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

Dennis Shasha (joint work with José Pablo Cambronero)

Courant Institute/New York University

June 6, 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)[14][8][3]

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[4]
- Modest syntactic and semantic extension to SQL 92: (i) Adds one clause: assuming clause (order) (ii) Provides order-senstive aggregates

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.

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
```

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
```

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.

Push selections inside joins [classical]

$$t' \leftarrow \sigma_W(sort_S(t_1 \bowtie t_2))$$
$$\rightarrow$$
$$t' \leftarrow \sigma_{W''}(sort_S(\sigma_{W'}(\sigma_{W_1}(t_1) \bowtie \sigma_{W_2}(t_2))))$$

Selections before the first order-dependent aggregate can be pushed down to join arguments, if all columns for a selection pertain to a single argument. Equality-based selections are pushed down (W_1 and W_2). W' contains single-argument selections, which are pushed below the join while preserving helpful indexes.

Reorder selections [classical]

- Selections are reordered, while maintaining semantics, to use helpful indices

$$\sigma_W(t)$$
$$\rightarrow$$
$$W' \leftarrow [W_1, W_2, ..., W_n]$$
$$W'' \leftarrow \Sigma_i^n reorder(W_i)$$
$$\sigma_{W''}(t)$$

where W' is partitioned at each order-dependent aggregate, guaranteeing safe commutation of selections. *reorder* rearranges selections so as to take advantage of indices.

Sequential Implementation

- 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)

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, but query language doesn't suport common functionality like joins
- Influxdb[1]: Limited query language, no user-defined functions, no arbitrary sorting
- SciQL[3]: extends MonetDB[7] 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[10][15], not query plans

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]
- *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: Pandas Results

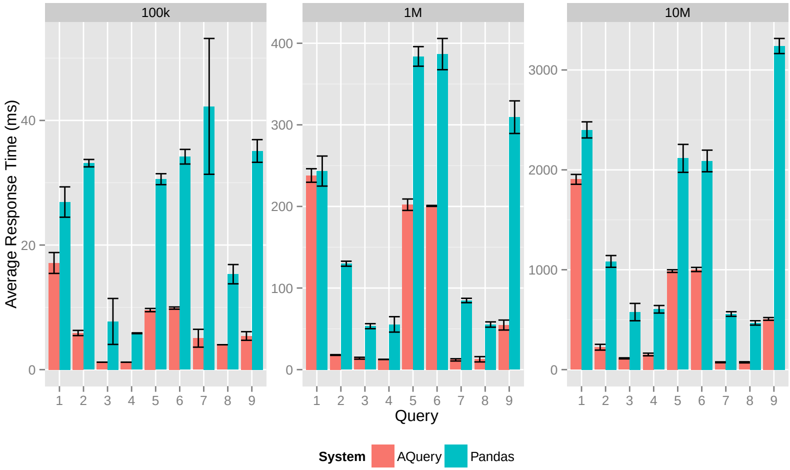


Figure 1: AQuery is faster with stock history of 100K, 1M and 10M rows across all queries. In various of these, AQuery's average response time is orders of magnitude shorter.

Finance Benchmark: Pandas Results

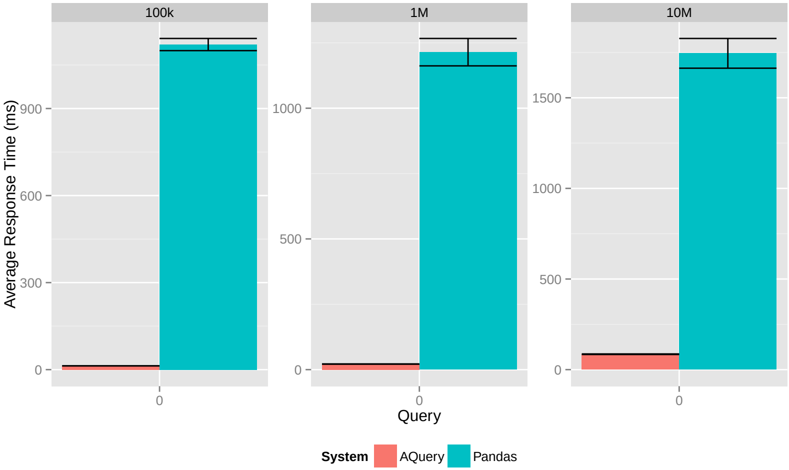


Figure 2: AQuery is faster with stock history of 100K, 1M and 10M rows across all queries. In various of these, AQuery's average response time is orders of magnitude shorter.

Finance Benchmark: MonetDB Results

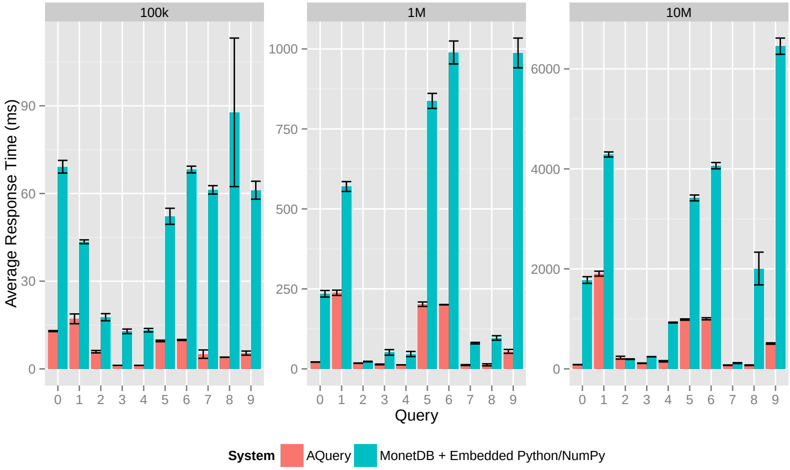


Figure 3: AQuery is faster across the board for 100K rows of stock history. When we increment to 1M AQuery remains faster in 8 of 10 queries, and comparable in the remaining 2. At 10M rows, AQuery is slightly slower for query 2, comparable for query 7, and faster in all others.

Finance Benchmark: Sybase IQ Results

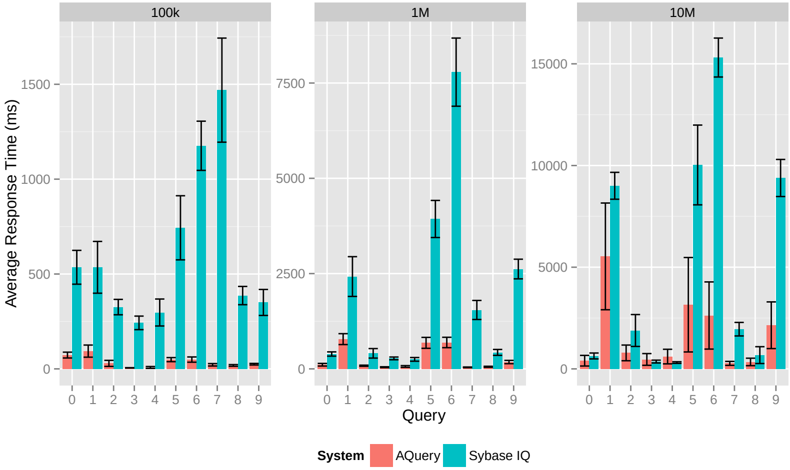


Figure 4: With 100K and 1M rows, AQuery outperforms Sybase IQ in all of the queries evaluated. At 10M rows, performance is a bit more varied, with larger standard errors, but on average AQuery is faster in 8 of the 10 benchmark queries.

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

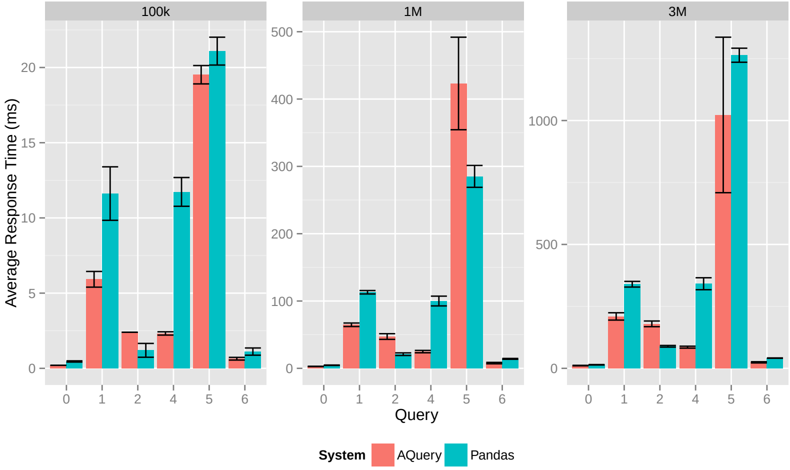


Figure 5: For 100K rows, AQuery is on average faster in 6 of 7 cases. For 1M and 3M rows, AQuery is faster in 5 of the 7 operations evaluated.

Pandas Benchmark: AQuery Results

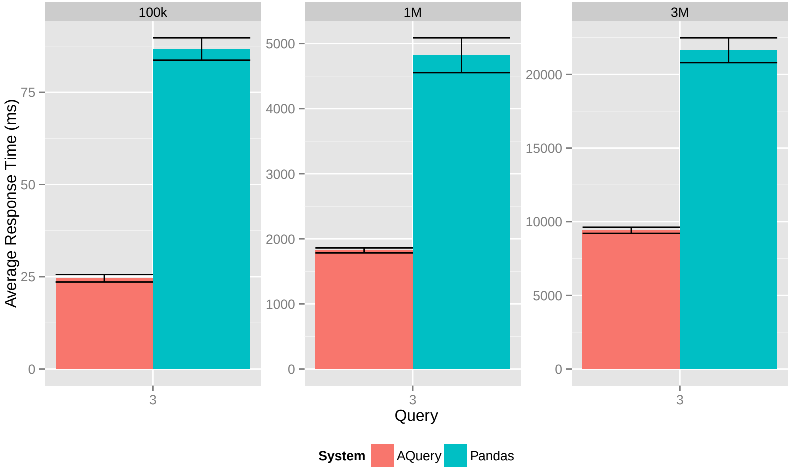


Figure 6: For 100K rows, AQuery is on average faster in 6 of 7 cases. For 1M and 3M rows, AQuery is faster in 5 of the 7 operations evaluated. The first set of graphs excludes query 3, for ease of reading, given the vastly different response time.

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

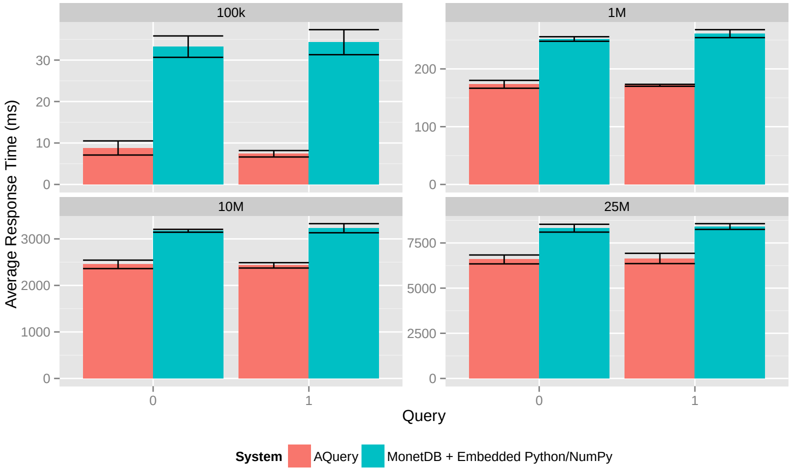


Figure 7: AQuery outperforms in all the dataset sizes evaluated. While the advantage narrows with larger data, we highlight AQuery's implementation is currently using a naive quantile calculation that involves sorting the entire array.

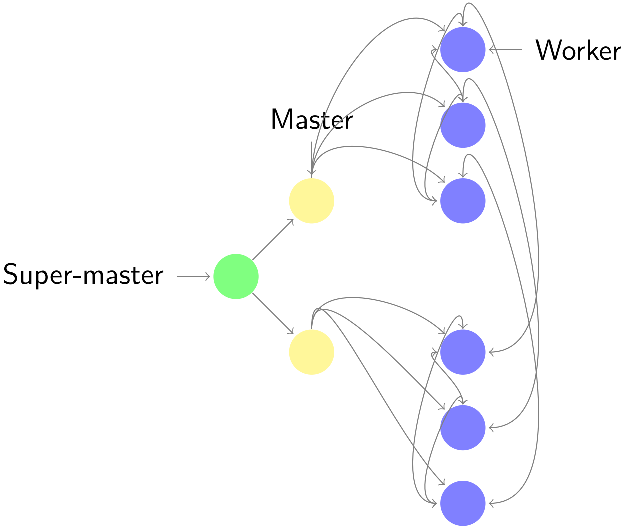
Parallel AQuery: newest work

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

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



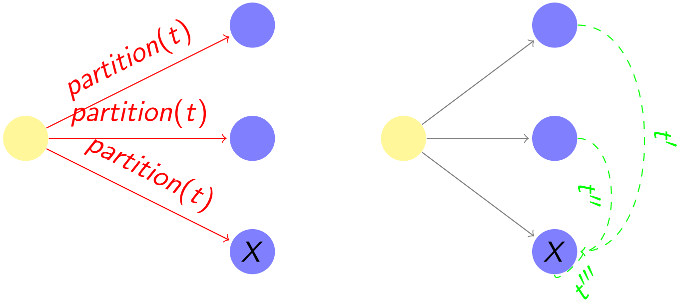
Parallel Primitives

- 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.
- Intra-cohort



Staged Reduce [classical]

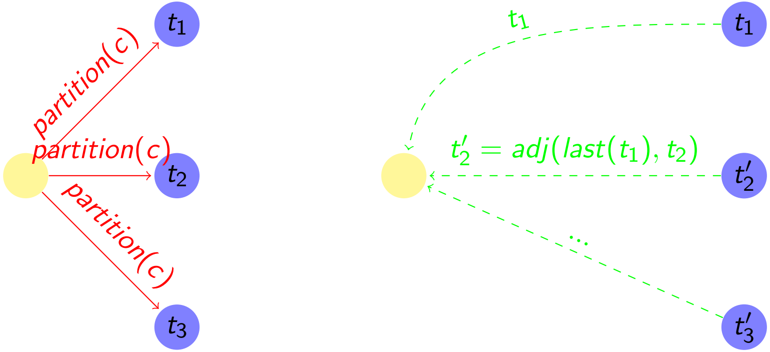
- Each worker does its own reduction.
- Optionally, stage reduced results into smaller and smaller summaries (e.g. for a global sum)

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

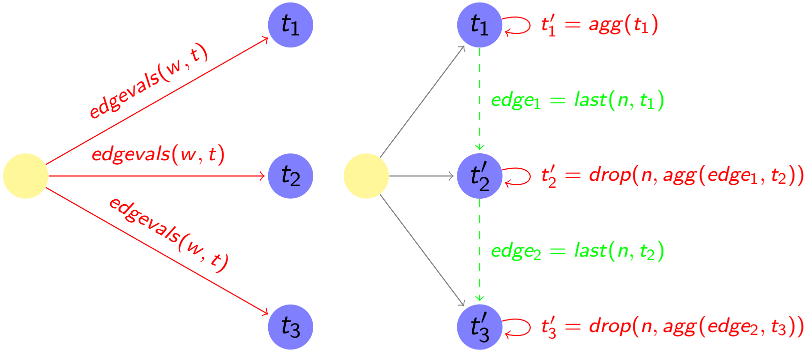


Edge-Extension

- 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

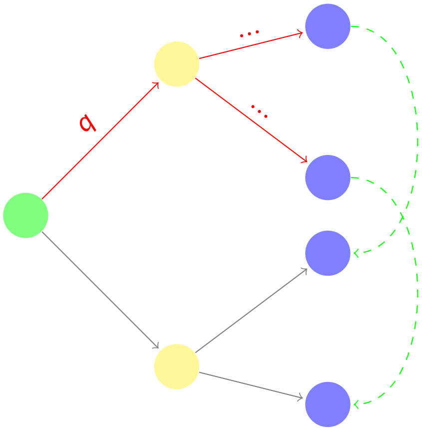
Edge-Extension

- $drop(x, y)$: 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



Synchronize

- Maintains replication
- Upon a write-query q, results are copied from each worker in the cohort to all of their respective counterparts
- Guarantees results available for later queries



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)

Parallel.q Performance: Setting

- We provide preliminary measurements using parallel.q-formulated queries for our finance benchmark
- Compare use of parallel.q vs. standard query formulation
- Multi-user setting, 20 cores, 18 workers, 2GB RAM per-core, 32-bit q executable
- Query execution time averaged over 5 iterations
- We scaled up parameters in queries from standard benchmark (i.e. more tickers and more days in interval-based queries)

Parallel.q Performance: Query description

- Query 0: Weekly/monthly/yearly price aggregates for 5000 stocks over 10 years
- Query 1: Adjust prices/volumes for splits over 1800 period for 5000 stocks
- Query 2: Difference in high/low price on stock split dates for 5000 stocks over a specified period of time.
- Query 3 + 4: Index calculation for 1000 and 5000 stocks
- Query 5: 21-day and 5-day moving average prices for 5000 stocks in 1800 day period (adjusted for splits)
- Query 6: Find intersection of averages in Query 5 (without using previously calculated results)
- Query 7: Simple trading strategy for 1000 tickers based on moving average crossings over a year
- Query 8: Pairwise correlation for 1000 stocks over 2 years
- Query 9: Yearly dividends and annual yield for 5000 stocks with no splits in 20 year period.

Parallel.q Performance: Results

Table 1: Performance for 20MM row historical price table (avg. execution time in ms)

Query	parallel.q	standard
0	5762.8	7163
1	8415.4	Out of memory
2	592.6	2724.4
3	77.8	50.2
4	153.4	126.4
5	5970.8	Out of memory
6	3780.2	Out of memory
7	567.6	481.6
8	2782.4	3758.2
9	3805	1859.6

Parallel.q Performance: Results (Continued)

Table 2: Performance for 200MM row historical price table (avg. execution time in ms)

Query	parallel.q	standard
0	9081	Out of memory
1	17546	Out of memory
2	2667	9063
3	202.6	181
4	859	920
5	14960	Out of memory
6	7637.4	Out of memory
7	776.8	711
8	3194.8	4031
9	10045.8	5259

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.
- Incorporate time series machine learning primitives.
- For this evening: tango

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.