AQuery, a query language for order in data analytics: Language, Optimization, and Experiments

ABSTRACT

The continuing success of the relational model results from the happy combination of expressive power and simplicity of expression. These virtues result from the fact that the model is based on a single data type and a few operations: unordered tables which can be selected, projected, joined, and aggregated. We contend that the unordered aspect of this model is in fact unnecessary for simplicity and needlessly limits the expressive power, making it difficult to express queries on ordered data such as time series data and other sequence data.

AQuery, introduced theoretically in [Lerner and Shasha(2003)], is a modest syntactic and semantic extension to SQL 92. AQuery supports ordered tables, called *arrables* (for array tables), which can be organized based on the values in one or more columns. Specifically, AQuery adds one clause (an assuming order clause) to SQL 92, and new (both system and user-defined) order-sensitive aggregates, such as moving average and running difference.

This paper reviews the semantics of AQuery both formally and through examples. The paper's first contribution is to explain the special optimization problems that arise because of order, the transformations that mitigate some of these problems, and an optimization framework to use those transformations. The paper's second contribution is to show the viability of the model through experiments comparing AQuery against other popular data analytic systems including Sybase IQ, Python's popular Pandas library and MonetDB using the union of benchmarks that those systems use themselves. On the same hardware, AQuery is overall significantly faster than all of those systems while offering simplicity of expression. Our examples come predominantly from finance and networking to show the variety of applications that can be handled.

AQuery itself is written using standard compiler tools, so could provide a front end to a wide variety of array systems.

1. INTRODUCTION

SIGMOD '16 San Francisco, CaliforniaUSA

ACM ISBN 978-1-4503-2138-9.

DOI: 10.1145/1235

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. The reason is that relational systems are based on an unordered table model, so order is provided by an ORDER BY clause basically as an afterthought. Such application programmers therefore either build their own time series database system or access data in their relational system and then use a separate language to process it (commonly, R, Python, Matlab, or even Java).

We assert (we are not alone in this [Seshadri et al.(1996)Seshadri, Livny, and Ramakrishnan] [Ng(2001)] [Kersten et al.(2011)Kersten, Zhang, Ivanova, and Nes]) that rethinking the relational model to allow order to be treated as a first class concept would open these applications to relational-like systems without unduly complicating the languages.

Here for example is a query that computes the average of end-of-day prices of a stock in SQL 92. Consider a table of the form $prices(ID, Date, EndOfDayPrice)$.

```
SELECT avg(EndOfDayPrice) as avg_px
FROM prices
WHERE ID = ‘‘AAPL’’
```

Here is the same query written in AQuery.

```
SELECT avg(EndOfDayPrice) as avg_px
FROM prices
WHERE ID = ‘‘AAPL’’
```

It's identical. Here is the SQL 99 expression of the query that computes the three day moving average of prices of stocks over a year.

```
SELECT Date,
avg(EndOfDayPrice) OVER (
        ORDER BY Date ROWS BETWEEN 2
            PRECEDING AND CURRENT ROW
        ) as ap
FROM prices
WHERE
Date BETWEEN ‘2014-01-01’ AND ‘2015-01-01’
```

By contrast, here is the AQuery formulation of that same moving average query.

```
SELECT Date, avgs(EndOfDayPrice,3) as ap
FROM prices
ASSUMING ASC Date
WHERE
Date BETWEEN ‘2014-01-01’ AND ‘2015-01-01’
```

Note that all that AQuery did was add an **ASSUMING** clause to the simple average query that says intuitively ”order this table by date" and a moving average aggregate that makes sense given this order.

While the semantics of this query say to order the whole table, we in fact need to order only the particular columns required by the query result (in this case the end-of-day price). So we can already see one order-related optimization. An additional obvious optimization is to recognize that a table that is already ordered consistently with the ASSUMING clause requires no further sorting.

Less obvious optimizations include figuring out when to perform the ordering relative to a join. For example, if we have a hash index on a join column, we might want to join before we order. So we have built a set of transformations to reflect possible optimizations. These generalize common relational transformations such as "select before join".

While our transformations are novel, our optimization strategy is currently rule-based. Transitioning to a cost-based optimization strategy is not hard, but that is not the research contribution we make here.

As for experiments, we build our system on top of a free version of "q", an array programming language provided by kx systems (www.kx.com) mostly for finance. This enables us to compare our system with other systems used in data science.

Section 2 will discuss related work in both language design and query optimization. Section 3 will present the semantics of AQuery and give examples. Section 4 will discuss the transformations we have implemented and our simple rule-based optimization strategy. Section 5 will present experiments comparing AQuery against MonetDB, Pandas, and Sybase IQ on the same hardware. Upon final publication, our software and data will be available to make our experiments fully reproducible.

2. 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, including, for example, joins and order-sensitive aggregates.

For example, DruidIO [Yang et al.(2014)Yang, Tschetter, Léauté, Ray, Merlino, and Ganguli] developed an open source data store for exploratory analytics. Their design is column-oriented, as is AQuery's, but their query language does not support common analytic functionality such as joins. Instead their focus is on data ingestion and real-time processing.

Another open-source timeseries solution, Influxdb, provides a simple-to-deploy solution because it has no external dependencies [Influxdb(2015 (accessed November 6, 2015))]. In contrast to other options, Influxdb supports a SQL-like language, but with restricted expressability. For example, user-defined functions have been discussed [Influxdb(2013-2015 (accessed November 7, 2015)a)], but are not yet supported. User-defined functions are useful to implement readable and efficient analytical tasks such as financial trading strategies. Further, Influxdb does not support sorting a table by multiple columns, which complicates queries for multidimensional time series data [Influxdb(2013-2015 (accessed November 7, 2015)b)].

Array-oriented languages, often based on APL, have long held a prominent position in time series analytics. Recent work has sought to bring their expressive power to databases. SciQL [Kersten et al.(2011)Kersten, Zhang, Ivanova, and Nes] extends MonetDB [Nes and Kersten(2012)] with arrays as first-class entities and provides elegant idioms for their use in scientific applications. Similarly, the system currently underlying AQuery's execution, q [Whitney(2009 (accessed November 6, 2015))], enables succint algorithm expression through its array orientation.

Excellent optimization work has focussed on reliability and scalability [Pelkonen et al.(2015)Pelkonen, Franklin, Teller, Cavallaro, Huang, Meza, and Veeraraghavan] [StumpleUpon(2015 (accessed November 6, 2015))], but not on query plans. AQuery attempts to encompass both traditional (i.e. unordered table) query transformations and order-specific transformations extending and realizing the previous work [Lerner and Shasha(2003)].

Anecdotal evidence for Python's increasing role in data science includes the scores of articles [Cass(2015 (accessed November 7, 2015))] that indicate its popularity among programmers. Python's simple and expressive syntax, in combination with its productive open source community, has led to the creation of a well-known library for data analysis called Pandas. Pandas provides in-memory array-oriented data structures called data frames [pandas development team(2015 (accessed November 7, 2015))], along with tools to manipulate them.

Pandas is explicitly aimed at achieving high performance. Critical parts of its implementation have been optimized using Cython, C, and numpy (an array oriented library for scientific computing) [Oliphant(2006)]. Finally, Pandas has seen heavy adoption in the finance community (Pandas was in fact developed at AQR, a financial institution). The overarching goals of simplicity and performance, combined with the domain and scale of adoption ([Sta(2015 (accessed November 7, 2015))] shows 17,229 questions tagged as Pandas in stackoverflow.com), justifies a comparison between AQuery and Pandas.

Major database vendors have also implemented column-oriented solutions. Sybase IQ is one of the best known. The system has become popular for basic business analytics across different industries, with the overarching goal of providing "high-performance decision-support" [SAP(2013 (accessed November 8, 2015)a)] . Like AQuery, Sybase IQ attempts to provide users with easy-to-use analytic functionality, allowing developers to extend the platform with user-defined functions that allow deeper analysis. These user-defined functions may be implemented in Java/C/C++ [SAP(2013 (accessed November 8, 2015)b)] which are then linked to SQL functions. Sybase IQ has found significant use within the financial services industry: it underlies Sybase RAP, which provides a range of risk, trading, and support analytics targeted at finance [SAP(2015 (accessed November 8, 2015))].

3. AQUERY SEMANTICS AND EXAMPLES

In this section, we explore the semantics of the AQuery language and show examples where appropriate.

The *Arrable*, short for array-table, is the key data type underlying AQuery. Unlike its traditional SQL counterpart [Codd(1970)], the rows in an arrable are ordered. Further, an arrable permits individual elements to be non-atomic, so a query can, for example, place an array as a valid value in an arrable's column.

Given arrables of the form $t_i(c_1, ..., c_{n_i})$, boolean expressions of the form $w_j(t_i.c_1, ..., t_i.c_{n_i})$, expressions of any type $p_j(t_i.c_1, ..., t_i.c_{n_i})$ and $g_j(t_i.c_1, ..., t_i.c_{n_i})$ a query in AQuery can be broken down into various separate clauses, implementing various of the operations defined in [Lerner(2003)]:

- A from-clause, that can consist of a single arrable t_i, or various arrables combined using $\times(t_1, ..., t_n)$ (cross-product) or join operators, such as $\bowtie_{w_j,...,w_k}(t_i, ..., t_n)$.
- An assuming-clause, which declares (and enforces) the appropriate ordering of an arrable. Assuming clauses consist of a series of *ASC/DESC* statements along with the appropriate column to use for the ordering.
- A where-clause, that enforces selection $\sigma_{w_j,...,w_k}$.
- A group-by clause, that performs groupings along expressions g_j, and results in a nested arrable
- A having-clause, which performs selections based on groupings by applying predicates of the form w_j.
- A projection-clause, which projects a series of expressions p_j over an arrable

The assuming clauses constitutes the main extension to SQL-92. We now explore each clause in further detail.

3.1 From-Clause

All AQuery queries have as a source one or more arrables, which may include temporary local tables to achieve some of the functionality of nested queries.

Semantically, the from-clause is executed first, in order to materialize the data required by the following steps. While users' joins performed using explicit *INNER JOIN/ FULL OUTER JOIN* operations are performed in the order provided, joins performed using the cartesian product operator and predicates are subject to reordering by the optimizer.

So

```
SELECT *
FROM
prices INNER JOIN
     sales INNER JOIN dividends USING Id
USING TradeDate
```

would first join *sales* and *dividends*, and then join prices, as these operations are right-associative in AQuery. The query below, however, could be subject to reorderings.

```
SELECT *
FROM prices, sales, dividends
WHERE sales.Id = dividends.Id AND prices.
    TradeDate = sales.TradeDate
```

3.2 Assuming-Clause

An arrable $t_i(c_1, ..., c_{n_i})$ can be ordered by various of its columns, $c_j, ..., c_k$. Furthermore, the ordering can be ascending or descending along the values in the various columns. The assuming clause allows a query to specify the expected order for the data and presumes this order is available for all operations. Semantically, this takes place after execution of the from-clause.

Assume an arrable
$network(base\_station, date, hour\_stamp, num\_cons)$
represents network loads across various base stations

```
SELECT avgs(num_conns , 24)
FROM network
ASSUMING ASC date, ASC hour_stamp
WHERE base_station=1
```

Figure 1: A networking query: 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

In the simple example in Figure 1, (a copy of) the network arrable is sorted by ascending date, and within each date, by ascending hour stamp. This then lets the user calculate a moving average with window size 24 (ie. 24 hours) over the number of connections for base station 1.

The sorted order of an arrable (if any) is maintained as meta-data, so the result of this query is identified as being sorted by *(asc date, asc hour_stamp)*. This may be useful for future queries.

3.2.1 Order and Operations: Dependence, Preservation, Equivalence

The ordered nature of AQuery in turn means that the behavior of operations relative to order can be characterized as either order-dependent or order-independent and either order-preserving or non-order-preserving.

Definition 1. *An operation op is* order-independent *if for all different orderings a_1 and a_2 of an arrable a, applying op to a_1, denoted op(a_1), yields the same result as op(a_2) after some permutation of op(a_1).*

For example, a standard group by aggregate is order-independent. If this doesn't hold then *op* is *order-dependent.* For example, moving average is order-dependent.

Definition 2. *An operation is* order-preserving *if it preserves the pre-existing order in an arrable. So if row r_i precedes r_j in the arrable before an order-preserving operation and both rows survive the operation, then r_i will precede r_j in the result.*

For example, a selection that scans sequentially through a table and outputs rows that meet constraints as it encounters them will be order-preserving. A selection that uses a non-clustered index will likely be non-order-preserving.

Definition 3. *We call two arrables a and b* permutation-equivalent *if some permutation of a yields b. So they must contain the same number of rows and there must be a 1-1, onto mapping among the rows. In the sequel, when we say that a query $q1$ is semantically equivalent to a query $q2$, we mean that $q1$ applied to any state s will yield a permutation-equivalent result as $q2$ appied to s.*

3.3 Where-Clause

Similarly to SQL-92, AQuery permits users to specify selections in the where-clause by specifying a series of predicates that must hold on the resulting arrable. Semantically, the where-clause applies after the assuming clause has executed. This implies that the where-clause conditions must semantically be order-preserving. Consider,

```
SELECT max(volume)
FROM stocks
```

```
ASSUMING ASC TradeDate
WHERE
ticker = ''IBM'' AND price>prev(price)
```

along with three tuples in our arrable:

```
(''HP'',  2015-11-16, 100)
(''IBM'', 2015-11-17, 150)
(''IBM'', 2015-11-18, 170)
```

The *prev* operation results in an array of the same size, with all elements shifted forward one position, such that given an array a, $prev(a)[i] = a[i-1]$, with the value at $prev(a)[0]$ replaced with a *null* marker.

The AQuery semantics are that the predicates are applied in the order they appear. So, semantically, the predicate on "IBM" executes to completion, then, in a second pass, the prev is applied. This will yield just the last tuple, because the IBM part of the query will leave just the last two tuples. By contrast, the query

```
SELECT max(volume)
FROM stocks
ASSUMING ASC TradeDate
WHERE
price>prev(price) AND ticker = ''IBM''
```

would find the last 2 tuples in a first pass, and its second pass would simply return them. This observation implies that the where-clause plays a key role in determining which attributes to order and when.

3.4 Group-By and Having-Clauses

A group-by clause specifies a series of expressions g_j, each as a function of arrays in the arrable, that are then used to group the arrable into nested arrable subsets corresponding to the unique product of values across the expressions g_j. So for example:

```
SELECT base_station, date, hour_stamp,
   num_conns
FROM network
GROUP BY base_station
```

would take the first arrable in Figure 2 and return the second nested arrable.

The Having-clause allows users to specify predicates on the nested arrable and refine the arrable prior to projection. Implicitly, operations taking place in the having-clause are modified automatically to handle the nested nature of the arrable. This introduces the AQuery algebra modifier *each*, a modifier shared with AQuery's ancestor k [Whitney(2009 (accessed November 6, 2015))]. *each* takes an expression $e(c_i, ..., c_j)$ and applies it to a nested arrable by applying it to the fields associated with each group, which are themselves arrays.

For example, the query in Figure 3 would first group the arrable, resulting in the nested arrable shown in Figure 2, and then the *each* modifier applies over the operations in the $HAVING$ clause, allowing the user to apply the *sum* aggregate over each nested arrable and then select those subsets that have at least 50 connections.

Semantically, the group-by clause is executed after the where-clause, followed by the having-clause.

base_station	date	hour_stamp	num_conns
1	04/01/15	1	10
1	04/01/15	2	20
2	04/01/15	1	30
2	04/01/15	2	40

base_station |	date	hour_stamp	num_conns
1 |	[04/01/15,04/01/15]	[1, 2]	[10, 20]
2 |	[04/01/15,04/01/15]	[1, 2]	[30, 40]

Figure 2: Grouping may generate nested arrables in which field values are arrarys

```
SELECT base_station,
COUNT(*) as ct
FROM network
GROUP BY base_station
HAVING sum(num_conns) >= 50
```

base_station |	ct
2 |	2

Figure 3: *each* allows us to adapt function behavior for nested arrables, so powerful aggregates can be applied on nested arrables intuitively

3.5 Projection-clause

The projection clause takes a series of expressions $e_i(c_j, ..., c_k)$ and applies them one at a time to form the output arrable. The operations in the projection-clause may be order-dependent as in the moving average examples we've already seen. Semantically, the projection clause is the final evaluation step prior to returning the query result to the user.

3.6 UDFs, Local Queries, and Array Extraction

To provide an expressive analytics platform, AQuery allows the definition of user-defined functions (UDFs) using the AQuery language. UDFs are treated identically to built-ins, meaning they can be used wherever a normal expression would be permitted.

While nested queries are not permitted in AQuery, the WITH clause incorporates query-local temporary tables into a larger table. For example, the following SQL query:

```
SELECT sum(volume) FROM
   (SELECT *
    FROM sales
    WHERE TradeDate BETWEEN ''01/01/2014''
         AND ''12/31/2014''
    ) y14
 INNER JOIN prices USING Id
 WHERE price > 100
```

becomes the following AQuery query:

```
WITH
  y14 AS (
        SELECT *
        FROM sales
        WHERE TradeDate BETWEEN
            ''01/01/2014'' AND
            ''12/31/2014''
        )

SELECT sum(volume)
FROM y14 INNER JOIN prices USING Id
```

```
WHERE price > 100
```

y14's scope is limited to statements that follow it within the *WITH* and up until the main query, which returns the resulting arrable to the user.

Finally, it is possible to bind the columns in a query's result with stand-alone variables, allowing users to utilize query results in further computations as arrays.

```
EXEC ARRAYS
SELECT avgs(px, 7) as moving_average
FROM prices
ASSUMING ASC TradeDate
```

allows the user to extend the environment with a variable *moving_average*, which has a value bound to a copy of the array *moving_average* in the resulting arrable. The user is now free to use this variable in other queries or functions.

4. TRANSFORMATIONS

In this section, we introduce the transformations applied to queries under this implementation of AQuery. We demonstrate their semantic equivalence where necessary and justify their use. Finally, we describe the current heuristic rule-based optimization strategy employed when analyzing a new query provided by the user. As mentioned above, the eventual goal is cost-based optimization.

4.1 Eliminating Sorts

Suppose an arrable t is already ordered by $t.c_i, ..., t.c_j$. If a query's assuming clause indicates a required order $(t.c_k, ..., t.c_l)$, where, $t.c_k, ..., t.c_l$ is a prefix of the existing order, we don't need to sort t. This is the simple optimization mentioned in the introduction. Note however that this implies that all operations must be order-preserving.

4.2 Sequence Selection

As discussed in the semantics section, AQuery's selection predicates are interpreted as a sequence of selections, in the spirit of relational cascades [Elmasri and Navathe(2014)]. Like relational systems, AQuery reorders the selections as long as these reorderings result in semantically equivalent selections.

Assume a sequence of selection predicates $W = (w_1, ..., w_n)$. If there does not exist an i such that w_i's result depends on the order of elements in the array, then the sequence W can be freely rearranged, just as in relational systems.

Now assume a sequence of selection predicates $W = (w_1, ..., w_j, ..., w_n)$, such that w_j represents the first and only selection that has inter-row dependencies (i.e. depends on order). We can split this sequence into 3 subsequences, $(w_1, ..., w_{j-1})$, (w_j), and $(w_{j+1}, ..., w_n)$. We can freely reorder selections within the first subsequence, and within the last subsequence, as they reduce to the first case. We do not move operations from one subsequence to another, as this might change the semantics as we saw in the IBM/prev example of section 3.3.

The analysis extends inductively to selections with multiple predicates with inter-row dependence. The intuition can be summarized as: inter-row dependent (i.e. order-dependent) selections act as boundaries for selection re-ordering.

As in relational optimizers, we perform re-orderings such that selections involving columns with useful index information evaluate first, this reduces the data we need to work with and allows us to take advantage of existing index information.

Theorem 4.1. *Suppose a query q contains a where-clause consisting of a sequence of selection predicates $W = (w_1, ..., w_n)$ and predicates with order-dependent operations $P_{od} = (p_1, ..., p_k)$ at positions $I = (i_{p_1}, ..., i_{p_k})$. A query $q2$ that reorders predicates in W within the subsequences demarcated by I has the same semantics as q provided that, in $q2$, all predicates including and following the first order-dependent predicate are order-preserving with respect to the assuming clause.*

Proof Sketch. Query q has the semantics that the assuming clause is executed first and all operations are order-preserving and executed in sequence. Each order-independent predicate applies to tuples one at a time. This implies that two such predicates can be applied in any order to an input arrable a to yield permutation-equivalent results. In $q2$, an order consistent with the assuming-clause is established before the first order-dependent predicate and preserved thereafter. Thus q and $q2$ will be semantically equivalent. □

4.3 Delayed Sorting

Because order-preserving operations often lead to slower implementations than non-order-preserving ones, an optimizer will often delay sorting as much as possible. For example, given a sequences of selections $W = (w_1, ..., w_n)$ such that w_j represents the first predicate with order-dependent operations, we execute $(w_1, ..., w_{j-1})$ prior to sorting. This results in a smaller arrable to sort.

This heuristic requires adaptation in the face of joins.

Consider a join between arrables $t_1 \bowtie t_2$. Our system seeks to perform any index-based joins prior to the first order-dependent where-clause predicate. Furthermore, in order to reduce the size of the arrables joined we take selections prior to this first order-dependent predicate, so the optimizer uses the following rules:

- Evaluate equality selections before the join
- But perform the index-supported join before non-equality selections

This rule-based approach assumes that the benefit from the reduction in arrable size from a selection predicate involving equality will outweigh any benefits from the index-supported join.

Theorem 4.2. *Given a query q, we can execute any order-independent operations prior to the first order-dependent operation without sorting. Sorting and execution of the remaining operations is guaranteed to yield the same result as sorting prior to the first operation.*

4.4 Sorting Columns, Not Arrables

Often, *ASSUMING* does not require sorting the entire arrable. In any given query q over an arrable t, we can identify a subset of columns that require sorting $C_{od} = \{c | c \in t \wedge needs - sort(c)\}$. The $needs - sort$ predicate indentifies columns requiring sorting to create a query $q2$ that is semantically equivalent to q.

Consider the example in Figure 4, from a finance setting

Reading the where-clause from left-to-right, we can evaluate the first 2 predicates without sorting, as neither is order-dependent. The third predicate is order-dependent,

```
SELECT
getYear(TradeDate) as year,
max(avgs(ClosePrice, 10)) as max_mavg
FROM stock_history
ASSUMING ASC TradeDate
WHERE Id = ''IBM''
AND TradeDate BETWEEN ''2010-01-01''  AND
    ''2015-01-01''
AND sums(volume) >= 1000000
GROUP BY  getYear(TradeDate)
```

Figure 4: A query from finance, showing the use of order-independent and order-dependent predicates along with other clauses. Analyzing clauses left to right, it is clear we can evaluate the first two order-independent selection predicates in the where-clause. Prior to the first order-dependent predicate, we sort all attributes used in the remaining selections, group-by and projection clauses.

and thus at this point AQuery must collect all attributes that require sorting by inspecting the remaining selections in the where-clause, group-by and projection clauses. This results in $C_{od} = \{volume, TradeDate, ClosePrice\}$. Once we have sorted these columns, we can apply the remaining selections, group, and aggregate. The result is semantically equivalent to sorting from the start.

The strategy followed in analyzing Figure 4 is a unique case of the more general approach explained in Algorithm 1.

Theorem 4.3. *Given an AQuery query q, Algorithm 1 yields a query $q2$ that is semantically equivalent to q.*

Proof Sketch. $q2$ consists of two queries: an order-independent one $q2A$ and an order-dependent one $q2B$. $q2A$ includes expressions/clauses that don't require order up to the first order-dependent operation. $q2B$ sorts all attributes necessary from that point onwards in a manner consistent with the assuming clauses of q and all operations in $q2B$ are order-preserving. Combining these two is semantically equivalent to q. □

4.5 Full Strategy Sketch

Given a new query q, the heuristic optimization approach is sketched out in Algorithm 2.

5. EXPERIMENTS

In all our experiments, we measure average response time (in milliseconds) to evaluate performance of other systems relative to AQuery. For all experiments, the execution order of the queries is randomized.

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.

For all experiments we use the following systems/libraries:

- 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

```
Input  : An AQuery query $q$
Output : Order of evaluation and attributes needing sort
Clauses in $q$ are analyzed in the order: where-clause ($W$), assuming-clause ($A$), group-by-clause ($G$), having-clause ($H$), projection-clause ($P$).
Thus $G$ is "subsequent" to $W$ for example.
Each clause is analyzed from left-to-right.
case $W$ has first order-dependent predicate at $j$
    Evaluate all $W_{i<j}$ according to Theorem 4.1
    Sort all attributes in $W_{i>=j}$, and subsequent clauses
    Evaluate selections $W_{i>=j}$ according to Theorem 4.1 and subsequent clauses
end
case $G$ or $H$ have any order-dependent expression
    Evaluate all selections $W$ according to Theorem 4.1
    Sort all attributes in subsequent clauses
    Evaluate $G$ and subsequent clauses
end
case $P$ has any order-dependent expression
    Evaluate all selections $W$ according to Theorem 4.1
    Evaluate $G$ and $H$
    Determine attributes that need sorting by analyzing $P$, sort them
    Evaluate $P$
end
```

Algorithm 1: Identifying order of evaluation and attributes to sort.
Once an order-dependent predicate is found, all attributes in the following predicates in the where- clause and all attributes in subsequent clauses must be sorted. If there are no order-dependent predicates in the where clause, then if any predicate in the group-clause is order-dependent, then every attribute in the group, having, and projection clauses must be sorted. Once the sorting takes place, all operations must be order-preserving. So for example, selections in W after the first order-dependent predicate must all be order-preserving. Similarly for the remaining clauses.

```
Input  : An AQuery query $q$
Output : Heuristically optimized query plan
if from-clause involves crossproduct then
    Replace cross products and selection predicates within joins
if has foreign key joins then
    Replace explicit joins with pointer-based access for foreign keys, rather than materializing the join.
Execute Algorithm 1.
Return improved plan.
```

Algorithm 2: A Sketch of Optimizations

Query	Condensed Description
0	Get the closing price of a set of 10 stocks for a 10-year period and group into weekly/monthly/yearly aggregates. For each determine low/high/avg value. The output should be sorted by id and trade date.
1	Adjust all prices and volumes for a set of 1000 stocks to reflect the split events during a specified 300 day period.
2	For each of 1000 stocks, find the differences between the daily high/low on day of each split event during a specified period.
3	Calculate the value of the S&P500 for a specified day using unadjusted prices.
4	Calculate the value of the Russell 2000 for a specified day using unadjusted prices.
5	Find the 21-day and 5-day moving average price for a specified list of 1000 stocks during a 6-month period. (Use split adjusted prices)
6	(Based on the previous query) Find the points when the 5-day moving average intersects the 21-day moving average. The output is to be sorted by id and date.
7	Determine the value of $100,000 now if 1 year ago it was invested equally in 10 specified stocks. When 21-day moving average crosses over 5-month moving average the complete allocation for that stock is invested and when 21-day moving average crosses below 5-month moving average the entire position is sold.
8	Find the pair-wise coefficients of correlation in a set of 10 securities for a 2 year period. Sort by coefficient.
9	Determine the yearly dividends and annual yield for the past 3 years for all the stocks in the Russell 2000 index that did not split during that period.

Table 1: Financial Queries Description

- Sybase IQ version 16.0
- q version 3.2 2014.11.01
- AQuery compiler a2q version 1.0

Because time is the most common way to organize data and because finance provides many examples of data analysis on time series, we start with the following financial benchmark from Sybase [SAP(2008 (accessed November 8, 2015))], Table 1 briefly describes the queries. While these queries represent common analytic tasks in finance, it is easy to find analogous operations in other domains where ordered data matters.

5.1 The Sybase IQ Financial Benchmark

The results shown in this section correspond to simulated data. Some of the queries require random parameters, for example the starting and ending dates or a subset of stock identifiers. Given this, we perform various iterations of our experiments, consisting of generating data at different sizes (100K, 1M, and 10M observations) and randomly generating the query parameters multiple times. We present the average reponse time for the various systems in Figures 5, 6 and 7.

We note that AQuery is faster than Pandas on the financial benchmark in Figure 5, significantly outperforming across data sizes and queries.

Similarly, AQuery is faster than MonetDB in all queries for a data set of 100K rows. At 1M rows, AQuery is faster for 8 of 10 queries and comparably performant for the remaining 2. At 10M rows, we see a significant advantage in 7 of 10 queries, slightly slower performance in 1 query and comparable in another query.

Finally, when compared to Sybase IQ we note that AQuery is orders of magnitude faster on the 100K and 1M data sets. The performance is much more varied in the 10M data set, with wider standard errors. However, on average AQuery outperforms in 8 out of 10 queries.

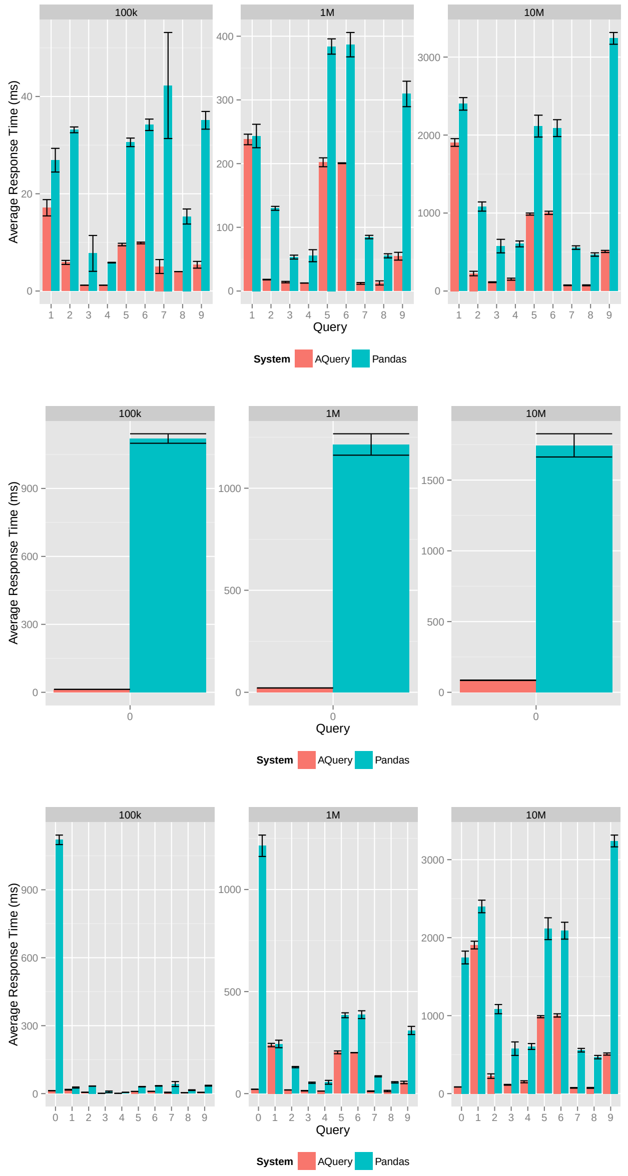

Figure 5: Pandas Comparison on Sybase IQ Financial Benchmark: AQuery is faster with stock history of 100K, 1M and 10M rows across all queries. In various of these, AQuery's average response time is orders of magnitude shorter.The first set of graphs excludes query 0, for ease of reading, given the vastly different response time. The second set shows query 0 separately. And the third shows all queries combined.

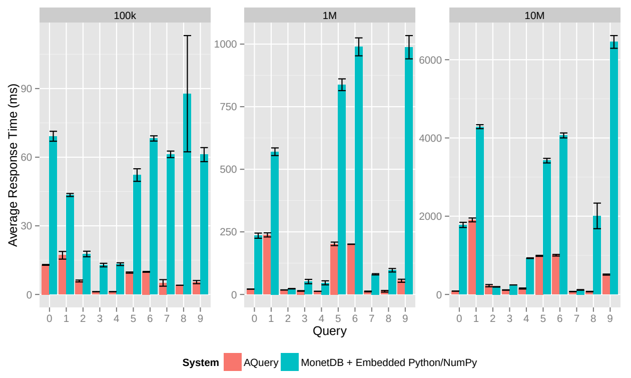

Figure 6: MonetDB Comparison on Sybase IQ Financial Benchmark: AQuery is faster across the board for 100K rows of stock history. When we increment to 1M AQuery remains faster in 8 of 10 queries, and comparable in the remaining 2. At 10M rows, AQuery is slightly slower for query 2, comparable for query 7, and faster in all others.

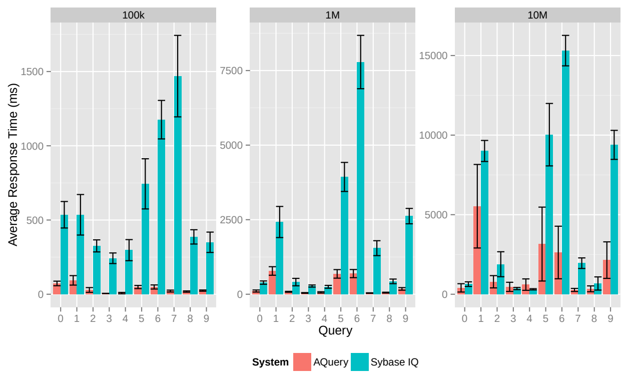

Figure 7: Sybase IQ Comparison on Sybase IQ Financial Benchmark: With 100K and 1M rows, AQuery outperforms Sybase IQ in all of the queries evaluated. At 10M rows, performance is a bit more varied, with larger standard errors, but on average AQuery is faster in 8 of the 10 benchmark queries.

Table 2: Pandas benchmarking: queries and descriptions

Query	Description
0	selection
1	group-by w/ multiple functions
2	count instances of each value
3	group-by along multiple columns
4	append to existing data
5	merge based on multiple columns
6	calculate standard deviation

5.2 Pandas Benchmark: Data Science Operations

In order to compare AQuery to Pandas, we pick a subset of operations that Pandas uses to track the library's historical performance evolution [the pandas development team(2011 (accessed November 18, 2015))]. The chosen subset represents common tasks in data science, for example: subsetting, grouping, summarizing, and merging data, amongst others.

Figure 8 shows the performance results in terms of average response time. The experiments are repeated along various baseline data sizes: 100K elements (as used in Panda's benchmarking), 1M, and 10M elements. For each of the data sizes, we randomly generate the data 5 times, on each data set we measure the average execution time for each query over 5 repeated evaluations. The results presented here represent an average of these times. Table 2 provides brief descriptions of each query.

We note that AQuery outperforms in 5 of 7 queries in most of the data sets we evaluated. Furthermore, AQuery's advantage in these queries is significant, with some running up to 3 times faster than in Pandas.

5.3 MonetDB Benchmark: Quantiles

MonetDB's ability to embed R [Mon(2014 (accessed November 18, 2015))], and more recently, Python/NumPy [Raasveldt(2015 (accessed November 06, 2015))], 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.

For comparison with MonetDB, we used MonetDB's benchmarking quantile computation, used in both [Mon(2014 (accessed November 18, 2015))] and [Raasveldt(2015 (accessed November 06, 2015))].

Specifically, we evaluated 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, which sorts a column and then takes the appropriate value. We calculate the 95th and 5th quantiles across random data.

Figure 9 shows our results, in terms of average response time in milliseconds. The experiments are repeated along various data sizes: 100K, 1M, 10M, and 25M values. For each data size, we generate 5 random data sets, and measure average execution time for each quantile calculation over 5 iterations. These response times are then averaged and appropriate standard error measures calculated.

In all cases, AQuery outperforms the equivalent calculation in MonetDB + embedded Python/Numpy, with the gap in performance narrowing in larger data sets. However, we remind the reader that the AQuery implementation uses a naive quantile function which sorts the data completely.

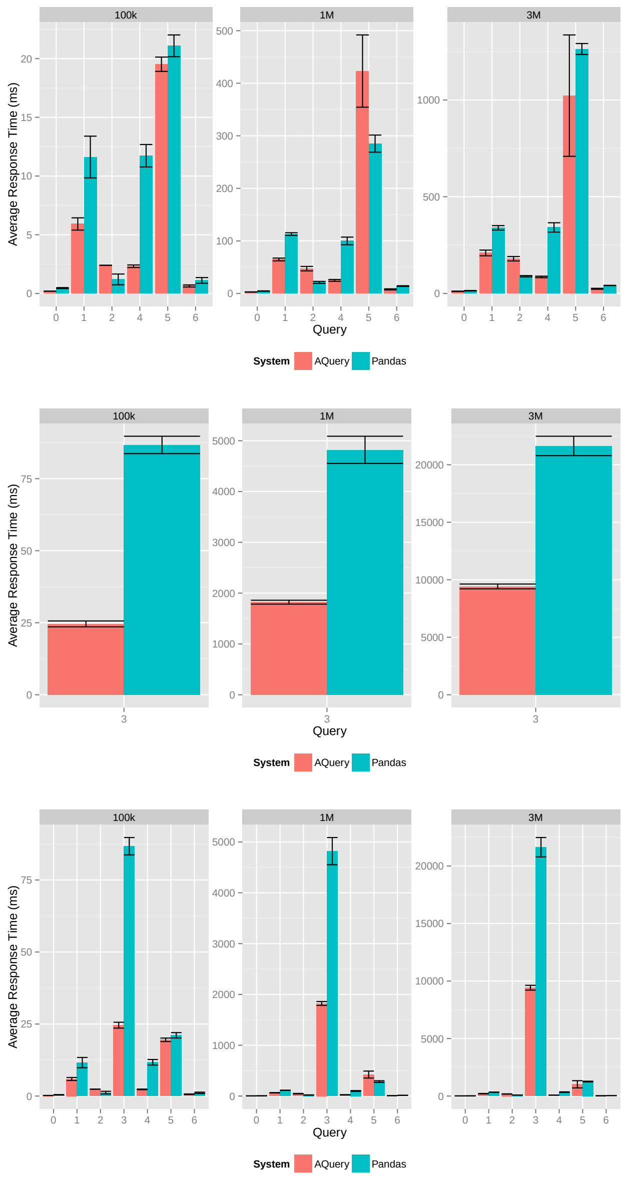

Figure 8: Pandas Benchmarking Operations: For 100K rows, AQuery is on average faster in 6 of 7 cases. For 1M and 3M rows, AQuery is faster in 5 of the 7 operations evaluated. The first set of graphs excludes query 3, for ease of reading, given the vastly different response time. The second set shows query 3 separately. And the third shows all queries combined.

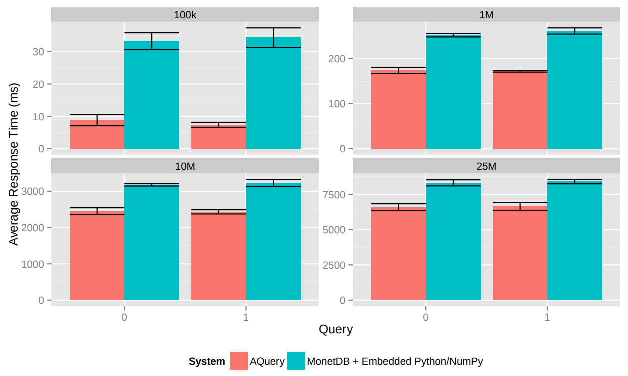

Figure 9: MonetDB Quantile Benchmarking: AQuery outperforms in all the dataset sizes evaluated. While the advantage narrows with larger data, we highlight AQuery's implementation is currently using a naive quantile calculation that involves sorting the entire array.

6. CONCLUSION

We have described a query language, transformations, and implementation designed to address the challenges of analyzing data that require order. AQuery shows that order can be introduced with minimal complication to the language and computational model. The paper describes the semantics of AQuery through examples and a simple formalism. The paper describes various heuristic optimization approaches implemented in our system and demonstrates their semantic correctness. Finally, the paper compares the performance of AQuery on known benchmarks with excellent alternative systems. We show that AQuery's simple model can yield good performance, providing data scientists and developers with an efficient and intuitive platform upon which to issue order-dependent queries.

7. APPENDIX

7.1 AQuery SybaseIQ Financial Benchmark Queries

7.1.1 Query 0

```
WITH
  Target(Id, TradeDate, ClosePrice)  AS
  (SELECT
  Id, TradeDate,ClosePrice
  FROM price
  WHERE Id IN stock10 AND
  TradeDate >= startYear10 AND
  TradeDate <= startYear10 + 365 * 10)

  weekly(Id, bucket,name,low,high,mean) AS
  (SELECT
  Id,
  timeBucket,
  "weekly",
  min(ClosePrice),
  max(ClosePrice),
  avg(ClosePrice)
  FROM Target
  GROUP BY Id, getWeek(TradeDate) as
      timeBucket)

  monthly(Id,bucket,name,low,high,mean) AS
  (SELECT
  Id,
  timeBucket,
  "monthly",
  min(ClosePrice),
  max(ClosePrice),
  avg(ClosePrice)
  FROM Target
  GROUP BY Id, getMonth(TradeDate) as
      timeBucket)

  yearly(Id,bucket,name,low,high,mean) AS
  (SELECT
  Id,
  timeBucket,
  "yearly",
  min(ClosePrice),
  max(ClosePrice),
  avg(ClosePrice)
  FROM Target
  GROUP BY Id, getYear(TradeDate) as
      timeBucket)

  SELECT
  Id, bucket, name, low, high, mean
  FROM
  CONCATENATE(weekly, monthly, yearly)
  ASSUMING ASC Id, ASC name, ASC bucket
```

7.1.2 Query 1

```
WITH
  pxdata(Id,TradeDate,HighPrice,LowPrice,
    ClosePrice,OpenPrice,Volume) AS
  (SELECT
  Id, TradeDate,
  HighPrice, LowPrice,
  ClosePrice, OpenPrice,
  Volume
  FROM price
  WHERE Id IN stock1000 AND
  TradeDate >= start300Days AND
  TradeDate <= start300Days + 300)

  splitdata(Id, SplitDate, SplitFactor) AS
  (SELECT
  Id, SplitDate, SplitFactor
  FROM split
  WHERE Id IN stock1000 AND
  SplitDate >= start300Days)

  adjdata(Id, TradeDate, AdjFactor) AS
  (SELECT
  Id, TradeDate, prd(SplitFactor)
  FROM
  pxdata INNER JOIN splitdata USING Id
  WHERE TradeDate < SplitDate
  GROUP BY Id, TradeDate)

  SELECT Id, TradeDate,
  HighPrice* fill(1, AdjFactor) as
      HighPrice,
  LowPrice * fill(1, AdjFactor) as
      LowPrice,
  ClosePrice * fill(1, AdjFactor) as
      ClosePrice,
  OpenPrice * fill(1, AdjFactor) as
      OpenPrice,
  Volume / fill(1, AdjFactor) as Volume
  FROM
  pxdata FULL OUTER JOIN adjdata
  USING (Id, TradeDate)
  ASSUMING ASC Id, ASC TradeDate
```

7.1.3 Query 2

```
WITH
  pxdata(Id,TradeDate,HighPrice,LowPrice)
      AS
  (SELECT
  Id, TradeDate,
  HighPrice, LowPrice
  FROM price
  WHERE Id IN stock1000 AND
  TradeDate BETWEEN startPeriod AND
      endPeriod)

  splitdata(Id,TradeDate,SplitFactor) AS
  (SELECT
  Id, SplitDate, SplitFactor
  FROM split
  WHERE Id IN stock1000 AND
  SplitDate BETWEEN startPeriod AND
      endPeriod)

  SELECT
  Id as Id,
  TradeDate as TradeDate,
  HighPrice - LowPrice as MaxDiff
  FROM pxdata INNER JOIN splitdata
  USING (Id, TradeDate)
  ASSUMING ASC Id, ASC TradeDate
```

7.1.4 Query 3

```
select avg(ClosePrice) as avg_close_price
FROM price
WHERE Id IN SP500 AND
TradeDate = startPeriod
```

7.1.5 Query 4

```
select avg(ClosePrice) as avg_close_price
```

```
FROM price
WHERE Id IN Russell2000 AND
TradeDate = startPeriod
```

7.1.6 Query 5

```
WITH
  pxdata(Id, TradeDate, ClosePrice) AS
  (SELECT
  Id, TradeDate, ClosePrice
  FROM price
  WHERE Id IN stock1000 AND
  TradeDate >= start6Mo AND
  TradeDate < start6Mo + 31 * 6)

  splitdata(Id, SplitDate, SplitFactor) AS
  (SELECT
  Id, SplitDate, SplitFactor
  FROM split
  WHERE Id IN stock1000 AND
  SplitDate >= start6Mo)

  splitadj(Id, TradeDate, ClosePrice) AS
  (SELECT
  Id, TradeDate,
  first(ClosePrice * prd(SplitFactor))
  FROM pxdata INNER JOIN splitdata USING
      Id
  WHERE TradeDate < SplitDate
  GROUP BY Id, TradeDate)

  avgInfo(Id,TradeDate,ClosePrice,m21,m5)
      AS
  (SELECT
  Id,
  TradeDate,
  fill(pxdata.ClosePrice, splitadj.
      ClosePrice),
  avgs(21,  fill(pxdata.ClosePrice,
      splitadj.ClosePrice)),
  avgs(5,  fill(pxdata.ClosePrice,
      splitadj.ClosePrice))
  FROM pxdata FULL OUTER JOIN splitadj
  USING (Id, TradeDate)
  ASSUMING ASC Id, ASC TradeDate
  GROUP BY Id)

  SELECT *
  FROM FLATTEN(avgInfo)
  ASSUMING ASC Id, ASC TradeDate
```

7.1.7 Query 6

```
WITH
  pxdata(Id, TradeDate, ClosePrice) AS
  (SELECT
  Id, TradeDate, ClosePrice
  FROM price
  WHERE Id IN stock1000 AND
  TradeDate >= start6Mo AND
  TradeDate < start6Mo + 31 * 6)

  splitdata(Id, SplitDate, SplitFactor) AS
  (SELECT
  Id, SplitDate, SplitFactor
  FROM split
  WHERE Id IN stock1000 AND
  SplitDate >= start6Mo)

  splitadj(Id, TradeDate, ClosePrice) AS
  (SELECT
  Id, TradeDate,
  first(ClosePrice * prd(SplitFactor))
  FROM
  pxdata INNER JOIN splitdata USING Id
  WHERE TradeDate < SplitDate
  GROUP BY Id, TradeDate)

  avgInfo(Id, TradeDate, ClosePrice, m21,
      m5) AS
  (select
  Id,
  TradeDate,
  fill(pxdata.ClosePrice, splitadj.
      ClosePrice),
  avgs(21,  fill(pxdata.ClosePrice,
      splitadj.ClosePrice)),
  avgs(5,  fill(pxdata.ClosePrice,
      splitadj.ClosePrice))
  FROM pxdata FULL OUTER JOIN splitadj
  USING (Id, TradeDate)
  ASSUMING ASC Id, ASC TradeDate
  GROUP BY Id)

  SELECT
  Id,
  TradeDate as CrossDate,
  ClosePrice
  FROM FLATTEN(avgInfo)
  WHERE Id = prev(Id) AND
  // cross over
  (prev(m5) <= prev(m21) & m5 > m21)
  OR
  // cross under
  (prev(m5) >= prev(m21) & m5 < m21)
```

7.1.8 Query 7

```
// encode execution of our trading
    strategy
FUNCTION
execStrategy(alloc,mavgday,mavgmonth,px) {
  buySignal := mavgday > mavgmonth;
  alloc * prd(
        CASE maxs(buySignal)
          WHEN TRUE THEN
            CASE buySignal
              WHEN TRUE THEN 1 / px
              ELSE px
            END
          ELSE 1
        END)
}

WITH
  pxdata(Id, TradeDate, ClosePrice) AS
  (SELECT
  Id, TradeDate, ClosePrice
  FROM price
  WHERE Id IN stock10 AND
  TradeDate >= max(TradeDate) - 365)

  splitdata(Id, SplitDate, SplitFactor) AS
  (SELECT Id, SplitDate, SplitFactor
  FROM split
  WHERE Id IN stock10 AND
  SplitDate >= max(SplitDate) - 365)

  splitadj(Id, TradeDate, ClosePrice) AS
  (SELECT
  Id, TradeDate,
  first(ClosePrice * prd(SplitFactor))
  FROM
  pxdata INNER JOIN splitdata USING Id
  WHERE TradeDate < SplitDate
  GROUP BY Id, TradeDate)

  adjpxdata(Id, TradeDate, ClosePrice) AS
  (SELECT
  Id, TradeDate,
  fill(pxdata.ClosePrice, splitadj.
     ClosePrice) as ClosePrice
  FROM pxdata FULL OUTER JOIN splitadj
  USING (Id, TradeDate))

  movingAvgs(Id, TradeDate, ClosePrice,
     m21day, m5month) AS
  (SELECT
  Id, TradeDate,
  ClosePrice,
  avgs(21, ClosePrice),
  avgs(160, ClosePrice)
  FROM adjpxdata
  ASSUMING ASC Id, ASC TradeDate
  GROUP BY Id)

  simulated AS (
  SELECT Id,
  execStrategy(10000, m21day, m5month,
     ClosePrice) as result,
  last(m21day) > last(m5month) as
     stillInvested
  FROM FLATTEN(movingAvgs)
  WHERE Id = prev(Id) AND
  (prev(m5month) <= prev(m21day) & m5month
      > m21day)
  OR
  (prev(m5month) >= prev(m21day) & m5month
      < m21day)
  GROUP BY Id  )


  latestPxs AS
  (SELECT *
   FROM  adjpxdata
   WHERE TradeDate=max(TradeDate))

  SELECT
  sum(
    fill(10000, result * CASE WHEN
        stillInvested THEN ClosePrice ELSE
         1 END)
  ) as stock_value
  FROM latestPxs FULL OUTER JOIN simulated
  USING Id
```

7.1.9 Query 8

```
FUNCTION covariance(x, y) {
  xmean := avg(x);
  ymean := avg(y);
  avg((x - xmean) * (y - ymean))
}

FUNCTION sd(x) {
  sqrt(covariance(x, x))
}

FUNCTION pairCorr(x, y) {
  covariance(x, y) / (sd(x) * sd(y))
}
<q>pairCorrEach:pairCorr'</q>

WITH
  stocksGrouped(Id, ClosePrice) AS
  (SELECT
  Id, ClosePrice
  FROM price
  ASSUMING ASC Id, ASC TradeDate
  WHERE Id IN stock10 AND
  TradeDate >= startYear10 AND
  TradeDate <= startYear10 + 365 * 2
  GROUP BY Id)

  pairsGrouped(Id1, Id2, ClosePrice1,
     ClosePrice2) AS
  (SELECT
  st1.Id, st2.Id,
  st1.ClosePrice, st2.ClosePrice
  FROM
  stocksGrouped st1, stocksGrouped st2
  GROUP BY st1.Id, st2.Id)

  corrTable(Id1, Id2, corrCoeff) AS
  (SELECT
  Id1, Id2,
  pairCorrEach(ClosePrice1, ClosePrice2)
  FROM pairsGrouped
  WHERE Id1 != Id2)

  SELECT *
  FROM corrTable
  ASSUMING ASC corrCoeff
```

7.1.10 Query 9

```
WITH
  DateInfo(startYear, startDate) AS
  (SELECT
  getYear(max(TradeDate) - 365 * 3), max(
     TradeDate) - 365 * 3
  FROM price)

  // we introduce a dummy column to avoid
     having q
  // group on distinct
  splitdata(Dummy, Id) AS
  (SELECT 1, distinct(Id)
  FROM split
  WHERE Id IN Russell2000 AND
  getYear(SplitDate) >= first(DateInfo("
     startYear")))

  nosplit_avgpx AS
  (SELECT
  Id, year, avg(ClosePrice) as avg_px
  FROM price
  WHERE Id IN Russell2000 AND
  TradeDate >= first(DateInfo("startDate")
     ) AND
  Id NOT IN splitdata("Id")
  GROUP BY Id, getYear(TradeDate) as year)

  divdata AS
  (SELECT
  Id, year, sum(DivAmt) as total_divs
  FROM dividend
  WHERE Id IN Russell2000 AND
  getYear(AnnounceDate) >= first(DateInfo(
     "startYear")) AND
  Id NOT IN splitdata("Id")
  GROUP BY Id, getYear(AnnounceDate) as
     year)

  SELECT
  Id, year, avg_px,
  total_divs, total_divs / avg_px as yield
  FROM nosplit_avgpx INNER JOIN divdata
  USING (Id, year)
```

8. REFERENCES

[Cass(2015 (accessed November 7, 2015))] S. Cass. *The 2015 Top Ten Programming Languages*, 2015 (accessed November 7, 2015). URL http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages.

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

[Elmasri and Navathe(2014)] R. Elmasri and S. B. Navathe. *Fundamentals of database systems*. Pearson, 2014.

[Influxdb(2013-2015 (accessed November 7, 2015)a)] Influxdb. *Influxdb Github Issues: Add support for custom functions #68*, 2013-2015 (accessed November 7, 2015)a. URL https://github.com/influxdb/influxdb/issues/68.

[Influxdb(2013-2015 (accessed November 7, 2015)b)] Influxdb. *Influxdb Github Issues: Add aggregate function top #409*, 2013-2015 (accessed November 7, 2015)b. URL https://github.com/influxdb/influxdb/issues/409.

[Influxdb(2015 (accessed November 6, 2015))] Influxdb. *InfluxDB: Overview*, 2015 (accessed November 6, 2015). URL https://influxdb.com/docs/v0.9/introduction/overview.html.

[Kersten et al.(2011)Kersten, Zhang, Ivanova, and Nes] M. Kersten, Y. Zhang, M. Ivanova, and N. Nes. Sciql, a query language for science applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 1–12. ACM, 2011.

[Lerner(2003)] A. Lerner. *Querying Ordered Databases with Aquery*. PhD thesis, Ph.D. Thesis, Ecole Nationale Superieure de Telecommunications, ENST-Paris, 2003.

[Lerner and Shasha(2003)] A. Lerner and D. 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.

[Mon(2014 (accessed November 18, 2015))] *Embedded R in MonetDB*. MonetDB, 2014 (accessed November 18, 2015). URL https://www.monetdb.org/content/embedded-r-monetdb.

[Nes and Kersten(2012)] S. I. F. G. N. Nes and S. M. S. M. M. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *Data Engineering*, page 40, 2012.

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

[Oliphant(2006)] T. E. Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[Oracle(2015 (accessed November 15, 2015))] Oracle. *MySQL: Handling of GROUP BY*, 2015 (accessed November 15, 2015). URL http://dev.mysql.com/doc/refman/5.7/en/group-by-handling.html.

[pandas development team(2015 (accessed November 7, 2015))] pandas development team. *pandas: powerful python data analysis toolkit (version 0.17.0)*, 2015 (accessed November 7, 2015). URL http://pandas.pydata.org/pandas-docs/stable/.

[Pelkonen et al.(2015)Pelkonen, Franklin, Teller, Cavallaro, Huang, Meza, T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan. Gorilla: a fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8(12): 1816–1827, 2015.

[Raasveldt(2015 (accessed November 06, 2015))] M. Raasveldt. *Embedded Python/NumPy in MonetDB*. MonetDB, 2015 (accessed November 06, 2015). URL https://www.monetdb.org/blog/embedded-pythonnumpy-monetdb.

[SAP(2008 (accessed November 8, 2015))] SAP. *Sybase IQ 15.3: Understanding User-Defined Functions*, 2008 (accessed November 8, 2015). URL http://infocenter.sybase.com/archive/index.jsp?topic=/com.sybase.dc00792_0100/html/rapug/rapug27.htm.

[SAP(2013 (accessed November 8, 2015)a)] SAP. *Introduction to SAP Sybase IQ: SAP Sybase IQ 16.0*, 2013 (accessed November 8, 2015)a. URL http://infocenter.sybase.com/help/topic/com.sybase.infocenter.dc38159.1600/doc/pdf/iqintro.pdf.

[SAP(2013 (accessed November 8, 2015)b)] SAP. *SAP Sybase IQ 16 In-Database Analytics Option Technical Overview*, 2013 (accessed November 8, 2015)b. URL http://www.sdn.sap.com/irj/scn/go/portal/prtroot/docs/library/uuid/30faeab1-7682-3010-8fb7-975050fa89ee?overridelayout=true.

[SAP(2015 (accessed November 8, 2015))] SAP. *Sybase RAP*, 2015 (accessed November 8, 2015). URL http://www.sap.com/pc/tech/database/software/trading-analytics-platform/index.html.

[Seshadri et al.(1996)Seshadri, Livny, and Ramakrishnan] P. Seshadri, M. Livny, and R. Ramakrishnan. *SEQ: Design and implementation of a sequence database system*. Citeseer, 1996.

[Sta(2015 (accessed November 7, 2015))] *pandas tag info*. StackOverflow, 2015 (accessed November 7, 2015). URL http://stackoverflow.com/tags/pandas/info.

[StumpleUpon(2015 (accessed November 6, 2015))] StumpleUpon. *FAQ*, 2015 (accessed November 6, 2015). URL http://opentsdb.net/faq.html.

[the pandas development team(2011 (accessed November 18, 2015))] the pandas development team. *Vbench performance benchmarks for pandas*, 2011 (accessed November 18, 2015). URL http://pandas.pydata.org/pandas-docs/vbench/.

[Whitney(2009 (accessed November 6, 2015))] A. Whitney. *Abridged Q Language Manual*, 2009 (accessed November 6, 2015). URL http://www.kx.com.

[Wickham(2009)] H. Wickham. *ggplot2: elegant graphics for data analysis*. Springer New York, 2009. ISBN 978-0-387-98140-6. URL http://had.co.nz/ggplot2/book.

[Yang et al.(2014)Yang, Tschetter, Léauté, Ray, Merlino, and Ganguli] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. 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.