Business Intelligence Workload Chokepoint Identification

LDBC SNB Interactive Chokepoints (http://ldbcouncil.org/sites/default/files/LDBC_D3.3.34.pdf)

- **Aggregation Performance**
  - **SNBI-1.1/TCPH-1.2: QOPT Interesting Orders**
    - This choke-point tests the ability of the query optimizer to exploit the interesting orders induced by some operators. Apart from clustered indexes providing key order, other operators also preserve or even induce tuple orderings. Sort-based operators create new orderings, typically the probe-side of a hash join conserves its order, etc.
  - **SNBI-1.2/TCPH-1.1: QEXE High Cardinality group-by performance**
    - This choke-point tests the ability of the execution engine to parallelize group-by's with a large number of groups. Some queries require performing large group-by's.
      In such a case, if an aggregation produces a significant number of groups, intra query parallelization can be exploited as each thread may make its own partial aggregation.
      Then, to produce the result, these have to be re-aggregated. In order to avoid this, the tuples entering the aggregation operator may be partitioned by a hash of the grouping key and be sent to the appropriate partition.
      Each partition would have its own thread so that only that thread would write the aggregation, hence avoiding costly critical sections as well. A high cardinality distinct modifier in a query is a special case of this choke point.
      It is amenable to the same solution with intra query parallelization and partitioning as the group-by.
      We further note that scale-out systems have an extra incentive for partitioning since this will distribute the CPU and memory pressure over multiple machines, yielding better platform utilization and scalability.
  - **SNBI-1.3: QEXE Complex aggregate performance**
    - This choke-point test the performance of the execution engine to perform complex aggregates. Many databases offer user defined aggregates and more complex aggregation operations than the basic count, sum, max and min, for example string concatenation aggregation operator. These types of aggregates are expected to benefit from the same optimizations as the basic built-in ones, for example partitioning.
  - **SNBI-1.4: QOPT Top-k push down**
    - Top-k push down. This choke-point tests the ability of the query optimizer to perform optimizations based on top-k selections. Many times queries demand for returning the top-k elements.
      Once k results are obtained, extra restrictions in a selection can be added based on the properties of the kth element currently in the top-k, being more restrictive as the query advances, instead of sorting all elements and picking the highest k.
  - **SNBI-1.5/TCPH-1.4: QEXE Dependent group-by keys**
    - This choke-point tests the ability of the query optimizer to exclude those functionally dependent group-bys. Sometimes queries require for group-by's on a set of columns and a key, where the value of the key determines the columns.
      In this situation, the aggregation operator should be able to exclude certain group-by attributes from the key matching.
  - **SNBI-1.6/TCPH-1.3: QEXE Low Cardinality group-by performance**
    - This choke-point tests the ability to efficiently perform group by evaluation when only a very limited set of groups is available. This can require special strategies for parallelization, e.g. pre-aggregation when possible. This case also allows using special strategies for grouping like using array lookup if the domain of keys is small.
- **Join Performance**
  - **SNBI-2.1/TCPH-2.3: QOPT Rich join order optimization**
    - This choke-point tests the ability of the query optimizer to find optimal join orders. A graph can be traversed in different ways. In the relational model, this is equivalent as different join orders.
      The execution time of these orders may differ by orders of magnitude. Therefore, finding an efficient join (traversal) order is important, which in general, requires enumeration of all the possibilities.
      The enumeration is complicated by operators that are not freely re-orderable like semi-, anti-, and outer- joins. Because of this difficulty most join enumeration algorithms do not enumerate all possible plans, and therefore can miss the optimal join order. Therefore, these chokepoint tests the ability of the query optimizer to find optimal join (traversal) orders.
  - **SNBI-2.2/TCPH-2.4: QOPT Late projection**
    - This choke-point tests the ability of the query optimizer to delay the projection of unneeded attributes until late in the execution. Queries where certain columns are only needed late in the query.
      In such a situation, it is better to omit them from initial table scans, as fetching them later by row-id with a separate scan operator, which is joined to the intermediate query result, can save temporal space, and therefore I/O.
      Late projection does have a trade-off involving locality, since late in the plan the tuples may be in a different order, and scattered I/O in terms of tuples/second is much more expensive than sequential I/O.
      Late projection specifically makes sense in queries where the late use of these columns happens at a moment where the amount of tuples involved has been considerably reduced;
      for example after an aggregation with only few unique group-by keys, or a top-k operator.
  - **SNBI-2.3/TCPH-2.1: QOPT Join type selection**
    - This choke-point tests the ability of the query optimizer to select the proper join operator type, which implies accurate estimates of cardinalities.
      Depending on the cardinalities of both sides of a join, a hash or an index index based join operator is more appropriate. This is especially important with column stores, where one usually has an index on everything. Deciding to use a hash join requires a good estimation of cardinalities on both the probe and build sides.
      In TPC-H, the use of hash join is almost a foregone conclusion in many cases, since an implementation will usually not

even define an index on foreign key columns.
There is a break even point between index and hash based plans, depending on the cardinality on the probe and build sides

- **SNBI-2.4/TCPH-2.2: QOPT Sparse foreign key joins**
  - This choke-point tests the performance of join operators when the join is sparse. Sometimes joins involve relations where only a small percentage of rows in one of the tables is required to satisfy a join. When tables are larger, typical join methods can be sub-optimal. Partitioning the sparse table, using Hash Clustered indexes or implementing bloom filter tests inside the join are techniques to improve the performance in such situations [7].

- **Data Access Locality**
  - **SNBI-3.1/TCPH-3.3: QOPT Detecting correlation**
    - This choke-point tests the ability of the query optimizer to detect data correlations and exploiting them. If a schema rewards creating clustered indexes, the question then is which of the date or data columns to use as key.
      In fact it should not matter which column is used, as range- propagation between correlated attributes of the same table is relatively easy. One way is through the creation of multi-attribute histograms after detection of attribute correlation.
      With MinMax indexes, range-predicates on any column can be translated into qualifying tuple position ranges. If an attribute value is correlated with tuple position, this reduces the area to scan roughly equally to predicate selectivity.
  - **SNBI-3.2: STORAGE Dimensional clustering**
    - This chokepoint tests suitability of the identifiers assigned to entities by the storage system to better exploit data locality.
      A data model where each entity has a unique synthetic identifier,
      e.g. RDF or graph models, has some choice in assigning a value to this identifier. The properties of the entity being identified may affect this, e.g. type (label), other dependent properties,
      e.g. geographic location, date, position in a hierarchy etc, depending on the application. Such identifier choice may create locality which in turn improves efficiency of compression or index access.
  - **SNBI-3.3: QEXE Scattered Index Access patterns**
    - This choke-point tests the performance of indexes when scattered accesses are performed. The efficiency of index lookup is very different depending on the locality of keys coming to the indexed access.
      Techniques like vectoring non-local index accesses by simply missing the cache in parallel on multiple lookups vectored on the same thread may have high impact.
      Also detecting absence of locality should turn off any locality dependent optimizations if these are costly when there is no locality. A graph neighborhood traversal is an example of an operation with random access without predictable locality.

- **Expression Calculation**
  - **SNBI-4.1/TPCH-4.2a: QOPT Common subexpression elimination**
    - This choke-point tests the ability of the query optimizer to detect common sub-expressions and reuse their results. A basic technique helpful in multiple queries is common subexpression elimination (CSE).
      CSE should recognize also that average aggregates can be derived afterwards by dividing a SUM by the COUNT when those have been computed.
  - **SNBI-4.2/TCPH-4.2d: QOPT Complex boolean expression joins and selections**
    - This choke-point tests the ability of the query optimizer to reorder the execution of boolean expressions to improve the performance. Some boolean expressions are complex, with possibilities for alternative optimal evaluation orders.
      For instance, the optimizer may reorder conjunctions to test first those conditions with larger selectivity [19].
  - **SNBI-4.3 QEXE Low overhead expressions interpretation**
    - This choke-point tests the ability to efficiently evaluate simple expressions on a large number of values. A typical example could be simple arithmetic expressions, mathematical functions like floor and absolute or date functions like extracting a year.

- **Correlated Sub-queries**
  - **SNBI-5.1/TCPH-5.1: QOPT Flattening sub-queries**
    - This choke-point tests the ability of the query optimizer to flatten execution plans when there are correlated sub-queries. Many queries have correlated sub-queries and their query plans can be flattened,
      such that the correlated sub-query is handled using an equi-join, outer-join or anti-join. To execute queries well, systems need to flatten both sub-queries, the first into an equi-join plan, the second into an anti-join plan.
      Therefore, the execution layer of the database system will benefit from implementing these extended join variants.
      The ill effects of repetitive tuple-at-a-time sub-query execution can also be mitigated if execution systems by using vectorized, or block-wise query execution, allowing to run sub-queries with thousands of input parameters instead of one.
      The ability to look up many keys in an index in one API call creates the opportunity to benefit from physical locality, if lookup keys exhibit some clustering.
  - **SNBI-5.2/TCPH-5.3: QEXE Overlap between outer and sub-query**
    - This choke-point tests the ability of the execution engine to reuse results when there is an overlap between the outer query and the sub-query. In some queries, the correlated sub-query and the outer query have the same joins and selections.
      In this case, a non-tree, rather DAG-shaped [21] query plan would allow to execute the common parts just once, providing the intermediate result stream to both the outer query and correlated sub-query,
      which higher up in the query plan are joined together (using normal query decorrelation rewrites).
      As such, the benchmark rewards systems where the optimizer can detect this and the execution engine supports an operator that can buffer intermediate results and provide them to multiple parent operators.
  - **SNBI-5.3/¿TCPH-5.2?: QEXE Intra-query result reuse**
    - This choke-point tests the ability of the execution engine to reuse sub-query results when two sub-queries are mostly identical.
      Some queries have almost identical sub-queries, where some of their internal results can be reused in both sides of the execution plan, thus avoiding to repeat computations.

- **Parallelism and Concurrency**

    - **SNBI-6.1/TCPH-6.3: QEXE Inter-query result reuse**
        - This choke-point tests the ability of the query execution engine to reuse results from different queries. Sometimes with a high number of streams a significant amount of identical queries emerge in the resulting workload.
          The reason is that certain parameters, as generated by the workload generator, have only a limited amount of parameters bindings.
          This weakness opens up the possibility of using a query result cache, to eliminate the repetitive part of the workload.
          A further opportunity that detects even more overlap is the work on recycling, which does not only cache final query results, but also intermediate query results of a "high worth".
          Here, worth is a combination of partial-query result size, partial-query evaluation cost, and observed (or estimated) frequency of the partial-query in the workload.
- **RDF and Graph Specifics**
    - **SNBI-7.1: QOPT Translation of internal ids to external ones [KILL WITH FIRE]**
        - This choke-point tests the ability of the query optimizer to delay the translation between internal and external entity ids to late in the query. Translate at point of minimum cardinality,
          e.g. after top k order by RDF and possibly graph models often use a synthetic integer identifier for entities, e.g. URI's .
          For presentation to the client applications, these identifiers must be translated to their original form,
          e.g. the URI string that was used when loading the data. This should be done as late as possible, or at the point of minimal cardinality in the plan.
    - **SNBI-7.2: QOPT Cardinality Estimation of transitive paths**
        - This choke-point tests the ability of the query optimizer to properly estimate the cardinality of intermediate results when executing transitive paths. A transitive path may occur in a "fact table" or a "dimension table" position.
          A transitive path may cover a tree or a graph, e.g. descendants in a geographical hierarchy vs. graph neighborhood or transitive closure in a many-to-many connected social network.
          In order to decide proper join order and type, the cardinality of the expansion of the transitive path needs to be correctly estimated.
          This could for example take the form of executing on a sample of the data in the cost model or of gathering special statistics, e.g. the depth and fan-out of a tree. In the case of hierarchical dimensions,
          e.g. geographic locations or other hierarchical classifications, detecting the cardinality of the transitive path will allow one to go to a star schema plan with scan of a fact table with a selective hash join.
          Such a plan will be on the other hand very bad for example if the hash table is much larger than the "fact table" being scanned.
    - **SNBI-7.3: QEXE Execution of a transitive step**
        - This choke-point tests the ability of the query execution engine to efficiently execute transitive steps. Graph workloads may have transitive operations, for example finding a shortest path between vertices.
          This involves repeated execution of a short lookup, often on many values at the same time, while usually having an end condition, e.g. the target vertex being reached or having reached the border of a search going in the opposite direction.
          For the best efficiency, these operations can be merged or tightly coupled to the index operations themselves. Also parallelization may be possible but may need to deal with a global state, e.g. set of visited vertices.
          There are many possible tradeoffs between generality and performance
    - **SNBI 7.4: QEXE Efficient evaluation of termination criteria for transitive queries**
        - This tests the ability of a system to express termination criteria for transitive queries so that not the whole transitive relation has to be evaluated as well as efficient testing for termination.
    - **SNBI 7.5: QEXE Path pattern reuse**
        - This choke-point tests the ability of the execution engine to reuse work across graph traversals **[TODO: complete in more detail]**
          E.g., when computing paths within a range of distances, it is often possible to incrementally compute longer paths by reusing paths of shorter distances that were already computed

TPC-H Interactive Chokepoints

Paper:

- http://oai.cwi.nl/oai/asset/21424/21424B.pdf
- https://www.researchgate.net/profile/Peter_Boncz/publication/291257517_TPC-H_Analyzed_Hidden_Messages_and_Lessons_Learned_from_an_Influential_Benchmark/links/5852dbf708ae95fd8e1d749b/TPC-H-Analyzed-Hidden-Messages-and-Lessons-Learned-from-an-Influential-Benchmark.pdf
- https://link.springer.com/chapter/10.1007%2F978-3-319-04936-6_5

Slides:

- http://www.tpc.org/tpctc/tpctc2013/slides_and_papers/005.pdf

- **Aggregation Performance**: Performance of aggregate calculations
    - **TPCH-P1.1 QEXE:** Ordered Aggregation
    - **TPCH-1.2 QOPT:** Interesting Orders
    - **TPCH-1.3 QOPT:** Small Group-by Keys (array lookup)
    - **TPCH-1.4 QEXE:** Dependent Group-By Keys (removal of)
- **Join Performance**: Voluminous joins, with or without selections
    - **TPCH-2.1 QEXE:** Large Joins (out-of-core)
    - **TPCH-2.2 QEXE:** Sparse Foreign Key Joins (bloom filters)
    - **TPCH-2.3 QOPT:** Rich Join Order Optimization
    - **TPCH-2.4 QOPT:** Late Projection (column stores)

- **Data Access Locality**: Non-full-scan access to (correlated) table data
  - **TPCH-3.1 STORAGE:** Columnar Locality (favors column storage)
  - **TPCH-3.2 STORAGE:** Physical Locality by Key (clustered index, partitioning)
  - **TPCH-3.3 QOPT:** Detecting Correlation (ZoneMap,MinMax,multi-attribute histograms)
- **Expression Calculation**: Efficiency in evaluating (complex) expressions
  - **TPCH-4.1 Raw Expression Arithmetic**
    - **TPCH-4.1a QEXE**: Arithmetic Operation Performance
    - **TPCH-4.1b QEXE**: Overflow Handling (in arithmetic operations)
    - **TPCH-4.1c QEXE**: Compressed Execution
    - **TPCH-4.1d QEXE**: Interpreter Overhead (vectorization; CPU/GPU/FPGA JIT compile)
  - **TPCH-4.2 Complex Boolean Expressions in Joins and Selections**
    - **TPCH-4.2a QOPT**: Common Subexpression Elimination (CSE)
    - **TPCH-4.2b QOPT**: Join-Dependent Expression Filter Pushdown
    - **TPCH-4.2c QOPT**: Large IN Clauses (invisible join)
    - **TPCH-4.2d QEXE**: Evaluation Order in Conjunctions and Disjunctions
  - **TPCH-4.3 String Matching Performance**
    - **TPCH-4.3a QOPT**: Rewrite LIKE(X%) into a Range Query
    - **TPCH-4.3b QEXE**: Raw String Matching Performance (e.g. using SSE4.2)
  - **TPCH-4.3c QEXE**: Regular Expression Compilation (JIT/FSA generation)
- **Correlated Subqueries**: Efficiently handling dependent subqueries.
  - **TPCH-5.1 QOPT**: Flattening Subqueries (into join plans).
  - **TPCH-5.2 QOPT**: Moving Predicates into a Subquery
  - **TPCH-5.3 QEXE**: Overlap between Outer- and Subquery
- **Parallelism and Concurrency**: Making use of parallel computing resources
  - **TPCH-6.1 QOPT**: Query Plan Parallelization
  - **TPCH-6.2 QEXE**: Workload Management
  - **TPCH-6.3 QEXE**: Result Re-use

	CP1.1	CP1.2	CP1.3	CP1.4	CP1.5	CP1.6	CP2.1	CP2.2	CP2.3	CP2.4	CP3.1	CP3.2	CP3.3	CP4.1	CP4.2	CP4.3	CP5.1	CP5.2	CP5.3	CP6.1	CP7.1	CP7.2	C
Q1		x										x		x							R.I.P		
Q2	x	x		x			x		x		x	x									R.I.P		
Q3										x	x	x		x		x			x	x	R.I.P		
Q4	x	x		x			x	x		x			x								R.I.P		
Q5		x		x	x		x	x	x	x			x						x	x	R.I.P		
Q6		x							x												R.I.P		
Q7		x							x			x	x							x	R.I.P		
Q8						x							x					x			R.I.P		
Q9		x		x			x		x	x											R.I.P		
Q10		x					x		x			x									R.I.P		
Q11	x						x	x	x		x	x								x	R.I.P		
Q12		x						x			x									x	R.I.P		
Q13		x						x	x			x								x	R.I.P		
Q14		x						x	x			x									R.I.P	x	
Q15		x							x			x	x						x	x	R.I.P		
Q16		x		x					x	x			x								R.I.P	x	
Q17	x								x												R.I.P		
Q18	x	x				x						x			x	x					R.I.P		
Q19	x			x			x		x	x			x				x				R.I.P		
Q20						x	x													x	R.I.P		
Q21		x					x		x	x		x	x				x		x		R.I.P		
Q22				x		x	x				x		x				x	x	x		R.I.P		
Q23						x			x	x			x			x					R.I.P		
Q24						x	x		x	x		x				x					R.I.P		
Q25		x					x	x		x			x				x		x			x	

Query 1 (Alex)

NOTE1: TPC-H CHOKEPOINTS NOT EVALUATED YET

- **[NEW CHOKEPOINT REQUIRED]** SNBI-1.2: QEXE High Cardinality group-by performance – in 3 years of data there are potentially 36 year groups, (ranges of) these could be used as keys to partition the messages for parallel execution.
- **[REMOVE]** SNBI-3.1: QOPT Detecting correlation – partition scheme (above) could be based on distribution of years in message creation dates for equal work assignment, e.g., if more messages appear in later years.
- SNBI-3.2: STORAGE Dimensional clustering – store messages in a way that makes grouping easier, e.g., sorted by grouping keys
- SNBI-4.1: QOPT Common subexpression elimination – average_message_length computation can be done once, as the very end, because sum_message_length & message_count are being tracked already
- **[NEW CHOKEPOINT FOR EXTRACTING YEAR]**

Query 2 (Arnau)

SNBI-1.1 The query asks for a group by, whose key is partially defined by the country of the person. Thus, if the first joins ensure a sorted output by country, then the group by can become cheaper. The same can happen with the month of the messages. If the messages are sorted by date, the access to the structure used to perform the aggregation will be more local. Even, I think that for the tags, it is possible to have an interesting order if the message X message-tag join is clustered by tag.

SNBI-1.2 The amount of different group-by keys for this query can be of more than 4M in the worst case. difficulting the locality if the aggregation structure does not fit the cache or memory in case of a very large dataset.

[Alex] "SNBI-1.3: QEXE Complex aggregate performance" <-- "age group of person" grouping key?

[Alex] "SNBI-1.4: QOPT Top-k push down" <-- with all the grouping keys and sorting criteria it's hard for my brain to know, but it seems like there may be potential for this – although maybe it is difficult because message_count is the first sorting criteria

SNBI-2.1 The messages table needs to be joined with the message/tags table. The optimizer should do this after joining the messages with the persons of the country, since doing the alternate way would consume more memory and produce a rather large table.

SNBI-2.3 I think that since the persons in the countries will be a rather small table compared to the messages table, a hash-join operator would be preferable, to hash over the person table and then filter those messages whose creator are in the hash table.

SNBI-3.1 Message's creation date is correllated with the messages table position. **Unconfirmed.**

SNBI-3.2 For those systems who have adjacencies materialized, having those nodes with similar attributes to have similar identifiers, will reduce the amount of data accessed when accessing the indexes. I think this chokepoint will be fullfilled by many or all of the queries.

Query 3 (Marcus)

(I took the choke point numbers from http://oai.cwi.nl/oai/asset/21424/21424B.pdf)

SNBI-1.1 Ordered Aggregation: can help, if domain of group-by key is large -> group by on tag name (cardinality is around 16K)

--> no chokepoint

CP1.2 Interesting orders: probably not helpful here (if join already provides order, this could be used in the aggregation)

--> no chokepoint

CP1.3 Small Group-by keys: there is only one group-by key (this one could however be represented by 14 bits instead of 64)

--> no chokepoint

CP2.2. Sparse Foreign Key Joins: not sure if this is applicable here

CP2.4. Only the tagname/creationdate (+join attributes) are of real interest, so columnar storage might help

--> chokepoint

CP3.1 Columnar Locality: Scan on creationdate, count aggregation on tags

--> chokepoint

CP3.2 Physical Locality by Key: Range-partitioning the table by creationdate is likely a good idea

--> chokepoint

CP3.3 Scattered index accesses - **Undecided**

CP4.1 Expression arithmetics: The query states that the second date has to be computed / doing this in a smart way will likely help here

--> chokepoint

SNBI-5.3

--> chokepoint

CP6.1 Query Plan Parallelization: Both subqueries (for the first and the second month) can be executed independently from each other

--> chokepoint

TPCH-6.1

--> It applies but should we include it in chokepoints?

Query 4 (Moritz)

- SNBI-1.1: Ordered Aggregation:

  - When joining post_tag with tags having the posts orderd by forum would allow for very cheap count aggregation

- SNBI-1.2: QEXE High Cardinality group-by performance.

  - This query requires e.g. a group by with high cardinality when counting the number of posts per forum with a given tag

- SNB-1.4

- SNBI-2.1: QOPT Rich join order optimization

  - When finding forums by country its important to first connect forums to persons and than to country, rather than persons to country and than forums which would be much more expensive

  - When counting posts the runtime should decide whether first filtering by forums or by tag.

-SNBI-2.2

-SNBI-2.4

-SNBI-3.3 I think the runtime has to do the right decisions concerning these chokepoints, but I think it should be trivial to decide correctly **Undecided**

[Marcus] SNBI-1.5: there seems to be at least a functional dependency between the forum id and the corresponding forum title (so grouping by both will probably result in grouping only by forum id)

[Marcus] SNBI-2.4: The selection on country name and tc name could result in sparse joins (?)

Query 5 (Alex)

NOTE1: ALL THINGS IN BOLD RED I AM UNSURE ABOUT, PLEASE DOUBLE CHECK THOSE ESPECIALLY

NOTE2: TPC-H CHOKEPOINTS NOT EVALUATED YET

- SNBI-1.2: QEXE High Cardinality group-by performance
  - First/top group key, person.id, is the only one necessary
  - There are many unique person.id values and no dependence between the aggregations of each
- SNBI-1.4: QOPT Top-k push down
  - Once 100 rows are retrieved, this part of the query "count number of Posts they made in any of those (most popular) Forums" does not need to perfect this check "in any of those (most popular) Forums" for any persons that have fewer Posts than the lowest top 100 result, regardless of the Forum those Posts are in (once adding the Forum constraint that count will only decrease)
- SNBI-1.5: QEXE Dependent group-by keys
  - Of all of these grouping keys "person.id, person.firstName, person.lastName, person.creationDate" only person.id is actually needed, because it is unique, the rest can be ignored
- SNBI-2.1: QOPT Rich join order optimization
  - First part of query "popularity of a Forum is measured by the number of members that Forum has from the given Country." has these joins
    (Forum)-[HAS_MEMBER]-(Person)
    (Person)-[IS_LOCATED_IN]-(City)
    (City)-[IS_PART_OF]-(Country)
  - Second part of query "count the number of Posts they made in any of those (most popular) Forums." has these joins
    (Forum)-[HAS_MEMBER]-(Person)
    (Person)-[HAS_CREATOR]-(Post)
    (Post)-[CONTAINER_OF]-(Forum)
  - The order these are executed in has large influence on query complexity
- SNBI-2.2: QOPT Late projection
  - you take "SNBI-1.5" into account, then of all the person attributes returned "person.id, person.firstName, person.lastName, person.creationDate" only "person.id" needs to be materialized during grouping
  - The rest can be materialized/projected for only the top 100 persons, and only at the very end, once
- SNBI-2.3: QOPT Join type selection
  - <honest>I was too lazy to think through all the possibilities but</honest> in this case I think it is more necessary to argue against

this choke point than to argue for it (i.e., "chokepoint until proven otherwise") because there are many different joins, the joins have massively varying cardinalities, and the systems running this benchmark have totally different "join" operator implementations

- SNBI-2.4: QOPT Sparse foreign key joins
  - How sparse does a join need to be for us to call it "sparse"? Is there some selectivity (right word?) threshold?
    // all forums <--> persons in country (see below)
    (Forum)-[HAS_MEMBER]-(Person)
    // all persons <--> cities of one country
    (Person)-[IS_LOCATED_IN]-(City)
    // persons of top 100 forums <--> their posts
    (Person)-[HAS_CREATOR]-(Post)
    [Moritz] probably not a choke point. From the paper description it sounds like this choke point is mostly concerned with joins from N to 1 where due to predicates most entites won't have the join partner. My intuition is that in this query this does not apply because the only selection happens at the 100 forums and thats probably the start of the join
- SNBI-3.1: QOPT Detecting correlation
  - Is there a correlation between "person.creationDate" and number of Posts created by that Person? If so, I think "SNBI-1.4: QOPT Top-k push down" can potentially be even more effective?
    [Alex] It may be more efficient to sort the persons after they have been retrieved for the most popular Forums, before counting their posts in the popular Forums
- SNBI-3.2: STORAGE Dimensional clustering
  - Could geographic/country be used in a clustered index with persons?
    [Moritz] In the first join for identifying the forums. Having persons clustered by country could avoid having to scan all the persons. So only much fewer persons would have to be loaded for this first join
- SNBI-3.3: QEXE Scattered Index Access patterns
  - Similar rationale to "SNBI-2.1: QOPT Rich join order optimization", e.g., when going from Forum to Person (to get members)
- SNBI-5.3/¿TCPH-5.2?: QEXE Intra-query result reuse
  - Should not be performed multiple times per post: (Post)-[CONTAINER_OF]-[(Forum)] "is post in top 100 forums?
- SNBI-6.1: QEXE Inter-query result reuse
  - (City)-[IS_PART_OF]-(Country) will never change
  - Repeating parameters from BI
- SNBI-7.1: QOPT Translation of internal ids to internal ones
  - Grouping by person.id could be done by internal ID of person records. External IDs do not need to be projected until sorting of the top 100 happens

Query 6 (Arnau)

SNBI-2.1 The optimizer should decide between selecting the messages per person, filtering them by tag and perform the counting vs starting by tag and for each message, count and aggregate per person. In the first case we would have a very good locality in the map used to keep the counts per person. In the second case the filtering of messages by tag would be faster. Possibly, the first case would be improved by creating a hash table first with those messages that has the tag, for discarding unnecessary messages fast. As a counterpart, we would still need to scan the whole messages table, which is hugh. Possibly, version 2 seems better  . Actually, in order to improve locality on the map used to count, the temporal message table containing the messages and persons, could be previously sorted. **NOT a CHOKEPOINT**

SNBI-1.2 One group per person, high cardinality

SNBI-2.3 An Index Join between Tags-Message table and Message Table.

SNBI-3.2 Having messages identifiers clustered per author, would improve the locality when accessing to the like indexes and reply indexes to count **NOT a CHOKEPOINT**

TCP-6.1 When counting and aggregating the number of likes per person message, and replies per person message, this could be parallelized, in such a way that one thread does the like count while another the comment count. Having such threads in different cores could maximize the caching of the indexes used to index the likes per message and the comments per message. Also the counting could be partitioned and then reduced using multiple threads **NOT a CHOKEPOINT**

TPC-6.3 There are not so many tags compared to the amount of messages, and the distribution of tag message is skewed. Also, some tags might be popular at a given moment (flashmob). In a suffitiently long run without updates, we could cache the results of this query for reuse. Even with updates, the counts could be easily maintained. **NOT a CHOKEPOINT**

Query 7 (Marcus)

Aggregations:
==========

SNBI-1.1:
keeping the derived person relations sorted will be beneficial for the joins and grouping on the person Ids

SNBI-1.2: High Cardinality Group-By

- computing the popularity of each person requires a join between posts and likes and a subsequent grouping by personId

SNBI-1.4:
- Since the score function is a derived aggregate, I'm not sure whether there is any early-out during the processing possible (maybe by sorting on the number of likes a person received)

Joins:
==========

SNBI-2.1:
The join order between the posts, the post_tags, and the tags likely benefits from a rich join order optimization by first evaluation the join on the (filtered) tag relation

TPCH-CP2.1 Large Joins: there are some large joins in the query (between posts and likes, between posts and posttags)
--> chokepoint

SNB-2.2:
Late projection will help here since mostly key columns are used to join, the only attribute column accessed is the tag name

SNBI-2.4:
the selective filter on the tag relation will probably result in many tuples finding no join partner when joined with post_tag

SNBI-3.2:
Clustering posts by tags

SNBI-6.1:
The popularity of persons can be computed offline and reused between multiple queries

Chokepoints: 1.2, 2.3, 3.2 and 3.3, 6.1, 7.1

Query 8 (Moritz)

Chokepoints: 1.6, 3.3, 5.2

Query 9 (Alex)

- SNBI-1.2/TCPH-1.1: QEXE High Cardinality group-by performance
  - Many forums
- SNBI-1.4: QOPT Top-k push down
  - There may be an opportunity for this.
    "For every forum, count number of Posts that have a Tag from TagClass1 (count1), and the number of posts that have a tag from TagClass2."
    These counts are of **(forum)-[CONTAINER_OF]-(post)-[HAS_TAG]-(tag)-[HAS_TYPE]-(tagClass)**
    If after calculating TagClass1 count that count is found to be low (e.g. 1), then for TagClass2 only **(forum)-[CONTAINER_OF]-(post)** (no tag check) may need to be performed, because that count already provides the upper bound. If that count is low then the score will be low. After once the top 100 have been collected the runtime should know the minimum score. Any forum that is guaranteed to not reach that score can be discarded early, e.g., when **(forum)-[CONTAINER_OF]-(post)-[HAS_TAG]-(tag)-[HAS_TYPE]-(tagClass)** is low AND **(forum)-[CONTAINER_OF]-(post)** count for TagClass2 is low.
  - This also means that it may be better to executes the TagClass with lowest count first (**see SNBI-2.1QOPT Rich join order optimization**)
- SNBI-2.1: QOPT Rich join order optimization
  - See last point, above
  - Also, executing the TagClass with (likely) lowest count first means that if one TagClass has count of zero it can discard that forum immediatelywhat
- SNBI-2.3: QOPT Join type selection
- SNBI-2.4/TCPH-2.2: QOPT Sparse foreign key joins
  - **This chokepoint depends on the TagClass parameters, which currently are not selected to match the chokepoint, but could be done if needed.**

Query 10 (Arnau)

1.2

2.1

2.3

3.2

Query 11 (Marcus)

1.1 <- Chokepoint

2.1 <- Chokepoint

2.2 <- Chokepoint (Tag.name)

2.3 <- Chokepoint

3.1 <- Chokepoint (Change selection of parameters accordingly)

3.2 <- Chokepoint (cluster Message-like join table by message, to make counting likes cheaper)

6.1 <- Chokepoint (computing incrementally the size of the intersection between tags of a reply and the message, or an index between persons to messages that qualify in terms of the tags they have with respect to the message they reply)

Query 12 (Moritz)

1.2 <- Chokepoint

2.2 <- Chokepoint

3.1 <- Chokepoint

6.1 <- Chokepoint

Query 13 (Alex)

1.2 <- Chokepoint

2.2 <- Chokepoint

2.3 <- Chokepoint

3.2 <- Chokepoint

6.1 <- Chokepoint

Query 14 (Arnau)

1.2 <- Chokepoint

2.2 <- Chokepoint

2.3 <- Chokepoint

3.2 <- Chokepoint

7.2 <- Chokepoint

7.3 <- Chokepoint

7.4 <- Chokepoint

Query 15 (Marcus)

1.2 <- Chokepoint

CP 2.2 Late projection <- **NOT A CHOKEPOINT**
- only join attributes are required (for persons only their ids and their places are needed)

2.3 <- Chokepoint

CP2.4 Sparse Foreign Key Joins <- **NOT A CHOKEPOINT**
- Persons could be partitioned by their country of residence

- Only persons within a country are considered / selection on the :1 side of the join

CP3.1 Detecting Correlation <- **NOT A CHOKEPOINT**
- friends tend to have more friends within the same country than in others (?)

3.2 <- Chokepoint

- If we have frienships sorted by first column first, and second column second, If identifiers to persons are assigned in such a way that given a person, his friends have close identifiers, when scanning the friendship table and probing against a Tree based set containing the persons in a given country.

3.3 <- Chokepoint

CP 5.3 Intra-query result re-use <- Chokepoint
- The friendships of all persons coming from the same country is acutally needed twice, once for computing the average number of friends and one for computing the number of friends for each person

CP 6.1 Inter-query result re-use <- Chokepoint
- the average number of friends per country could be easily cached and reused later on

Query 16 (Moritz)

CP1.2 <- Chokepoint

CP1.4 <- Chokepoint <- we can early skip Persons based on their message count, and even if they pass the early test, we can early skip the message tag join based on the already observed tags

CP2.3 <- Chokepoint

CP2.4 is a Chokepoint

CP3.3 is a Chokepoint

CP7.2 is a Chokepoint

CP7.3 is a Chokepoint

CP7.5 is a Chokepoint

Query 17 (Alex)

CP1.1 is a chokepoint - Having the adjecency list ordered by id and keeping this order between operators makes finding distinct triangles cheaper as a partial triangle will only need to be joined with neighbors that have a larger id.

CP2.3 is a chokepoint

Query 18 (Arnau)

CP1.1 is a chokepoint

CP1.2 is a chokepoint

CP1.6 is a chokepoint

CP3.2 is a chokepoint

Query 19 (Marcus)

CP1.2 is a chokepoint : exists clause is effectively a group-by + final grouping by person

CP1.4 is a chokepoint: if you start from persons filtered by birthday, one can apply top-k pushdown

CP2.1 is a choke point: there are multiple possible ways to start this query, either starting from the tag classes or from the persons born after a given birthday

CP2.3 is a choke point: choose between BFS and DFS makes a difference in evaluation

CP2.4 is a choke point: there are many joins involved in this query, some of them could potentially become sparse, i.e., between comments and persons

CP3.3 is a Chokepoint

CP5.1 IS PENDING

- it's possible to combine the two group bys at the end into one physical group by

CP6.1 is a chokepoint.

CP7.3 This query does not have a real transitive part but for the traversals it can make use of many of the optimization that are used in transitive queries.

CP7.4 TODO: Adjust 7.4 in spirit of the remark for 7.3 "This query does not have a real transitive part but for the traversals it can make use of many of the optimization that are used in transitive queries."

Query 20 (Moritz)

CP1.6: There are just 72 TagClasses

CP2.1: Proper join order impacts the performance.

CP6.1: We can materialize partial counts for each Tagclass, which can be updated everytime a message is added. When a query arrive, the counts are summed to compute the final counts.

Query 21 (Alex)

CP 1.2 is a chokepoint, as the size of the group by will linearly increase with the scale factor, and at some point it will be large enough.

CP2.1 is a chokepoint: So many joins you can play with depending on the cardinalities

CP2.3 is a chokepoint: In some places(specially message->likes) you have to properly chose between index based join or hash-join

CP2.4 is a chokepoint: The number of messages to join with likes is small compared to the total number

CP3.2 is a chokepoint: clustering by message would impact the performance

CP3.3 is a chokepoint: If an index is used for message->likes, access will be sparse

CP5.1 is a chokepoint: For zombie like count and total like count are correlated sub-queies that can be flattened

CP5.3 is a chokepoint: Simila to 5.1. Also, the persons that qualify for likes is similar to those that qualify for zombies (besides the country filter)

Query 22 (Arnau)

CP1.4 is a chokepoint: there are top-k push down opportunities if some interactions are met, we don't need to look at others

CP1.6 is a chokepoint: there as many groups as cities

CP2.1 is a chokepoint: there are many possible join orders

CP3.1 is a chokepoint: It is more likely that persons reply other persons that already replied to their messages. Detecting such correlation is needed for proper cardinality estimation

CP3.3 is a chokepoint: The use of index is prominent in the whole query, and as long as the prunning of the considered pairs goes, index accesses become more scattered

CP5.1 is a chokepoint:

CP5.2 is a chokepoint:

CP5.3 is a chokepoint: In general, there are many overalps and results reuse for the sub-queries

Query 23 (Marcus)

CP1.6 is a chokepoint: the number of counts is 12*110, so pretty low.

CP2.3 is a chokepoint: index vs hash-join depending on the country and the scale factor

CP2.4 is a chokepoint: the ratio of persons and messages is very mall

CP3.3 is a chokepoint: if an index is used to access messages, it is likely that these accesses are very sparse

CP4.3 is a chokepoint: the extraction of the month from dates

Query 24 (Moritz)

CP1.6 is a chokepoint: the number of groups is 36*5

CP2.1 is a chokepoint: depending on the tagclass its better to join continents and message first and then tags or the other way around

CP2.3 is a chokepoint: depending on the tagclass and thus the tags, an index join or a hash-join should be used

CP3.2 is a chokepoint: having messages clustered by date will improve locality

CP4.3 is a chokepoint: extracting year and months from dates

Query 25 (all)

CP1.2 is a chokepoint (many persons)

CP2.1 is a chokepoint (tons of joins)

CP2.2 is a chokepoint (tons of joins with different cardinalities)

CP2.4 is a chokepoint (very sparse joins between reduced set of messages which few results as an outcome of the join.)

CP3.3 is a chokepoint (scattered indexes access patterns if used index joins)

CP5.1 is a chokepoint

CP5.3

CP7.2

CP7.3