AQuery: A Query Language for Order in Data Analytics

New York University

February 14, 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) $\rightarrow$ simple algebra
- But ... 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.
- We (and others) contend that order can be introduced without affecting simplicity (and improving performance)[14][8][3]

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[18]: open source data store for analytics. Column oriented. Query language doesn't suport common functionality like joins
- Influxdb[1]: Efficient. No user-defined functions limited options for sorting.
- SciQL[3]: extends MonetDB[7] with first-class arrays for scientific applications. Expressive
- Focuses on reliability and scalability[10][15], simple query plans

AQuery

- introduced in [4]
- modest syntactic and semantic extension to SQL 92
- supports ordered tables, called *arrables* (for array tables): can be sorted by one or more columns
- Adds one clause: assuming clause (order)
- Provides order-sensitive aggregates, and incorporates their use into optimization strategies

AQuery: A Network Query

Assume table of the form *network*(*basestation*, *numconns*, *hourstamp*, *date*, ...). The user declares that the arrable should be sorted by date and hour stamp, selects data relevant to a particular base station and then calculates a moving average with window size 24.

```
SELECT basestation, avgs(numconns, 24)
FROM network
ASSUMING ASC date, ASC hourstamp
GROUP BY basestation
```

SQL-99: same network query

Assume table of the form *network*(*basestation*, *numconns*, *hourstamp*, *date*, ...). All the ordering happens in the last step. There may be missing opportunities for optimization.

```
SELECT ID, Date,
  AVG(numcons) OVER (
   ORDER BY date, hourstamp ROWS
    BETWEEN 23 PRECEDING AND CURRENT ROW
   ) as nc
  FROM network
  GROUP BY basestation
```

AQuery: Moving Variance Query

Assume a table of the form *prices*(*ID*, *Date*, *EndOfDayPrice*), calculate a 12-day moving average in returns for stock tickers AQuery uses assuming clause, order-dependent aggregate (vars, ratios), nested arrables

```
WITH
  variances(Date, ID, mv) AS (
    SELECT Date, ID,
     vars(12, ratios(1, EndOfDayPrice) - 1)
    FROM prices
    ASSUMING ASC Date
    GROUP BY ID
  )
SELECT * FROM FLATTEN(variances)
```

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
```

AQuery: Correlation Pairs

```
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
```

Optimizations

- Heuristic (currently rule-based)
- Eliminate unnecessary sorts (minimize sorts to relevant columns)
- Perform selections before sorts (exceptions apply with indices), while maintaining semantics
- Foreign-key joins replaced by pointer-based accesses
- Cross products + selection predicates $\rightarrow$ join
- More to come!

Implementation

- Standard compiler tools: C[2] + flex + bison[5]
- Execution engine: q[17]
- Workflow: write AQuery code, compiler generates optimized q code, execute using q interpreter
- Advantages: portability, transparency (user able to inspect generated q code)

Benchmarks

- Compare: AQuery, Python's Pandas[9], Sybase IQ[13], and MonetDB (with imbedded Python)[11]
- Experiments: financial benchmark from Sybase[12], MonetDB's benchmarking operation of quantile calculation, various Pandas benchmarking operations from Panda's historical performance benchmark[16]

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)
- Simulated data, randomized as necessary (various parameter values)
- data at different sizes (100K, 1M, and 10M observations)
- Present average response time

Finance Benchmark: Pandas Results

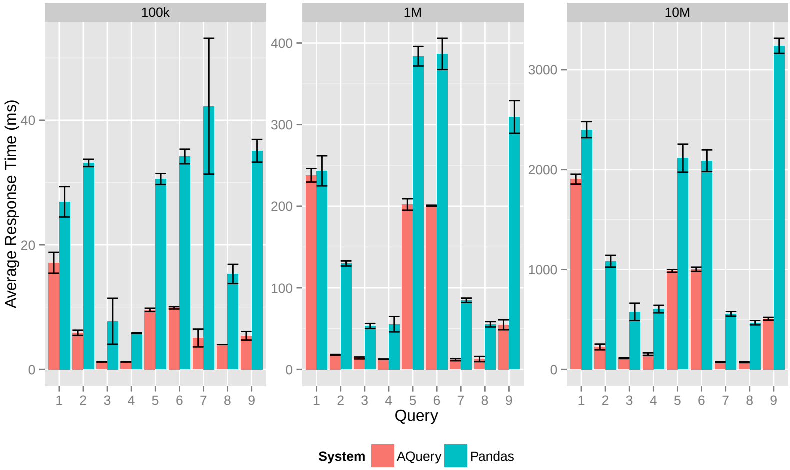

Figure 1: AQuery is faster with stock history of 100K, 1M and 10M rows across all queries. In various of these, AQuery's average response time is orders of magnitude shorter.

Finance Benchmark: Pandas Results

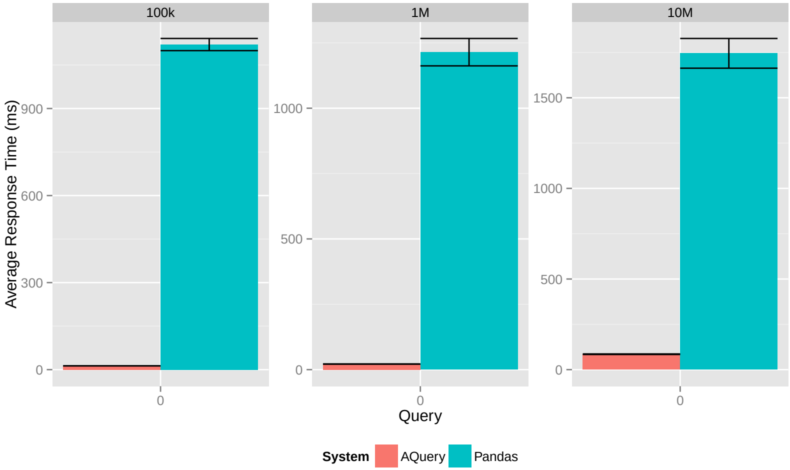

Figure 2: AQuery is faster with stock history of 100K, 1M and 10M rows across all queries. In various of these, AQuery's average response time is orders of magnitude shorter.

Finance Benchmark: MonetDB Results

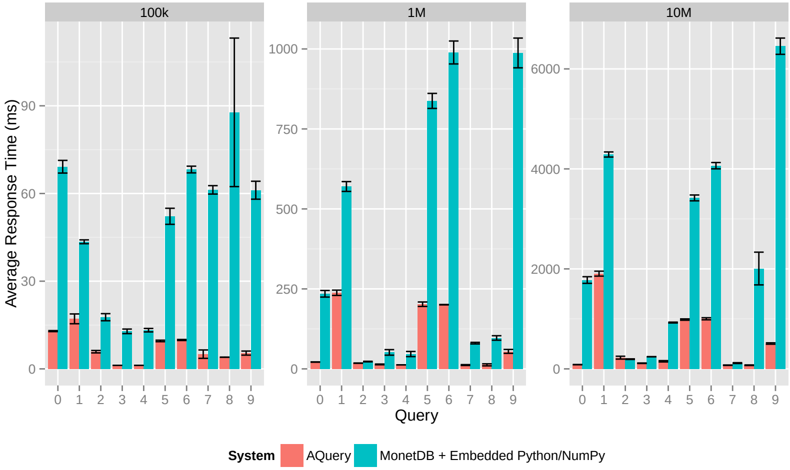

Figure 3: AQuery is faster across the board for 100K rows of stock history. When we increment to 1M AQuery remains faster in 8 of 10 queries, and comparable in the remaining 2. At 10M rows, AQuery is slightly slower for query 2, comparable for query 7, and faster in all others.

Finance Benchmark: Sybase IQ Results

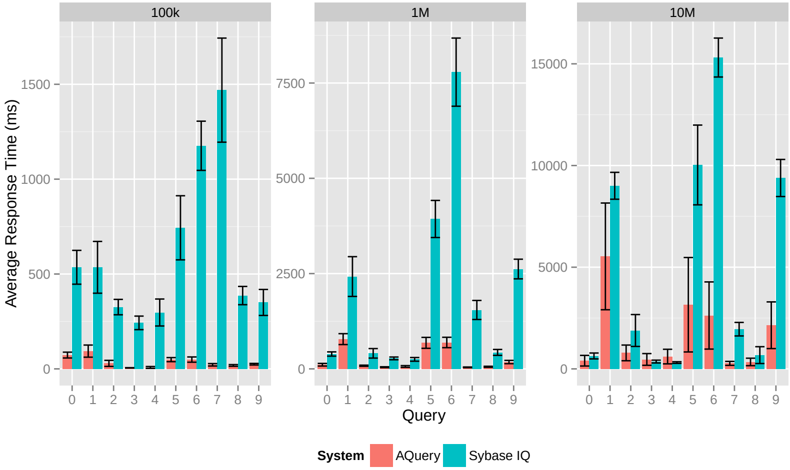

Figure 4: With 100K and 1M rows, AQuery outperforms Sybase IQ in all of the queries evaluated. At 10M rows, performance is a bit more varied, with larger standard errors, but on average AQuery is faster in 8 of the 10 benchmark queries.

Pandas Benchmark: Data Science Operations

- Picked a subset of operations used by Pandas to track library's historical performance evolution[16]
- 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

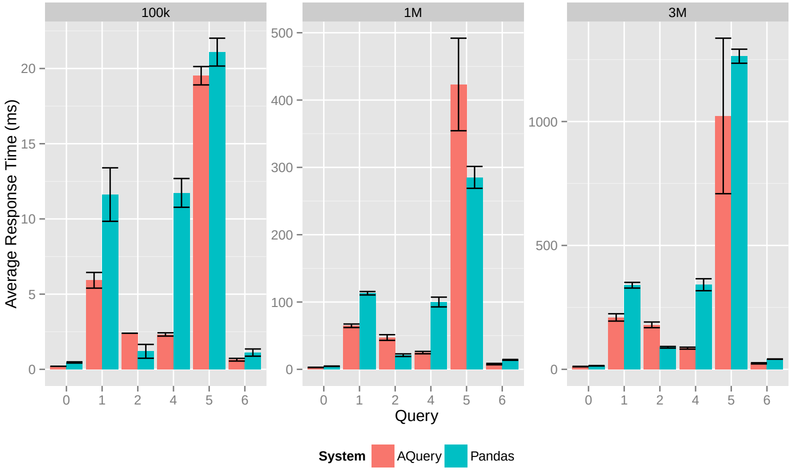

Figure 5: For 100K rows, AQuery is on average faster in 6 of 7 cases. For 1M and 3M rows, AQuery is faster in 5 of the 7 operations evaluated.

Pandas Benchmark: AQuery Results

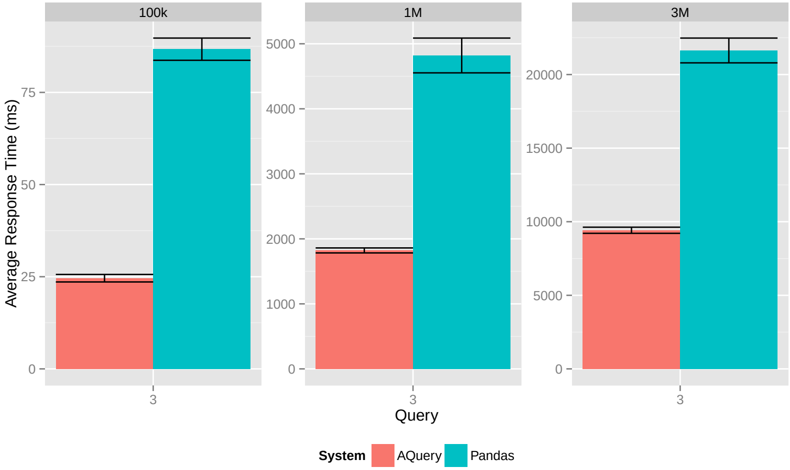

Figure 6: For 100K rows, AQuery is on average faster in 6 of 7 cases. For 1M and 3M rows, AQuery is faster in 5 of the 7 operations evaluated. The first set of graphs excludes query 3, for ease of reading, given the vastly different response time.

MonetDB Benchmark: Quantiles

- MonetDB's ability to embed R[6], and more recently, Python/NumPy [11], 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

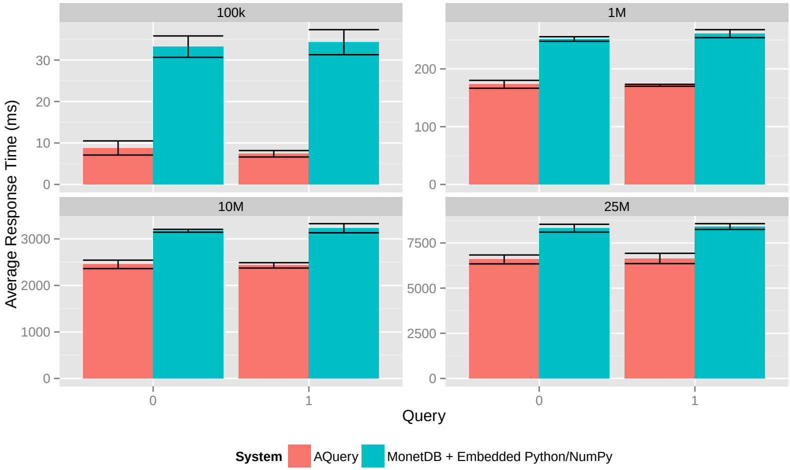

Figure 7: AQuery outperforms in all the dataset sizes evaluated. While the advantage narrows with larger data, we highlight AQuery's implementation is currently using a naive quantile calculation that involves sorting the entire array.

Demo

Video submitted as part of demonstration proposal (SIGMOD 2016), under review. We explore a series of simple financial trading strategies with real world data in AQuery.

Future work

- Explore further transformations
- Scalability
- Cost-based optimization
- Improved error reporting at compile time

References I

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.

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.

References II

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

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.

References III

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

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 IV

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).

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

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

References V

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).

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.