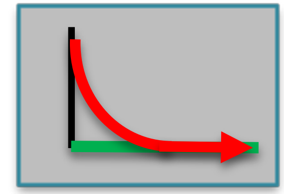


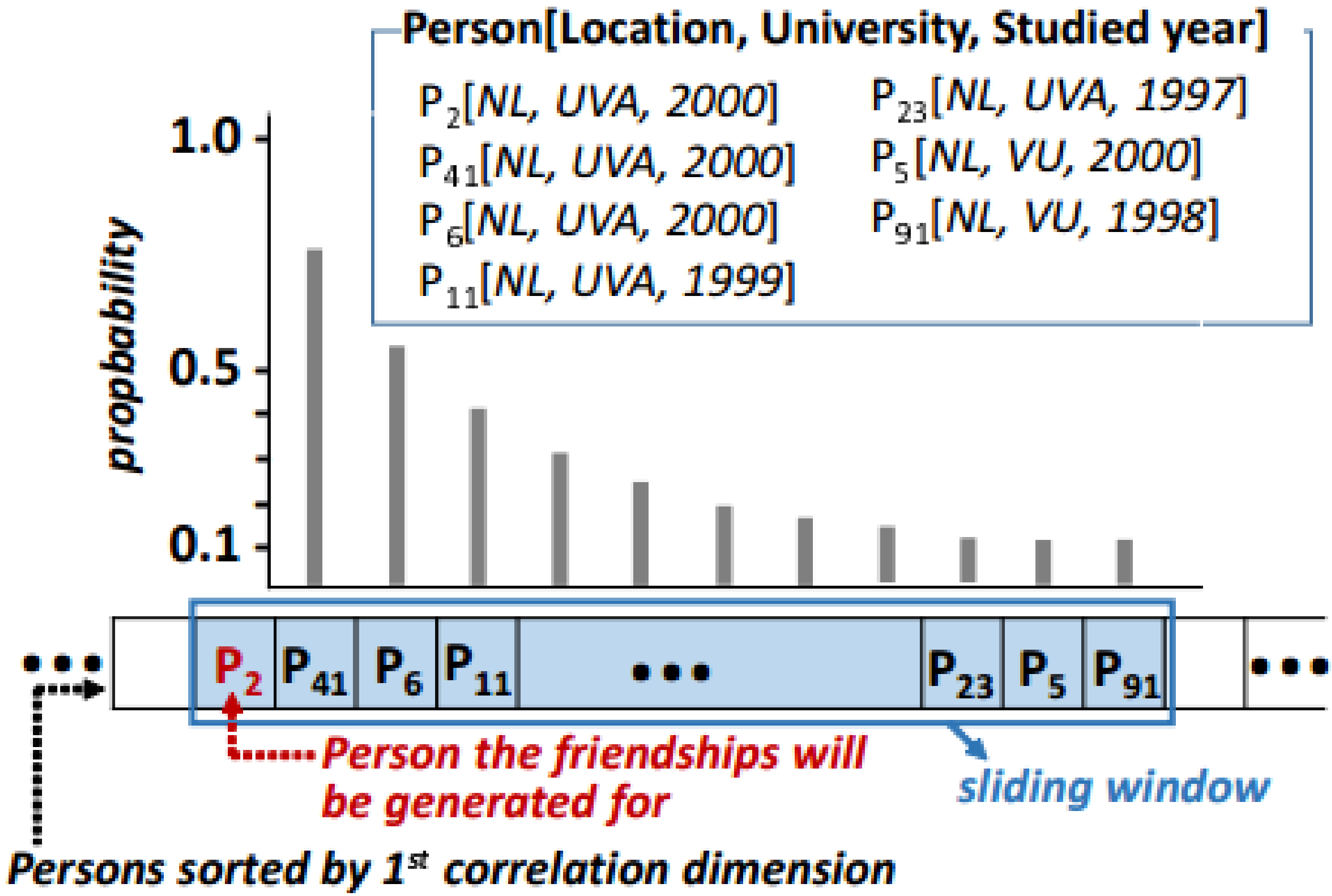
Trick: disregard nodes with too large similarity distance (only connect nodes in a **similarity window**)



Probability that two nodes are connected is **skewed** *w.r.t* the **similarity** between the nodes (due to probability distr.)

MapReduce data generation: one map pass per Correlation Dimension





TPCTC 2012: "S3G2: A Scalable Structure-correlated Social Graph Generator"

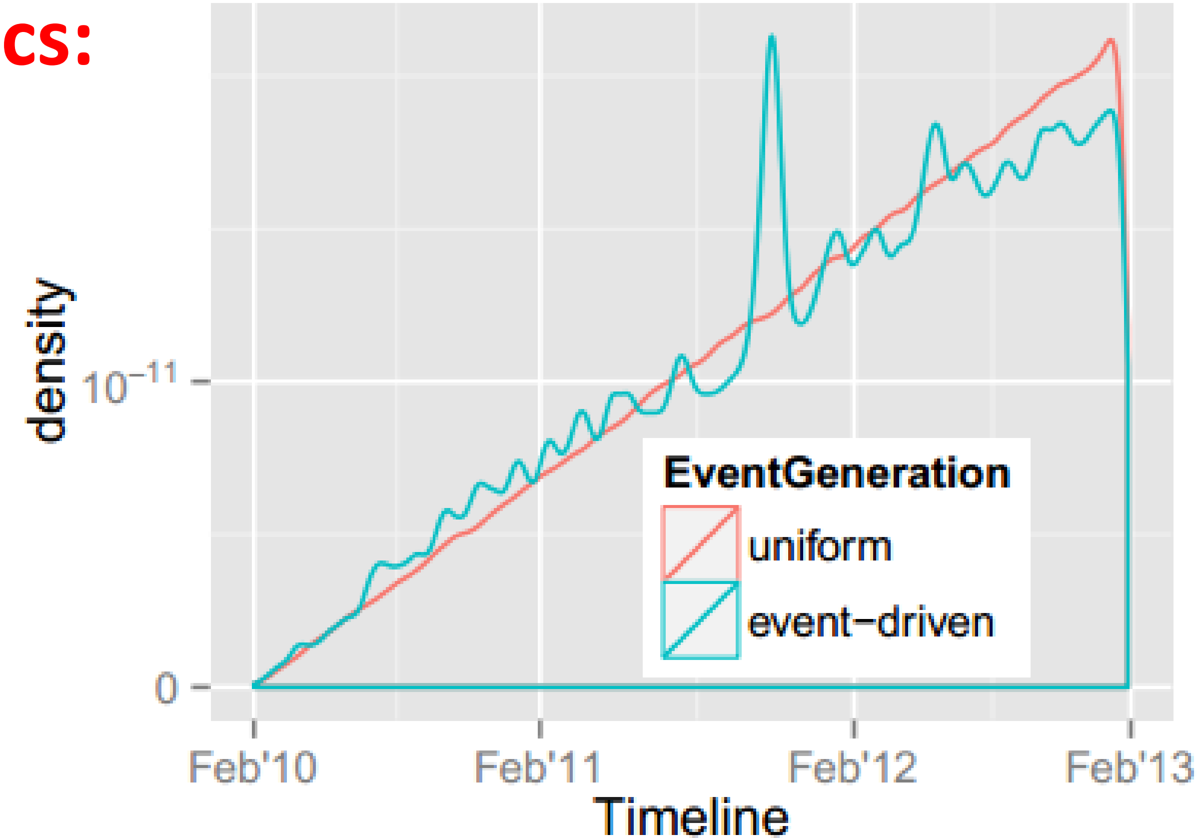
DATAGEN: social network generator

advanced generation of:

- network structure
  - Power law distributions, small diameter
- property values
  - realistic, correlated value distributions
  - **temporal correlations** / “flash mobs”

Temporal Effects (Flash Mobs)

- Forum posts generation spikes in time **for certain topics:**



DATAGEN: Scaling

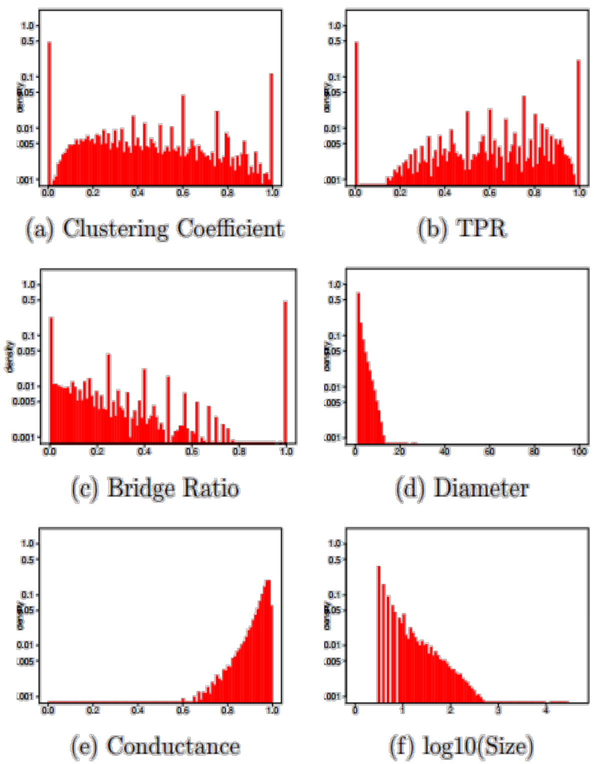
- Scale Factor (SF) is the size of the CSV input data in GB
- Some Virtuoso SQL stats at SF=30:

SFs	Number of entities (x 1000000)					
	Nodes	Edges	Persons	Friends	Messages	Forums
30	99.4	655.4	0.18	14.2	97.4	1.8
100	317.7	2154.9	0.50	46.6	312.1	5.0
300	907.6	6292.5	1.25	136.2	893.7	12.6
1000	2930.7	20704.6	3.60	447.2	2890.9	36.1

Table	**Size (MB)**	**Largest Index (MB)**
post	76815	ps_content (41697)
likes	23645	l_creationdate (11308)
forum_person	9343	fp_creationdate (5957)

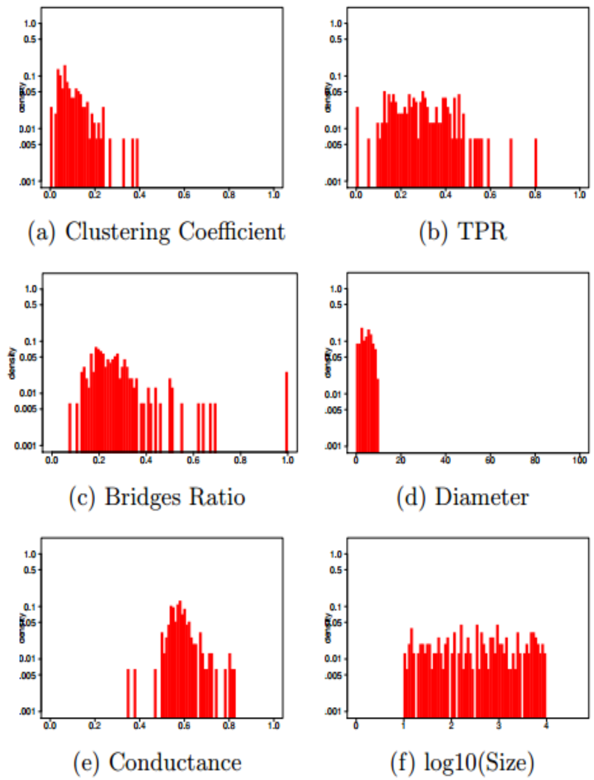
DATAGEN: Graph Characteristics

Livejournal



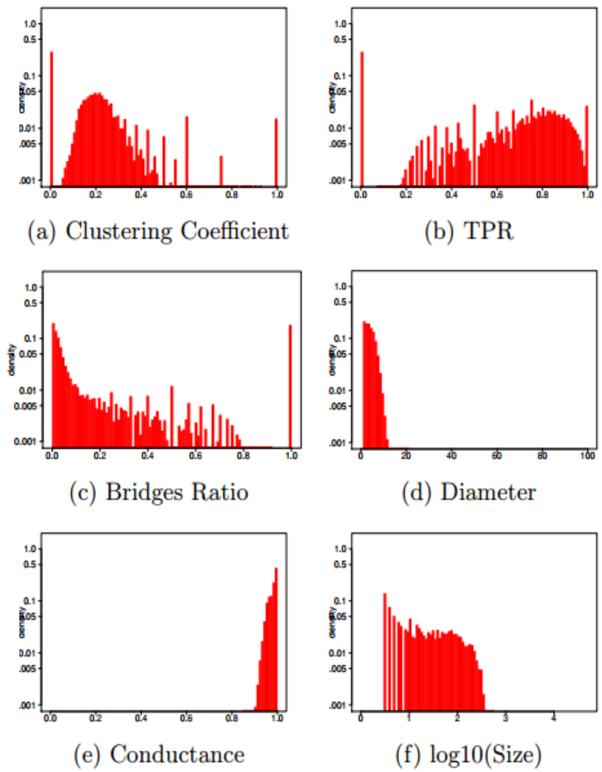
(a) Clustering Coefficient (b) TPR (c) Bridge Ratio (d) Diameter (e) Conductance (f) log10(Size)

LFR3 (synthetic)



(a) Clustering Coefficient (b) TPR (c) Bridges Ratio (d) Diameter (e) Conductance (f) log10(Size)

LDBC DATAGEN



(a) Clustering Coefficient (b) TPR (c) Bridges Ratio (d) Diameter (e) Conductance (f) log10(Size)

GRADES2014 “How community-like is the structure of synthetically generated graphs” - Arnau Prat(DAMA-UPC); David Domínguez-Sal (Sparsity Technologies)

Interactive Workload

MapReduce-base data generation

- Generate 3 years of network activity for a certain amount of persons
  - 33 months of data ➔ bulk load
  - 3 months of data ➔ insert queries
- Scalable (SF1000 in one hour on 10 small compute nodes)
  - can also be used without a cluster (pseudo-distributed)

During data generation, we perform Parameter Curation to derive suitable parameters for the complex-read-only query set

CWI

Centrum Wiskunde & Informatica

SNB Interactive Workload

Q1. *Extract description of friends with a given name* Given a person's `firstName`, return up to 20 people with the same first name, sorted by increasing distance (max 3) from a given `person`, and for people within the same distance sorted by last name. Results should include the list of workplaces and places of study.

Q2. *Find the newest 20 posts and comments from your friends.* Given a start `Person`, find (most recent) Posts and Comments from all of that Person's friends, that were created before (and including) a given `Date`. Return the top 20 Posts/Comments, and the Person that created each of them. Sort results descending by creation date, and then ascending by Post identifier.

Q3. *Friends within 2 steps that have recently traveled to countries X and Y.* Find friends and friends of friends of a given `Person` who have made a post or a comment in the foreign `CountryX` and `CountryY` within a specified period of `DurationInDays` after a `startDate`. Return top 20 Persons, sorted descending by total number of posts.

Q4. *New Topics.* Given a start `Person`, find the top 10 most popular Tags (by total number of posts with the tag) that are attached to Posts that were created by that Person's friends. Only include Tags that were attached to Posts created within a given `time interval`, and that were never attached to Posts created before this interval.

Q5. *New groups.* Given a start `Person`, find the top 20 Forums which that Person's friends and friends of friends became members of after a given `Date`. Sort results descending by the number of Posts in each Forum that were created by any of these Persons.

Q6. *Tag co-occurrence.* Given a start `Person` and some `Tag`, find the other Tags that occur together with this Tag on Posts that were created by start Person's friends and friends of friends. Return top 10 Tags, sorted descending by the count of Posts that were created by these Persons, which contain both this Tag and the given Tag.

Q7. *Recent likes.* For the specified `Person` get the most recent likes of any of the person's posts, and the latency between the corresponding post and the like. Flag Likes from outside the direct connections. Return top 20 Likes, ordered descending by creation date of the like.

Q7. *Recent likes.* For the specified `Person` get the most recent likes of any of the person's posts, and the latency between the corresponding post and the like. Flag Likes from outside the direct connections. Return top 20 Likes, ordered descending by creation date of the like.

Q8. *Most recent replies.* This query retrieves the 20 most recent reply comments to all the posts and comments of `Person`, ordered descending by creation date.

Q9. *Latest Posts.* Find the most recent 20 posts and comments from all friends, or friends-of-friends of `Person`, but created before a `Date`. Return posts, their creators and creation dates, sort descending by creation date.

Q10. *Friend recommendation.* Find a friend of a friend who posts much about the interests of `Person` and little about topics that are not in the interests of the user. The search is restricted by the candidate's `horoscopeSign`. Returns 10 Persons for whom the difference between the total number of their posts about the interests of the specified user and the total number of their posts that are not in the interests of the user, is as large as possible. Sort the result descending by this difference.

Q11. *Job referral.* Find a friend of the specified `Person`, or a friend of her friend (excluding the specified person), who has long worked in a company in a specified `Country`. Sort ascending by start date, and then ascending by person identifier. Top 10 result should be shown.

Q12. *Expert Search.* Find friends of a `Person` who have replied the most to posts with a tag in a given `TagCategory`. Count the number of these reply Comments, and collect the Tags that were attached to the Posts they replied to. Return top 20 persons, sorted descending by number of replies.

Q13. *Single shortest path.* Given `PersonX` and `PersonY` , find the shortest path between them in the subgraph induced by the

Q14. *Weighted paths.* Given `PersonX` and `PersonY` , find all weighted paths of the shortest length between them in the subgraph induced by the Knows relationship. The weight of the path takes into consideration amount of Posts/Comments exchanged.

Choke-Point: shortest paths

Q14. *Weighted paths.* Given PersonX and PersonY , find all weighted paths of the shortest length between them in the subgraph induced by the Knows relationship. The weight of the path takes into consideration amount of Posts/Comments exchanged.

- compute weights over a **recursive forum traversal**
  - on the fly, or
  - materialized, but then maintain them under updates
- **compute shortest paths** using these weights in the friends graph

SNB Interactive Workload

Q1. *Extract description of friends with a given name* Given a person's `firstName`, return up to 20 people with the same first name, sorted by increasing distance (max 3) from a given `person`, and for people within the same distance sorted by last name. Results should include the list of workplaces and places of study.

Q2. *Find the newest 20 posts and comments from your friends.* Given a start `Person`, find (most recent) Posts and Comments from all of that Person's friends, that were created before (and including) a given `Date`. Return the top 20 Posts/Comments, and the Person that created each of them. Sort results descending by creation date, and then ascending by Post identifier.

Q3. *Friends within 2 steps that have recently traveled to countries X and Y.* Find friends and friends of friends of a given `Person` who have made a post or a comment in the foreign `CountryX` and `CountryY` within a specified period of `DurationInDays` after a `startDate`. Return top 20 Persons, sorted descending by total number of posts.

Q4. *New Topics.* Given a start `Person`, find the top 10 most popular Tags (by total number of posts with the tag) that are attached to Posts that were created by that Person's friends. Only include Tags that were attached to Posts created within a given `time interval`, and that were never attached to Posts created before this interval.

Q5. *New groups.* Given a start `Person`, find the top 20 Forums which that Person's friends and friends of friends became members of after a given `Date`. Sort results descending by the number of Posts in each Forum that were created by any of these Persons.

Q6. *Tag co-occurrence.* Given a start `Person` and some `Tag`, find the other Tags that occur together with this Tag on Posts that were created by start Person's friends and friends of friends. Return top 10 Tags, sorted descending by the count of Posts that were created by these Persons, which contain both this Tag and the given Tag.

Q7. *Recent likes.* For the specified `Person` get the most recent likes of any of the person's posts, and the latency between the corresponding post and the like. Flag Likes from outside the direct connections. Return top 20 Likes, ordered descending by creation date of the like.

Q7. *Recent likes.* For the specified `Person` get the most recent likes of any of the person's posts, and the latency between the corresponding post and the like. Flag Likes from outside the direct connections. Return top 20 Likes, ordered descending by creation date of the like.

Q8. *Most recent replies.* This query retrieves the 20 most recent reply comments to all the posts and comments of `Person`, ordered descending by creation date.

Q9. *Latest Posts.* Find the most recent 20 posts and comments from all friends, or friends-of-friends of `Person`, but created before a `Date`. Return posts, their creators and creation dates, sort descending by creation date.

Q10. *Friend recommendation.* Find a friend of a friend who posts much about the interests of `Person` and little about topics that are not in the interests of the user. The search is restricted by the candidate's `horoscopeSign`. Returns 10 Persons for whom the difference between the total number of their posts about the interests of the specified user and the total number of their posts that are not in the interests of the user, is as large as possible. Sort the result descending by this difference.

Q11. *Job referral.* Find a friend of the specified `Person`, or a friend of her friend (excluding the specified person), who has long worked in a company in a specified `Country`. Sort ascending by start date, and then ascending by person identifier. Top 10 result should be shown.

Q12. *Expert Search.* Find friends of a `Person` who have replied the most to posts with a tag in a given `TagCategory`. Count the number of these reply Comments, and collect the Tags that were attached to the Posts they replied to. Return top 20 persons, sorted descending by number of replies.

Q13. *Single shortest path.* Given `PersonX` and `PersonY` , find the shortest path between them in the subgraph induced by the Knows relationships. Return the length of this path.

Q14. *Weighted paths.* Given `PersonX` and `PersonY` , find all weighted paths of the shortest length between them in the subgraph induced by the Knows relationship. The weight of the path takes into consideration amount of Posts/Comments exchanged.

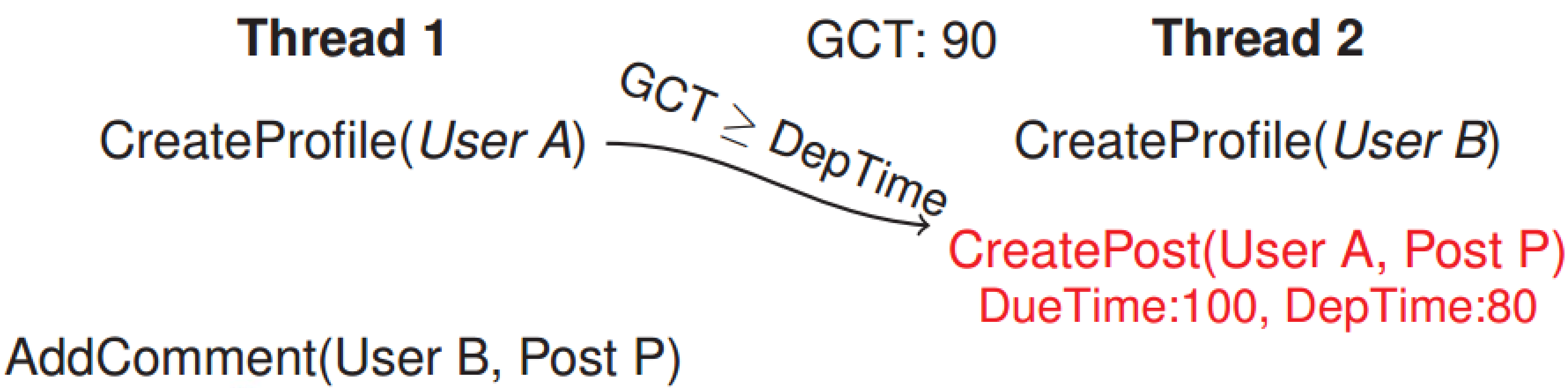
Choke-Point: **outdegree correlation**

Q3. *Friends within 2 steps that recently traveled to countries X and Y.* Find top 20 friends and friends of friends of a given `Person` who have made a post or a comment in the foreign `CountryX` and `CountryY` within a specified period of `DurationInDays` after a `startDate`. Sorted results descending by total number of posts.

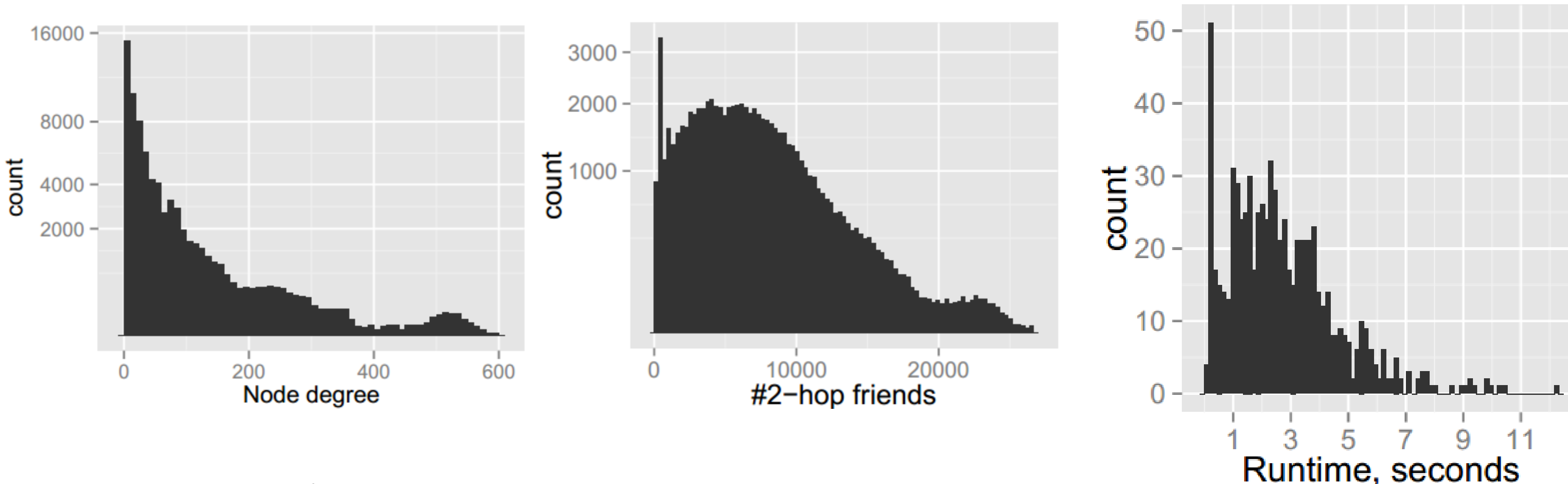
- Travel is correlated with location
  - People travel more often to nearby countries
- Outdegree after `(countryX,countryY)` selection varies a lot
  - **(Australia,NZ)**: high outdegree ("join hit ratio")
  - **(Australia,Belgium)**: low outdegree ← *different query plan, or navigation strategy likely wins*

SNB Query Driver

- Window-based parallel query generation
  - Problem: friends graph has complex dependencies (non-partitionable). Could cause large checking overhead.
  - Solution: Window based approach for checking dependencies (Global Completion Time)



Problem: Parameter Sensitivity



SNB Interactive Q5:

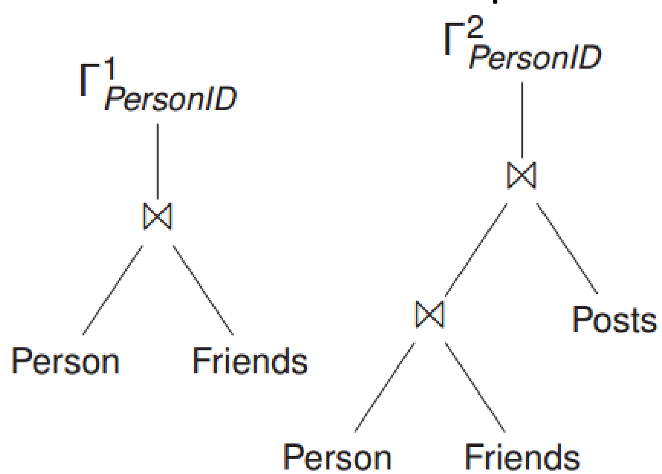
explores the 2-hop friend neighbourhood, of one start person

Observation: depending on the start person, there is a large runtime variance

Parameter Curation

TPCTC2014 “Parameter Curation for Benchmark Queries” Andrey Gubichev (TUM) & Peter Boncz (CWI)

- Example: Q3
  - Problem: value correlations cause very large variance
  - Solution: data mine for **stable** parameter **equivalence classes**



PersonID	$\bowtie_1$	$\bowtie_2$
...	...	...
1542	60	99
1673	60	102
7511	60	103
958	60	120
1367	61	101
...	...	...

- form sliding windows of rows
- pick sub-window with the smallest variance in the next column

Query Mix & Metric

Query Mix

- Insert queries (~10% of time):
  ➔ challenge: **execute parallel but respect data dependencies in the graph**
- Read-only Complex Queries (~50% of time)
  ➔ challenge: **generate query parameters with stable query behavior**
  Parameter Curation to find “equivalence classes” in parameters
- Simple Read-only Queries (~40% of time)
  – Retrieve Post / Retrieve Person Profile

Metric

- **Acceleration Factor** (AF) that can be sustained (+ AF/$ weighted by cost)
  – with 99th percentile of query latency within maximal query time

SNB Query Driver

- Dependency-aware parallel query generation
  - **Problem**: friends graph is non-partitionable, but imposes ordering constraints.

    Could cause large checking overhead, impeding driver parallelism.
  - **Solution**: Window-based checking approach for keeping driver threads roughly synchronized on a global timestamp.

    Is helped by DATAGEN properties that ensure there is a minimal latency between certain dependencies (e.g. entering the network and making friends, or posting on a new friend's forum). This minimal latency provides synchronization headroom.

Summary

- LDBC
  - Graph and RDF benchmark council
  - Choke-point driven benchmark design (user+system expert involvement)
- Social Network Benchmark
  - Advanced social network generator
    - skewed distributions, power laws, value/structure correlations, flash mobs
  - 3 workloads: Interactive (←focus of this paper), BI, Analytics
    - Interactive Query Mix & Metrics
    - Parallel Query Driver that respects dependencies efficiently
    - Parameter Curation for stable results

7th LDBC Technical User Community meeting
November 9+10 2015, IBM TJ Watson (NJ)

Assignment 1: Querying a Social Graph

www.cwi.nl/~boncz/snb-challenge

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.*

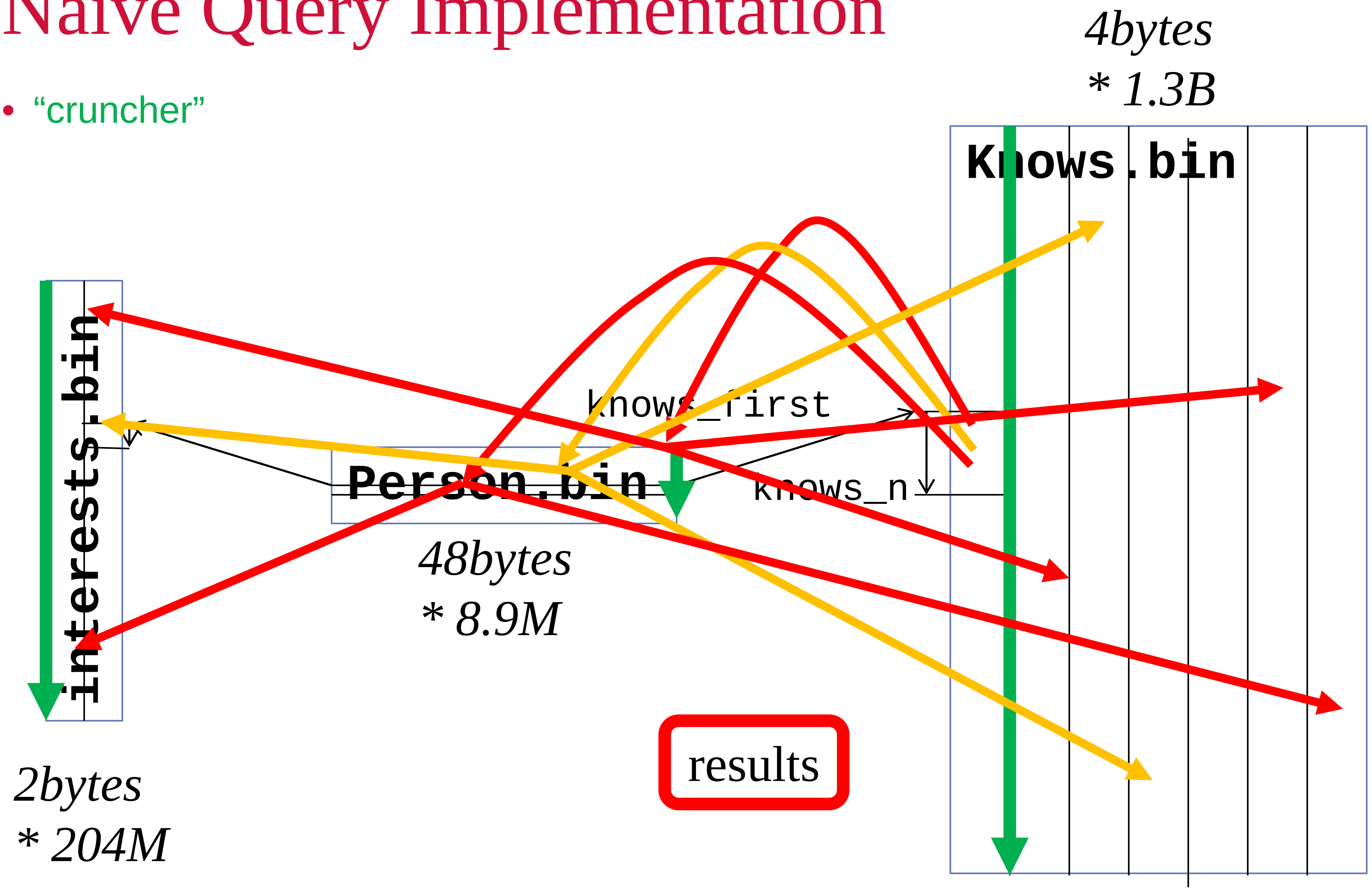
    For each F:

    - *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”



Improving Bad Access Patterns

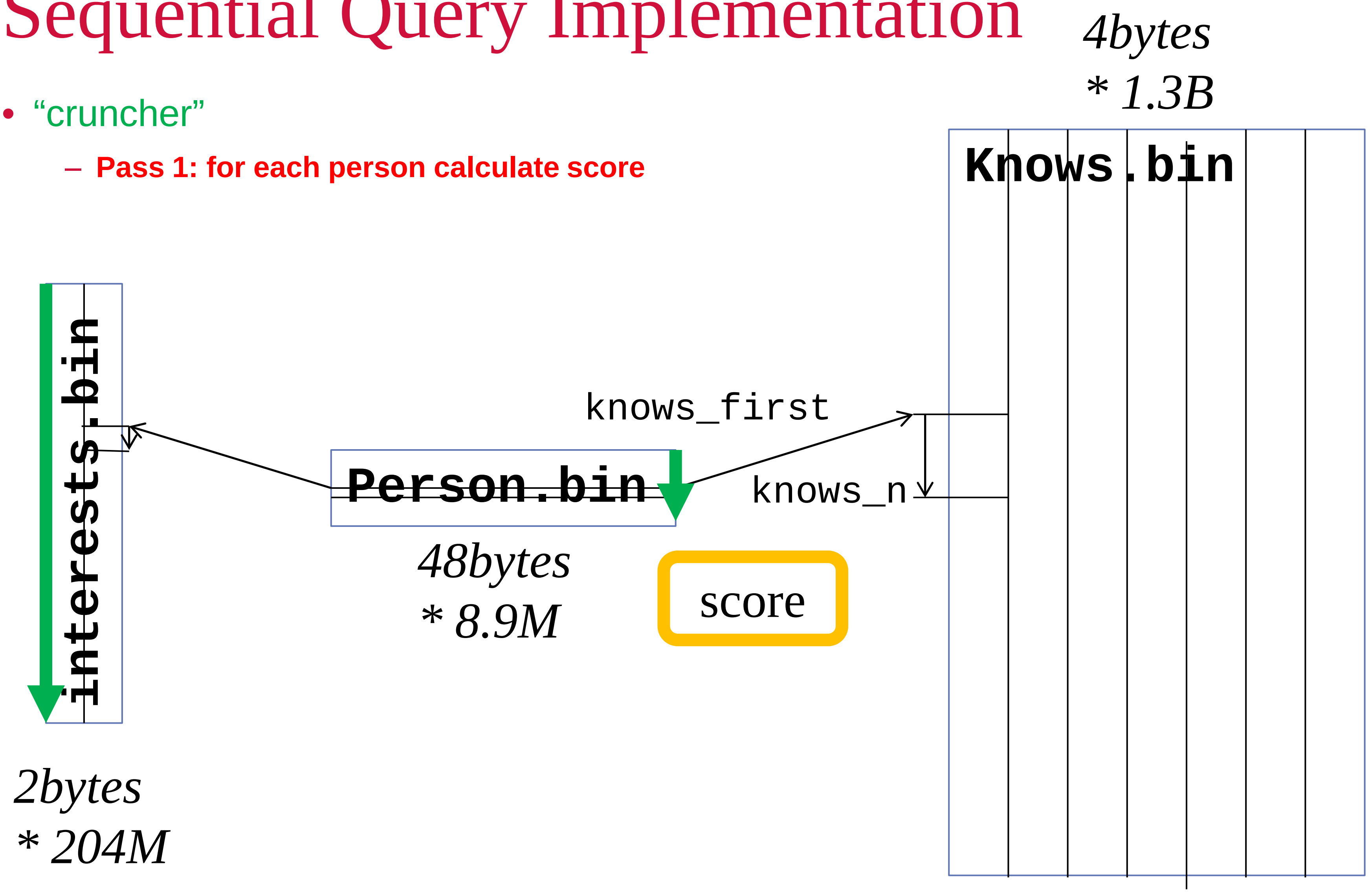
- Minimize Random Memory Access
  - Apply filters first. Less accesses is better.
- Denormalize the Schema
  - Remove joins/lookups, add looked up stuff to the table (but.. makes it bigger)
- Trade Random Access For Sequential Access
  - perform a 100K random key lookups in a large table

    ➔ 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

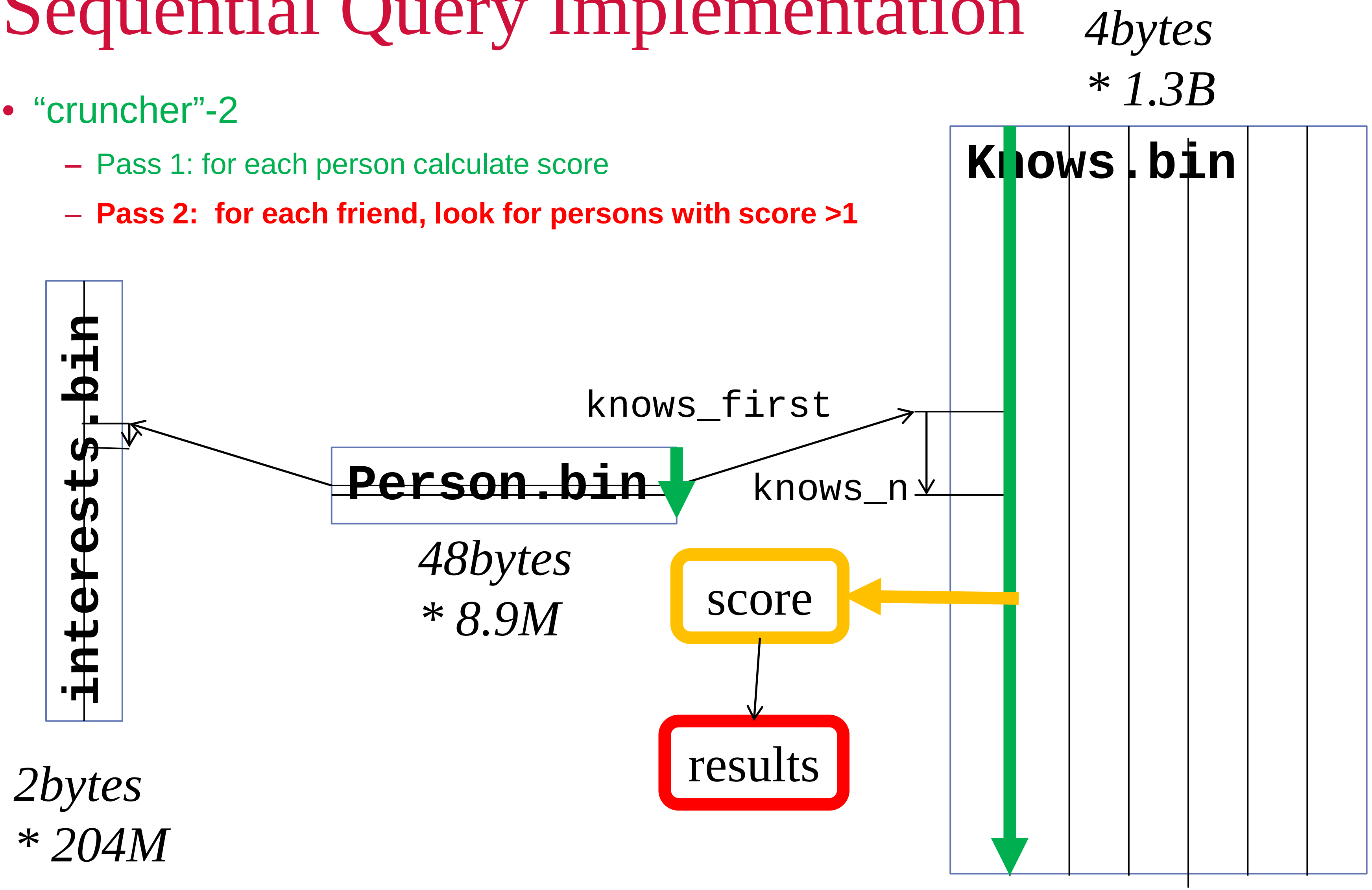
Sequential Query Implementation

- “cruncher”
  - Pass 1: for each person calculate score



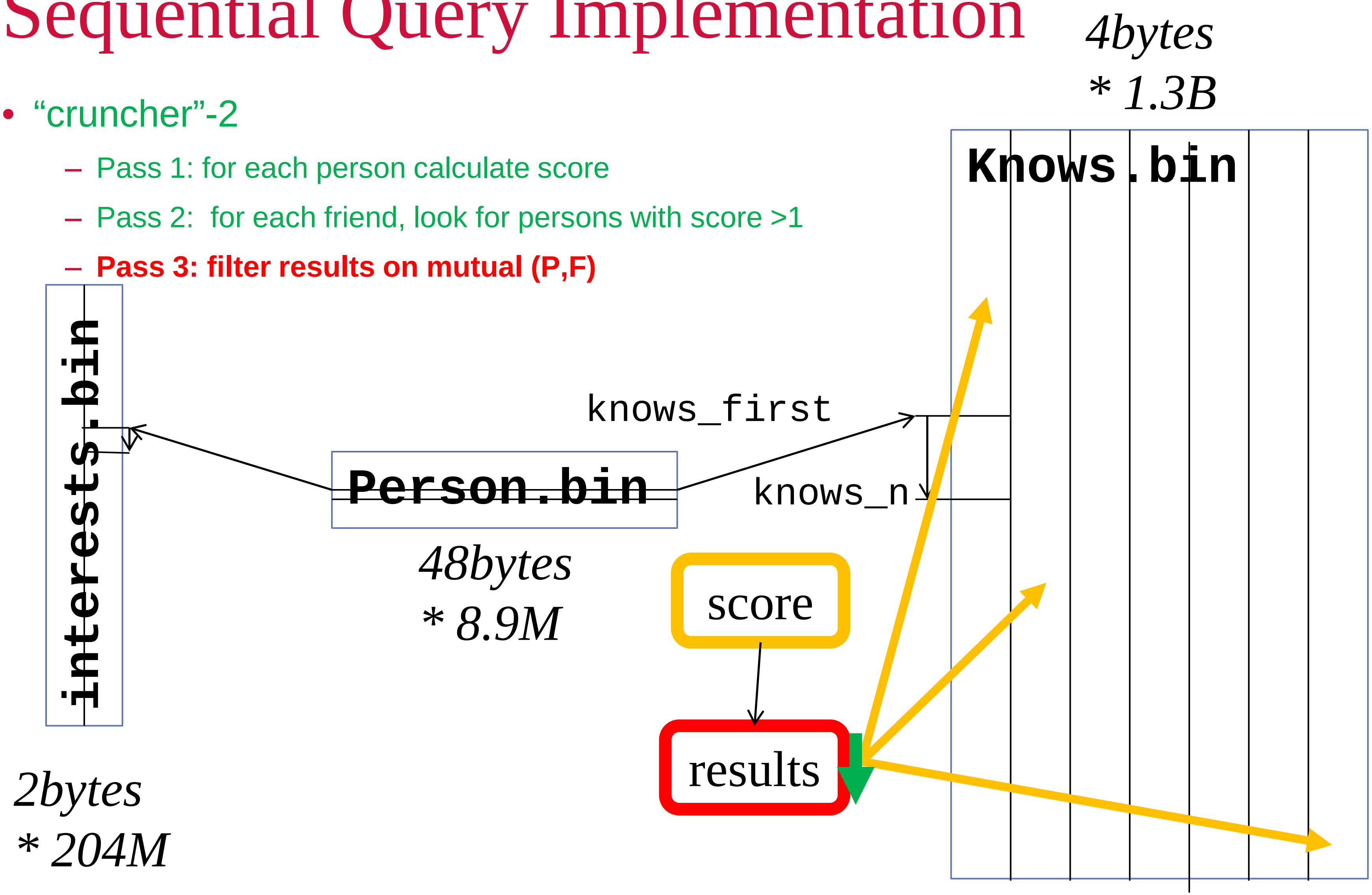
Sequential Query Implementation

- "cruncher"-2
  - Pass 1: for each person calculate score
  - **Pass 2: for each friend, look for persons with score >1**



Sequential Query Implementation

- "cruncher"-2
  - Pass 1: for each person calculate score
  - Pass 2: for each friend, look for persons with score >1
  - **Pass 3: filter results on mutual (P,F)**



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

The Naïve Implementation

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.

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

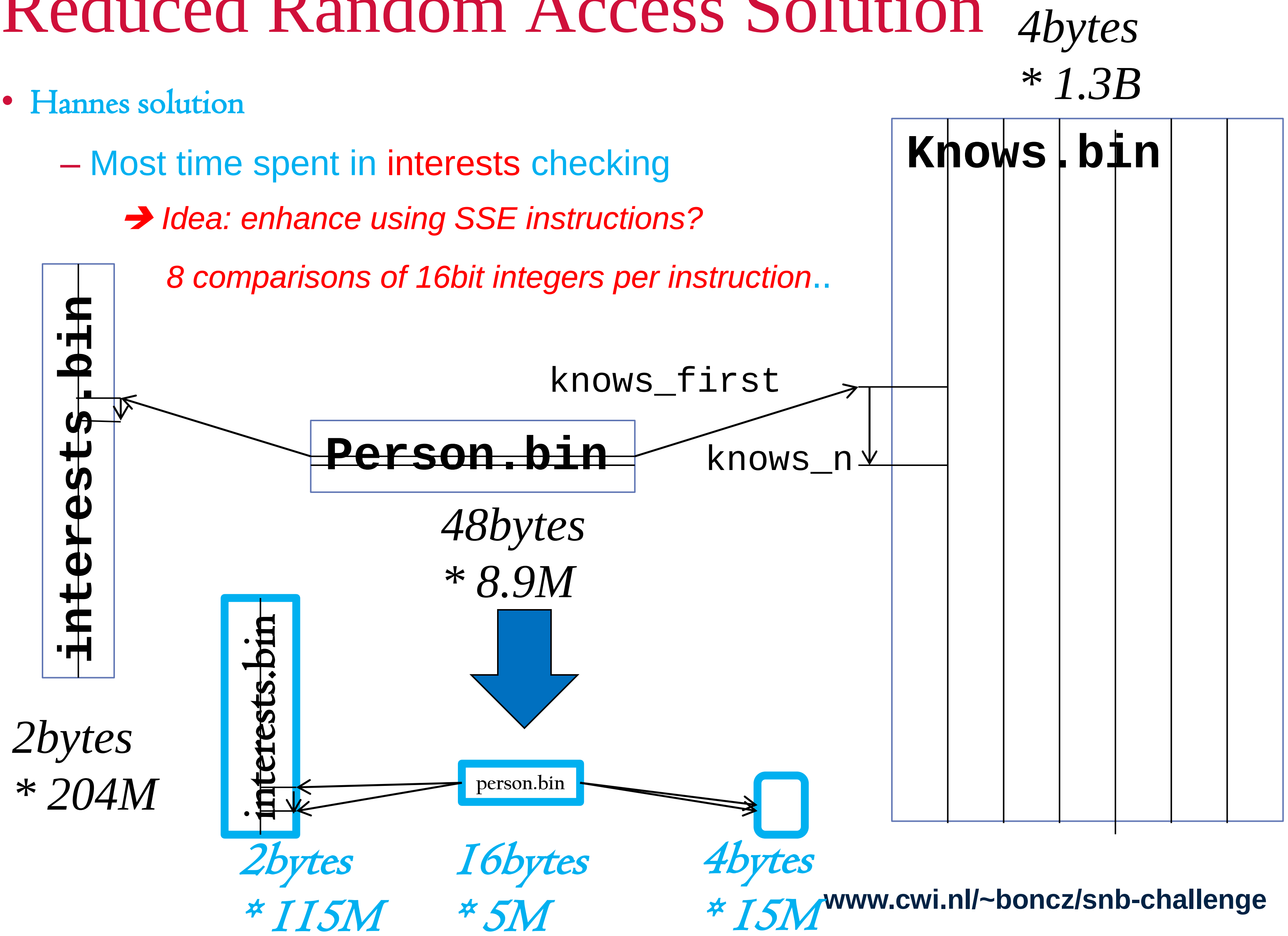
- knows.bin
  - is big (larger than RAM)
  - is accessed randomly
    - random access unavoidable (denormalization too costly)

Ideas:

- Only keep **mutual-knows**
- Only keep **local-knows**
  - Idea: remove knows where persons live in different cities (**50x less:** 150 ➔ 3 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*3M=15M knows = 60MB random access

Reduced Random Access Solution

- Hannes solution
  - Most time spent in interests checking
    - ➔ *Idea: enhance using SSE instructions?*
      8 comparisons of 16bit integers per instruction..



The Naïve Implementation

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***

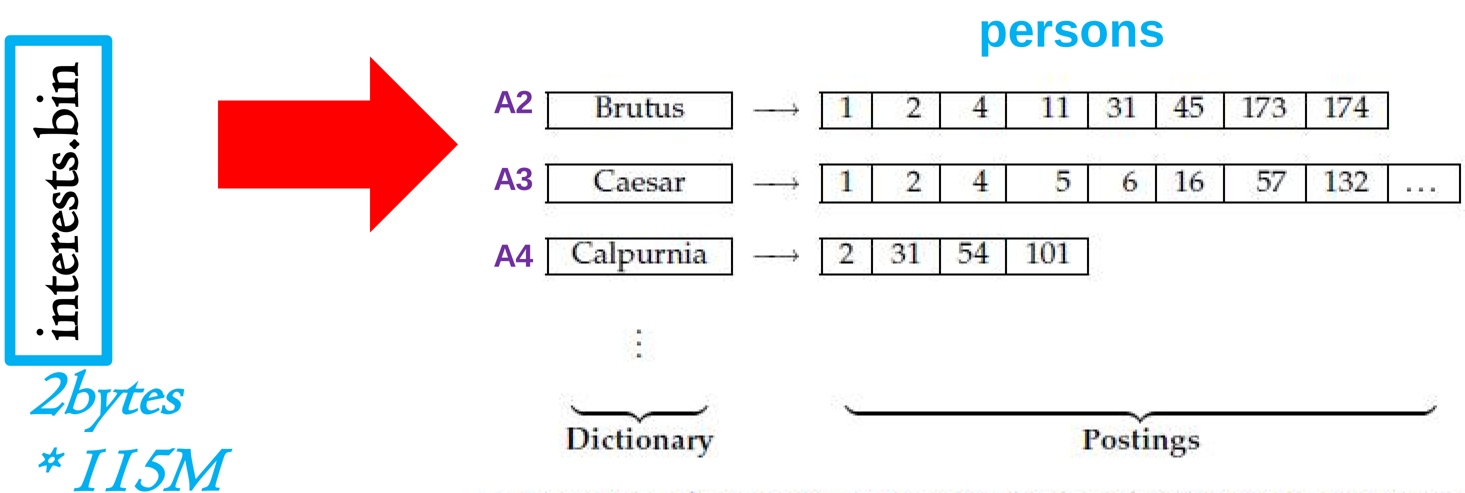
    if all this succeeds (score,P,F) is added to a result table.

Idea: using Inverted Files

The search engine data structure

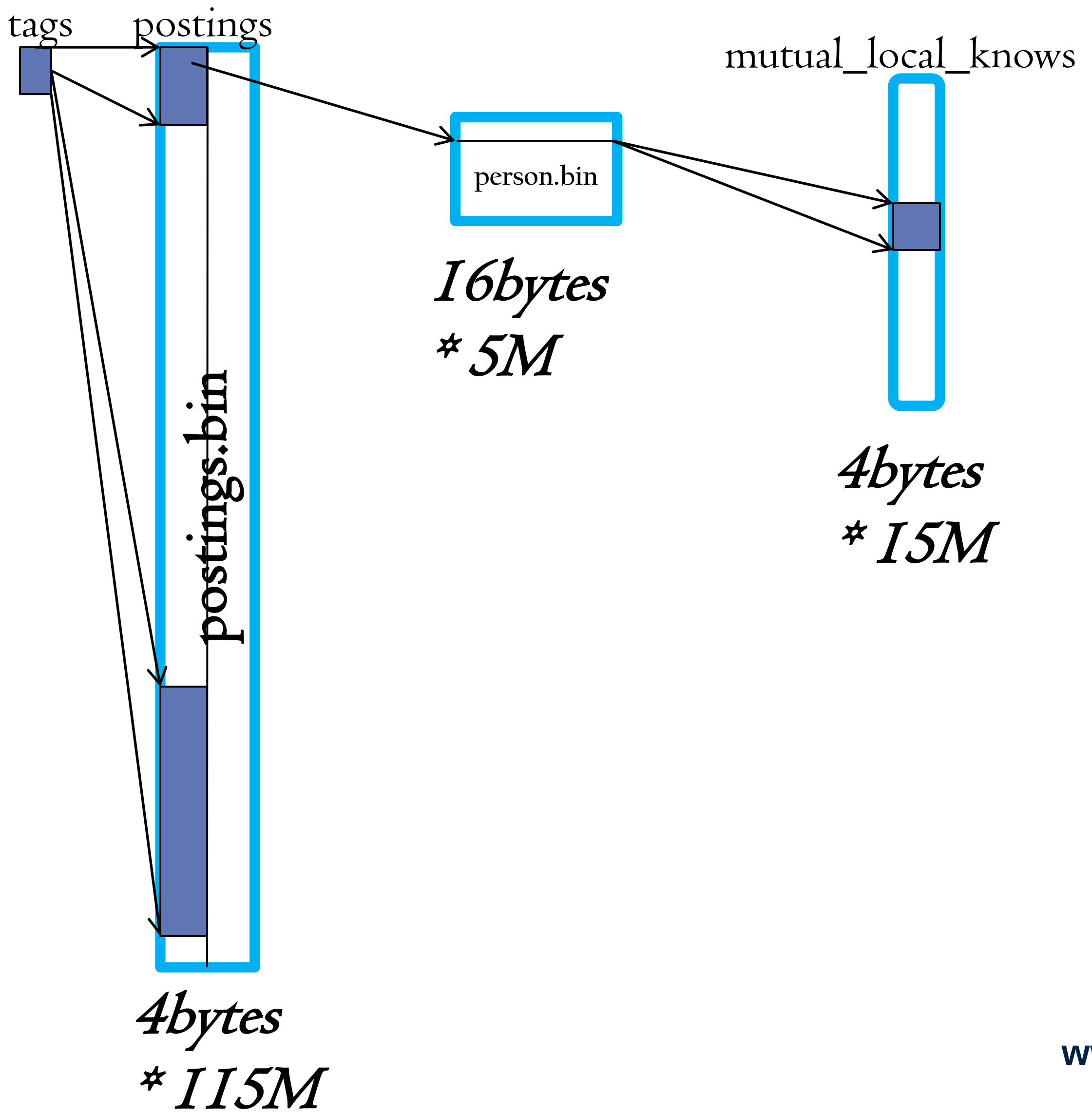
- For each term (keyword), a list of document IDs

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



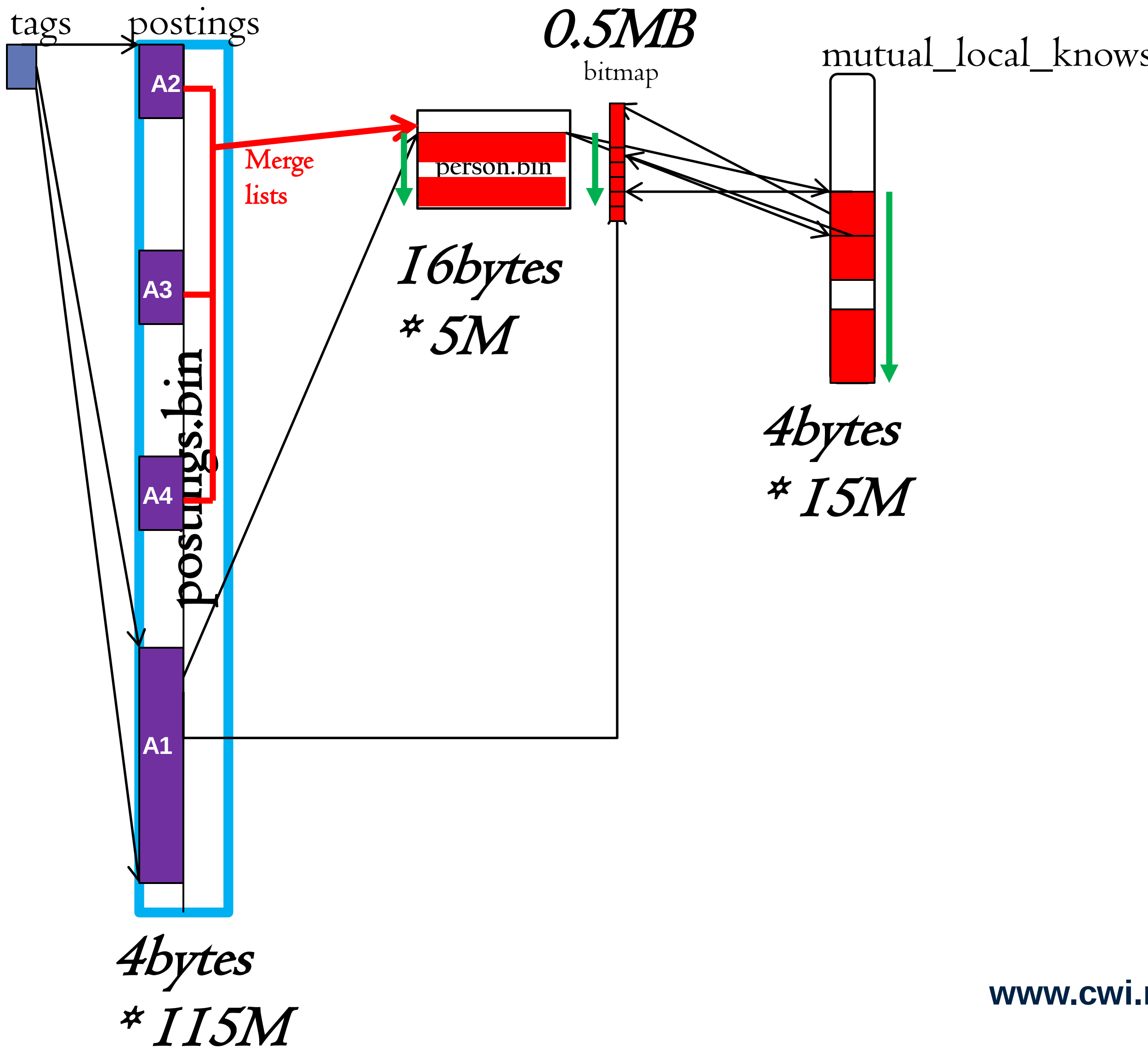
► **Figure 1.3** The two parts of an inverted index. The dictionary is commonly kept in memory, with pointers to each postings list, which is stored on disk.

Inverted File on Tags



CWI
Centrum Wiskunde & Informatica

Inverted File on Tags



www.cwi.nl/~boncz/snb-challenge

Inverted File Cruncher Implementation

*Create a **A1-bitmap** (1bit for each person) based on the inverted list of **A1***

- ***0.5MB bitmap (even fits CPU cache)***

*Merge inverted lists **A2,A3,A4** computing a **score** and for each person P*

- *for those with **score**>0, in **birthday range** and who are not in **A1-bitmap**:*
  - *visit all persons F known to P.*

    For each F:
    - *check whether F is set in **A1-bitmap***

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

Idea: use Table Partitioning

Goals:

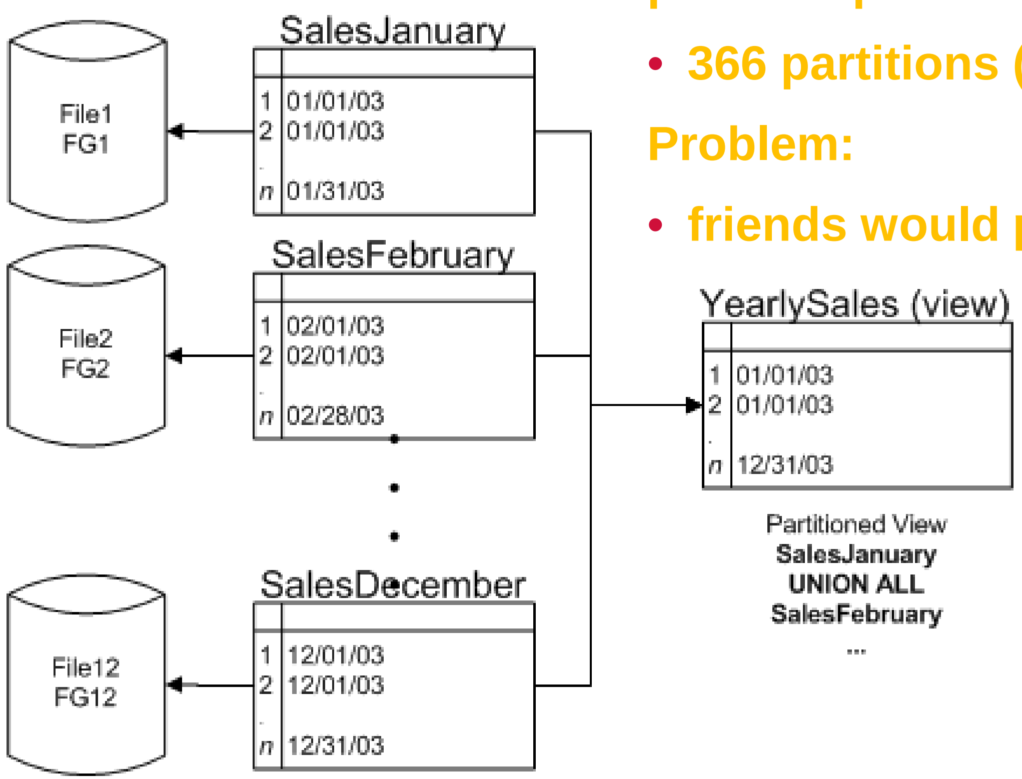
- make birthdate comparisons faster
- remove birthdate column (no longer needed, implicit)
- **Increase locality in person.bin and knows.bin**!

partition person.bin by birthdate

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

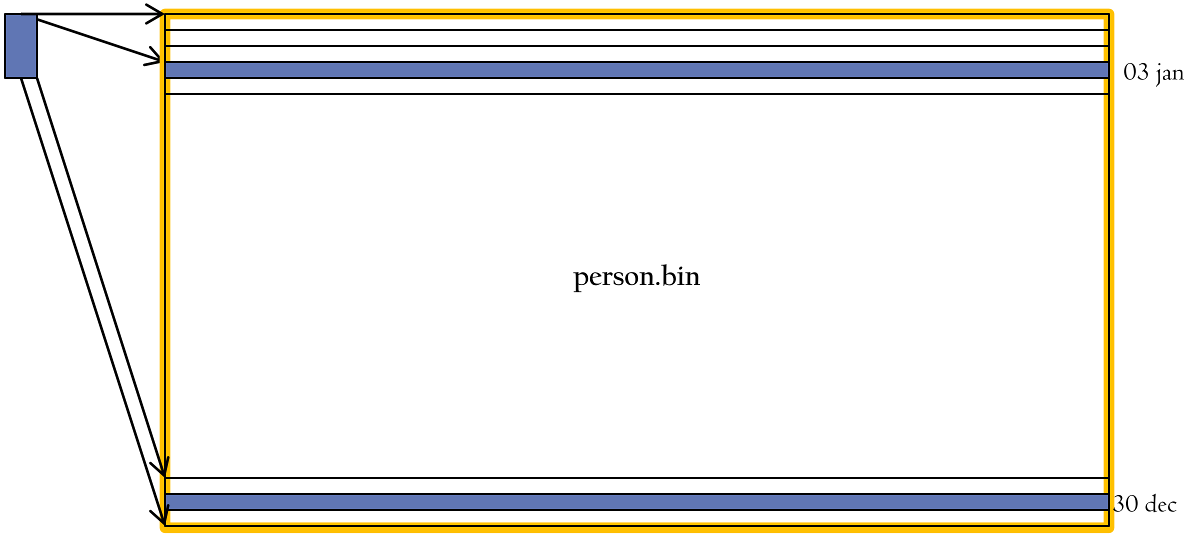
Problem:

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



www.cwi.nl/~boncz/snb-challenge

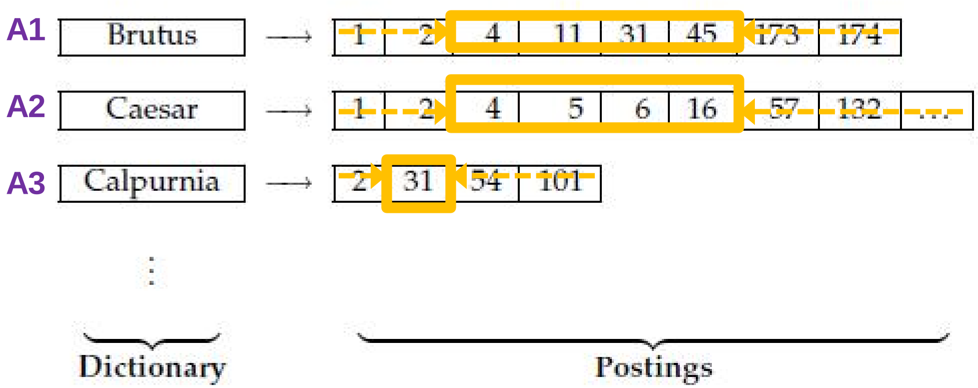
Inverted File on Tags



Inverted Files Revisted

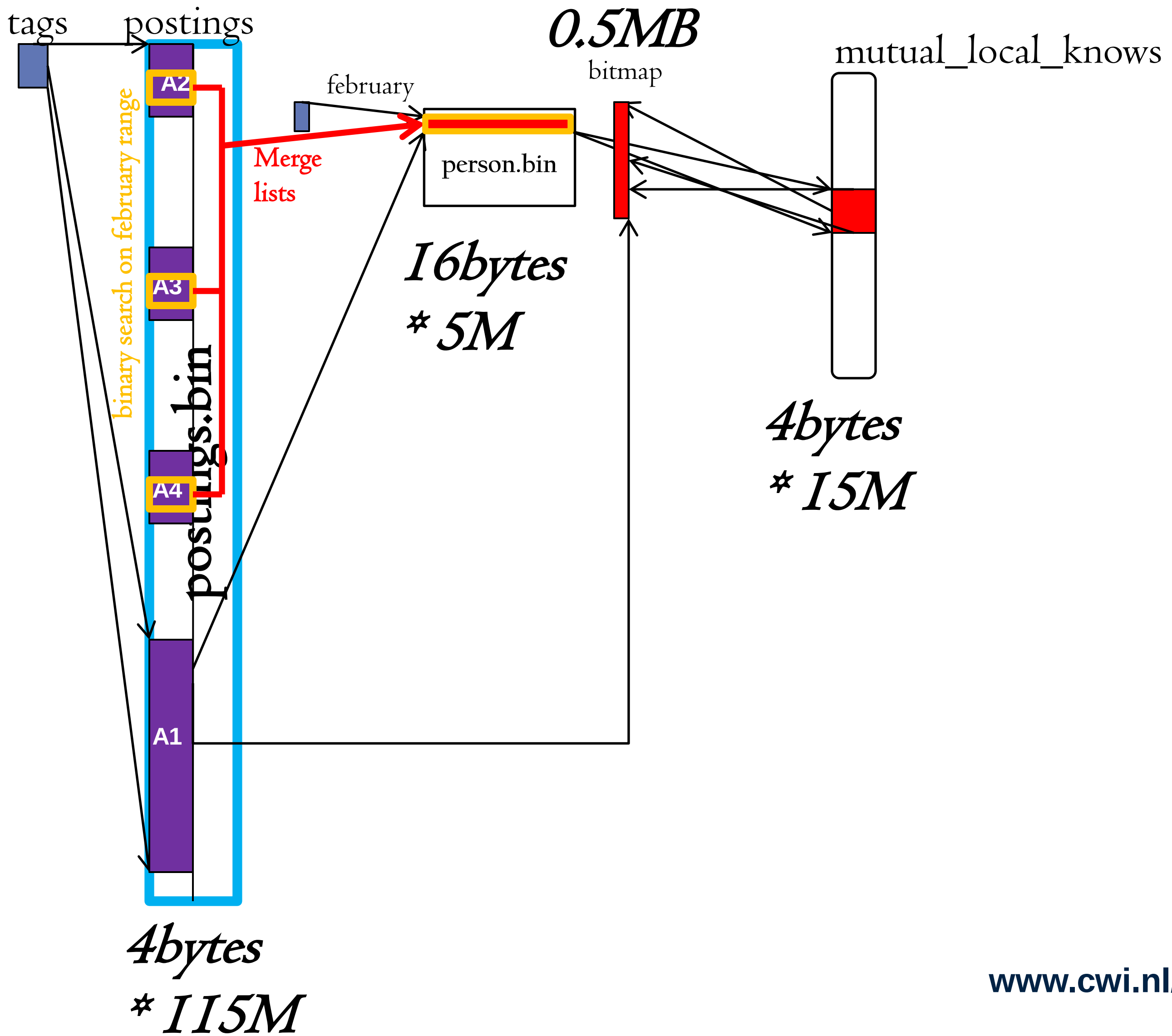
The birthdate clustering gives us for a **birthdate range** a **person range**

- 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 File on Tags



Clustered + Inverted File Cruncher

"Peter approach"

*Create a **A1-bitmap** (1bit for each person) based on the inverted list of **A1***

0.5MB bitmap (even fits CPU cache)

***Binary search** (restrict on **birthdate range**) and merge inverted lists **A2,A3,A4** computing a **score** and for each person P*

- *for those with **score**>0, who are not in **A1-bitmap**:*
  - *visit all persons F known to P.*

    For each F:
    - *check whether F is set in **A1-bitmap***

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