AQuery: A Query Language for Order in Data Analytics: Language, Optimization, and Experiments

José Pablo Cambronero and Dennis Shasha

Courant Institute/New York University

September 20, 2016

Introduction

- Success of the relational model results from happy combination of expressive power and simplicity
- Single data type + few operations (select/project/join/aggregate) $\rightarrow$ simplicity
- Programmers of applications that depend on ordered events face a dilemma
- They would like to use a relational database system, but the model makes it hard to express queries over order.
- AQuery (and others) embodies philosophy that order can be introduced without affecting simplicity (and improving performance)[24][14][8]

AQuery: Sales Query

Please return the running three month moving average of sales.

```
SELECT month, avgs(sales, 3)
FROM Revenue
ASSUMING ASC month
```

AQuery: Sales Query

Please return the running three month moving average of sales.

```
SELECT month, avgs(sales, 3)
FROM Revenue
ASSUMING ASC month
```

The assuming clause creates an arrable ordered by month and the running average query avgs performs the calculation.
That's (most of) AQuery!

AQuery

- Modest syntactic and semantic extension to SQL 92
- Replaces unordered relational tables by ordered tables (*arrables* which stands for array-tables), which can be sorted by one or more columns[10]
- Modest syntactic and semantic extension to SQL 92: (i) Adds one clause: assuming clause (order) (ii) Provides order-senstive aggregates (iii) Go into and out of first normal form.

SQL 92: Sales Query – inefficient AND incorrect

Please return the running three month moving average of sales.

```
SELECT t1.month, t1.sales,
(t1.sales+t2.sales+t3.sales)/3
FROM Revenue t1, Revenue t2, Revenue t3
WHERE t1.month - 1 = t2.month and
t1.month - 2 = t3.month
```

Three-way join (inefficient) and misses the first two months. Can be written correctly in SQL 99 but complex and inefficient.

AQuery: Moving Variance Query

Assume a table of the form *prices*(*ID*, *Date*, *EndOfDayPrice*) with the last ten years' data. Calculate a 12-day moving variance in returns for stock tickers Leverages: assuming clause, order-dependent aggregate (vars over 12 previous value, ratios based on consecutive days). Gives for each ID, a vector of Dates and variances.

```
    SELECT ID, DATE,
    vars(12, ratios(1, EndOfDayPrice) - 1)
    FROM prices
    ASSUMING ASC Date
    GROUP BY ID
```

SQL-99: Moving Variance Query

Assume a table of the form *prices*(*ID*, *Date*, *EndOfDayPrice*), calculate a 12-day moving average in returns for stock tickers

```
SELECT ID, Date,
  VARIANCE(rets) OVER (
    ORDER BY Date ROWS
    BETWEEN 11 PRECEDING AND CURRENT ROW
    ) as mv
  FROM
  (SELECT
    curr.Date, curr.ID,
    curr.EndOfDayPrice /
    prev.EndOfDayPrice − 1 as rets
    FROM
    prices curr LEFT JOIN prices prev
    ON curr.ID = prev.ID
    AND curr.Date = prev.Date + 1)
    GROUP BY ID
```

Optimizations for both sequential and parallel implementations

- Rule-based optimization for predictability
- Tranformation rules yield demonstratable advantages
- Rules implemented as rewrites on abstract syntax tree.

Sort minimization [new, but clear]

- Detect order-dependent vs order-independent operations
- Sort only columns upon which operations are order-dependent.
- $od(t)$ returns all columns affected by order-dependence, and necessary to maintain semantics

```
SELECT ... FROM t ASSUMING S ....
```

$$sort_S(t)$$
$$\rightarrow$$
$$sort_S(od(t)), (columns(t) \setminus od(t))$$

Push selections [classical]

- Generally perform selections before sorting and joins
- Except when doing so loses the benefits of indexes.

$$t' \leftarrow \sigma_W(sort_S(t))$$
$$\rightarrow$$
$$t' \leftarrow \sigma_{W''}(sort_S(\sigma_{W'}(t)))$$

where W' includes all selections up to first use of an order-dependent aggregate, and W'' contains remaining selections.

AQuery: Sales Query (again)

Please return the running three month moving average of sales.

```
SELECT month, avgs(sales, 3)
FROM Revenue
ASSUMING ASC month
```

The assuming clause creates an arrable ordered by month and the running average query avgs performs the calculation. Sort only month and sales by month.

Sequential Implementation

- Pure Scala implementation of compiler
- Execution engine: q[28]
- Workflow: write AQuery code, compiler generates optimized q code, execute using q interpreter
- Advantages: portability, transparency (user able to inspect generated q code)

Related Work

- Among the excellent work in the development of time series databases, much has focused on developing architectures that allow for scalability and performance for simple queries, rather than developing a performant language supporting complex queries
- DruidIO[30]: open source data store for analytics. Column oriented, but query language doesn't suport common functionality like joins
- Influxdb[6]: Limited query language, no user-defined functions, no arbitrary sorting
- SciQL[8]: extends MonetDB[13] with first-class arrays for scientific applications, allowing direct manipulation of array and matrix structures. Comparable in expressability to AQuery, but AQuery is designed to be a natural extension of sql (and is faster).
- Excellent work but focused on reliability and scalability[18][26], not query plans

Benchmarks

- Compare: AQuery, Python's Pandas[17], Sybase IQ[21], and MonetDB (with imbedded Python)[19]
- Experiments: financial benchmark from Sybase[20], MonetDB's benchmarking operation of quantile calculation, various Pandas benchmarking operations from Panda's historical performance benchmark[27]
- *We compare on our competitors' benchmarks!*

Experimental Setup

Experiments against Pandas and MonetDB are run in a single-user setting on a MacBook Air with a 2-Core 1 .7 GHz Intel Core i7 processor, with 8GB of memory. The Sybase IQ comparison is performed on a multi-user linux system with 4 16-Core 2.1 GHz AMD Opteron 6272 processors, with 256GB of memory.

- Pandas version 0.17.0
- Numpy version 1.10.1
- Python version 2.7.5
- MonetDB version 1.7, built from the *pyapi* branch that allows for embedded Python
- Sybase IQ version 16.0
- q version 3.2 2014.11.01
- AQuery compiler a2q version 1.0

Finance Benchmark

- Common financial operations (e.g. adjust prices for stock events, find crossing points of moving averages, summarize prices across different time horizons, test trading strategies)
- Simulated data, randomized as necessary (various parameter values) data at different sizes (100K, 1M, and 10M observations)
- Present average response time
- Data and sequential system soon available.

Finance Benchmark: Sybase IQ Results

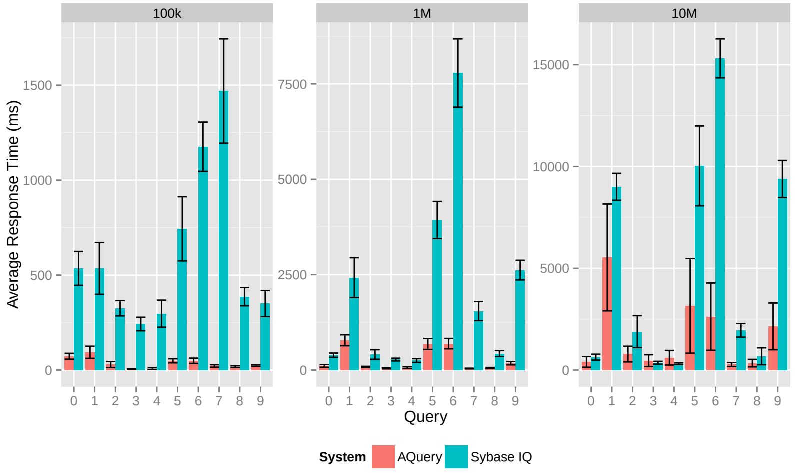
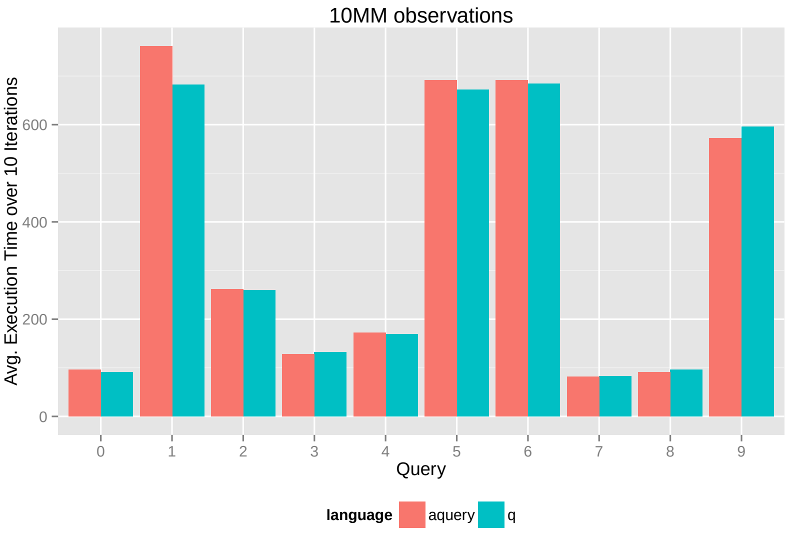


Figure 1: With 100K and 1M rows, AQuery outperforms Sybase IQ in all of the queries evaluated. At 10M rows, performance is a bit more varied, with larger standard errors, but on average AQuery is faster in 8 of the 10 benchmark queries.

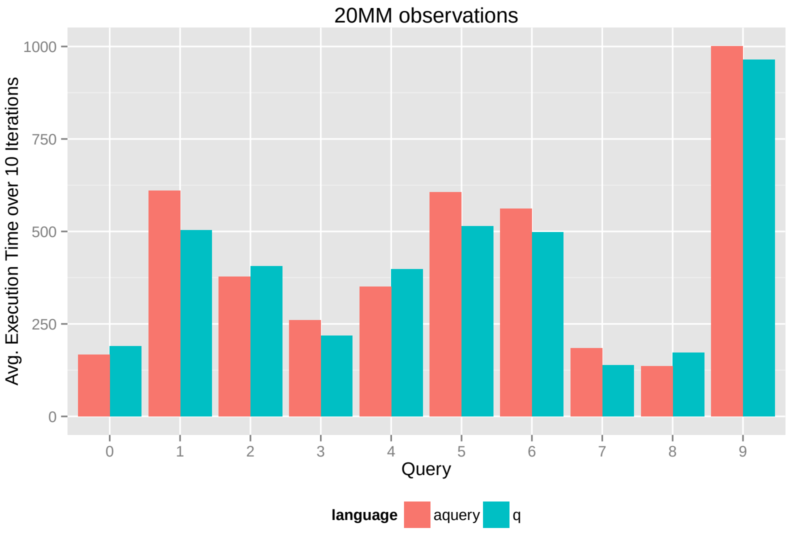
How does AQuery stack up against q?: Finance Benchmark

- Performance on most queries is comparable
- There is some overhead in managing certain simple aquery data structures
- Current joins available: equi-join and full outer join. Increasing expressiveness of joins would erase most of remaining gap
    - Gap is most evident in queries 1, 5, 6, which use `lj` in the q version

How does it stack up against q?: Finance Benchmark



How does it stack up against q?: Finance Benchmark



Decomposing our query

- Of course, anything AQuery writes, you can write
- But that doesn't mean it won't require keeping track of lots of details, or that reasoning on the fly will give correct and efficient results. AQuery does that for you, e.g.,
  - When function is not order-dependent, push selections below sort
  - Sort only relevant columns
  - If already sorted, don't resort.

Parallel AQuery: newest work

- Simple architecture, allows deeper reasoning for query generation/transformation
- Novelty: Explores order-based optimizations in a distributed setting

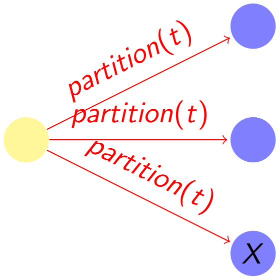
Parallel Primitives

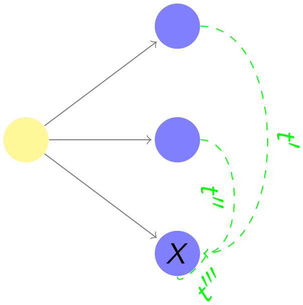
- Encapsulate all parallelism, allowing compositional reasoning
    - Shuffle
    - Map (-Reduce)
    - Carry-lookahead
    - Edge-extension

**Note on diagrams in following slides: red/solid lines represent instructions sent across nodes, while green/dashed lines represent data sent across nodes*

Map [classical]

- Predicate based partitioning of say table t – like the map in the classic map-reduce.





Staged Reduce [classical]

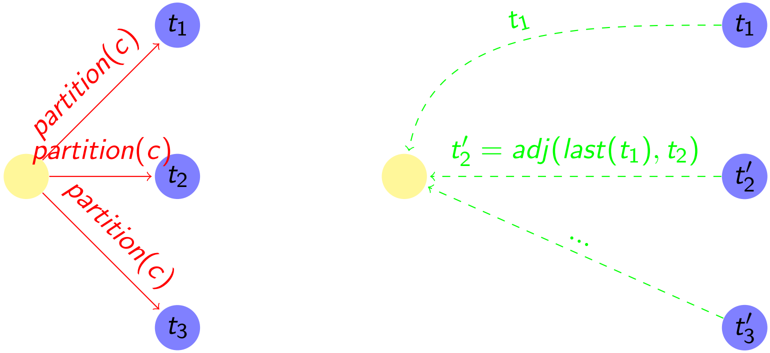
- Each worker does its own reduction.
- Optionally, stage reduced results into smaller and smaller summaries (e.g. for a global sum)

Carry-Lookahead Calculations [new]

- Some operations lend themselves to parallelizing intermediate results followed by adjustments
- Example: Running (i.e., cumulative) sum of stock volumes entails partitioning into separate chunks of time, performing running sum in each chunk and then adding the intermediate results. Like a carry-lookahead adder.
- Effectively, a map-reduce operation with: order-dependent scan + adjustment function as a reduction operation

Carry-Lookahead Calculations

- *partition*(c): initial partition on column c
- *adj*(x, y): adjusts y by combining with x

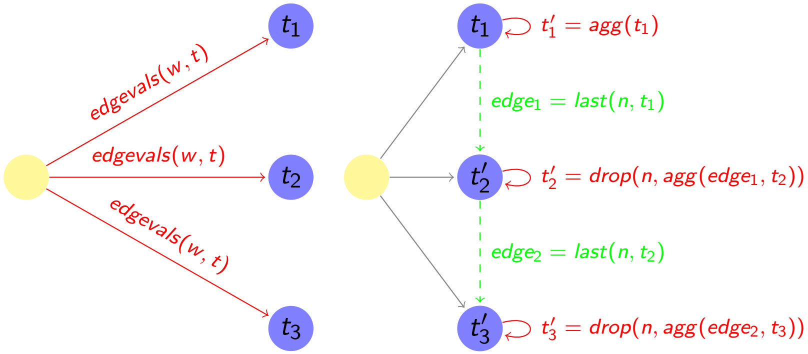


Edge-Extension

- Window-based operations abound in order-dependent data analysis
- Example: 7-day moving average of stock prices
- Dependencies across worker processes
- Solution: extend partitioned data with necessary replicated data (maintaining order of tuples)
- Allows parallelized window-based computation

Edge-Extension

- $drop(x, y)$: drop first x tuples of y
- $last(x, y)$: last x tuples of y
- Results can be kept in worker processes, or sent back to master (yellow) if these are final results



Implementation

- Developed open-source library implementing primitives: parallel.q
- Composes primitives to yield: distributed sorting, distributed grouping, distributed crossing, distributed reference joins, in addition to standard selections/projections/etc
- Standalone library allows users to write distributed queries in an intuitive fashion
- Parallel AQuery translates standard queries into calls to parallel.q, modularizing distributed logic
- Prior optimizations still apply (as rewritten abstract syntax tree)

Exploring performance in parallel.q

- Setup: 30 million float point numbers in-memory across 3 worker processes
- Experiments: Compare parallel.q performance versus serial q. Serial q collects data from workers and computes centrally, meanwhile parallel.q allows expressing the same in-memory operations over the distributed dataset
- End Goal: AQuery compiler should translate the same simple query into parallel.q formulation

- Experiment 1: Last value in running average (carry-operation)
- Experiment 2: Max value in 10-element moving average (edge-extension)

Experiment 1: last of running average

Target AQuery (note that this translation has not yet been implemented, and parallel.q has been written manually)

```
SELECT last(avgs(vals)) FROM nums
```

Experiment 2: Max of moving averages

Target AQuery (note that this translation has not yet been implemented, and parallel.q has been written manually)

```
SELECT max(avgs(10, vals)) FROM nums
```

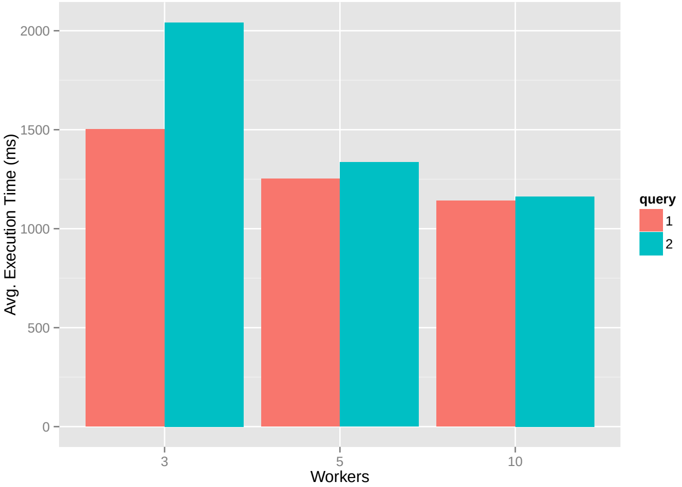
Performance Overview

Table 1: parallel.q allows users to take advantage of parallelism for in-memory operations that otherwise require collecting (average execution time ms)

Experiment	parallel.q	standard q
1	1016.5	1213.4
2	1574.9	1876.6

Performance Overview

We evaluate parallel.q scalability by testing with 3, 5 and 10 worker processes, on a machine with 12 cores. The combined workers contain a total of 100MM floating point numbers in-memory



Conclusions

- AQuery is a linguistically simple high performance database system for time series and other ordered data.
- The concept of arrables and assuming and moving averages constitute the backbone of the system
- Some new optimization problems can be handled with simple powerful primitives.
- Here is a demo of the sequential version:

Future Work

- Improve parallel system performance.
- Implement translation for parallel version
- Incorporate time series machine learning primitives.

References I

[]

Stephen Cass.
The 2015 Top Ten Programming Languages, 2015 (accessed November 7, 2015).

Edgar F Codd.
A relational model of data for large shared data banks.
Communications of the ACM, 13(6):377–387, 1970.

Ramez Elmasri and Shamkant B Navathe.
Fundamentals of database systems.
Pearson, 2014.

Influxdb.
Influxdb Github Issues: Add aggregate function top #409, 2013-2015 (accessed November 7, 2015).

References II

Influxdb.
Influxdb Github Issues: Add support for custom functions #68, 2013-2015 (accessed November 7, 2015).

Influxdb.
InfluxDB: Overview, 2015 (accessed November 6, 2015).

Brian W Kernighan, Dennis M Ritchie, and Per Ejeklint.
The C programming language, volume 2.
prentice-Hall Englewood Cliffs, 1988.

M Kersten, Ying Zhang, Milena Ivanova, and Niels Nes.
Sciql, a query language for science applications.
In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 1–12. ACM, 2011.

References III

Alberto Lerner.
Querying Ordered Databases with Aquery.
PhD thesis, Ph.D. Thesis, Ecole Nationale Superieure de Telecommunications, ENST-Paris, 2003.

Alberto Lerner and Dennis Shasha.
Aquery: Query language for ordered data, optimization techniques, and experiments.
In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 345–356. VLDB Endowment, 2003.

John Levine.
Flex & Bison: Text Processing Tools.
" O'Reilly Media, Inc.", 2009.

References IV

MonetDB.
Embedded R in MonetDB, 2014 (accessed November 18, 2015).

Stratos Idreos Fabian Groffen Niels Nes and Stefan Manegold Sjoerd Mullender Martin Kersten.
Monetdb: Two decades of research in column-oriented database architectures.
Data Engineering, page 40, 2012.

Wilfred Ng.
An extension of the relational data model to incorporate ordered domains.
ACM Transactions on Database Systems (TODS), 26(3):344–383, 2001.

References V

Travis E Oliphant.
A guide to NumPy, volume 1.
Trelgol Publishing USA, 2006.

Oracle.
MySQL: Handling of GROUP BY, 2015 (accessed November 15, 2015).

pandas development team.
pandas: powerful python data analysis toolkit (version 0.17.0), 2015 (accessed November 7, 2015).

Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan.
Gorilla: a fast, scalable, in-memory time series database.
Proceedings of the VLDB Endowment, 8(12):1816–1827, 2015.

References VI

Mark Raasveldt.
Embedded Python/NumPy in MonetDB.
MonetDB, 2015 (accessed November 06, 2015).

SAP.
Sybase IQ 15.3: Understanding User-Defined Functions, 2008 (accessed November 8, 2015).

SAP.
Introduction to SAP Sybase IQ: SAP Sybase IQ 16.0, 2013 (accessed November 8, 2015).

SAP.
SAP Sybase IQ 16 In-Database Analytics Option Technical Overview, 2013 (accessed November 8, 2015).

SAP.
Sybase RAP, 2015 (accessed November 8, 2015).

References VII

Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan.
SEQ: Design and implementation of a sequence database system.
Citeseer, 1996.

StackOverflow.
pandas tag info, 2015 (accessed November 7, 2015).

StumpleUpon.
FAQ, 2015 (accessed November 6, 2015).

the pandas development team.
Vbench performance benchmarks for pandas, 2011 (accessed November 18, 2015).

Arthur Whitney.
Abridged Q Language Manual, 2009 (accessed November 6, 2015).

References VIII

Hadley Wickham.
ggplot2: elegant graphics for data analysis.
Springer New York, 2009.

Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli.
Druid: a real-time analytical data store.
In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 157–168. ACM, 2014.

AQuery: Correlation Pairs (for self-study)

```
WITH
  stocksGrouped(ID, Ret) AS (
    SELECT ID,
     ratios(1, EndOfDayPrice) - 1
    FROM prices
    ASSUMING ASC ID, ASC Date
    WHERE Date >= max(Date) - 31 * 6
    GROUP BY ID)

  pairsGrouped(ID1, ID2, R1, R2) AS (
    SELECT st1.ID, st2.ID,
     st1.Ret, st2.Ret
    FROM
     stocksGrouped st1, stocksGrouped st2)

SELECT ID1, ID2,
 cor(R1, R2) as coef
FROM FLATTEN(pairsGrouped)
WHERE ID1 != ID2
GROUP BY ID1, ID2
```

Finance Benchmark: Pandas Results

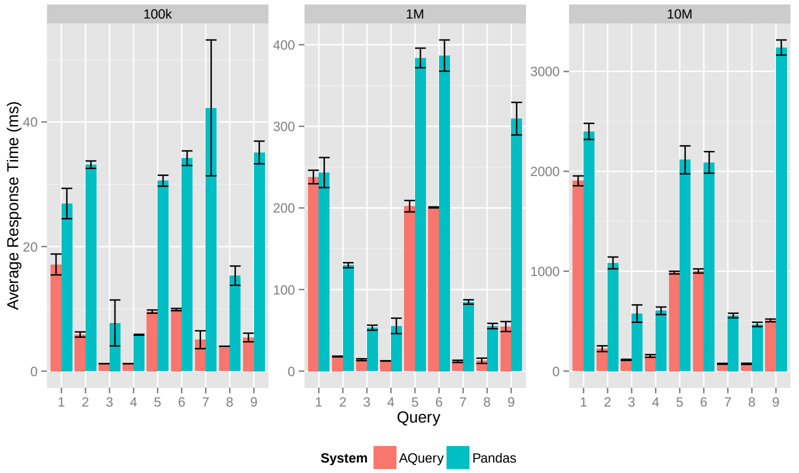


Figure 2: AQuery is faster with stock history of 100K, 1M and 10M rows across all queries. In various of these, AQuery's average response time is orders of magnitude shorter.

Finance Benchmark: Pandas Results

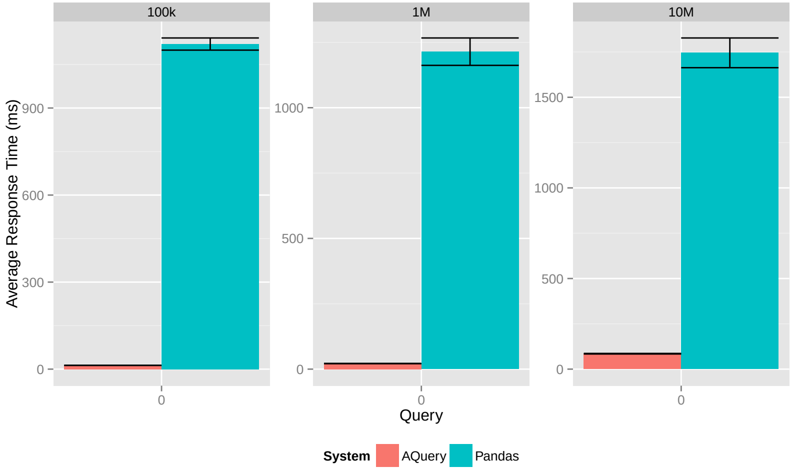


Figure 3: AQuery is faster with stock history of 100K, 1M and 10M rows across all queries. In various of these, AQuery's average response time is orders of magnitude shorter.

Finance Benchmark: MonetDB Results

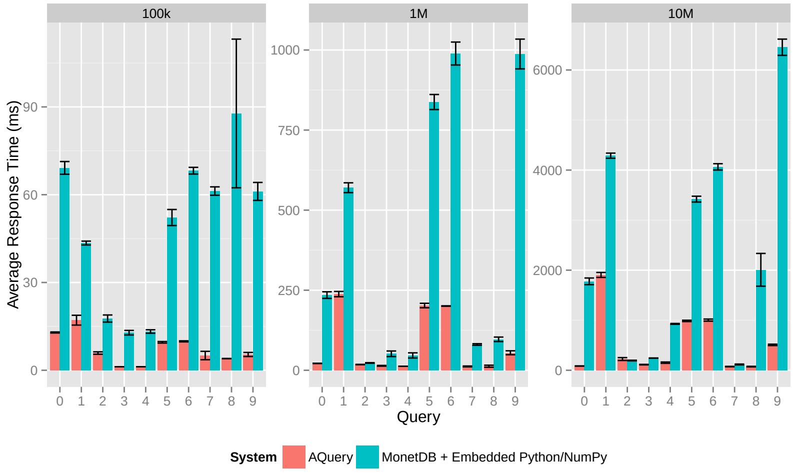


Figure 4: AQuery is faster across the board for 100K rows of stock history. When we increment to 1M AQuery remains faster in 8 of 10 queries, and comparable in the remaining 2. At 10M rows, AQuery is slightly slower for query 2, comparable for query 7, and faster in all others.

Pandas Benchmark: Data Science Operations

- Picked a subset of operations used by Pandas to track library's historical performance evolution[27]
- Represents common tasks in data science, for example: subsetting, grouping, summarizing, and merging data, amongst others.
- Various baseline data sizes: 100K elements (as used in Panda's benchmarking), 1M, and 10M elements
- Randomly generate data and repeat experiments

Pandas Benchmark: AQuery Results

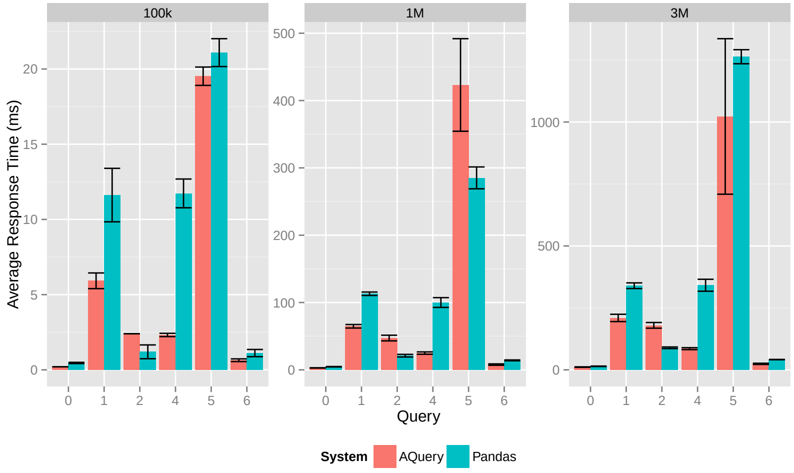


Figure 5: For 100K rows, AQuery is on average faster in 6 of 7 cases. For 1M and 3M rows, AQuery is faster in 5 of the 7 operations evaluated.

Pandas Benchmark: AQuery Results

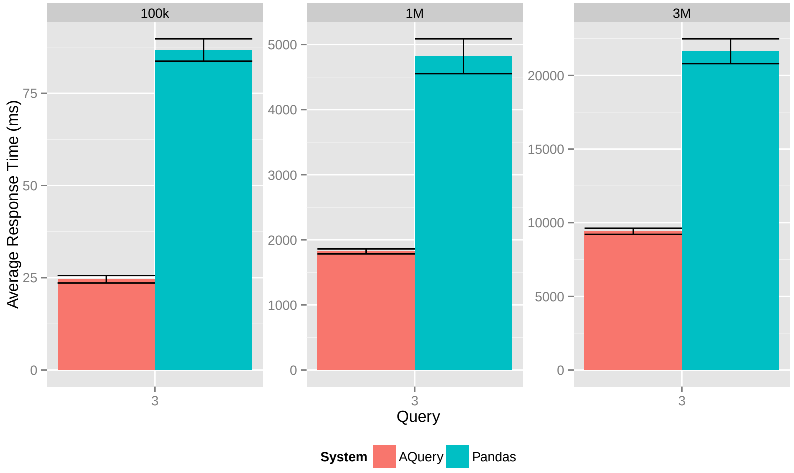


Figure 6: For 100K rows, AQuery is on average faster in 6 of 7 cases. For 1M and 3M rows, AQuery is faster in 5 of the 7 operations evaluated. The first set of graphs excludes query 3, for ease of reading, given the vastly different response time.

MonetDB Benchmark: Quantiles

- MonetDB's ability to embed R[12], and more recently, Python/NumPy [19], directly into a query makes it a very flexible and appealing approach for data scientists and developers looking to integrate their data storage/query and analysis tools.
- AQuery's performance in quantile calculation compared to MonetDB's performance using a performant NumPy function. On the AQuery side, we implement a naive quantile function
- 100K, 1M, 10M, and 25M values
- Repeatedly generate random data sets

MonetDB Benchmark: AQuery Results

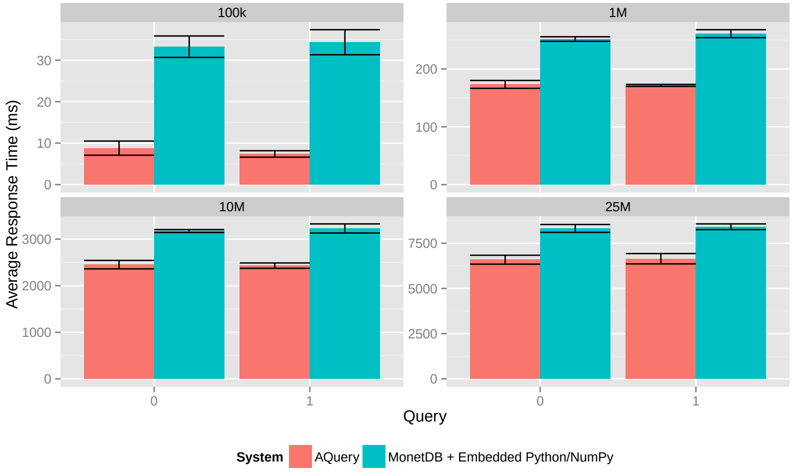


Figure 7: AQuery outperforms in all the dataset sizes evaluated. While the advantage narrows with larger data, we highlight AQuery's implementation is currently using a naive quantile calculation that involves sorting the entire array.

A simple example

We explore a simple example, transformations, and resulting code.

```
<q>
\s 10
n:`int$5e6;
t:([]c1:n?100; c2:n?100; c3:n?100;
    c4:n?100; c5:n?100; c6:n?100);
t:update c2:`g#asc c2 from t
</q>

// identity
function f(x){x}

select
sums(c3), max(c4)
from t
assuming asc c1, desc c2
where f(c1)>=50 and c2 > 50
```

A simple example: execution time

We consider various q implementations

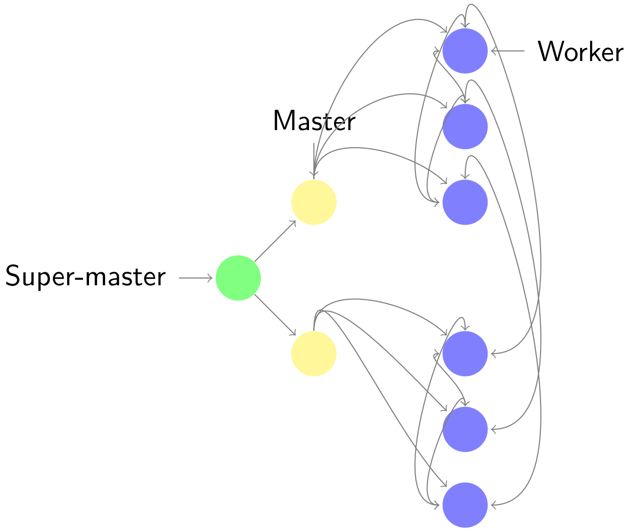
```
// "declarative"
.kdb.q0:{select sums c3, max c4 from `c1 xasc `c2
    xdesc t where 50<=f c1, c2>50}
// select before sort
.kdb.q1:{select sums c3, max c4 from `c1 xasc `c2
    xdesc select from t where 50<=f c1, c2>50}
// reorder selections
.kdb.q2:{select sums c3, max c4 from `c1 xasc `c2
    xdesc select from t where c2>50, 50<=f c1}
```

```
q)\ts:10 .aq.q0[]
1961 150996080
q)\ts:10 .kdb.q0[]
10935 872416128
q)\ts:10 .kdb.q1[]
3558 218104736
q)\ts:10 .kdb.q2[]
3255 218104736
```

Parallel AQuery: Architecture

- Supermaster-master-worker architecture
- Supermaster: Communicates with user and assigns queries provided by user to masters (each associated with one cohort of workers)
- Each cohort has the same data as each other cohort.
- Reads go to one cohort and writes to all.

Parallel AQuery: Sample Architecture


Worker
Master
Super-master