AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments

Alberto Lerner
Ecole Nationale Superieure
de Telecommunications
Paris - France
lerner@cs.nyu.edu

Dennis Shasha
New York University
New York - USA
shasha@cs.nyu.edu

TR2003-836
Courant Intitute of Mathematical Sciences
New York University

AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments*

Alberto Lerner†
Ecole Nationale Superieure
de Telecommunications
Paris - France
lerner@cs.nyu.edu

Dennis Shasha
New York University
New York - USA
shasha@cs.nyu.edu

Abstract

An order-dependent query is one whose result (interpreted as a multi-set) changes if the order of the input records is changed. In a stock-quotes database, for instance, retrieving all quotes concerning a given stock for a given day does not depend on order, because the collection of quotes does not depend on order. By contrast, finding the five price moving average in a trade table gives a result that depends on the order of the table. Query languages based on the relational data model can handle order-dependent queries only through add-ons. SQL:1999, for example, permits the use of a data ordering mechanism called a "window" in limited parts of a query. As a result, order-dependent queries become difficult to write in those languages and optimization techniques for these features, applied as pre- or post-enumerating phases, are generally crude. The goal of this paper is to show that when order is a property of the underlying data model and algebra, writing order-dependent queries in a language can be natural as is their optimization. We introduce AQuery, an SQL-like query language and algebra that has from-the-ground-up support for order. We also present a framework for optimization of the order-dependent queries categories it expresses. The framework is able to take advantage of the large body of query transformations on relational systems while incorporating new ones described here. We show by experiment that the resulting system is orders of magnitude faster than current SQL:1999 systems on many natural order-dependent queries.

*Work supported in part by U.S. NSF grants IIS-9988636 and N2010-0115586.

†This work was done while this author was visiting NYU

1 Introduction

Querying ordered data arises naturally in applications ranging from finance to molecular biology to network management. A financial analyst may be interested in moving averages within or correlations among price time series. A biologist may be interested in frequent nucleic acid motifs. A network manager may be interested in packet flow statistics. Several extensions to SQL have been suggested that are able to express such order-dependent queries [16, 2, 14, 11].

SQL:1999, through its OLAP amendment, is the first such language to gain commercial acceptance [11]. It provides this facility through the following new order-aware features: ordering in the SELECT clause (OVER...WINDOW construct), a notion of row numbering, and an ARRAY data type. Unfortunately, these extensions, however expressive, result in complex formulations of even simple queries. The complex formulation is then difficult to optimize.

1.1 Motivational Queries and Problems

Consider the schema Trades(ID, tradeDate, volume, price, timestamp), where ID is the ticker symbol of a traded security, timestamp identifies the date and time of a particular trade (tradeDate is a human-readable form of the day portion), volume is the number of shares that exchanged hands in the trade, and price is the price of the trade.

Consider the following query: for a given stock and a given date, find the best profit one could obtain by buying it and then selling it later that day (short selling – in which an item is sold before it is bought – is disallowed). Algorithmically, the solution is straightforward: compute the profit resulting from selling at each trade t by subtracting the price at t by the mini-

mum price seen up until t. The answer to the query is the maximum of these profits. In SQL:1999 this query would look like[1]:

```
[SQL:1999]
SELECT   max(running_diff)
FROM     (SELECT ID, tradeDate
                 price - min(price) OVER
                    (PARTITION BY ID, tradeDate
                    ORDER BY timestamp
                    ROWS UNBOUNDED PRECEDING)
                    AS running_diff,
          FROM    Trades ) AS t1
WHERE    ID = 'ACME' AND tradeDate = '02/18/03'
```

The nesting here is necessary because a windowed function (min(price) OVER ...) cannot be an argument of an aggregating function (max(running_diff)). One opportunity for optimization is to push the outer query's selection (ID='ACME' AND tradeDate='02/18/03') to the inner query. Although possible in this particular case, pushing a selection over a projection containing a generic expression involving windowed functions (SELECT ... price - min(price) OVER ...) requires deep analysis. As of this writing, the commercial optimizers we have tested do not do such an optimization.

As another example, consider the schema Packets(pID, src, dest, length, timestamp), where pID identifies a packet exchanged between a source (src) and a destination (dest) host. Length refers to the size of the packet and timestamp to the moment this packet was exchanged. A "flow" from a source s to a destination d ends if there is a 2-minute gap between consecutive packets from s to d. A network administrator wants to know the count of packets and their average length within each flow. This is a simplified version of a real network monitoring query [3]. Basically, this query needs to group packets of each flow, and compute the count and average needed. Finding the flows is very hard to express, though, because it involves order. In SQL:1999 that query would look like:

```
[SQL:1999]
WITH
 Prec (src, dest, length, timestamp, ptime) AS
 (SELECT  src, dest, length, timestamp,
          avg(ts) OVER
                   (PARTITION BY src,dest
                   ORDER BY timestamp
                   ROWS BETWEEN 1 PRECEDING
                   AND 1 PRECEDING)
 FROM     Connections),
 Flow (src, dest, length, timestamp, flag) AS
 (SELECT  src, dest, length, timestamp,
          CASE WHEN timestamp-ptime > 120 THEN 1
               ELSE 0
          END
 FROM     Prec),
 FlowID (src, dest, length, timestamp, fID) AS
 (SELECT  src, dest, length, timestamp,
          sum(flag) OVER
                   (ORDER BY src, dest, timestamp
                   ROWS UNBOUNDED PRECEDING)
 FROM     Flow)
SELECT   src, dest, avg(length), count(timestamp)
FROM     FlowID
GROUP    BY src, dest, fID
```

The first sub-query, Prec, creates a new column, ptime, containing the previous packet's timestamp within each source and destination. Next, the Flow sub-query adds a flag column that is turned true (1) at each packet whose difference to the preceding one exceeds two minutes; otherwise the flag is turned to false (0). Next, the FlowID sub-query sums these flags cumulatively, creating an auxiliary flow ID, fID. The main query uses these results.

Optimization of this query should seek to reduce the work required by PARTITION BYs and ORDER BYs. That is hard because the windows defined in Prec and in FlowID have slightly different sliding parameters. The commercial optimizer we tested made it with two sorts before the grouping was done.

The problem with these queries is then two-fold. Their expression in SQL:1999 is complex and therefore both hard to read and difficult to optimize[2].

We have designed a data model, language, and system where expressions dependent on order are natural to write, fostering the exposure of idioms that the optimizer can exploit. We explain the data model and language, then illustrate the good performance of the system by experiment.

1.2 AQuery: First Look

In our data model, tables are not viewed as multisets, but rather as ordered entities that we call *arrables* (standing for array-tables). An arrable's ordering may be defined at creation time using an ORDERED BY clause and can later be altered.

Our query language, *AQuery*, is a semantic extension of the multiset relational model (i.e., SQL 92) without order-related extensions, but including the classical SELECT-FROM-WHERE-GROUP BY-AGGREGATE-HAVING clauses. The main extensions are based on a new clause called ASSUMING ORDER which defines the order of the arrables identified in the FROM clause. Predicates and expressions, in whichever clause they are, can count on the order defined by the ASSUMING clause, leading to the natural expression of order-dependent queries as we show in section 2.

[1] The queries shown here use somewhat advanced SQL:1999 features. The unfamiliar reader is encouraged to refer to [11].

[2] We will argue in section 5 that several optimization techniques for specific cases, such as sort elimination, do exist, but have so far been discussed in isolation.

It is up to the optimizer to match the input arrables' existing order with a query's ASSUMING ORDER and to decide whether and when further sorting is necessary, as we show in section 3. This flexibility stems from the fact that in the AQuery algebra, each operator has an order-preserving and an order-oblivious variation. The transformations use equivalences to move the sort over the operators possibly entailing a change in variation. We show that this schema is able to integrate new transformations with classical ones.

Our experiments, in section 4, show orders of magnitude differences between AQuery's and current commercial SQL:1999's renditions of queries and show that the transformations generate highly efficient plans.

In section 5, we identify several languages that also considered order as a first-class concept and from which AQuery draws inspiration. In the same section, we also comment on the sources for some of our optimization techniques.

Finally, section 6 closes this paper stating our conclusions.

2 Data Model and Algebra

2.1 Arrables and Order

Definition 1 (Arrables). Let $\mathcal{T}$ be a set of types in which each $t \in \mathcal{T}$ corresponds to a basic type (e.g., integer, boolean, etc) or to a one-dimensional array made up of elements of basic types. Let A be a finite, unbounded array of elements of a type $t \in \mathcal{T}$. The cardinality of A is its number of elements. The k-th element of A is denoted by $A[k]$, and k is said to be an *index* or *position* in A. Indexes start at 0. An *arrable* r is a collection of named arrays $A_1, \cdots, A_n$ that have the same cardinality, and such that each A_i, $1 \leq i \leq n$, is of a type of $\mathcal{T}$. □

Definition 2 (Arrable Indexing). The k-th record of an arrable r is formed by the k-th element of each of r's component arrays. This operation, denoted *indexing*, is represented as $r[k] = \langle A_1[k], \cdots, A_n[k] \rangle$. □

Observe that if $A_1, \cdots, A_n$ are all vectors (i.e., their elements are all scalars), arrables have the appearance of tables as we know them. We will show shortly how arrables having vector elements can be used. Because an arrable consists of arrays and arrays are ordered, an arrable is ordered.

Definition 3 (Ordered by). An arrable r may be ordered by a subset of its arrays, $B_1, \cdots, B_m \subseteq A_1, \cdots, A_n$. If the ordering is ascending and k_1 and k_2 are two indexes of r and $k_1 < k_2$, then either (i) $B_1[k_1] = B_1[k_2], \cdots, B_m[k_1] = B_m[k_2]$ or (ii) there exists a i, $1 \leq i \leq m$, such that $B_i[k_1] < B_i[k_2]$ and if $i > 1$ then $B_1[k_1] = B_1[k_2], \cdots, B_{i-1}[k_1] = B_{i-1}[k_2]$. The definitions are symmetric for descending orders, but for purposes of explanation, we will consider order to be ascending throughout this section. □

For instance, if we wanted Trades to be ordered by ID, timestamp, we would say: Trades(ID, tradeDate, volume, price, timestamp) ORDERED BY ID, timestamp[3].

Definition 4 (Order-Equivalence). Two arrables r and s are *order-equivalent* with respect to the list of attributes $B_1, \cdots, B_m$, denoted $r \equiv_{B_1,\cdots,B_m} s$, if both r and s are ordered by $B_1, \cdots, B_m$ or if the latter is a prefix of both orderings and there exists a permutation of r that is identical to s. (We need the permutation because several rows may have the same $B_1, \cdots, B_m$ values.) When r and s are multi-set-equivalent (i.e., they contain the same set of tuples and for each tuple in the set, they contain the same number of instances of that tuple), we say that $r \equiv_{\{\}} s$. □

2.2 Array-Typed Expressions

A row-oriented language (i.e., one in which variables range over tuple values) requires joins or other auxiliary constructs to express calculations that depend on relationships between different tuples. For example, consider a query to find the pair-wise time-consecutive price differences of a stock in a schema such as Trades.

In SQL:1999, a construct called WINDOW can define a set of rows involved in the calculations of every row being considered. To find a previous price a WINDOW would establish an appropriate sort order, timestamp in our case, and the number of rows to be considered, here just the previous one. In [16, 14], order is intrinsic to the sequence and ordered-relation data models, respectively. Even so, obtaining a previous price would entail self-joining Trades with a shifted-by-one-position copy of it using the operators "offset" [16] and "SHIFT" [14].

By contrast, AQuery algebra's and language's semantics are column-oriented in that variables do not range over individual arrays' elements – *variables are bound to entire arrays at once.* For instance, the above pair-wise difference can be captured by an expression – price - prev(price). No additional construct is needed. This expression should be interpreted as follows. Price is an array. Next, prev of an array A is an array such that $\text{prev}_A[i] = A[i-1]$ if $i > 0$ and null otherwise. Finally, for arrays A and B such that $|A| = |B|$, minus (-) is element-wise subtraction. sf Deltas(price) is the abbreviation of the above expression. Hence, array expressions express inter-tuple calculations that otherwise would require additional language constructs.

AQuery provides built-in functions that take advantage of its column-oriented semantics. (They also are excellent targets for optimization, as discussed in the next section.) Functions that compute running aggregates follow this "s"-as-suffix pattern. A running minimum over an array A, mins(A), is $\text{mins}_A[i] = \min(A[i], \text{mins}_A[i-1])$ for $0 < i < |A|$ or $A[i]$ for $i =$

[3] We are omitting the typing information here for convenience. A complete definition would include also NULL and referential integrity information.

0. A running sum over an array A, denoted sums(A), is $\text{sums}_A[i] = A[i] + \text{sums}_A[i-1]$ for $0 < i < |A|$, or $A[i]$ for $i = 0$. Some running aggregates can be computed over sliding windows. For instance, a running average using a fixed-sized window of w positions over an array A is denoted avgs(w, A) and is defined as $\text{avgs}_{w,A}[i] = \text{sum}(A[i-(w-1)]..A[i])/w$, for $w-1 \leq i < |A|$ or $\text{sum}(A[0]..A[i])/i$ for $0 \leq i < w-1$[4].

Functions that reduce an array's cardinality are called edge functions (i.e., they keep either the beginning or the end of an array). For instance, first n positions of an array A, denoted first(n, A), is $\text{first}_{A,n} = A[0..n-1]$. Similarly, $\text{last}_{A,n} = A[|A|-n..|A|-1]$. A more complete set of functions can be found at [10].

Definition 5 (Order-Dependency). An expression e that maps a list of arrays to an array is said to be *order-independent* if for all operand arrays A_i, $1 \leq i \leq m$, where m is the degree of the expression, and for any corresponding permutations A_i^{perm} $e(A_1, \cdots, A_m) \equiv_{\{\}} e(A_1^{perm}, \cdots, A_m^{perm})$. For example, min() is order-independent. An expression that is not order-independent is *order-dependent*. For example, running minimum, mins(), is order-dependent. □

2.3 An Algebra and Query Language

The AQuery algebra supports the operators of the relational algebra. Such operators take array-expressions as arguments, and most have order-preserving and order-oblivious variations. Some operators are order-generating in that they output their results in a specified order, e.g., ascending by name.

Definition 6 (Projection). Let r be an arrable and $e = e_1, \cdots, e_m$ be a list of expressions involving r's arrays, such that $|e_1| = \cdots = |e_m|$. An order-preserving projection of r over e, denoted $\pi_e^{op}(r)$, is defined as follows[5]:

```
projection(e,r)
1. s:= empty arrable
2. for i = 0 to |r|-1
3.     s[i] := <e1[i],···,em[i]>
4. end for
5. output s
```

If any e_i is order-dependent, the projection is said to be order-dependent. An order-oblivious projection, denoted $\pi_e(r)$, is one where $\pi_e(r) \equiv_{\{\}} \pi_e^{op}(r)$, i.e. they are multi-set equivalent but need not be in the same order. □

Definition 7 (Selection). Let r be an arrable and p be a predicate mapping a list of r's arrays into an array of booleans, such that $|r| = |p|$. An order-preserving selection of r over p, denoted $\sigma_p^{op}(r)$, is defined as follows. (Note that we are using lists and arrays interchangeably.)

```
selection(p,r)
1. s:= empty arrable
2. for i = 0 to |r|-1
3.     if p[i] is true
4.         append r[i] to s
5.     end if
6. end for
7. output s
```

As for a projection, a selection can be order-dependent, and either order-preserving or order-oblivious. □

An arrable's sort order is a property that can be manipulated on a per-query basis.

Definition 8 (Sort). Sorting an arrable $r(A_1, \cdots, A_n)$ over $B_1, \cdots, B_m \subseteq A_1, \cdots, A_n$, denoted $\text{sort}_{B_1,\cdots,B_m}(r)$, is order-equivalent with respect to $B_1, \cdots, B_m$ to having r declared as ORDERED BY $B_1, \cdots, B_m$. Sort is necessarily an order-generating operation. □

Having defined these operators, it is now possible to show AQuery's rendition of the best-profit query.

```
SELECT    max(price - mins(price))
FROM      Trades
          ASSUMING ORDER timestamp
WHERE     ID = 'ACME' AND tradeDate = '02/18/03'
```

AQuery clauses (SELECT, FROM, ...) are processed in the same order as SQL's. Semantically, ASSUMING ORDER is translated to a sort after the FROM clause is computed. It enforces the desired order for the query and it obliges future clauses (WHERE, GROUP BY, HAVING, and SELECT) to be translated to order-preserving algebra variations. (This is the required semantics. Optimization may avoid performing the sort this early as we show later.)

Note that due to the column-oriented semantics of AQuery, the mins() function is called only once and takes the whole price vector as an argument. Subtracting a vector (mins(price)) from another (price) with the same cardinality is a standard array expression [1] as is taking the max() of the resulting vector.

The above query is translated to the AQuery algebra as follows, letting e= max(price - mins(price)) and p= (ID = 'ACME') ∧ (tradeDate = '02/18/03'), as:

$$\pi_e^{op}(\sigma_p^{op}(\text{sort}_{timestamp}(\text{Trades})))$$

Let us now continue our tour of AQuery operators.

Definition 9 (Grouping). Let r be an arrable, and

[4]Such a definition is commonplace in Finance applications. Other domains may require avgs() to return NULLs on positions where the window is incomplete. In any case, it is often convenient to have the running average return an array the same size of its argument.

[5]Note: We are aware of developments in semantics of array operations using algebraic and category theoretic formalisms, but for purposes of this exposition we found the following procedural presentation to be most suitable. For an alternative presentation, see http://cs.nyu.edu/cs/faculty/shasha/papers/aquery.html.

$g(G_1, \cdots, G_m)$ be an arrable of the same cardinality as r. The arrable g is called the *grouping arrable* in that $g[i]$ is the *group value* that $r[i]$ maps to. (In most cases, g will be an order-preserving projection of some of the named arrays of r, but G_i may also be the result of more complex expressions.) The order-preserving group-by of r over g, denoted gby_g^{op}, is defined as follows

```
group-by(g,r)
1.  groups := empty arrable
2.  s:= empty arrable
3.  for i = 0 to |r|-1
4.      if g[i] in groups
5.          j:= index of g[i] in groups
6.          for each array A in r
7.              if A is not a grouped-by column
8.                  concat r[i].A to s[j].A
9.              end if
10.         end for
11.     else
12.         append g[i] to groups
13.         append r[i] to s
14.     end if
15. end for
16. output s
```

Step 13 above forms a single element list (or equivalently a vector). Step 8 concatenates to that list. The result is that fields may consist of vectors. As before, group by is order-dependent if any of its grouping expressions is. Group-by can also have an order-oblivious variation.

For reasons that will become clear on the next section, it is convenient to have an order-generating version of group-by. Semantically, such a group by delivers the results ordered by the grouping expression. □

Intuitively, grouping in AQuery partitions the operand arrable into disjoint sub-arrables that share the same g record values. It then transforms each sub-arrable into a single record by replacing each non-grouped column (in the sub-array) by its equivalent array-typed value. This process is depicted in Figure 1 using the grouping expression of the network management query. Recall that this query involves a group-by over source host, destination host, and a flow ID between them. Flow ID is order-dependent – a new flow between a pair of hosts starts whenever a consecutive timestamp difference is greater than 120 seconds. In AQuery, such a grouping expression corresponds to the arrable src, dest, sums(deltas(ts)>120). Figure 1(a) shows how this expression is computed, supposing the Trades arrable is sorted over src, dest, and timestamp and that the boolean TRUE carries a value of 1, and the FALSE, 0.

The arrable we see in 1(b) is the grouped one. Note that the columns of Packets that are not columns of g (the grouping arrable) have arrays within fields. Because fields may be arrays (though not arrables), aggregate functions may apply over an entire column or over each field. To express the latter, AQuery provides an operator modifier called *each* that applies functions to each array-valued element of a column.

Definition 10 (Each Modifier). Let r be an arrable, and *col* be a parameter of an any function F. The execution of F modified by 'each' is defined as follows:

```
each(F, col, r)
1.  s:= array of size |r| consisting of empty lists
2.  for i = 0 to |r|-1
3.      s[i] := F(r.col[i])
4.  end for
5.  output s
```

An "each-ed" operator is necessarily an order-preserving one. □

Each is a way of applying an operator to each field of a grouped column. In figure 1(c) we see that avg() was applied to each of the array-values of the column length. And similarly to count(timestamp). Having defined the operators involved in the network management query, we can now show its AQuery rendition.

```
SELECT    src, dest, avg(length), count(timestamp)
FROM      Packets
          ASSUMING ORDER src, dest, timestamp
GROUP     BY src, dest, sums(deltas(timestamp) > 120)
```

The algebraic version of the network management query, supposing that e= src, dest, avg(length), count(timestamp) and g= src, dest, sums(deltas(timestamp)>120), looks like:

$$\pi_e^{each}(gby_g^{op}(\text{sort}_{src,dest,timestamp}(\text{Packets})))$$

Cross-product ($\times$) in AQuery is order-oblivious and hence has the same definition as in the relational algebra. By contrast, joins have both order-preserving and order-oblivious variations.

Definition 11 (Join). A *left-right order-preserving* join of arrables r and s on join predicate p, denoted $r \bowtie_p^{lrop} s$, is defined in the following way.

```
join(p, r, s)
1.  o:= empty arrable
2.  for i = 0 to |r| - 1
3.      for j = 0 to |s| - 1
4.          if p(r[i],s[j]) is true
5.              append concat r[i] with s[j] to o
6.          end if
7.      end for
8.  end for
9.  output o
```

Let $A_1, \cdots, An$ be the attributes of r. A *left order-preserving* join, $r \bowtie_p^{lop} s$, is one that is order-equivalent with respect to $A_1, \cdots, An$ to its left-right counterpart. □

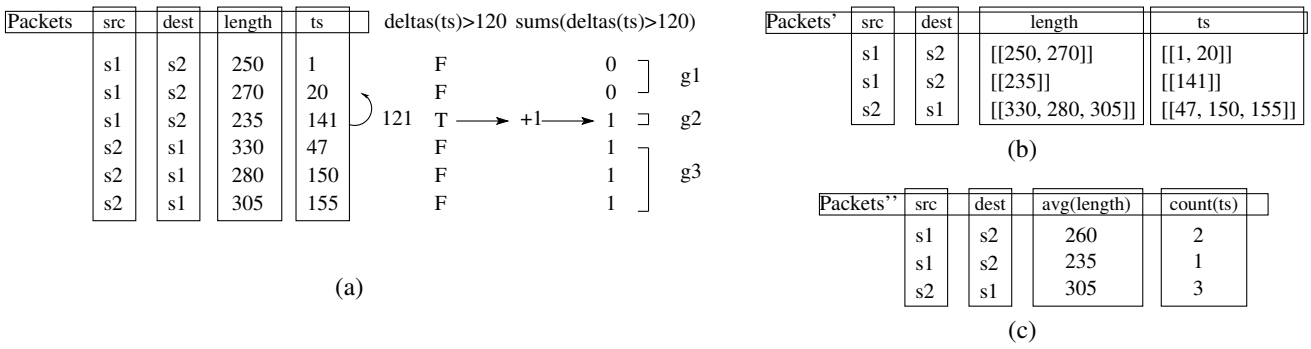

Figure 1: *Grouping Trades over src, dest, sums(deltas(ts) > 120): (a) computing the groups; (b) replacing the groups by single records; and (c) aggregating using the each modifier*

The join variations' functionality will become clear when we discuss optimization. For now, let us introduce another query to exemplify the use of a cross-product.

```
SELECT    t.ID, last(10, price)
FROM      Trades t, Portfolio p
          ASSUMING ORDER timestamp
WHERE     t.ID= p.ID
GROUP BY t.ID
```

Semantically, the query first performs a cross-product ($\times$) between Trades and Portfolio. Cross-product in AQuery is order-oblivious. Next, the ASSUMING clause imposes the desired sort order and the join predicate is applied. Then, the resulting arrable is partitioned into groups according to ID values. The assumed order is preserved within each group. The last() function "trims" each array-valued priced column to a maximum of the ten last positions of each price array. Letting e= ID, last(10,price) and p= Trades.ID=Portfolio.ID, this query can be represented as:

$$\pi_e^{each}(Gby_{ID}^{op}(\sigma_p^{op}(\text{sort}_{timestamp}(\text{Trades} \times \text{Portfolio}))))$$

More advanced arrable manipulation features of AQuery along with the complete, formal definition of AQuery is in [10]. Our purpose here was to show that by defining ordering explicitly and by adopting column-oriented semantics, AQuery allows order-dependent queries to be naturally expressed.

3 Query Transformations

Sort elimination and move-around are known techniques[17, 15] that can be applied to AQuery optimization. AQuery's semantics and, in particular, the edge built-in functions (e.g., first, last) allow other aggressive order-related optimizations as well. We introduce these new techniques through examples.

3.1 Implicit Selections and Sort-edge

Let Connections(host, client, timestamp) ORDERED BY host be an arrable that stores the client addresses that accessed services in a set of hosts. Suppose one wants to find the last client that connected to server "atlas." In AQuery:

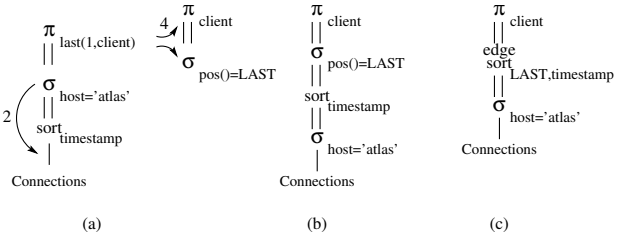

Figure 2: *Implicit selection and early sort-edge optimizations*

```
SELECT    last(1, client)
FROM      Connections
          ASSUMING ORDER timestamp
WHERE     host = 'atlas'
```

We are going to show plans in the usual diagrammatic way. The above query's initial plan is depicted in Figure 2(a). We introduce some auxiliary notation as follows. Double arcs mean order-preservation. Arrows represent the net effect of the application of a transformation. Each arrow is annotated with the corresponding transformation number. The formal descriptions of the transformations are given in Table 1 (informal descriptions are given when appropriate). We say pos(r) = i when we refer to the record $r[i]$. The special indexes for an arrable r, FIRST and LAST, are 0 and $|r|-1$, respectively. Finally, order(r) returns the list of attributes the arrable r is ORDERED BY.

A regular selection such as $\sigma_{host='atlas'}$ can be pushed down over a sort [15]. Transformation 2 in Table 1 is a slight variation of that transformation here in that order-preservation or obliviousness is made explicit. There are two advantages to commuting the sort and the selection here: the selection can benefit from the existing order on Connections (host), and delaying the sort reduces the number of records that would need to be sorted.

The projection $\pi_{last(1,client)}$ includes an implicit selection, i.e., it is only interested in one client. This is a

Sort Reduction/Elimination	
(1) $\text{sort}_A(r) \equiv_{order(r)} r$	if A is a prefix of $order(r)$
Selection	
(2) $\sigma_p^{op}(\text{sort}_A(r)) \equiv_A \text{sort}_A(\sigma_p(r))$	if p is *not* order-dependent
(3) $\sigma_{p_1}^{op}(\sigma_{p_2}^{op}(r)) \not\equiv_{\{\}} \sigma_{p_2}^{op}(\sigma_{p_1}^{op}(r))$	if p_i is order-dependent
Projection	
(4) $\pi_{e[i]}^{op}(r) \equiv_{order(r)} \pi_e^{op}(\sigma_{pos()=i}(r))$	e is an expression over r's arrays
Join and Semi-Join	
(5) $\text{sort}_A(r \bowtie_{A=B} s) \equiv_A \text{sort}_A(r) \bowtie^{lop}_{A=B} s$	if A,$B \in$ schema of r,s, resp.
(6) $\text{sort}_A(r \ltimes_{A=B} s) \equiv_A \text{sort}_A(r) \ltimes^{lop}_{A=B} s$	if A,$B \in$ schema of r,s, resp.
(7) $\sigma^{op}_{A=(B[i])}(r) \equiv_{order(r)} r \ltimes^{lop}_{A=B} \sigma_{pos()=i}(r)$	if A,$B \in$ schema of r
(8) $\sigma_p^{op}(r \bowtie^{lop}_{A=B} s) \equiv_{order(r)} \sigma_p^{op}(\sigma_p^{each}(\text{gby}_A(r)) \bowtie^{lop}_{A=B} s)$	if A,$B \in$ schema of r,s, resp., p is 'pos()=FIRST' or 'pos()=LAST', and B is unique
Group-By	
(9) $\text{gby}_A^{op}(\text{sort}_{A,B}(r)) \equiv_{A,B} \text{sort}_B^{each}(\text{gby}_A^{og}(r))$	

Table 1: *A subset of the equivalences between sort and remaining algebra operators*

particularity of AQuery's column-oriented semantics – a projection over a function that itself does a selection. The transformation 4 in Table 1 is a new transformation that replaces a projection with indexing by a pure projection plus a selection of the desired positions. If such positions are on an end of the operand array, we call this selection an *edge selection*. The result of applying this transformation is seen in Figure 2(b)[6].

The advantage of isolating the edge-selection from the original projection is that while the latter can't be moved around easily, the former can. In this example, the existence of an edge selection after a sort suggests that there is no need to sort all the input just to use some of the elements [2]. In AQuery, the physical operator *sort-edge* implements the logical pattern $\sigma^{op}_{edge-condition}(sort(r))$. The sort-edge uses a modified heapsort to keep the top (or bottom) n elements, as appropriate. This is similar to the approach used in [2] except that we modify the heapsort to make it stable[7]. The final plan shown in Figure 2(c)

3.2 Sort Splitting

Consider again the arrable Connections ORDERED BY host. The following query finds all the clients that connected to the last host to which a client hooked in.

```
SELECT    client
FROM      Connections
          ASSUMING ORDER timestamp
WHERE     host = last(1,host)
```

[6]A word of caution: Had we separated the implicit edge selection before moving the regular selection, they would be adjacent. Regular and order-dependent selections do not commute as inequivalence 3 in Table 1 shows.

[7]A stable sort is one that does not change the original order of records having identical value on the sorted key. Heapsort is not naturally stable. It becomes stable if one concatenates a tuple ID to the key.

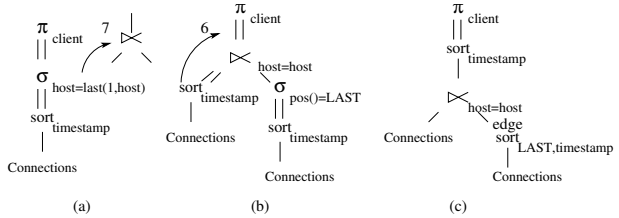

Figure 3: *Sort-splitting optimization*

Note that the predicate host = last(1,host) makes sense as an array expression. The array host is compared to the single element array (treated as a scalar) last(1, host) resulting in an array of booleans. Positions that map to false are eliminated by the WHERE clause.

An initial plan for this query appears in Figure 3(a). Timestamp is not a prefix of $order(Connections)$, thus the sort over timestamp may be required. However, host is a prefix of $order(Connections)$, and therefore the selection $\sigma_{host=last(1,host)}$ may take advantage of it. This is where the split-sort technique comes in. Here is how.

If A and B are arrays of an arrable r, a selection $\sigma_{A=(B[i])}(r)$ can be replaced by a semi-join as described by transformation 7 in table 1. The benefit of the semi-join is that we can now manipulate order on each of the semi-join's arguments differently.

Figure 3(b) shows the result of applying that transformation. Note that $last(1, host) = host[LAST]$. Let's analyze each side of the semi-join in turn. On the right-hand side we have the pattern edge-selection / sort, which can be efficiently implemented, as we have discussed. By contrast, the left-hand-side sort changes what could be an interesting order to the semi-join operation. We can thus defer it until after the join. The transformation 6 in table 1 commutes a semi-join

and a sort. In words, sorting a semi-join is equivalent to sorting its left stream and then performing an order-preserving semi-join under certain conditions and those conditions hold here. The net effect here is that computing the semi-join predicate is facilitated by existing order and that sorting over timestamp has to be done just over records generated by the semi-join – much cheaper than the original join over the whole arrable. The resulting plan appears in Figure 3(c).

A contrasting technique where a big sort is exchanged by a number of smaller ones is described next.

3.3 Sort Embedding and Edgeby

Consider the arrable Trades(ID, date, price, timestamp), this time with no determined ORDERED BY. (Often trades arrive in a "near-timestamp" order.) The query seeks the ten most recent prices for each security ID. In AQuery:

```
SELECT    ID, last(10,price)
FROM      Trades
          ASSUMING ORDER ID, timestamp
GROUP     BY ID
```

An initial plan for this query is shown in Figure 4(a). We can separate the implicit selection from the projection as we did before. The resulting plan appears in Figure 4(b).

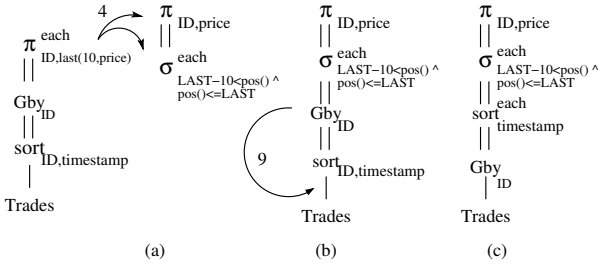

Figure 4: *Sort-embedding optimization*

But group-by and sort need not be executed in this expensive way. Let's enumerate a few options. First, an edge-selection applied to groups is an idiom, called edgeby, that can be highly optimized. *Edgeby* is a physical operator capable of implementing the logical pattern $\sigma^{each}_{edge-condition}(\text{Gby}(r))$. Instead of separating all elements of an arrable into groups just to use a slab of them (e.g., first n, last n, drop n, etc), edgeby discards, on-the-fly, elements for groups that already fulfill the edge condition. Depending on this condition, edgeby can scan arrables backwards or forwards. In fact, edgeby can be parameterized to perform grouping followed by any possible edge selection. In our example, we need the end of an array (i.e., last() over ascending order) of prices for each ID. A backwards edgeby ID on the sorted Trades keeps a record if it belongs to a heretofore unseen ID or a group having less than ten records.

An alternative is to delay sort until after group by ID is done. Sort would have to be applied then only within each group. It may be much cheaper to do several small sorts than a single big one. This is what we call sort embedding. Moreover, for this particular query the smaller sorts would be then followed by edge selections – sort-edge would apply.

The transformation 9 in table 1 allows such a commute of a sort with a group-by. Note that (a) group by must deliver its results in the same order it is grouping by over (an order generating operator, as described in section 2.3); and (b) grouping must be over a prefix of sort's arguments. The result of this transformation is shown in Figure 4(c). Note how a double-arc connects group-by and sort-each, because this instance of group by is order generating. Ultimately, these options allow the optimizer to decide which of the patterns, edge selection/sort or edge selection/group-by, is faster under the existing circumstances.

3.4 Early Edge Selection

Let us look at another scenario where an edge selection may reduce cardinality early in a query.

We use the arrable Trades here as well, but we now assume it is ORDERED BY timestamp. The arrable Portfolio(ID, name, tradedSince) ORDERED BY ID stores the subset of securities with which an analyst deals. Name is a unique identifier of securities in Portfolio. So is ID. To retrieve the last price of a security by its name one would do:

```
SELECT    last(1, price)
FROM      Trades, Portfolio
          ASSUMING ORDER timestamp
WHERE     Trades.ID=Portfolio.ID
          AND name = "DataOrder"
```

An initial plan for this query is depicted in Figure 5(a). A heuristic to improve performance applies here: The regular selection can be commuted with the sort and then pushed down over the join. The result is seen in Figure 5(b).

Upon detection of an appropriate existing order (i.e., ORDERED BY of Trades matches ASSUMING ORDER of the query), the optimizer would try to eliminate the sorting completely. Transformation 5 in table 1 commutes a join with a sort while still keeping track of order. That is a slight variation of a transformation in [15] in which order-preservation is made explicit. As Trades is already ORDERED BY timestamp, that sort may be eliminated, as transformation 1 in Table 1 determines. The result is seen in Figure 5(c).

This query also contains a projection-with-selection, and again we can brake them apart. The consequent presence of the edge selection after the join suggests that we may not need to perform the edge selection in its entirety. Portfolio.ID is a key and so it guarantees that each record in Trades will match at

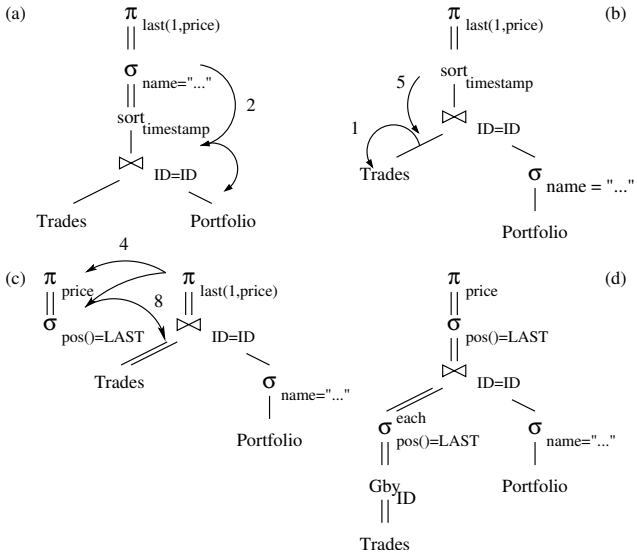

Figure 5: *Early edgeby optimization*

most one record in Portfolio. (Foreign key joins are among the most frequent of equijoins.) In these conditions we could push down this edge selection in the following way. For each ID in Trades, find its last record. This can be done by grouping Trades by ID and edge-selecting each last record. As discussed before, this can be done rather efficiently. By joining the result with the right-hand side stream we would get much less records. The final selection would then pick the desired price. This is what transformation 8 in table 1 does. The final plan is shown in Figure 5(d).

These transformations generate orders of magnitude improvement over a direct syntactic implementation of the AQuery calculus as well as over the SQL:1999 formulations. We illustrate this assertion in the next section.

4 Experimental Results

We have made three basic assertions in this paper relating to performance: (1) An array-based data model results in simpler queries than a multiset model with add-ons. Simpler queries may be easier to optimize. (2) Order-based expressions reveal idioms to which we can apply transformations. (3) The transformations exploit these opportunities to generate highly efficient plans for realistic queries. This section attempts to test these assertions.

4.1 Implementation Aspects

All our experiments were conducted on a Pentium III-M 1.13Mhz with 1 gigabyte of memory running Linux with no setting of process priorities. Each query was run in single-threaded mode. The timings reported here correspond to wall clock timings. Each experiment was repeated 10 times and were averaged for this presentation.

4.2 Performance Measurements

AQuery's rendition of the best-profit and the newtork-management queries contained no nesting and were straightforward to optimize. Table 2 compares performance results of AQuery plans versus an SQL:1999 commercial optimizer's.

Query	AQuery	SQL:1999
Best Profit	0.04 sec	12 sec
Network Management	8 sec	31 sec

Table 2: *Performance of AQuery's vs. SQL:1999 Optimizer plans*

The best-profit query numbers were generated using a Trades arrable/table with 1 million trades divided evenly among 100 securities. The time difference is due to AQuery's ability to push down the selection predicate. Recall that this query was interested in profits for a given security for a given date. An index facilitates the evaluation of the predicate. By contrast, the SQL:1999 optimizer could not move the selection, because the SQL:1999 representation used a complicated nesting structure.

For the network management query we used a Connections arrable/table with 1 million connections that took place among 100 distinct hosts. AQuery's plan was faster because in its formulation there was only one sort to be done, the one caused by the ASSUMING clause. Group-by in AQuery depends – and thus benefits – from that ordering. Conversely, the SQL:1999 optimizer had to figure out how to deal with two unrelated WINDOWS specifications. That resulted in having two distinct sorts before the processing of the group by, which did not benefit from them.

Moral: AQuery's structural simplicity helps in finding better plans.

In most cases, an edgeby requires a small fraction of the time required to perform the associated group-by, if done in its entirety as shown in Figure 6(a). We used the arrable Trades with 1 million records divided evenly among 10, 100, 1000, and 10000 securities. An edgeby over security ID with varying slab sizes is tested. The more records edgeby can discard, the faster its response time. For instance, when only a few distinct securities are used, groups are large, and therefore most records fall off the slabs even for the biggest slab sizes tested, greatly improving performance. As the groups get smaller (i.e., more distinct securities are used), highly selective slabs gets better performance. A degenerate case is seen where a 100-slab is taken from groups that are themselves 100 records wide. Edgeby doesn't improve performance here – but doesn't hurt either.

The idea behind the sort embedding technique is that a sort can be delayed until after a group by, and can be replaced by several sorts over the grouped columns (sort-each's). Figure 6(b) characterizes the

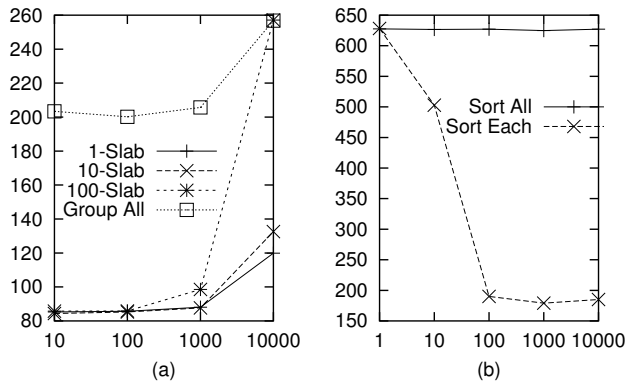

Figure 6: *Efficiency of work reduction techniques. Vertical axis are time in milliseconds. (a) Several sized edgeby slabs of Trades by ID where the horizontal axis is the number of distinct securities. (b) Embedding sorts within distinct numbers of groups along the horizontal axis.*

performance gains of sort-eaches as compared to the entire sort they replace. We used arrables of 1 million records and varied the number of groups. When only one group exists, there's no point in applying the technique – but, again, there's no penalty in doing so. Trading one big sort by several smaller ones starts to payoff whenever more than 10 groups exist.

Sort-edge presented similar results as those of sort-stop [2] and we omit these results.

Moral: Reduction of work due to edge selections or sort handling techniques is significant.

AQuery's optimizer integrates sort with the standard relational operators such as selection and group by. For instance, in the query of section 3.1 it was able to apply selection push-down over a sort. Figure 7(a) shows that this technique was particularly efficient for arrable instances where the number of distinct hosts was greater than 10. But plans using a plain, regular sort on instances with less than 10 distinct hosts would still perform poorly. By identifying the implicit edge selection of that query and by using it to reduce the number of records to sort, Aquery generated an optimized plan that performed best in all the instances tested.

Figure 7(b) shows the performance gains of applying the sort splitting technique to the example query of section 3.2. The efficiency of the optimized plan stems from delaying the enforcing of the ASSUMING order up until after the semi-join reduces the number of records to be sorted. The gains stabilized at instances with 100 or more distinct hosts because at this point the cost of the query is dominated by the semi-join itself as opposed to the sort of its results. Note that application of this technique whenever the number of hosts is too low (e.g. just one) may represent an unnecessary overhead – although a small one. So, this technique depends on the data distribution, underscoring the need for cost-based optimization.

Figure 7(c) shows the comparative performance of plans for the example query of section 3.3. The naive plan sorts the whole arrable, groups the entire result and applies the edge-selection only at the end. Cost remains rather high, even when the edge selection removes most records. By contrast, the optimized plan trades one big sort for several smaller ones – sort-edges, in fact. Thus, even in the degenerate case where each group has only one record (i.e., number of distinct hosts is equal to the cardinality of the arrable), the optimized plan saves the cost of a big sort. The curves show order of magnitudes difference at instances with small number of distinct hosts.

Finally, Figure 7(d) shows the results for the naive and the optimized plans for the example query of section 3.4. By applying an edgeby early in the plan, the number of records that have to be joined is considerably reduced. The optimized plan also takes advantage of the existing order, eliminating any sort altogether. The result is consistently faster response times.

Moral: AQuery transformations bring substantial performance improvements, especially when used with cost-based query optimization.

5 Related Work

Whereas SQL:1999 is the commercially most significant implementation of ordered queries, other systems, both commercial and research, have proposed many excellent ideas.

5.1 Optimization Techniques

Sort order has always been treated in the optimization process as a physical property to be included in a plan (if not specified by SQL's ORDER BY clause) to support an efficient algorithm such as merge join. Mechanisms such as Starburst's "glue" [9] or Volcano's "enforcer" [5] made sure a sort step was added whenever an efficient algorithm required it.

Sort was never fully integrated with other operators, though [17] added an order-optimization step before plans were enumerated in the context of DB2's optimization process. This step may dramatically improve queries that have order requirements due to the clause ORDER BY, GROUP BY, or the DISTINCT modifier. But order optimization in [17] was still a separate optimization step in that transformations involving sort elimination or reduction (i.e., sort over fewer attributes) were not considered at the same time transformations involving other algebraic operators were. By contrast, because we consider sort operators with other operators in the spirit of [15], we are able to discover techniques such as sort splitting or sort embedding (i.e., transformations 7 and 9, respectively, on Table 1).

In [2] a clause STOP AFTER was suggested which was capable of limiting the resulting cardinality of

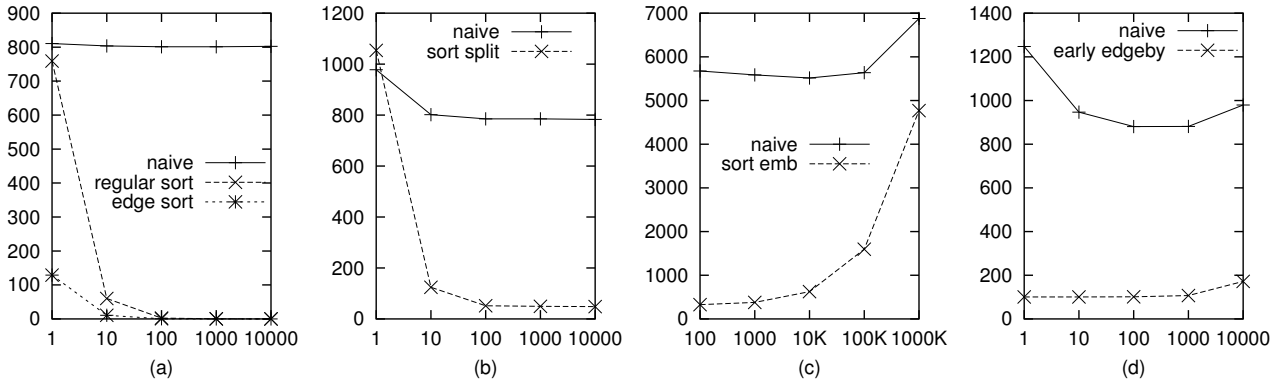

Figure 7: *Optimized and Non-optimized versions of plans. Vertical axis is time in milliseconds. Horizontal axis is the number of hosts for (a) and (b) over which the 1 million connections are divided and the number of securities for (c) and (d) over which the 1 million trades are divided. Plans for (a) are shown in figure 2, for (b) in 3, for (c) in 4, and for (d) in 5.*

queries whose results were ordered (by an ORDER BY). When only the *top-k* tuples of a query's result need be consumed, queries can run much faster. Our sort-edge is similar to stop-sort. A difference between that work and ours is that, again, their order optimization process happens as a separate, isolated phase. Because we integrate the two, AQuery uses the stop-after idea within groups as well. After all cardinality restriction is a generally useful technique.

The first work, to the best of our knowledge, to provide optimization of order in an integrated framework was [15]. Most of their list-based transformations apply to AQuery. The transformations we presented here extend their framework with several new techniques (e.g., transforamtion 8 in Table 1, the early-edgeby one).

Other optimization rules related to order or ordered structures were suggested in the context of their corresponding query languages, which we discuss next.

5.2 Languages

AQuery is a descendent of KX systems's KSQL [7] from which AQuery takes its arrable notion – a fully vertically partitioned implementation of tables, each of whose columns is an array. AQuery differs from KSQL by trying to preserve the SQL flavor to a much greater extent than KSQL, by the use of the ASSUMING clause to make the use of order declarative, by the introduction of transformations, and by the exploitation of a cost-based framework for optimization.

The sequence query languages SEQUIN [16] and SRQL [14] are precursors of SQL:1999 in that they are SQL dialects that handle order-based queries. They are true to the spirit of [13] which showed that order was a much needed feature in queries. SEQUIN treats sequences as an extended abstract data type, although a sequence can serve as the sole source of data for a query. SRQL was inspired by SEQUIN and treats tables as ordered relations. AQuery borrowed from these languages the early introduction in a query of an order defining clause. Both SEQUIN and SRQL keep the tuple semantics of SQL, as opposed to the vector (column) processing of AQuery. A consequence is that several valid vector expressions in AQuery are invalid in these languages, e.g. max(price - mins(price)). Another difference is in the way non-1NF relations/arrables are handled. In [16], if a table has a sequence attribute, SEQUIN is used to express predicates over the sequence, while SQL is used over the table. That can lead difficult-to-read queries and complex optimization transformations. Further, SRQL did not pursue non-1NF ordered relations, whereas we have found them very useful.

Non-SQL efforts at query languages and data models based on arrays have been suggested before [8, 12]. Neither AQL [8] nor AML [12] provides a declarative mechanism to define the order in which the queries manipulate data. Queries process data in the order it is stored. While that makes sense for databases of raster images [12] or scientific data in CDF format [8], it makes less sense in general data processing. Nevertheless, one could argue that these languages could easily incorporate a sort function and express most, if not all, of the queries here. We agree. We would welcome such extensions because the implementors may then arrive at interesting optimizations that would be complementary to ours. Finally, we have a bias for pragmatic reasons for an SQL dialect, but reasonable people can differ on this point.

A particularly inspiring feature of the AQL optimizer is that it has the powerful capability of optimizing operators (or newly added functions) on the calculus level, i.e., by application of variations of λ-calculus reductions over the operators definitions. Reductions help find syntactically simpler forms of an expression while keeping its semantics intact. We have not yet fully exploited that ability in AQuery. On the other hand, we have shown that, for instance, the sort split-

ting technique requires more than simplifying an expression. It involved transforming what was one sort plus a selection in a semi-join plus two sorts plus a selection – and that resulted in sorting fewer tuples than the simpler expression. AML by contrast uses a fixed set of transformation rules aimed generically at function application on array slabs. A complete fusion of these ideas requires more exploration.

6 Conclusion

AQuery builds on previous language and query optimization work to accomplish the following goals:

1. Incorporate order in a declarative fashion to a query language (using the ASSUMING clause) built on SQL 92.

2. Introduce a query semantics that while keeping compatibility to SQL's allows inter-tuple calculations that do not require query nesting.

3. Add order-dependent functions (e.g. sums, first) that are natural and flexible to express, for instance, top-k elements of groups queries.

4. Create a simple, yet powerful optimization framework that results in performance that is an order of magnitude faster than commercial SQL:1999 systems for natural queries. Edge optimization and sort splitting and embedding seem to be particularly promising for order-dependent queries.

Our future research work concerns further work in query optimization especially involving order-preserving joins. We are also insterested in the implementation of continuous queries in an environment where historical data may have one sort order and new data another.

References

[1] Budd, T., "An APL Compiler." Springer-Verlag, April, 1988.

[2] Carey, M.J., and Kossmann, D., "On Saying 'Enough Already!' in SQL." SIGMOD Int'l Conf. on Management of Data, 219–230, 1997.

[3] Cranor, C., Gao, Y., Johnson, T., Shkapenyuk, V., and Spatscheck, O., "Gigascope: High Performance Network Monitoring with an SQL Interface." SIGMOD Int'l Conf on Management of Data, 623–623, 2002.

[4] Claussen, J., Kemper, A., Kossmann, D., and Wiesner, C. "Exploiting early sorting and early partitioning for decision support query processing." VLDB Journal, 9(3): 190–213, 2000.

[5] Graefe, G., and McKenna, W.J., "The Volcano Optimizer Generator: Extensibility and Efficient Search." ICDE Int'l Conf on Data Engineering, 209–218, 1993.

[6] Knuth, D.E., "The Art of Computer Programming." V3, Sorting and Searching, 2nd Ed,. Addison Wesley Longmand, 1998

[7] KX Systems. "KSQL Reference Manual, Version 2.0." http://www.kx.com

[8] Libkin, L., Machlin, R., and Wong, L., "A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques." SIGMOD Int'l Conf on Management of Data, 228–239, 1996.

[9] Lohman, G.M., "Grammar-like Functional Rules for Representing Query Optimization Alternatives." SIGMOD Int'l Conf on Management of Data, 18–27, 1988.

[10] Lerner, A., and Shasha, D., "The AQuery Language for Querying Ordered Data." In preparation.

[11] Melton, J., "Advanced SQL:1999 – Understanting Object-Relational and Other Advanced Features." Morgan Kaufmann Publishers, September, 2002.

[12] Arunprasad P. Marathe and Kenneth Salem. "Query Processing Techniques for Arrays." The VLDB Journal, 11:68–91, 2002.

[13] Maier, D., and Vance, B., "A Call to Order." PODS Symp on Principles of Database Systems, 1–16, 1993.

[14] Ramakrishnan, R., Donjerkovic, D., Ranganathan, A., Beyer, K.S., and Krishnaprasad, K., "SRQL: Sorted Relational Query Language." SSDBM Int'l Conf on Scientific and Statistical Database Management, 84–95, 1998.

[15] Slivinskas, G., Jensen, C.J., and Snodgrass, R.T., "Bringing Order to Query Optimization." SIGMOD Record, 31(2):5–14, 2002.

[16] Seshadri, P., Livny, M., and Ramakrishnan, R. "The Design and Implementation of a Sequence Database System." VLDB Int'l Conf on Very Large Data Bases, 99–110, 1996

[17] Simmen, D.E., Shekita, E.J., and Malkemus, T., "Fundamental Techniques for Order Optimization." EDBT Int'l Conf on Extending Database Technology, 625–628, 1996.