# Graph Processing on an "almost" Relational Database

Ramesh Subramonian
Oracle Labs
ramesh.subramonian@oracle.com

## ABSTRACT

It would be hard to disagree with the contention that graph processing (whether it be of connections between people, shopping habits, ...) allows the creation of valuable data-driven products and insights. There is less consensus on the systems that make it easy to analyze these graphs. In this paper, we argue that a relational database (actually, a close approximation to one) is well suited for many graph processing applications. We restrict our claims to the following case, which, we believe, dominates the nature of much data analyses — *the data does not change during the analysis in response to external events.* We present representative examples from our work at LinkedIn, the world's largest professional network. We present performance results for these examples using `Q`, a single-node, analytical database.

## Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Big Data, Column-Stores, Relational Databases

## 1. INTRODUCTION

Data represented in graphs is particularly interesting because of the insights that can be gleaned from the interconnectedness of the nodes, not just the nodes (and their attributes) themselves. However, while traversal of the edges is easy to do conceptually, it is hard to do at scale in a performant manner. This has prompted much specialized work in processing large graphs.

Some researchers have focused on converting problems from a relational model to a graph model arguing that "graph database management systems provide an effective and efficient solution to data storage where data are highly connected and systems need to scale to large data sets." [10]. In contrast, we believe that converting from a graph model to a relational model actually simplifies analysis and does not crimp scalability.

In our experience, a significant simplification that is often justified when looking for insights into the data is that the data does not change during the period of analysis. This is not always true. Examples where this assumption is unjustified are (i) monitoring for fraud, abuse, risk, (ii) high-frequency trading. However, we would contend that it is the norm more than the exception that static, but not stale, data is sufficient. Examples are when one is trying to tune the behavior of a large web application (such as LinkedIn) to make it more (i) engaging to users (ii) more targeted for advertisers/marketers.

When the above assumption is justified, one can leverage this assumption to create very simple yet powerful and performant analytical systems. This is the approach we have taken in the development of an almost relational database (Section 2), tuned for analytics. Using some non-trivial real-world applications, we hope to persuade the reader of the soundness of this approach.


http://dx.doi.org/10.1145/2621934.2621935
*GRADES2014,* June 22, 2014, Snowbird, Utah, USA.


### 1.1 Contributions

This paper makes the following three contributions.

First, we present several interesting applications that are powered by the ability to analyze graph data of various types. Examples of graph data we have analyze include (i) connections between users — the well known "social graph" (Section 4.1) (ii) endorsements — where user $x$ endorses user $y$ for skill $z$ (Section 4.3) (iii) page traversals — where nodes are pages of a web application and an edge from $x$ to $y$ (suitably annotated with user and time information) represents that a user visited page $y$ after page $x$ (Section 4.2)

Second, in earlier work [3], we have developed a moderately general purpose vector programming model for several data transformation tasks at LinkedIn. Here, we show that this can be used for agile and efficient programming of graph-data problems. Following Iverson [5], we suggest that the vector programming language proposed be judged in terms of the following.

1. ease of expressing constructs arising in problems
2. suggestivity
3. ability to subordinate detail
4. economy
5. amenability to formal proofs

In Section 4, we interleave actual code with pseudo-code to show the ease with which a natural way of decomposing the problem leads to working code. In practice, we find that a relatively small number of operators is adequate. When efficiency demands, it is possible to add custom operators (as in Step 4 of Section 4.1) as long as they conform to the requirements of Section 2.1.

Third, we provide performance results (Section 5) to bolster our contention that one can perform realistic and fairly complex analyses of large graph data on a using a minor variation of the relational database model. In particular, our results are obtained on a single, moderately powerful machine, although that is not a restriction of the approach we advocate.

## 1.2 Related Work

In comparison to Green Marl [4], which is a domain specific language for graph algorithms, `Q` is a general purpose language for the manipulation of tables. We believe that graphs can be represented (and efficiently queried) as tables. In Section 4, we provide a diverse set of examples as evidence that this is often the case.

The database discussed in this paper shares some of the design principles of SAP HANA [9] and some significant differences. In common, "data can be queried and manipulated in the same place without having to convert it into a different format" and "processing (is provided) directly within the database core instead of creating a new layer on top of it" and "universal tables for storing vertices and edges, with each attribute being mapped to a table column". However, we do not support data changes nor do we provide specialized "graph routines". Instead, we show how existing table manipulation routines suffice in most cases and how user-defined functions can fill in the gaps when needed.

Several authors [1, 6, 7, 8] have posited that for many practical problems, a single machine is up to the task of analyzing large graphs, both in terms of storage and run times. The one-machine approach, while it has its limitations, has significant advantages in terms of simplicity. While we do not take a position on the one-versus-many debate in this paper, our results (Section 5) show that the relational model is capable of very efficient implementations that allow large problems to be solved on a single machine.

## 1.3 Organization

The paper is organized as follows. Section 2 describes the analytical database used. Section 3 describes the data used for the experiments. Section 4 presents some interesting applications that require the analysis of graph data. Section 5 presents and discusses performance results.

# 2. Q — THE ANALYTICAL DATABASE

Q is a high-performance "almost-relational" or "semi-relational" analytical, single-node, column-store database.

By "analytical", we mean that it is not meant for transactional usage — data is loaded infrequently. The relatively high cost of data change is amortized over the analysis workload. While `Q` can handle data changes, it is expensive and therefore inadvisable to use `Q` in situations where the data changes are frequent.

By "almost-relational", we mean that it would more correctly be called a "tabular model" [2]. As Codd states, "Tables are at a lower level of abstraction than relations, since they give the impression that positional (array-type) addressing is applicable (which is not true of n-ary relations), and they fail to show that the information content of a table is independent of row order. Nevertheless, even with these minor flaws, *tables are the most important conceptual representation of relations, because they are universally understood.*"

That said, the tabular model implemented by `Q` satisfies Codd's requirements for a data model, which are

1. structural part: A collection of data structure types (the database building blocks)
2. manipulative part: A collection of operators or rules of inference, which can be applied to any valid instances of the data types listed in (1), to retrieve, derive, or modify data from any parts of those structures in any combination desired;
3. integrity part: A collection of general integrity rules, which implicitly or explicitly define the set of consistent database states or changes of state or both — these rules are general in the sense that they apply to any database using this model

## 2.1 Operations in Q

The basic building block is a table which can be viewed as a collection of fields or columns, each of which has the same number of cells or values. This allows us to use notation of the form (i) $T[i].f$ which is the value of the $i^{th}$ row of column $f$ of table $T$ or (ii) $T[f = v]$ which is the subset of rows of table $T$ where column $f$ has value $v$.

Every operation of `Q` consists of reading one or more columns from one or more tables and producing one or more columns in an existing table or a newly created one. This relatively simple statement makes for great simplicity, both in terms of implementation and programming (Section 2.3). There are a few exceptions to this rule e.g.,

1. computing an associative operation on a column e.g., sum, min, max,ldots
2. meta-data querying e.g., are the values of a column unique, describe a table, show tables ...
3. input routines — from CSV, from Hadoop, ...
4. output routines — print text, dump binary data, ...

## 2.2 Implementation Details

A full description of the architecture of `Q` is out of the scope of this paper. It is a column-store, where each column is stored in binary format as a file on the file system. `Q` does not explicitly load all necessary data into memory — instead, it leaves it to the OS to load the buffer cache with the relevant portions of disk resident files. The `Q` kernel endeavors (not always wholly successfully as in operations like sort, permute, ... ) to respect locality of reference, thereby improving performance.

## 2.3 Programming Q

Q is programmed with a vector language, not with traditional SQL. Using Codd's terminology, it is a "double-mode" data language i.e., it is usable in two modes (i) interactively at a terminal and (2) embedded in an application program written in a host language. In the second mode, the series of `Q` instructions are embedded in a shell script or the program of a scripting language e.g., PHP.

Since a full listing of the operations is not possible, we confine ourselves give the reader a flavor for them.

1. `sort T1 f1 f2 g1 g2 A_` would mean that we read columns `f1,f2` in table `T1` and create columns `g1,g2`, such that `g1` is a permutation of `f1` sorted in ascending order (specified in the "A" of the last argument) and field `g2` is permuted, as a drag-along field (specified by the underscore of the last argument)

2. `shift T1 f1 n g1 v` creates a new field `g1` in table `T1` as $T[i].g_1 \leftarrow T[i-n].f_1$ and $\forall j : 1 \leq j \leq n : T[j].g_1 \leftarrow v$
3. `w=x?y:z T w x y z` creates a new field `w` in table `T` as follows. $\forall i : 0 \leq i < |T| : \ T[i].x = \text{true} \ \Rightarrow T[i].w \leftarrow T[i].y$; else, $T[i].w \leftarrow T[i].z$

# 3. DATA

Following Linus Torvalds' dictum that "Bad programmers worry about the code. Good programmers worry about data structures and their relationships.", this section is used to explain the input data. This will make it easier to understand the problem definitions and algorithms of Section 4. Tables and fields are created in one of the following ways:

1. `load`: created in initial load from data warehouse (often HDFS)
2. `post-proc`: created in post-processing, in order to accelerate access. Since `Q` does not support any structure other than a table, the notion of the traditional index is supported in the form of additional tables/fields.

## 3.1 Notations

Since tables (unlike relations) permit positional addressing, we need some new terminology.

1. *IndexA*, Definition 2. As an example, assume table $T_1$ has 3 rows and column $f_1$ with values $[30, 20, 10]$ in that order. As an example, assume table $T_2$ has 5 rows and column $f_2$ with values $[10, 20, 30, 20, 10]$ and column $f_I$ with values $[2, 1, 0, 1, 2]$
2. *IndexB*, Definition 3. Acts like a foreign key to an implicit field with values $0, 1, 2, \ldots$
3. *IndexC*, Definition 4. Acts like a foreign key

DEFINITION 1. *Q supports 4 kinds of integers of size 1 byte, 2 bytes, 4 bytes and 8 bytes. These are referred to as I1, I2, I4, I8. Q supports 2 kinds of floating point numbers of size 4 bytes and 8 bytes. These are referred to as F4, F8.*

DEFINITION 2. $f_I = IndexA(T_2, f_2, T_1, f_1) \Rightarrow (\forall i : T_2[i].f_2 = v \Rightarrow T_1[T_2[i].f_I].f_1 = v)$

DEFINITION 3. $f_I = IndexB(T_2, T_1) \Rightarrow \forall i : T_2[i].f_I \in \{0, |T_1|\}$

DEFINITION 4. $f_2 = IndexC(T_2, T_1, f_1) \Rightarrow \forall i \exists j : T_2[i].f_2 = T_1[j].f_1$

## 3.2 TX

Table $T_X$ records activity of a user in terms of pages visited. We synthesized $1\text{B} = 2^{30}$ rows. A row of the form $(sid = s, mid = m, pid = p, time = t)$ indicates that user $m$ visited page $p$ at time $t$ in session $s$. Rows are sorted (i) primary on *sid* ascending (ii) secondary on *time* ascending

1. *sid*, `load`, I8
2. *mid*, `load`, I4, $mid = IndexC(T_X, T_M, mid)$
3. *time*, `load`, I8
4. *pid*, `load`, I2, $pid = IndexC(T_X, T_P, pid)$, where $T_P$ is a table that maps integers to strings (e.g., 1 = Home Page, ...)

## 3.3 TC

Table $T_C$ stores connections between users i.e., it lists the edges between nodes listed in $T_M$. We synthesized $16\text{B} = 2^{34}$ rows. A row of the form $(from = f, to = t)$ indicates that $f$ is connected to $t$. Note that a connection is stored as both $(f, t)$ and $(t, f)$.

1. *from*, `load`, $from = IndexC(T_C, T_M, mid)$
2. *to*, `load`, $from = IndexC(T_C, T_M, mid)$
3. *to_idx*, `post-proc`, $to\_idx = IndexA(T_C, to, T_M, mid)$
4. *fromto*, `post-proc`, *from* (I4) and *to* (I4) packed into I8

The table is sorted in ascending order with *from* being the primary sort key and *to* being the secondary sort key. Field *from* can be deleted after post-processing is complete.

## 3.4 TE

$T_E$ stores endorsements. We synthesized $4\text{B} = 2^{32}$ rows. A row of the form $(from = f, to = t, skill = s)$ indicates that user $f$ endorsed user $t$ for skill $s$.

1. *from*, `load`, I4, $from = IndexC(T_E, T_M, mid)$
2. *to*, `load`, I4, $to = IndexC(T_E, T_M, mid)$
3. *skill*, `load`, I4, $skill = IndexC(T_E, T_S, sid)$, where $T_S$ is a table that maps integers to strings (e.g., 1 = Java, 2 = PHP)

$T_E$ is sorted (i) ascending on *skill* as primary key and (ii) sorted ascending on *from* as secondary key and (iii) sorted ascending on *to* as tertiary key.

## 3.5 TE1

Table $T_{E_1}$ is derived from $T_E$ and contains the endorsements received by a user. A row of the form $(mid = m, skill = s, cnt = n)$ indicates that user $m$ was endorsed $n$ times for skill $s$.

1. *mid*, `post-proc`, I4, $mid = IndexC(T_{E_1}, T_M, mid)$
2. *skill*, `post-proc`, I4, $skill = IndexC(T_{E_1}, T_S, sid)$
3. *cnt*, `post-proc`, I4, positive integer

## 3.6 TM

Table $T_M$ lists the users. Note that $(T_M, T_C)$ together constitute the "social graph". We synthesized $256\text{M} = 2^{28}$ rows.

1. *mid*, `load`, unique, I4, sorted ascending
2. *first name*, `load`, text
3. *last name*, `load`, text
4. $C_l$, `post-proc`, I8, $C_l = IndexB(T_M, T_C)$
5. $C_u$, `post-proc`, I8, $C_u = IndexB(T_M, T_C)$
6. $E_{1l}$, `post-proc`, I4, $E_{1l} = IndexB(T_M, T_{E_1})$
7. $E_{1u}$, `post-proc`, I4, $E_{1u} = IndexB(T_M, T_{E_1})$

The table is sorted in ascending order on *mid*. Consider row $i$ in TM. Let $m = T_M[i].mid$.

- $T_M[i].C_l$ points to the first row in $T_C$ where $from = T_M[i].mid$
- $T_M[i].C_u$ points to the last row (+1) in $T_C$ where $from = T_M[i].mid$
  - $(T_M[i].C_u - T_M[i].C_l)$ is the number of connections (out-edges) of $T_M[i].mid$
  - Hence, $\forall j : T_M[i].C_l \leq j < T_M[i].C_u : \ T_C[j].from = T_M[i].mid$

- $E_{1l}$ points to the first row in $T_{E_1}$ where $mid = T_M[i].mid$
- $E_{1u}$ points to the last row (+1) in $T_{E_1}$ where $mid = T_M[i].mid$
  - $T_M[i].E_{1u} - T_M[i].E_{1l}$ is the number of unique skills that member $T_M[i].mid$ has been endorsed for
  - $\forall j : T_M[i].E_{1l} \leq j < T_M[i].E_{1u} : \ T_{E_1}[j].mid = T_M[i].mid$

# 4. GRAPH PROBLEMS

## 4.1 Computing Second Degree Network

This is a useful sub-routine for other graph computations (algorithm in Figure 1, results in Section 5.2). Given $m \in T_M.mid$, find $F_2(m) = F_1(m) \cup M_2$ where $F_1(m) \subseteq T_M.mid$ and $M_2 \subseteq T_M.mid$ such that

1. $\forall j : T_C[j].from = m \wedge T_C[j].to = y \Rightarrow y \in F_1(m)$
2. $\forall j, k : T_C[j].from \in M_1 \wedge T_C[j].to = y \Rightarrow y \in M_2$

Steps 4, 5 of Figure 1 can be accelerated by implementing it as a user-defined function (UDF), instead of iterating Steps 1,2, 3 for each row of TD1, is significantly faster. As a `Q` sysadmin, the use of UDFs is discouraged but permitted. The SQL programmer would likely have used a "self-join" to implement this algorithm.

## 4.2 Incremental Path Navigation

A web site can be considered as a graph where each page is a node and edges between nodes, when suitably labeled, indicate the activity of users as they navigate between pages. Given (1) a sequence of pages, $(p_1, p_2, \ldots, p_i)$ and (B) a new page $p_X$, the aim is to (i) determine the top $n$ most likely nodes that were visited next by users who had earlier traversed $(p_1, \ldots p_i)$ and (ii) determine the number of users who traversed the path $(p_1, \ldots p_i, p_X)$. The algorithm (and code in `Q`) for this problem is in Figure 2.

## 4.3 Filtered Endorsements

On LinkedIn, it is possible for users to endorse others users for skills. The number of endorsements that user A has for skill B can be considered as a signal that A actually has skill B. However, I may not trust all the endorsements that A has garnered, since I may not necessarily trust the endorser. Instead, I might wish to count only those endorsements made by people whose opinion I value e.g., (i) my first degree network, $F_1(m)$ (ii) people who work for the same company that I do. Informally, the problem can be stated as follows. Given (1) a skill, $s \in T_S.sid$, and (2) a set of users I trust, stored in field $mid$ in table $T_U$, find the top $n$ users ranked on the number of endorsements they have received for skill $s$ from users in $T_U$ Algorithm in Appendix, Figure 3.

## 4.4 People You Should Know (PYSK)

The commonality between recommendations like customized deals and "People You May Know" (PYMK) is that it suggests the creation of an edge, (between two people or between a person and a product or ...) to a user who is one of the nodes of the edge. A variation of this theme is to suggest the creation of an edge to a user who is **not** one of the nodes. This is equivalent to Jim introducing Jack to Jane (who currently do not know each other) because Jim knows that Jack and Jane both share an interest in golf. Algorithm in Appendix, Figure 4.

# 5. RESULTS AND DISCUSSION

## 5.1 Machine Configuration and Data Sizes

For confidentiality reasons, all data described in this paper is synthetic but representative, and is in no way related to actual user-supplied data. Data sizes were chosen simply to demonstrate the extent to which the system could be pushed and are not representative of the scale at which LinkedIn is operating — see Section 3 for sizes used. The machine used for timing has a single 24-core Intel Xeon 2.93 GHz CPU with 64 GB RAM and 0.5 TB disk.

## 5.2 Second Degree Network

In Table 1, we see that the performance (for the problem of Section 4.1) is quite acceptable even for very large networks. The time taken is dependent on the size of the first and second-degree networks. Timings reported are from "warm" runs. "Cold" runs can take as much as 40 times that of the "warm" run. In our example, all 16B edges can fit into main memory, although we do not explicitly "malloc" them as such, using an 'mmap" instead.

As the size of the second degree network scales, the bottleneck is the de-duplication. Initial results from using a GPU to de-dupe ( using a sort) indicate that we could reduce the times reported here by 50%.

| $1^{st}$ degree | $2^{nd}$ degree | time in msec |
|---|---|---|
| 120 | 64349 | 8.070 |
| 263 | 112213 | 12.53 |
| 505 | 334246 | 41.51 |
| 1021 | 694644 | 80.13 |
| 2053 | 1166594 | 166.4 |
| 4091 | 1956817 | 259.4 |
| 8199 | 4069339 | 1363 |
| 16378 | 8319301 | 1516 |

**Table 1: Performance Results for Section 4.1**

## 5.3 Interactive Path Navigation

For 1B rows of $T_X$ and 2K rows for $T_P$, a query takes an average of 2.5 seconds for the first iteration and 3.0 for subsequent ones. Using the strip mining technique (Section 5.4) reduces this to 1.3 seconds and 1.6 seconds respectively.

## 5.4 Materialization

The discerning reader would have noticed that the read-/write style of operations (Section 2.1) leads to materialization of temporary columns, which can be expensive. Our solution is what we call "compound expressions" where the programmer explicitly demarcates sections of `Q` commands between `start_compound` and `stop_compound` directives. We use strip mining to reduce writing to memory/disk. Not all commands (e.g., sort, permute, ...) can be put within such a zone Currently, the system does not automatically detect such zones. Hence, as of today, this option is reserved for advanced programmers.

## 5.5 Batch Execution

Q favors an incremental programming style, where the programmer writes a single operation, waits for it to terminate, performs queries to confirm that it succeeded as

expected and then proceeds to the next step. While we find this a great productivity boost compared to a batch programming style suggested by Hadoop, it means several database invocations are necessary to execute a script. Once a script is debugged and ready for production use, `Q` allows a "batch" command, which allows the user to invoke the database once to perform several operations. This is reminiscent of SAP's HANA where "WIPE permits multiple complex operations to be combined within a single statement, thereby reducing the need for several roundtrips between the application and the database system"[9].

### 5.6 Issues in Programming in `Q`

While we have no wish to engage in religious wars on programming style, we believe that the examples (Figure 1, 3) have provided empirical evidence that `Q` meets Iverson's criteria (Section 1.1).

In our experience, data analysts often have a lot of intuition about the data that they are dealing with. `Q` encourages the user to push this knowledge into the code, without hard-coding values. For example, in many cases (not all), one can often eliminate the long tail of the distribution and create much more compact encodings of the data. While `Q` does not do compression, one can choose integers of different sizes (Definition 1). Since most database operations are I/O bound, the single biggest speedup is often obtained by scanning as little data as possible.

One might argue that `Q` is a retrograde step from SQL where a higher level declaration of intent is translated by an optimizing compiler into an execution plan. However, our experience is that the loss in expressivity is offset by

1. the agility of development that interactivity allows
2. the efficiency of the final plan.
3. the "plans" or scripts are rarely so long as to be onerous to write and maintain

## 6. CONCLUSION

The examples presented in this paper, while simplistic for ease of exposition, represent the potential that lies in the analysis of graph data. The nature of such analyses are that they are interactive and iterative but are not time sensitive in that the data does not need to be the most current. Given the above, (almost) relational databases allow agile and efficient development of algorithms for analysis, thereby aiding the ability to explore different hypotheses quickly.

## 7. REFERENCES


[1] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *SC10*, 2010.

[2] E. F. Codd. Relational database; a practical foundation for productivity. *Communications of the ACM*, 25(2):109–117, February 1982.

[3] R. S. et al. In data veritas — data driven testing for distributed systems. In *6th International Workshop on Testing Database Systems*, 2013.

[4] S. Hong, H. Chafi, and E. Sedlar. Green-marl: A DSL for easy and efficient graph analysis. In *ASPLOS*, 2012.

[5] A. Iverson. Notation as a tool of thought. *Communications of the ACM*, 23(8):444–465, 1980.

[6] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, 2012.

[7] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *XXXX*, 9999.

[8] A. Rowstron, D. Narayanan, A. Donnelly, G. OShea, and A. Douglas. Nobody ever got fired for using hadoop on a cluster. In *1st International Workshop on Hot Topics in Cloud Data Processing*, 2012.

[9] M. Rudolf, M. Paradies, C. Bornhvd, and W. Lehner. Synopsys: Large graph analytics in the SAP HANA database through summarization. In *GRADES*, 2013.

[10] R. D. Virgilio, A. Maccioni, and R. Torlone. Converting relational to graph databases. In *GRADES*, 2013.

1 Find $i$ such that $T_M[i].mid = m$. Fast because $T_M$ is sorted on *mid*.

```
i=‘q f_to_s TM mid "op=[get_idx]:val=[$m]"‘
```

2 Find range, $[T_{C_l}, T_{C_u}]$, of rows in $T_C$ that contain edges out of $m$

```
TC_lb=‘q f_to_s TM TC_lb "op=[get_val]:idx=[$i]"‘
TC_ub=‘q f_to_s TM TC_ub "op=[get_val]:idx=[$i]"‘
```

3 Create TD1 = $F_1(m)$ by copying above row range for field *to_idx* of table $T_C$

```
q copy_fld_ranges TC to_idx "" $TC_lb $TC_ub TD1
```

4 Repeat previous steps for each $m' \in F_1(m)$ (i.e., each row of TD1) to create TD2
By using field `to_idx` and not field `to`, we avoid searching $T_M$ for each entry of TD1
Implemented as “user-defined function”

5 De-dupe the results of above

```
q mk_uq TD2 mid Tout mid
```

**Figure 1: Code for Problem 4.1**

1 As a pre-processing step, create a boolean column SameSessAsPrev that is
1 true if the current row and the previous row are in the same session.

```
q f1opf2 TX sid 'op=[shift]:val=[1]:newval=[0]' prev_sid
q f1f2opf3 TX sid prev_sid == same_sess_as_prev
```

2 Let $x_i$ be a boolean field in $T_X$ that marks the occurrence of a sequence $(p_1, \ldots p_i)$
2 Create boolean field $x_j = x_{i+1}$ to select rows such that
2 (i) page matches $p_X$ (ii) $x_i$ of previous row is true
2 (iii) *sid* of previous row is same as that of current one

```
q f1s1opf2 TX pid $pX '==' y
q f1f2opf3 TX y Previous '&&' z
q f1f2opf3 TX z SameSessAsPrev '&&' == xj
```

3 Summing $x_j = x_{i+1}$ yields the number of occurrences of the pattern

```
q f_to_s TX xj sum
```

4 find top $n$ most popular **next** pages as follows

```
q f1opf2 TX xj 'op=[shift]:val=[1]:newval=[0]' x
q f1f2opf3 TX x SameSessAsPrev '&&' == y
q count TX pid y TP cnt
q sortf1f2 TP cnt idx A_
q pr_fld TP idx:cnt | head -$n
q rename TP y Previous // Set up for next iteration
```

**Figure 2: Appendix: Code for Problem 4.2**

0 Input is skill, $s$, and trusted users $T_U$, table with one column = *mid*

1 find $i$ such that $T_S[i].sid = s$. Fast because $T_S$ is sorted on *sid*.

```
i=‘q f_to_s TS sid "op=[get_idx]:val=[$s]"‘
```

2 Find range, $[T_{E_l}, T_{E_u}]$, of rows in $T_E$ that contain endorsements for skill $s$

```
TE_lb=‘q f_to_s TS TE_lb "op=[get_val]:idx=[$i]"‘
TE_ub=‘q f_to_s TS TE_ub "op=[get_val]:idx=[$i]"‘
```

3 Copy endorsements for skill $s$ into temp table, $T_1$

```
q copy_fld_ranges TE from "" $TE_lb $TE_ub T1
q copy_fld_ranges TE to "" $TE_lb $TE_ub T1
```

4 Using a join, create boolean field $x \in T_1$ such that $T_1[i].mid \in T_U.mid \Rightarrow T_1[i].x \leftarrow 1$

4 Operation is fast since $T_1$ is sorted ascending on *from* and $T_U$ is sorted ascending on *mid*

4 Sorted order of T1 is consequence of TE being sorted on skill first and `from` second

4 Operation `copy_fld_ranges` preserves sort order

```
q fop TU mid "op=[sort]:ordr=[asc]"
q srt_join TU mid "" T1 from x exists
```

5 Copy the users who were endorsed for skill $s$ by some user in $T_U$ into another temp table, $T_2$

5 Users can occur multiple times in $T_2$ if endorsed by $> 1$ user in $T_U$

```
q copy_fld T1 to x T2
q rename T2 to mid
```

6 Create temp table $T_3$ which contains unique values of $mid \in T_2$ and their respective counts

```
q fop T2 mid sortA
q count_vals T2 mid "" T3 mid cnt
```

7 Sort $T_3$ descending on *cnt* and print top $n$

```
q sortf1f2 T3 cnt mid D_
q pr_fld T3 mid:cnt | head $n
```

**Figure 3: Appendix: Code for Problem 4.3**

| | |
|---|---|
| 1 | Let $T_1$ be first degree network of $m$ |
| 2 | Let $T_2$ be all edges between friends of $m$ |
| | `q crossprod T1 mid mid T2 m1 m2 "" "" upper_triangular` |
| 3 | Let $T_3 = T_2 - T_C$ be all such edges that don't already exist |
| | `q f1f2opf3 T2 m1 m2 concat m1m2` |
| | `q srt_join TC fromto "" T2 m1m2 x exists` |
| | `q f1opf2 T2 x 'op=[!]'  notx` |
| | `q copy_fld T2 m1m2 notx T3` |
| 4 | Let $T_5 \subseteq T_{E_1}$ focus attention on endorsements of members in$T_1$ |
| 4 | Recall $T_{E_1}$ is sorted on skill primary and mid secondary |
| 4 | Hence, $T_5$, created by `copy_fld` has same sort property |
| 4 | We create $T_4$ which is unique skills in $T_5$ and their counts |
| | `q fop T1 mid 'op=[sort]:order=[asc]'` |
| | `q srt_join T1 mid "" TE1 mid x exists` |
| | `q copy_fld TE1 skill x T5` |
| | `q copy_fld TE1 mid x T5` |
| | `q sortf1f2 T5 skill mid A_` |
| | `q count_vals T5 skill T4 skill cnt` |
| 5 | $T_6$ has one row for each endorsement that $m_1, m_2$ have in common |
| 5 | This is accomplished with a cross-product with a variation. |
| 5 | It is performed in chunks (one for each skill); chunk size given by `cnt` in $T_4$ |
| | `q crossprod T5 mid mid T6 m1 m2 T4 cnt upper_triangular` |
| 6 | Now we count number of common endorsements |
| | `q f1f2opf3 T6 m1 m2 concat m1m2` |
| | `q fop T6 m1m2 'op=[sort]:order=[asc]'` |
| | `q srt_join T6 m1m2 "" T3 m1m2 num cnt` |
| 7 | Number of common endorsements is measure of strength of connection between $m_1, m_2$ |
| 7 | To find the members with the most in common, we sort |
| | `q sortf1f2 T3 cnt m1m2 D_` |
| 8 | Unpack m1m2 (I8) into m1 (I4) and m2 (I4) |
| | `q f1opf2f3 T3 m1m2 unconcat m1 m2` |

**Figure 4: Appendix: Code for Problem 4.4**