# Small is Beautiful: a Contrarian Approach to Big Data

Ramesh Subramonian
NerdWallet

Indrajeet Singh
NerdWallet

Krushnakant Mardikar
GS Labs

Tara Mirmira
UC Berkeley

Srinath Krishnamurthy
GS Labs

## ABSTRACT

The unspoken secret of "big data" is that while data warehouses are exploding in size, the actual amount of data needed for many analytics and machine learning problems is better characterized as "medium data". We define "medium data" as data large enough that one must be cognizant of performance but small enough that a single machine is adequate for interactive usage. This realization has motivated the design and development of Q, an analytical database. It has its own language with an extensible set of operators. The language is inspired by Ousterhout's dichotomy, using a scripting language (Lua) with a fast interpreter (LuaJIT) as the front-end and a compiled language (C) for the back-end. A fundamental realization that has influenced Q's design is that data practitioners often build up tremendous intuition about the data sets that they are working with. Q encourages the use of this intuition to gain performance at relatively low complexity. Q exposes its software architecture in a disciplined and layered fashion. By this we mean that there is a continuum of optimizations that become possible as one is willing to engage with more of the details of the system. The other central requirement guiding Q's development is ease of development and interactive development, because most data science activities involves trial and error, changing requirements, etc. To support this, we have an extreme focus on performance and have built a Python interface so that users can use their favorite IDEs as well as Jupyter notebooks.

**ACM Reference format:**
Ramesh Subramonian, Indrajeet Singh, Krushnakant Mardikar, Tara Mirmira, and Srinath Krishnamurthy. 2018. Small is Beautiful: a Contrarian Approach to Big Data. In *Proceedings of x, x, x,* 5 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

It is undeniable that there has been an explosive growth in the amount of data available for analysis. Solutions that deal with the reliable storage of large amounts of data have seen tremendous improvements, as also tools to analyze large data sets. What has sometimes been obscured is the understanding of the actual reality of the life of a data scientist or analyst. More often than not, the actual amount of data that needs to be analyzed is in the order of terabytes[Dit15]. The user may spend several months on this sliver of data, augmenting it or eliminating parts of it as insights emerge. Therefore, the mantra of "velocity, volume and variety", while entirely true for data in general, does not translate as readily to the sustained, concentrated effort that goes into most work that we have encountered. The familiarity gained about the nature of the data while working on a given problem for a length of time should not be discounted. The user becomes aware of what approximations can be made, what dynamic range is truly needed, at what point can a value be saturated, etc.

The fact that we are better off scaling up before scaling out has been suggested by several authors [RND+12, Dit15, MIM15]. Fault tolerance on large distributed systems is a difficult problem and adds significant system complexity but is it really needed? If we consider the mean time between failure of a single server and the cost of a "re-do" in the event of failure, it is hard to justify that investment. Further, the emergence of GPUs — both their increased compute power and the increased bandwidth to the GPU — pack the single server solution with considerable punch.

While Q shares many design goals with [PTS+17], there are some key differences. (1) Q's RTS performs data layout and locality optimizations, using hints provided by the user such as marking Vectors as ephemeral or indicating that memo-ization is not needed. (2) Because we control the data layout, we have been able to use [Sak18] to offload computationally intensive activites to the GPU.

In this paper, we present Q, our answer to both technological trends and actual user needs. Q is an analytical database that invites the participation of the user in tuning performance and writing new operators.

The simplicity and power of Q does not come for free. We have abandoned many cherished tenets of databases, like fault tolerance, multi-tenancy, concurrency, etc. While it is hard to argue against these features, our experience leads us to question as to whether they truly assist the work of the data scientist, the business analyst, the machine learning model builder.

*x, x*

DOI: 10.1145/nnnnnnn.nnnnnnn

## 2 Q'S RUN TIME SYSTEM

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

(1) By "analytical" we mean that data changes at the user's behest e.g. loading a data set.
(2) By "almost-relational" we mean that it would more correctly be called a "tabular model" [Cod82]. 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 ...even with these minor flaws, **tables are the most important conceptual representation of relations**, because they are universally understood."
(3) By "single-node", we mean that Q does not distribute computation across machines. The flow of execution is inherently single-threaded. However, OpenMP and CUDA are heavily used *within* individual operations so as to exploit multi-core systems as well as GPUs.
(4) By "column-store", we mean that Q provides the Lua programmer with the Vector type, each individual element of which is a Scalar. A table in Q is a Lua table of Vectors, where a Lua table is an associative array, like a Python dictionary.

### 2.1 Q: a Lua extension/Python module

It is useful to think of Q as a domain specific language, targeted for data manipulation. In contrast with Wevers' work [Wev14] on developing a persistent functional language with a relational database system inside, Q works within the context of Lua, while inspired by functional programming ideas like memo-ization, lazy evaluation, etc. More accurately, Q is a C library, embedded within an interpreted language. We chose to embed within an existing language because (i) we did not have to write a custom compiler (ii) it allowed us to leverage a rich eco-system of libraries, debuggers, web development environments, allowing the programmer to blur the distinction between application logic and database programming.

We chose Lua because it was designed specifically as both an embedding and embedded language [IFC11, IFC18] and it had a wickedly fast interpreter, LuaJIT. In typical workloads, 90% of the time is usefully spent in the computational code; the balance 10% splits roughly 20:80 between the run time system and the Lua glue logic. Among other things, the glue logic supports operator serves as the bridge between the dynamic typing that Q supports and the static typing required by the C code underneath.

Recognizing that Python is the dominant language of data science, we built a Python interface so that Q can be imported as just another module.

Data is stored in memory and, when necessary, is persisted (uncompressed) in binary format to the file system. This allows us to quickly access data by mmapp-ing the file. Like kdb+ [Bor15], one can think of Q as an in-memory database with persistent backing to the file system.

### 2.2 Vectors

A vector is essentially a map $f(i)$ such that given $i, 0 \leq i < n$, it returns a value of a given type. The main types that Q supports are four variants of integers (1, 2, 4, and 8 byte) and two variations of floating point (single and double precision). In addition, it supports bit vectors and constant length strings. There is limited support for variable length strings, which are used primarily as dictionaries.

Note that Q has 6 types of numbers, in contrast with Lua which uses a single type `number`, internally a double precision floating point. This is because data bandwidth plays a significant role in determining performance, as illustrated by Nvidia's introduction of half precision floating point [HW17]. The user is encouraged (but not required) to use the smallest type that supports the actual dynamic range required.

When a vector is created, we need to specify (i) its type (ii) whether it has null values (iii) how it will be populuated. We can either (a) "push" data to it, much like writing to a file in append mode or (b) we can provide a generator function, which is invoked with the chunk index, $i$, as parameter and which knows how to generate the $i^{th}$ chunk.

Vectors are evaluated lazily. Hence, a statement like `x = Q.const({len = 10, qtype = I4, val = 0})` does not actually create ten 4-byte integers with value 0 as one might suspect. Data is populated only when `eval()` is explicity invoked on the vector or the data is implicitly required by some other operator e.g. `Q.print_csv(x)`

Vectors are processed in chunks. Consider an expression like $\sum(a+b\times c)$, written in Q as `d = Q.sum(Q.add(Q.mul(b,c), a))`. When `d` is eval'd, computation alternates between the `mul, add, sum` operators processing chunks of data at a time until there are no more. The chunk size, $n_C$, is chosen large enough that it is amenable to vectorization and parallelization and small enough that its memory consumption is low.

Vectors are not mutable (with few exceptions) and must be produced sequentially. In other words, the $i^{th}$ element must be produced before the $i + 1^{th}$. Vectors operate in "chunks" of a fixed size. Let us say that the chunk size is 64K and that we have produced 65K elements. In that case, the current chunk would have only 1K elements. Whether one can get access to an element in the previous chunk depends on whether the vector has been "memo-ized". The default behavior, with a concomitant performance hit, is to memo-ize. However, when the programmer is aware that the vector

will be consumed in a streaming fashion, they set memo to false.

Memo-izing is done by appending previous chunks in binary format to a file. Subsequent reads of this vector are done by mmap-ing the file. Not all algorithms are readily transformed into streaming operations e.g. sort. There are a few cases where we support modifying a vector after it has been fully materialized by opening it in write mode and mmap-ing it.

Mmap-ing gives us the illusion of a linear address space. This is useful to incoporate algorithms and libraries that have not been written with streaming in mind. For example, we have used this to call functions in the GNU Scientific Library and LAPACK from Q.

Q's run time is an alternate approach to "stream fusion" [MLJ17]. In that paper, the authors identify this as a technique that allows a compiler to " cope with boxed numeric types, handle lazy evaluation, and eliminate intermediate data structures".

## 2.3 Reducers

A Reducer is a construct that takes one or more Vectors as input and produces one or more Scalars as output, similar to "Builders" in [PTS+17].

The simplest Reducers are those for `min`, `max`, `sum` and other associative functions. In this case, the Reducer takes 1 Vector as input and returns 2 Scalars, one being the value of the function being computed and the other being the number of non-null values observed.

An example of a more advanced Reducer is `minK`. Given 2 Vectors, $d$ and $g$, minK returns 2 Lua tables of Scalars, $d'$ and $g'$ such that $d'$ contains the $k$ smallest values of $d$ and $g'$ contains the corresponding values of $g$. See Section 3.1 for a use case.

A few notes about Reducers: (a) The value of a Reducer can be queried even before it has been evaluated completely, a facility that has proven useful while debugging (Section 3.1) (b) Given that Reducers can return a table of Scalars, it is possible to convert a Lua table of scalars, $s1, s2, \ldots$ into a vector by $Q$.pack($\{s1, s2, \ldots\}$). Equivalently, $S = Q$.unpack($x$) creates a Lua table of Scalars from a Vector $x$.

# 3 SAMPLE APPLICATIONS

In this section, we discuss some well known algorithms in the context of Q, both to describe how these problems are addressed in Q and to motivate the features of Q that simplify implementation.

## 3.1 k-Nearest Neighbors

A key subroutine of the `knn` algorithm [HTF09] is finding the class labels of the $k$ points closest to the point $x$ being classified. We assume that we have $n$ points in $m$-dimensional space, represented as a Lua table, $T$, of $m$ Vectors of length $n$

Note that the operations are performed in column order rather than the more intuitive row order. Row ordering means computing the distance from $x$ to a point in $T$ completely before proceeding to the next point. We invite the reader to judge from Figure 1 the merits of our claim that the programming burden of being aware of data layout and operation ordering is not onerous.

Given Q's lazy evaluation in chunks, note that the operation is actually performed in batches. In other words, had we written `d1, g1 = Q.mink(d, g, k)` and then done `g1:next()` instead of `g1:eval()`, we would have gotten the class labels of the $k$ points closest to $x$ from the **first** $n_C$ points of $T$.

```
local function knn(
  T, -- table of m Vectors of length n
  g, -- Vector of length n
  x, -- Lua table of m Scalars
  k -- number of neighbors we care about
  )
  local n = g:length()
  local m = #T
  local d = Q.const({val = 0, qtype = "F4", len = n})
  for i = 1, m do -- for each dimension
    d = Q.vvadd(d, Q.sqr(Q.vssub(T[i], x[i])))
  end
  return Q.mink_reducer(d, g, k)
end
return knn
```

**Figure 1: core of k-nn algorithm in Lua**

## 3.2 Logistic Regression

A key subroutine in logistic regression [HTF09] is improving the estimates of the coefficients, $\beta$, using the Newton-Raphson algorithm. In the course of this, we need to compute $\frac{1}{1+e^{-x}}$ and $\frac{1}{(1+e^{-x})^2}$. We use this to motivate the "conjoin" feature of Q. Performing common sub-expression elimination as in Figure 2 yields a 2x speedup.

*3.2.1 Lock step evaluation.* However, the code in Figure 2 has a subtle but critical bug when the number of elements in $x$ exceeds the chunk size. If we were to call `eval()` on $y$, then we would end up consuming sucessive chunks of $t_3$. Now, if we were to call `eval()` on $z$, we would fail when requesting the first chunk of $t_3$, since it has **not** been memo-ized. One solution is to ensure that $y$ and $z$ are evaluated in lock-step, after they have been created, as in Figure 3.

```
t1 = Q.vsmul(x, Scalar.new(-1, fldtype)):memo(false)
t2 = Q.exp(t1):memo(false)
t3 = Q.incr(t2):memo(false)
t4 = Q.sqr(t3):memo(false)
y  = Q.reciprocal(t3)
z  = Q.reciprocal(t4)
```

**Figure 2: Common sub-expression elimination**

```
cidx = 0  -- chunk index
repeat
  ly = y:chunk(cidx)
  lz = z:chunk(cidx)
  cidx = cidx + 1
until ( ly == 0 )
```

**Figure 3: Lock step evaluation**

```
-- indicate no memo-izing needed
y:set_memo(false); z:set_memo(false);
-- indicate that y,z need to be evaluated together
conjoin({y, z})
-- z not fully evaluated
assert(z:is_eov() == false)
-- evaluate y when needed
y:eval()
-- z fully evaluated as a consequence of y eval
assert(z:is_eov() == true)
```

**Figure 4: Conjoined Vectors**

The problem with this solution is that the burden of lock-step evaluation falls on the Q programmer. This is remedied by the `conjoin` function (Figure 4)

# 4 POWER TO THE PEOPLE

It is well accepted that large data systems benefit significantly from careful optimization of data movement. Optimizing compilers and query plan rewriters aim to do this automatically. Q takes a fundamentally different approach. It is our belief that the choreography of computations can be left to the database programmer **if** they have (i) some understanding of the underyling system architecture and (ii) relatively simple knobs to influence the run time system.

In many analytical tasks, one repeatedly performs very similar operations on slowly changing data. In these cases, it is not onerous to maintain (and periodically refresh) summary statistics that can significantly speed up more complex operations. For example, sort is trivially parallelized if one has a rough distribution of the keys. Note that the fidelity demanded of these summary statistics is often low — they need to be only as good as the use to which they are put.

## 4.1 Building indexes — she social graph

One of the problems we investigated at LinkedIn was using the social graph to guide and power the design of data-driven products. Assume that a social graph is presented as a a table, `E`, with 2 columns, `from` and `to`. Q supports the creation of auxiliary data structures to enable fast access but, in keeping with our minimalist philosophy, does not prvoide them out of the box. We describe the pre-processing performed to provide efficient access.

(1) `f, t = Q.sort2(E.from, E.to, "ascending")`, sorts `E` with `from` as primary key and `to` as secondary key
(2) `V.m = Q.unique(f)` creates a Vector of member IDs, sorted ascending.
(3) For each member in `V.m`, we determine the contiguous edges out of it by
    (a) `V.lb = Q.join(f, V.m, min_index)`
    (b) `V.ub = Q.join(f, V.m, max_index)`
    Now, we can assert that member `m = V.m[i]` is connected to members `t[V.lb[i] .. V.ub[i]]`

# 5 CONCLUSION

For some time now, data analysis has been straining to slip the surly bonds of relational databases and SQL, as evidenced by the proliferation of systems inspired by [DG04]. Nonetheless, as Santayana postulated and [PPR+09] demonstrated, those who don't remember the past are condemned to repeat it. Despite this, we believe that the development of Q, is **not** an extraordinary act of hubris. Instead, it is a judicious application of well-understood techniques to a limited but meaningful domain, encompassing the day to day activities of the data scientist e.g., ingestion, cleaning, validation, preparation, feature engineering, model building, exploration, ...

The development of Q has been motivated by understanding, as data science practitioners, of what is truly needed to be effective. We have found that engaging the user at the right level of abstraction, rather than shielding them from system internals, makes for simple, performant code. It has allowed for the rapid deployment of highly performant analytical capabilities that service the needs of the business in a timely manner. At the same time, Q's parsimonious use of computing resources shows that the pursuit of a profit motive is not inconsistent with a recognition that computing uses material and energy resources [Sch99, NTP+18] and that there are "planetary boundaries that must not be transgressed" [Roc09].

## REFERENCES


[Bor15] Jeffrey Borror. *Q for Mortals Version 3: An introduction to Q programming*. q4m LLC, 3rd. edition, 2015.

[Cod82] E. F. Codd. Relational database: a practical foundation for productivity. *Commun. ACM*, 25(2), February 1982.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *6th Symposium on Operating Systems Design and Implementation*, San Francisco, December 2004.

[Dit15] Jens Dittrich. The case for small data management. In *7th Biennial Conference on Innovative Data Systems Research*, Asilomar CA, January 2015. ACM.

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *Elements of Statistical Learning*. Springer Series in Statistics. Springer, 2nd. edition, 2009.

[HW17] Nhut-Minh Ho and Weng-Fai Wang. Exploiting half precision arithmetic in nvidia GPUs. In *IEEE High Performance Extreme Computing Conference*, Waltham, MA, September 2017. IEEE.

[IFC11] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes. Passing a language through the eye of a needle. *Commun. ACM*, 54(7), July 2011.

[IFC18] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes. A look at the design of lua. *Commun. ACM*, 61(11), November 2018.

[MIM15] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems*, Kartause Ittingen, May 2015. USENIX.

[MLJ17] Geoffrey Mainland, Roman Leshchinskiy, and Simon Peyton Jones. Exploiting vector instructions with generalized stream fusion. *Commun. ACM*, 60(5):83–91, May 2017.

[NTP+18] Bonnie Nardi, Bill Tomlinson, Donald Patterson, Jay Chen, Daniel Pargman, Barath Raghavan, and Birgit Penzenstadler. Computing within limits. *Commun. ACM*, 61(10):86–93, October 2018.

[PPR+09] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, Providence, Rhode Island, June 2009.

[PTS+17] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, and Matei Zaharia. Weld: A common runtime for high performance data analytics. In *8th Biennial Conference on Innovative Data Systems Research*, Chaminade CA, January 2017. ACM.

[RND+12] Antony Rowstron, Dushyanth Narayanan, Austin Donnelly, Greg O'Shea, and Andrew Douglas. Nobody ever got fired for using hadoop on a cluster. In *1st International Workshop on Hot Topics in Cloud Data Processing*, Bern, Switzerland, April 2012. ACM.

[Roc09] J Rockstrom. A safe operating space for humanity. *Nature*, 461:472–475, 2009.

[Sak18] Nikolay Sakharnykh. Everything you need to know about unified memory, March 2018.

[Sch99] Ernst F. Schumacher. *Small is Beautiful — Economics as if People Mattered*. Hartley and Marks, 1999.

[Wev14] Lesley Wevers. Persistent functional languages: Toward functional relational databases. In *Sigmod PhD Symposium*, Snowbird, UT, June 2014.