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
The LDBC
Social Network Benchmark
Interactive Workload

http://www.ldbcouncil.org
Social Network Benchmark: schema
Database Benchmark Design

Desirable properties:
• Relevant. ➔ “Choke Points”
• Representative.
• Understandable.
• Economical.
• Accepted.
• Scalable.
• Portable.
• Fair.
• Evolvable.
• Public.


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!

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

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

Q10

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

Q10

TPCTC 2013: “TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark”
CP1.4 Dependent GroupBy Keys

• Functional dependencies:

\[ c_{\text{custkey}} \rightarrow c_{\text{name}}, \ c_{\text{acctbal}}, \ c_{\text{phone}}, c_{\text{address}}, \ c_{\text{comment}}, \ c_{\text{nationkey}} \rightarrow n_{\text{name}} \]

• Group-by hash table should exclude the colored attrs \( \rightarrow \) 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!

TPCTC 2013: “TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark”
CP2.2 Sparse Joins

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

Q21

- 200M tuples probed
- 8M tuples result
- 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

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_linenstatus, 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
WHERE l_shipdate <= date '1998-12-01' - interval '[DELTA]' day (3)
GROUP BY l_returnflag, l_linenstatus
ORDER BY l_returnflag, l_linenstatus
```

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

```
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.

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’

SELECT personID from person
WHERE firstName = ‘Cesare’ AND addressCountry = ‘Italy’

- Query optimizers may underestimate or overestimate the result size of conjunctive predicates
Data correlations **between attributes**

```sql
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'
```
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'

- 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

- 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 Dictionary**

  \{"Andrea", "Anna", "Cesare", "Camilla", "Duc", "Joachim", "Leon", "Or"

- **Probability density function**

- **Ranking Function** \( R(\text{gender}, \text{country}, \text{birthyear}) \)
  - gender, country, birthyear \( \rightarrow \) correlation parameters

#### How to implement \( R() \)?

We need a table storing:

<table>
<thead>
<tr>
<th>Gender</th>
<th>X</th>
<th>Country</th>
<th>X</th>
<th>BirthYear</th>
<th>X</th>
<th>D</th>
</tr>
</thead>
</table>

#### 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](http://dbpedia.org))

<table>
<thead>
<tr>
<th>(person.location, person.gender)</th>
<th>person.firstName</th>
<th>(typical names)</th>
</tr>
</thead>
<tbody>
<tr>
<td>person.location</td>
<td>person.lastName</td>
<td>(typical names)</td>
</tr>
<tr>
<td>person.university</td>
<td>(nearby universities)</td>
<td></td>
</tr>
<tr>
<td>person.company</td>
<td>(in country)</td>
<td></td>
</tr>
<tr>
<td>person.languages</td>
<td>(spoken in country)</td>
<td></td>
</tr>
<tr>
<td>person.language</td>
<td>person.forum.message.language</td>
<td>(speaks)</td>
</tr>
<tr>
<td>person.interests</td>
<td>person.forum.post.topic</td>
<td>(in)</td>
</tr>
<tr>
<td>post.topic</td>
<td>post.text</td>
<td>(DBpedia article lines)</td>
</tr>
<tr>
<td></td>
<td>post.comment.text</td>
<td>(DBpedia article lines)</td>
</tr>
<tr>
<td>person.employer</td>
<td>person.email</td>
<td>(@company, @university)</td>
</tr>
<tr>
<td>(friendship.userId1, friendship.userId2)</td>
<td>friendship.terminator</td>
<td>(= one of the two)</td>
</tr>
<tr>
<td>message.photoLocation</td>
<td>message.latitude</td>
<td>(matches location)</td>
</tr>
<tr>
<td></td>
<td>message.longitude</td>
<td>(matches location)</td>
</tr>
<tr>
<td>friendship.requestDate</td>
<td>friendship.approveDate</td>
<td>(&gt; )</td>
</tr>
<tr>
<td></td>
<td>friendship.deniedDate</td>
<td>(&gt; )</td>
</tr>
<tr>
<td>person.birthDate</td>
<td>person.createdDate</td>
<td>(&gt; )</td>
</tr>
<tr>
<td>person.createdDate</td>
<td>person.forum.message.createdDate</td>
<td>(&gt; )</td>
</tr>
<tr>
<td>forum.createdDate</td>
<td>person.forum.createdDate</td>
<td>(&gt; )</td>
</tr>
<tr>
<td>message.createdDate</td>
<td>message.travelTime</td>
<td>(&gt; )</td>
</tr>
<tr>
<td></td>
<td>forum.post.createdDate</td>
<td>(&gt; )</td>
</tr>
<tr>
<td></td>
<td>forum.groupmembership.joinedDate</td>
<td>(&gt; )</td>
</tr>
<tr>
<td></td>
<td>message.comment.createdDate</td>
<td>(&gt; )</td>
</tr>
</tbody>
</table>
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

- Student "Anna"
  - University of Leipzig
  - 1990

- Student Laura
  - University of Leipzig
  - 1990

- <Britney Spears>
  - University of Leipzig
  - 1990

- <Britney Spears>
  - University of Amsterdam
  - Netherlands

- "University of Leipzig"
  - "1990"

- "University of Leipzig"
  - "1990"

- "University of Amsterdam"
  - "Netherlands"
**Simple approach**

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

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

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:

<table>
<thead>
<tr>
<th>SFs</th>
<th>Nodes</th>
<th>Edges</th>
<th>Persons</th>
<th>Friends</th>
<th>Messages</th>
<th>Forums</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>99.4</td>
<td>655.4</td>
<td>0.18</td>
<td>14.2</td>
<td>97.4</td>
<td>1.8</td>
</tr>
<tr>
<td>100</td>
<td>317.7</td>
<td>2154.9</td>
<td>0.50</td>
<td>46.6</td>
<td>312.1</td>
<td>5.0</td>
</tr>
<tr>
<td>300</td>
<td>907.6</td>
<td>6292.5</td>
<td>1.25</td>
<td>136.2</td>
<td>893.7</td>
<td>12.6</td>
</tr>
<tr>
<td>1000</td>
<td>2930.7</td>
<td>20704.6</td>
<td>3.60</td>
<td>447.2</td>
<td>2890.9</td>
<td>36.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SFs</th>
<th>Number of entities (x 1000000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>post</td>
<td>post_content</td>
</tr>
<tr>
<td>likes</td>
<td>likes_likes</td>
</tr>
<tr>
<td>forum_person</td>
<td>forum_person_forum_person_creationdate</td>
</tr>
</tbody>
</table>
DATAGEN: Graph Characteristics

Livejournal  
LFR3 (synthetic)  
LDBC DATAGEN

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
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.

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 relationship. The weight of the path takes into consideration amount of Posts/Comments exchanged.

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
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 descending by Post id.

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.

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 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 $(\text{country}_X, \text{country}_Y)$ selection varies a lot
  - $(\text{Australia}, \text{NZ})$: high outdegree ("join hit ratio")
  - $(\text{Australia}, \text{Belgium})$: low outdegree $\Rightarrow$ 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

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

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

- 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
The Naïve Implementation

The “cruncher” program

Go through the persons $P$ sequentially

- counting how many of the artists $A_2, A_3, A_4$ 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 $A_1$
  - check whether $F$ also knows $P$

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

• “cruncher”

2bytes
* 204M

48bytes
* 8.9M

4bytes
* 1.3B

results

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

```
2bytes * 204M
48bytes * 8.9M
4bytes * 1.3B
```
Sequential Query Implementation

• “cruncher”-2
  – Pass 1: for each person calculate score
  – Pass 2: for each friend, look for persons with score >1

2bytes
* 204M

interests.bin

Person.bin

48bytes
* 8.9M

score

results

4bytes
* 1.3B

Knows.bin
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 \textit{in birthday range}

- count how many of the artists $A_2,A_3,A_4$ are liked as the score
  for those with score $>0$ and who do not like $A_1$:
  - visit all persons $F$ known to $P$.

  For each $F$:
  - checks on \textit{equal location}
  - check whether $F$ already likes $A_1$
  - check whether $F$ also knows $P$

  if all this succeeds $(\text{score},P,F)$ 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**)
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..

```
Person.bin
```

```
interests.bin
2bytes * 204M
```

```
Knocks.bin
4bytes * 1.3B
```

```
Knows.bin
4bytes * 8.9M
```

```
interests.bin
2bytes * 115M
```

```
person.bin
16bytes * 5M
```

```
4bytes * 15M
```
The Naïve Implementation

The “cruncher” program

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

• count how many of the artists $A_2,A_3,A_4$ are liked as the score
  for those with score $>0$ and who do not like $A_1$:
  – visit all persons $F$ known to $P$.

For each $F$:

• checks on equal location
• check whether $F$ already likes $A_1$
• 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 persons

![Diagram of Inverted Index](https://www.cwi.nl/~boncz/snb-challenge)

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

- tags
- postings

- postings.bin
- 4 bytes
- * 115M

- person.bin
- 16 bytes
- * 5M

- mutual_local_knows
- 4 bytes
- * 15M

www.cwi.nl/~boncz/snb-challenge
Inverted File on Tags

- tags
- postings
- person.bin
- postings.bin
- bitmap
- mutual_local_knows

- 16 bytes * 5M
- 4 bytes * 115M
- 0.5 MB
- Merge lists

www.cwi.nl/~boncz/snb-challenge
Inverted File Cruncher Implementation

Create a **A1-bitmap** (1 bit 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
Inverted File on Tags

Person.bin

03 Jan

30 Dec
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)
Inverted File on Tags

- **tags**
- **postings**
  - **bits**
    - **february**
      - **bitmap**
        - **person.bin**
          - **16 bytes**
            - **5M**
          - **4 bytes**
            - **115M**
          - **0.5MB**
          - **binary search on february range**
          - **Merge lists**
          - **mutual_local_knows**
          - **4 bytes**
            - **15M**
Clustered + Inverted File Cruncher

“Peter approach”

Create a **A1-bitmap** (1 bit 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.