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 query 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 more 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

Programmers of applications that depend on ordered events in time 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 "or-

SIGMOD '16 San Francisco, CaliforniaUSA

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

DOI: 10.1145/1235

der 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. We compare with both.

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 ([pan(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 $\bowtie_{w_i,...,w_j}(t_i, ..., t_n)$.
- 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 $\sigma_{w_i,...,w_j}$.
- 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 (we explain the approach in section 4).

```
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, $c_i, ..., c_j$. 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

$network(base\_station, date, hour\_stamp, num\_cons)$

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

Figure 1: A networking query

represents network loads across various base stations

In this simple example, (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, as show in Figure 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.

Intuitively, 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.

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.

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.

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.

We note the two possible interpretations of this query, first assuming we treat the selection predicates as a bag, and then as a sequence:

- Bag: select all tuples in stocks that have "IBM" as a ticker **and** saw a price increase. Given our tuples described above, our selection results in the latter 2 tuples.
- Sequence: select all tuples that have "IBM" as ticker **and then** select all tuples that saw a price increase. This results in just the last tuple being selected.

The AQuery semantics are that the predicates are applied in the order they appear. So, tuples having "IBM" as ticker are found and then, semantically, in a second pass over the result, the prev is applied. This will yield just the last tuple whereas 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.

3.4 Group-By and Having-Clauses

A group-by clause specificies 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 a 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 apply the *each* modifier to the *sum* aggregate, allowing us to sum over the nested arrable, resulting in the aggregated table shown.

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

3.5 Projection-clause

The projection clause takes a series of expressions $e_i(c_j, ..., c_k)$ and projects them in a semantically order-preserving way

Figure 2: Grouping generates nested arrables

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 3: *each* allows us to adapt function behavior for nested arrables

```
SELECT sum(num_conns)
FROM network
GROUP BY base_station
```

base_station |	num_conns
1 |	30
2 |	50

over an arrable. The operations in the projection-clause may be order-dependent as in the 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

In an effort to provide a robust 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, simple syntax for creating query-local temporary tables allows users the same level of expressiveness. 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
```

becomes

```
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
```

y14's scope is limited to statements that follow it within the *WITH* and up until the main (non-local) query, which appear last.

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 broader heuristic optimization strategy employed when analyzing a new query provided by the user.

4.1 Eliminating Sorts

An arrable t can be 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 can eliminate the sort. This is the simple optimization mentioned in the introduction.

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. *Given 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 $I = (i_{p_1}, ..., i_{p_k})$, reordering predicates in W within the subsequences demarcated by I maintains the same semantics of selection as executing the predicates in sequential order.*

PROOF. Reordering between indices in I reduces to the semantics of treating a sequence of selections as a bag of selections, as there is no inter-row dependency (i.e no order dependence). Since the selections are chained by conjunction, their satisfaction is independent of the order of the selections within that bag. Thus this strategy maintains the original semantics.

□

4.3 Delayed Sorting

Because order-preserving operations often lead to slower implementations than non-order-preserving ones, we tend to 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, and we:

- 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 for the correct execution of q, where correct means an execution resulting in exactly the same result as if we had sorted the entire arrable.

Given a query $q(c_1, ..., c_j)$ on table t, we sort the relevant columns, C_{od}, and ignore the remainder. To the extent that $|C_{od}|$ is smaller than $|t|$, this can result in significant performance gains. Furthermore, note that C_{od} need not include all columns used in the query, but rather possibly a subset of these (including $\emptyset$).

This of course implies we must be able to infer columns that require sorting.

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 we have not yet encountered an order-dependent predicate that requires our data to be sorted. The third predicate is order-dependent, and thus at this point AQuery must collect all attributes

Figure 4

```
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) >= 1e6
GROUP BY  getYear(TradeDate)
```

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.

Algorithm 1: Sorting Necessary Attributes

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).
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 remaining clauses
  Evaluate selections $W_{i>=j}$ according to Theorem 4.1 and remaining clauses
end
case *G or H have any order-dependent expression*
  Evaluate all selections W according to Theorem 4.1
  Sort all attributes in remaining clauses
  Evaluate G and remaining 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

THEOREM 4.3. *Given an AQuery query q, following Algorithm 1 is equivalent to sorting the arrable from the start and making all operations order-preserving.*

PROOF. The proof is simple, we only evaluate expressions/clauses that don't require order up to the first order-dependent operation. Clearly, if they are order-independent, they result in the same arrable with/without sorting up to a permutation. Sorting all attributes necessary from this point onwards is equivalent to sorting the entire arrable, as we only use a subset of attributes. □

4.5 From Cross to Join and Choosing Joins

A common optimization in traditional database system involves taking a cartesian product (aka cross) and selection operations to create a join operation [Elmasri and Navathe(2014)]. Furthermore, joins are often commuted when possible to create more efficient operations.

In the case of AQuery, joins using *INNER JOIN/FULL OUTER JOIN* expressions are executed in the order provided by the user, but joins performed implicitly by the cross operator (,) and selection predicates can be reorganized by the system. Given AQuery's column orientation and the semantics of selections explained above, the predicates available to perform the join(s) must come before the first inter-row-dependent selection. We employ a greedy approach detailed in Algorithm 2 to order these joins. The general idea is: greedily select the join that allows us to execute the largest number of equality-based selections on the resulting arrable.

Given that these predicates do not depend on order, their execution on subsets of the completely joined data set results in an equivalent arrable to crossing and then evaluating the predicates.

Algorithm 2: Greedy Selection of Join Order for Cross Product Optimization

Input : A set of arrables $T = \{t_1, ...t_n\}$ combined with $\times$ operator, and a sequence of equality-based predicates $W = (w_1, ..., w_j)$
Output : J join order
Initialize: $J = T$
repeat
  if $\{w|$ *can join remaining arrables and is equality based* $\} = \emptyset \wedge |J| > 1$ **then**
    $J = J_1 \times J_2 ... \times J_{|J|}$
  else
    chosen join $= \arg\max_{w,i,j} = |w|$ in $t_i \bowtie_w t_j$, where $t_i, t_j \in J$ and all w are equality selections
    $J = J \setminus \{t_i, t_j\}$ from chosen join
    $J = J \cup t_i \bowtie_w t_j$ from chosen join
    $W = W \setminus w$ from chosen join
  end
until $|J| = 1$;
return J

THEOREM 4.4. *If the set of tables implicitly joined by cartesian products and selection predicates has size n, the Algorithm 2 has time complexity* $O(n^3)$

Given that most from-clauses involving implicit joins typically consist of only a few arrables, the time complexity is not onerous. Furthermore, this decision is performed at compile time, and imposes no additional costs on the query execution.

4.6 Full Strategy Sketch

Given a new query q, the heuristic optimization approach is sketched out in Algorithm 3.

5. EXPERIMENTS

In all our experiments, we use a measure called average response time ratio to evaluate performance of other systems relative to AQuery. An average response time ratio above 0

Algorithm 3: A Sketch of Optimizations

Input : An AQuery query q
Output : Heuristically optimized query plan
if *from-clause involves cross product* **then**
  Execute Algorithm 2, modify selection predicate sequence accordingly.
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.

means the system is on average slower than AQuery, while conversely an average response time ratio below 0 means the system is on average faster than AQuery.

All experiments are run in a single-user setting, running on a MacBook Air with a 2-Core 1 .7 GHz Intel Core i7 processor, with 8GB of memory, with the exception of the Sybase/AQuery comparison. The performance for the latter comparison is measured in 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 0.17.0
- Numpy 1.10.1
- Python 2.7.5
- MonetDB version 1.7, built from the *pyapi* branch that allows for embedded Python
- q 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 randomly simulated data. Some of the queries require random parameters, for example 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 elements) and randomly generate the query parameters multiple times. We present the average reponse time ratio for the various systems relative to AQuery in Figures 5,6 and 7.

We note that AQuery is faster than Pandas on the financial benchmark (as indicated by the higher average response ratio in Figure 5, significantly outperforming across data sizes and queries.

Similarly, AQuery is faster than MonetDB in all queries at the 100k and 1M data ranges. Once we perform experiments with a stock history of 10M observations, AQuery outperforms in 9 out of 10 queries.

Table 1: Financial Queries Description

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.

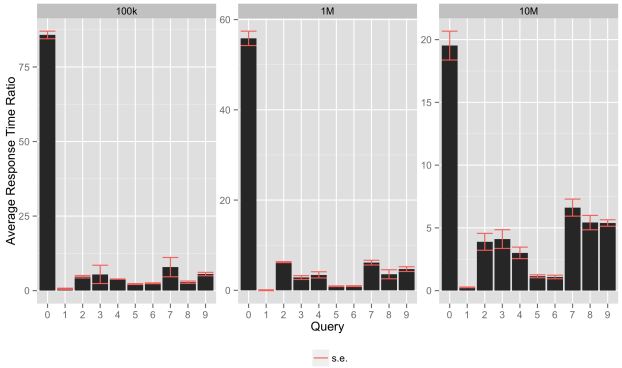

Figure 5: Pandas Comparison on Sybase IQ Financial Benchmark

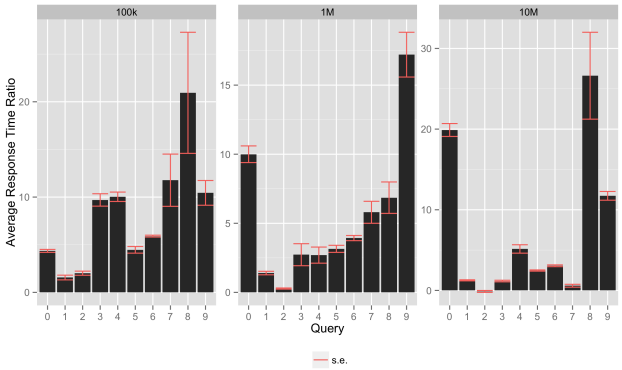

Figure 6: MonetDB Comparison on Sybase IQ Financial Benchmark

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 very wide standard errors. However, on average AQuery outperforms in 8 our of 10 queries.

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))]. While many of these are great to track and tune the library's historical development, they are not of immediate interest in a system comparison to AQuery. Given this, we pick a subset of operations that represent common tasks in data science, for example: subsetting, grouping, summarizing, and merging data, amongst others.

Figure 8 shows the performance results relative to AQuery's own execution 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 the Pandas to AQuery execution time ratio across each data size. Table 2 provides brief descriptions of each query.

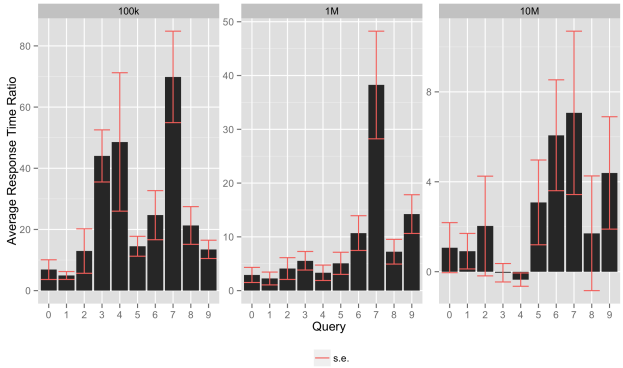

Figure 7: Sybase IQ Comparison on Sybase IQ Financial Benchmark

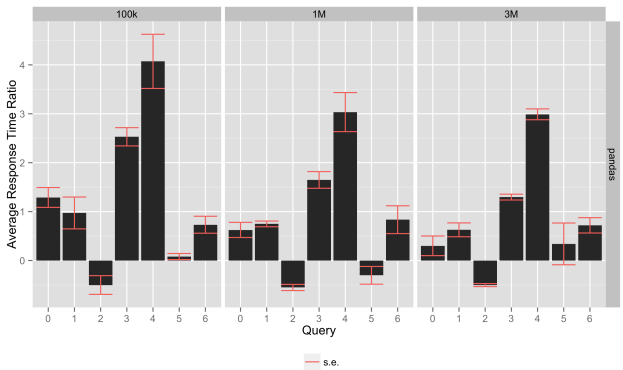

Figure 8: Pandas Benchmarking Operations

We note that AQuery outperforms substantially in 5 of 7 cases in most of the data sets we evaluated. However, AQuery's advantage in the remaining queries is significant, with some queries running up to 3 times slower 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 alternative 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 quan-

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

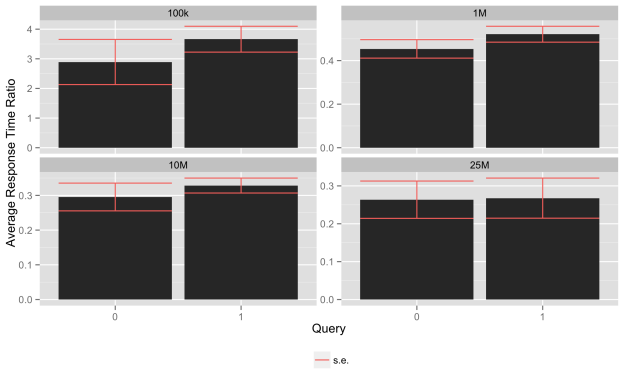

Figure 9: MonetDB Quantile Benchmarking

tile 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 ratio. 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. The response time ratios 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 substantially in larger data sets. However, we remind the reader that the AQuery implementation uses a naive quantile function which sorts the data completely.

6. CONCLUSION

We have explored a query language and system designed to tackle the challenges of analyzing data that require order. Expressing many of the common and necessary operations is made difficult by traditional relational database systems. AQuery shows that order can be introduced without complicating the language and computational model. We explored in detail the semantics associated with AQuery and the particularities that arise from order. We showed various of the heuristic optimization approaches implemented in our system and prove their semantic correctness. Finally, we compared performance on various benchmarks with various excellent and popular alternative systems. We show that AQuery's simple model can yield good performance, providing data scientists and developers with an alternative to express their ordered experiments.

7. REFERENCES

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

[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.

[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.

[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.

[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.