Querying Ordered Databases with AQuery

Alberto Lerner

June 22, 2003

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Order-Dependent Queries

An order-independent query is one for which the results (interpreted as a multiset) do not change if the order of the input records change. In a stock-quotes database, for instance, calculating the maximum price of a stock in a given day is order-independent. Regardless of the order in which records are examined, their maximum is the same. Relational databases support order-independent queries extremely well.

By contrast, finding the price changes of a stock over many days depends on order. Such a query is therefore *order-dependent*. Order-dependent queries arise naturally in many application domains. In finance, an analyst often looks at *n-moving averages* over price time series, which is the average of a price and its n predecessors, calculated for each price in the series [18]. The analyst may also be interested in correlations among time series, which requires prices to be in time order [39]. In network management, an administrator may want to analyze packet logs for statistics or security purposes. Statistics may involve breaking sessions between any pairs of hosts down into "flows" (sub-sessions), a flow-separator occurring whenever a packet and its predecessor are more than a given time interval apart [8]. A security check of the log may look for a port scanning attempt, in which a same client sends a succession of packets to different ports on a given host [9]. Again, packet ordering is relevant. In the relational storage of XML, the order of XML elements and attributes need to be encoded [36]. In Biology, frequent nucleic acid motifs are of interest. In Linguistics, texts are scanned linearly. In epidemiology, unusual spikes in emergency room visits may suggest the start of an epidemic. The reader may imagine many other applications.

Many queries whose natural formulation requires order can be expressed using query languages based on multisets. For instance, suppose a stock's quotes and their timestamps are stored in a table Quotes and that one wants to obtain each quote's predecessor. Joining Quotes with itself (using as a predicate the maximum

timestamp that is less or equal to the current quote's timestamp) would give the desired result. If done often enough, a reasonably skilled SQL writer would recognize the predecessor idiom at once. As a practical matter however, the more structurally complex a query's rendition is (joins or nested sub-queries), the more difficult it is to optimize (join elimination or query un-nesting).

On the other hand, multiset query languages have a long and illustrious history. Order should therefore be inserted in a language through careful design. This thesis is the result of our studies into building such a language. We present here not only the language itself, but also the underlying data model that makes it coherent, the optimization techniques that make it efficient, and the system that implements it all. We call this framework AQuery.

1.2 Principles and Goals

AQuery is a language in which order-dependent queries can be expressed naturally. AQuery's design began with these principles:

- Declarative Order – A query is able to define the order it requires records to be processed, regardless of the way the records are stored. One implication is that a query's results can be made independent of the underlying storage strategies. For instance, stocks' quotes are naturally generated in time order and could conceivably be stored so. But queries may require quotes to be in stock ID and time order; queries may simply declare so.

- Ubiquitous Order – The assumption that data is in a declared order is valid everywhere in a query, be it in a calculation involved in the result, a filter that depends on order, or still a grouping or aggregation operation. Order in AQuery can simply be counted upon everywhere.

- Conciseness and Lucidity – The most common order idioms are easily expressed – without auxiliary language constructs – in AQuery. The order idioms are not buried under or masked by a query's structure. For instance, if a filter is dependent on the order declared in its query and order is needed in no other context, then this is evident from the query structure alone. The rationale here is that if a query looks simple, then it should be easy to understand and optimize.

- SQL Compatibility – SQL practitioners should be able to familiarize themselves with AQuery with very little effort.

These principles provide a natural basis for optimization. By declaring the order it requires, a query allows the optimizer to determine whether data is already organized in a convenient order. If it is, sorting may be eliminated altogether.

If data is not in an appropriate order, then the AQuery optimizer may still have some space to maneuver. For instance, if a join followed by an aggregation are

involved in a query, but only the latter is order-dependent, then the optimizer may pick among several alternatives. It may perform the join in an "order-cavalier" fashion and then sort immediately before the aggregation. Or it may sort the relations in an appropriate way, perform an order-preserving join, and then finally aggregate. Or it may take advantage of the underlying order the relations are stored in and propagate that order until the aggregation. This choice is a cost-based one.

In many ways, we integrate the results from previous work: some focussed on linguistic aspects alone without considering optimization [29]; others did consider optimization but would not allow order handling in a declarative fashion [31, 21]; still others focused on order management in query optimization but for non-order-aware languages [33] or for languages with simple order extensions [35].

To the best of our knowledge, AQuery is the first comprehensive effort at addressing order-dependent queries' needs from the data model to the query language to the optimization process. As a result, AQuery expresses most order-dependent queries evaluated in a more concise way than other languages, and has delivered orders of magnitude performance gains for order-dependent queries as compared to robust commercial SQL:1999 query optimizers.

1.3 Thesis Overview

The first chapter of this thesis investigates the difficulties that arise in writing order-dependent queries in SQL. It then brings a comparative study of new languages that address that difficulty. From the study, we draw a powerful set of order-manipulation mechanisms but we conclude that such a set is not entirely available in any given language. We also observe that the most successful attempts at supporting order were those that replaced multisets in the data model by some form of array.

Chapter 3 defines a data model built around arrays that not only supports all the relational operators, but also order-preserving variations of them. The model provides several other features that are essential for expressing order-manipulations of realistic queries. The chapter also describes the AQuery language syntax and semantics, the latter by mapping queries into the algebra defined. The languages studied previously are then compared against AQuery on the basis of simplicity and conciseness.

The optimization of order-dependent queries is presented in Chapter 4. The chapter starts by reviewing the techniques applied to order (sort) management and shows that they also apply to AQuery. However, order optimization is usually done in a pre- or post-plan enumeration step. We advocate considering sort as any other operator in the enumeration process and present a set of new query transformations using this approach. The new transformations bring orders of

magnitude improvement to the performance of some plans. For each new transformation we conduct a performance evaluation study.

Chapter 5 describes the architecture and design of a system that implements the AQuery language and model. It is a fully functional system. Its most salient design aspect is that it is fully based on vector-processing techniques. The system manipulates data in a vertically partitioned fashion and translates queries into sequences of vector-operations over columns. The system itself is written in the very same vector-oriented language to which it translates queries.

A performance comparison, both qualitative (query plans) and quantitative (response times), between AQuery and an commercial SQL:1999 optimizer is described in Chapter 6. SQL:1999 gained order-manipulation capabilities through a late amendment [16] and therefore constitutes the first order-aware language to gain commercial acceptance. The comparison shows that AQuery is more amenable to order optimization, for its order idioms are more compact and easier to identify. Its plans were simpler than and in some cases almost two orders of magnitude faster than SQL:1999's.

Finally, chapter 7 concludes by describing our ongoing work and future research possibilities.

Chapter 2

State of the Art

2.1 Introduction

This chapter opens our investigation into querying ordered databases by addressing the following questions: Does a query language need specific features in order to express order-dependent queries? If yes, which ones? Is there any already existing query language with such features?

The best way to understand how hard – and why – it is to formulate queries that involve order is to try a few examples in SQL:92. Take a table Sales(month, sales) that stores for each month the amount sold in that month. For simplicity, assume that months are represented by integers starting with 1. Now, try to find the difference in sales between each month and its predecessor. (We encourage the reader to spend a minute trying to write this query.)

The first problem lies in the "predecessor" part of the query. How does one express such a property in a language that is based on (multi-) sets? Elements in a set are not ordered unless an ordering relationship is provided. Here the order can be obtained through months numbers. Again, for simplicity, let us assume that months are numbered consecutively with no gaps. Therefore one can find a previous month by simply subtracting 1 from the current month. Let us leave the boundary issue of month 1 aside for a brief moment.

The second problem is that *row-oriented languages* such as SQL – in which variables iterate over rows – cannot easily access two sales amounts (rows) at once. To subtract a previous month's sales from the current's, both sales must be in the same row. Producing such a configuration requires a self-join such as the following.

[**SQL:92**] – Delta Sales Query (naive)

```
SELECT t1.month, t1.sales - t2.sales AS delta
FROM   Sales t1, Sales t2
WHERE  t1.month - 1 = t2.month
```

Sales	month	sales
	1	100
	2	120
	3	140
	4	140
	5	130

Result	month	delta
	2	20
	3	20
	4	0
	5	–10

Figure 2.1: A Sales table instance and the result of the delta sales query

Note that the first month has no predecessor thus it was eliminated by the join. Fixing it requires replacing the join by an outer join and handling the case where t2.sales would be null.

[**SQL:92**] – Delta Sales Query

```
SELECT t1.month,
       t1.sales - CASE
                    WHEN t2.sales is null THEN 0
                    ELSE t2.sales
                  END
FROM   Sales t1 LEFT OUTER JOIN Sales t2
       ON t1.month - 1 = t2.month
```

We call such a manipulation of column values at different rows *Inter-row Operations*. In this particular case we are executing a *running delta* operation. It is one of the most common order manipulations.

Another quite frequent operation category is that of the *Running Aggregate Operations*, which brings together a number of cumulative operations such as moving averages. A *moving average* over a column divides its elements into groups of rows that are not necessarily disjoint ("windows") and calculates the average of each group. For instance, if one wanted to compute a 3-month sales moving average, one would find the two previous sales for each month, would form a group with the current month, and would then calculate the average of that group. This operation would be repeated for each month.

The previous query can be used as the basis for this one, except that an extra outer join would be required so to have the 3-sales window. Managing the null values becomes more complex, too. The new query looks like

[**SQL:92**] – 3-month Sales Moving Average Query

```
SELECT t1.month,
       t1.sales +CASE
                   WHEN t2.sales is null AND
                        t3.sales is null
                        THEN 2*t1.sales
                   WHEN t2.sales is not null AND
                        t3.sales is null
```

```
                    THEN t2.sales + (t1.sales+t2.sales)/2
               ELSE   t2.sales + t2.sales
             END
FROM  Sales t1 LEFT OUTER JOIN Sales t2
      ON t1.month - 1 = t2.month
      LEFT OUTER JOIN Sales t3
      ON t1.month - 2 = t3.month
```

Despite the complex syntax of their SQL renditions, the queries so far are quite simplistic. To show a more realistic example based on the operation categories presented so far, we turn our attention to stock trading. An increasingly popular way of trading in a very dynamic market is "day-trading." It consists of buying and selling shares within very short periods of time – often in the very same day – aiming at immediate albeit modest profits. To verify whether a trader made a profitable enough transaction, one may wish to know what would be the best profit that could be done by buying and then selling a given stock in the same day.

In this query we use the schema Ticks(ID, date, volume, price, timestamp), which stores all quotes and trades for securities negotiated in a trade floor or trading system. ID is the ticker symbol of a security, the code by which a security is identified; timestamp is the date and time of a particular quote or trade; date is the human-readable form of the day portion of the timestamp; volume is the number of shares that are being negotiated; and price is the value per share.

A strategy to solve this query would be to calculate, for each row of Ticks, what would have been the minimum price seen for that security up until that tick. This operation is called a running minimum – another running aggregate – and requires ticks to be sorted by ID and timestamp order. Subtracting each Tick's price by its running minimum yields the profit made by buying at the lowest price up until that tick and selling at that time. The maximum of such differences is the best profit. The graph in Figure 2.2 illustrates this approach by showing the price curve of a stock on a given date along with its price running minimum curve. The difference between the curves depict profits at each tick. For brevity, we omit the solution of this query in SQL:92.

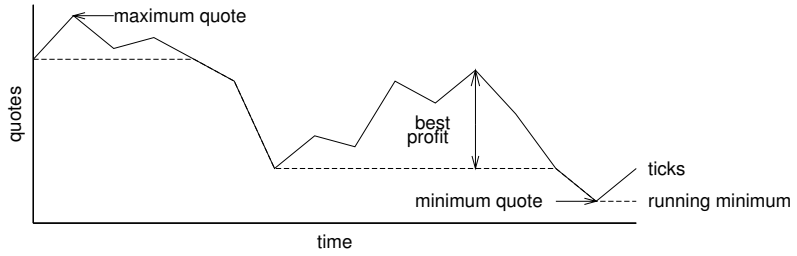


Figure 2.2: A stock's price curve and its running minimum

The above examples show that writing queries that deal with order may result in complex SQL. That is not without consequences to the query's optimization process – upon which declarative query languages depend so heavily to be efficient. From a strictly algorithmic point of view, all the queries shown here could use solutions (query plans) with linear time complexity in the database size. Had the tables been in an appropriate order, running deltas and running minimums would have required a single pass on the data. Yet whenever expressed in SQL, even state-of-the-art commercial optimizers are not able to find such plans and eliminate the (algorithmically unnecessary) joins.[1]

We can now go back to our first question. Do query languages need specific mechanisms to express order operations? If one wants order-dependent queries to be easy to read and to optimize, then yes. Now, which mechanisms would accomplish that?

Supporting order-dependent queries means supporting an ordered data structure and providing a query language and algebra equipped to deal with it [23]. There is not consensus on whether that structure is a list, an array, an arbitrary combination thereof, or still some other structure or on whether that query language should be based on SQL.

Several alternatives have been investigated and we can find inspiring insights in the database literature. We divide these works in three categories: those that added ordered structures and adapted SQL accordingly, those that designed a whole data model and a language around a given structure, and finally the standard SQL approach itself, which we describe next.

2.2 Standard SQL with Late Order

Order support in SQL:1999 [15] comes indirectly from a language mechanism whose purpose is to define a *sliding window*. This mechanism was in fact a late addition to the standard [16] and was conceived more to deal with analytical queries (OLAP) needs than to address more general order-dependent queries.

One way to present SQL:1999's sliding windows is as a means of transforming aggregate functions into running aggregates. For example, to compute the 3-month sales moving average query stated previously, we define a window whose size is of 3 positions (rows), starting two months before the current one and finishing in the latter. This definition requires months to be sorted on ascending order. Applying the aggregate function avg() over each distinct window yields the desired running average. This query in SQL:1999 would look like

[SQL:1999] – 3-month Sales Moving Average Query
```
SELECT month, sales, avg(sales) OVER ( ORDER BY month
```

[1] As of this writing, the DBMS products we tested picked join-based (non-linear) solutions.

```
                                    ROWS BETWEEN 2 PRECEDING
                                      AND CURRENT ROW)
FROM  Sales
```

This query may not look familiar to the SQL practitioner. Having in a SELECT list the aggregate function avg() and columns that were not GROUP-ed BY may look at first as an error. It is not; this query returns the same result as its SQL:92 counterpart. The OVER clause specifies a sliding window which modifies the function avg() to a running aggregate. The change is less in the way an average is computed than in how many times this function is called. The OVER clause associates to each row a window and then causes the aggregate function to be called on that window. The net effect is that avg() is called as many times as there are rows.

Although being quite expressive, window usage is subject to a few syntactical restrictions [26]. For one, they can be used only in the SELECT clause. For another, they can modify only aggregate functions. Thus, for instance, to find a "predecessor" element as was the case in the delta sales query specified previously, some creativity is involved. Defining a single-element sliding-window consisting of the element that precedes the current and applying min() to such a window would returns that very element. The delta sales query in SQL:1999 looks like

[**SQL:1999**] – Delta Sales Query
```
SELECT month, sales - min(sales) OVER ( ORDER BY month
                                        ROWS BETWEEN 1 PRECEDING
                                          AND 1 PRECEDING)
FROM  Sales
```

Some DBMS vendors provide non-standard "windowed functions" that make the above query more intuitive. For instance, Oracle 9i has a pair of functions, lag() and lead(), that retrieve values by their relative position [28]. The standard itself defined other additional functions specifically to be used over windows, but the presentation of those is beyond the scope of this discussion.

Of interest to our study is to evaluate whether the window mechanism itself helps or hinders order-dependent queries. One way to assess it is to write the best profit query in SQL:1999. Recall that it relied on finding a running minimum of prices for each stock. The difficulty here is to define a sliding window that takes care not to mix ticks of different stocks. Enforcing a sort order or a window size alone would not suffice. We therefore resort to a more advanced way to define a window which involves partitioning data. Our point becomes clearer if we try to find the best profit of all stocks rather than just of a single one. This query in SQL:1999 looks like

[**SQL:1999**] – Best Profit Query
```
SELECT  ID, max(running_diff)
```

```
FROM ( SELECT ID, date,
              price - min(price) OVER ( PARTITION BY ID, date
                                        ORDER BY timestamp
                                        ROWS UNBOUNDED PRECEDING)
              AS running_diff
       FROM   Ticks ) AS t1
WHERE  date = '05/11/2003'
GROUP  BY ID
```

To transform the aggregate function min() in the desired running minimum, we partition data by ID and date. This means that the current row's stock ID and quoting date values are considered when forming that row's window. Only after eliminating all elements that do not belong to the current row's partition does the sort order get enforced. We also used a cumulative way to form a window in which it starts at the first row of the current partition seen and ends at the current row (ROWS UNBOUNDED PRECEDING).

Because an aggregate function (max()) cannot take a running aggregate function (min(price) OVER $\cdots$) as an argument, this query is a nested one. The inner block of the query calculates the running difference while the main query performs grouping and aggregation. The impact of nesting is felt both on readability and on optimization. We show in Chapter 6 how a commercial optimizer missed optimization opportunities due to the structure of the query.

Sliding windows are not the only mechanism in which order is involved in SQL:1999. The language incorporated a new array type; columns can now hold arrays as opposed to just scalars. The manipulation of "array-fields" in SQL:1999 is very limited, though. There were academic prototypes that exploited arrays fields better than SQL:1999. These will be discussed in the next section.

2.3 SQL Dialects over Ordered Structures

Academic prototypes had defined dialects of SQL with order features before SQL:1999 was amended with OLAP functions. SEQUIN, or the PREDATOR system to be precise, suggested a pacific cohabitation of sequences and relations through the use of "enhanced" ADTs [31]. SRQL and its underlying algebra did not make distinctions between relations and sequences, treating the former as a degenerate case of the latter [29]. Let us discuss each language in turn.

2.3.1 SEQUIN

The PREDATOR database system introduced the concept of "enhanced" abstract data types (E-ADT) [31]. By enhanced it meant that each new data type carried more than a collection of methods that could manipulate it. It carried a particular

query language and even an optimizer of its own. SEQUIN is PREDATOR's language designed to deal with its sequence E-ADT.

A sequence in a SEQUIN query occupies the same place a table would in SQL. The main difference is that sequences are ordered [30] and thus the FROM clause supports special joins based on the ordering domain of the participating sequences. This feature comes in handy to express the delta sales query. We show the SEQUIN rendition of the query below

[**SEQUIN**] – Delta Sales Query
```
PROJECT t1.month, (t1.sales - t2.sales)
FROM    Sales AS t1, PREVIOUS(Sales) AS t2
```

When more than one sequence appears in the from clause an implicit natural join on their positions takes place, i.e. first element joins with first element, second element with second element, and so on. The modifiers NEXT, PREVIOUS, and OFFSET in the FROM clause shift entire sequences so that different alignments can be used in the joins.

The PROJECT clause is similar to the SQL's SELECT. It implicitly assumes that elements of the sequences are ordered. Herein lies both an advantage and a problem. The advantage is that enforcing order early in a query makes order visible to all subsequent clauses. For instance, had we wanted to find the months whose delta sales were greater than a given threshold, we would have put the expression 't1.sales - t2.sales > threshold' in the WHERE clause. In SQL:1999, where order is supported only in the SELECT clause, this query would have required nesting. The disadvantage of implicit order is that order is not declarative. By looking at the query alone one does not know in which order the Sales sequence is. Should the order of Sales be changed for some reason, the result of this query would have been affected. Had the query specified in which order it expected to handle data, such problem would not have happened.

An interesting order feature is that of the mixing of SEQUIN (sequences) and SQL (sets) in a query. In the following example suppose Ticks' is a table that holds two columns: an ID one, and a sequence of quote-price elements in timestamp order called priceSeq. To query the minimum price of every security within a given interval one would write in SEQUIN the following query.

[**SEQUIN**] – Mixed SQL and SEQUIN Query
```
SELECT  ID, SEQUIN ( "PROJECT min(price)
                     FROM    $1
                     WHERE   date BETWEEN
                             '01/01/2003' AND '03/31/2003",
                   T'.priceSeq)
FROM    Ticks' AS T'
```

The keyword 'SEQUIN' in the query could be seen as a function call to SEQUIN's query processor. Its first argument is the query to be executed over sequence data. The second argument is the sequence data itself corresponding to ticks of the security's row that is being processed. The $1 in the query is replaced with that parameter.

This is quite a useful feature for applications that manipulate sets of sequences, e.g. Finances and Biology. Nevertheless, nesting an entire query in the SELECT list of another is arguably hard to read.

2.3.2 SRQL

SRQL is a SQL dialect that also manipulates sequences and relations. It does so by representing both by the same underlying structure. A sequence is a sorted relation, in which a list of attributes exists that defines the order of its records. A relation is simply a degenerate case in which such a list is empty. That and other simplifications made the writing of order manipulations in SRQL easier.

A SRQL query structure resembles that of a SEQUIN query structure. The main improvement was to allow a query to declare in which order it requires data to be processed rather than to assume that such order comes automatically from the underlying data structures. The chosen order is enforced early in the query and, as in SEQUIN, all remaining clauses can count on it. SRQL has operators to shift sequences much as SEQUIN does, but they are not confined to the FROM clause. To illustrate the use of these mechanisms we show the SRQL rendition of the delta sales query below

```
[SRQL] – Delta Sales Query
SELECT S.month, (S.sales - SHIFT(S,-1).sales)
FROM   Sales
       SEQUENCE BY month AS S
```

In the query, the Sales relation is "sequenced" according to month order and the result is bound, a row at a time, to the tuple variable 'S'. The expression 'SHIFT(S,-1).sales' reads "the previous tuple to the one pointed by the tuple variable S." SRQL is the first language we present that is able to express this query without explicitly resorting to a join. From an optimization point of view such a query rendition makes the order idioms clear – it exposed both the order in which rows are to be processed and what order manipulations are involved in the query.

SRQL also supports the concept of sliding windows. The window definition, much simpler and thus less powerful than SQL's, has to adopt the same order specified in the SEQUENCE BY clause of the query. The following SRQL rendition of the 3-month sales moving average shows the use of windows.

[**SRQL**] – 3-Month Sales Moving Average Query

```
SELECT month, AVG(sales) OVER -2 TO 0
FROM   Sales
       SEQUENCE BY month
```

The OVER clause defines a window based on relative positions. In this case, avg() will be called once per row and will be passed a window that ranges from two rows before the current (-2) until the current row (0).

We don't know the personalities involved, but it seems likely that SQL:1999 borrowed the OVER constructs from SRQL. The concept here is clearer, though, because the sort order is valid throughout the entire query as opposed to just within the window. It remains that, for the same reason as SQL:1999, queries such as the best profit one would require nesting to be expressed in SQRL.

SRQL has borrowed several mechanisms from SEQUIN, but it gave up an important one: the ability to manipulate order (sequences) within a field.

2.4 Array-Based Querying Systems

Arrays are ubiquitous in application domains such as scientific computing and finance. Attempts were made to develop data models and query languages that revolve around arrays that would support such applications. Order is intrinsically involved in array manipulation and therefore we investigate such effort.

2.4.1 AQL

Array Query Language (AQL) is a language that manipulates multidimensional arrays [21]. Underneath AQL there is a data model based on the nested relational calculus of [6] with the addition of primitives to handle arrays.

AQL syntax is based on set comprehensions [5]. A comprehension has the form { f | q_1, q_2,$\cdots$, q_n }, where qualifiers qi can be either predicates or generators. A generator is an expression of the form x $\leftarrow$ A (reads x draws from A) that sequentially binds elements of the collection A to variable x. The qualifiers are evaluated from left to right. A binding is propagated until a predicate evaluates to false under this binding. The function f in the head of the comprehension is evaluated under all bindings that survived the predicates. The AQL rendition of the delta sales query shown below illustrates the use of comprehensions.

[**AQL**] – Delta Sales Query

```
{ ( month, sales[i] - sales[i-1] ) |
     [ \i : (\month,\sales) ] ← Sales,
     i > 0 }
```

The first qualifier is a special generator that binds array elements to variables. We assume here that Sales is an array of records (month, sales). The generator binds at once the position of a record to the variable 'i', and the record's components to variables 'month' and 'sales'. A backslash preceding a variable signals that the latter is being bound at that qualifier. The second qualifier is a predicate that filters out the very first record. Array indexes in AQL start at 0. This will avoid an out-of-bounds error in the head of the comprehension. It calculates the delta for a given month and puts the result in a record format.

Note that AQL generators use implicit order. The language could easily be extended with a sort user-defined function that would explicitly enforce the desired order for a query.

AQL is the first language described so far that supports an indexing operation – that is, a positional access to an element through a subscript. The previously introduced languages do support some notion of row numbering but would not allow it to be used to directly access an element. As the query demonstrates, such a feature comes with the responsibility of avoiding out-of-bounds errors.

2.4.2 KSQL

KSQL is the SQL dialect of the database management system KDB [20]. KDB has a fully vertically partitioned implementation of tables in which each column is a 1-dimensional array. For that, KDB tables have been called arrables (array-tables) [37].[2] An arrable's rows are intrinsically ordered.

What makes KSQL very effective for queries that other languages can express only through nesting is its column-oriented semantics. It allows a single block rendition of the best profit query.

[**KSQL**] – Best Profit Query
```
SELECT max(price - mins price) BY ID
FROM   Ticks
```

The variables in the query refer to entire columns rather than to single values. The construct 'BY' has the same effect of SQL's GROUP BY. Therefore 'price' in the SELECT clause is bound to a vector of prices partitioned by ID. The function mins() is being implicitly called once for each price partition and its result is subtracted from that very same price partition. The function max() then returns the highest difference of price for that partition.

KSQL can express a large set of order-dependent queries in a very concise way. But the language does not support declarative order; a change in the underlying order of Ticks would affect the result of this query. In addition, KSQL's syntax is somewhat far from the classical structure SQL practitioners are used to seeing.

[2] This term was actually first coined in [32].

2.5 Discussion

The investigation we have presented here is far from exhaustive but brings a significant cross-cut of order-manipulation mechanisms in existing query languages. We briefly cite other pertinent work. Array manipulation, in particular image pixel arrays, motivated at least two other works. AML [25] is a framework for generic function application over multidimensional arrays. It is based on a small, flexible set of algebraic operators in which several image manipulations can be encoded. RasQL [2] also manipulates image pixel arrays but using a SQL dialect. Both languages do not support declarative order. Ordered relations were also used elsewhere. In [27] the relational model was extended by providing the facility of user-defined orderings over data domains. The resulting model was called Ordered Relational Model. It supports a variation of SQL called Ordered SQL that could express the queries seen here, although with a peculiar syntax. Finally, sequences were used in an extension of Datalog called Sequence Datalog [4]. The latter kept the same syntax as Datalog but added two types of enriched terms, one capable of extracting sub-sequences from a given sequence (indexed sequenced terms) and the other capable of concatenating sequences (constructive sequence terms). Order wise, these works did not add any new mechanism that wasn't mentioned here.

The table 2.1 presents a qualitative comparison of the languages described previously. For each language we state which underlying data model and data structure it is based upon and what querying style and semantics it adopts. We also determine for each of them what order manipulation mechanism they incorporate. By "declarative order" we refer to the ability of explicitly stating the order in which data is supposed to be processed by a query. By "running aggregates" we mean the support for cumulative aggregates (e.g., running minimum) and windowed aggregates (e.g., moving average). By "inter-row" we refer to the ability of using values of two or more distinct rows in a single expression. Finally, "best profit query" shows whether a language's rendition of it requires a nested structure.

The table shows that there is no absolute best language. That answers our final question regarding whether any one given language would incorporate all necessary order manipulation mechanisms. AQL is arguably the most expressive of the languages – comprehensions have shown to be as expressible as languages such as OQL [13]. The missing linguistic features – declarative order and running aggregates – could be implemented through the addition of user-defined functions. We show in Chapter 4 that such user additions may be less than satisfactorily handled by the optimization process. We also have a bias for pragmatic reasons for SQL and its dialects, but reasonable people can differ on this point. KSQL has proven to be quite appropriate for order-dependent queries and was the only SQL dialect capable of expressing the best profit query in a single block.

Linguistic differences notwithstanding, arrays were the structure that provided the greater flexibility. Any operation that can be done on an ordered set, on a sequence, or on a table can also be done on an array representation of those structures. The inverse doesn't hold. Moreover, array indexing (positional access) can be used as a primitive to express every order-manipulation mechanism studied here. It was shown that indexing comes with the risk of having run-time (out of bounds) error, though.

In conclusion, we claim that arrays are the best suited data structure to support order-manipulations. We seek a language that can take advantage of the flexibility of arrays, and can express order-dependent queries in a natural, clear way.

	Underlying Data Model	Main Data Strucutre	Query-Language Style	Semantics
SQL:1999	Relational	table	SQL	row
SEQUIN	Object-Relational	sequence	SQL dialect	row
SRQL	Ordered Relations	ordered sets	SQL dialect	row
AQL	Nested Relational	arrays	comprehensions	row
KSQL	Relational	arrable	SQL dialect	column

	Declarative Order	Order in all clauses	running aggregates	inter-row	best profit query
SQL:1999	yes	no	yes	window	nested
SEQUIN	no	yes	yes	join	nested
SRQL	yes	yes	yes	yes	nested
AQL	UDF	yes	UDF	yes	single block
KSQL	no	yes	yes	yes	single block

Table 2.1: Comparative table of languages with order constructs

Chapter 3

AQuery Syntax and Semantics

3.1 An Array-Based Data Model

AQuery is a SQL dialect that is based upon an ordered data structure called an *arrable*, for array-table. Informally, an arrable is a collection of named arrays that, in their simplest form, are vectors of elements of a base type. In this form, an arrable is essentially a table organized by columns. An arrable's arrays may assume more complex shapes, though. They may contain array-valued fields themselves, but nesting beyond this point is not allowed.

Ticks	ID	price	date	ts
	ACME	12.02	05/11/03	1
	WXYZ	43.23	05/11/03	2
	ACME	12.04	05/11/03	5
	ACME	12.05	05/11/03	9
	WXYZ	43.22	05/11/03	13

(a)

Series	ID	price	date	ts
	ACME	[12.02 12.04 12.05]	05/11/03	[1 5 9]
	WXYZ	[43.23 43.22]	05/11/03	[2 13]

(b)

Figure 3.1: Example of two well-formed arrables

Figure 3.1 shows examples of two well-formed arrables. Both store stock quotes, but each in a particular shape. The column ID refers to the security identifier of a quote, price is the quote itself, and date and timestamp record the moment the quote occurred. Formally, arrables can be described as follows.

Definition 3.1 (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 of elements from a basic type. Let A be a finite array of elements of a type $t \in \mathcal{T}$. The cardinality of A is the number of elements in A's first dimension. 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 an array of type $t_i \in \mathcal{T}$. □

Definition 3.2 (Arrable Indexing) – The k-th *row* 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 $r[k] = \langle A_1[k], \cdots, A_n[k] \rangle$. □

For instance, Ticks[0] corresponds to the record $\langle ACME, 12.02, 05/11/03, 1 \rangle$, Ticks [1] to $\langle WXYZ, 43.23, 05/11/03, 2 \rangle$, and so on.

Because an arrable consists of arrays and arrays are ordered, an arrable's rows are ordered.

Definition 3.3 (Ordered by) – An arrable r may be (lexicographically) 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 $B_1[k_1] = B_1[k_2], \cdots, B_{i-1}[k_1] = B_{i-1}[k_2]$. Or, put informally, the tuples in the ordered projection of r onto $B_1, \cdots, B_m$ are lexicographically ordered. The definitions are symmetric for descending orders, but for the purpose of exposition, we will consider order to be ascending throughout this chapter. □

For instance, the arrable Ticks shown in Figure 2.1(a) could be defined as Ticks(ID, price, date, timestamp) ORDERED BY timestamp.[1]

The AQuery language borrows its syntax from SQL but it permits a query to specify the order in which it wishes to process rows. It does so through a new clause called ASSUMING ORDER introduced between the FROM and the WHERE clauses. The ASSUMING ORDER clause's semantic effect is to sort data immediately after the Cartesian product indicated by the FROM clause.

Definition 3.4 (Sort) – Let $r(A_1, \cdots, A_n)$ be an arrable and $B_1, \cdots, B_m \subseteq A_1, \cdots, A_n$. By a sort of r over $B_1, \cdots, B_m$, we mean a permutation s of r that is ORDERED BY $B_1, \cdots, B_m$. □

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

For instance, to find ACME's quotes ordered by timestamp, one would write

```
[AQuery]
SELECT price
FROM   Ticks
       ASSUMING ORDER timestamp
WHERE ID = 'ACME'
```

The order declared in the query is independent of that of the arrable to which it refers. A smart optimizer should be able to detect the coincidence and take advantage of it. (The topic of AQuery optimization will be further explored in Chapter 4.)

By enforcing order in the FROM clause of a query, all of the subsequent clauses may take advantage of it. We believe though that order throughout all clauses is a necessary, but not a sufficient condition for having clear, concise order-dependent queries.

3.2 Column-Oriented Semantics

One problem in expressing order-dependent queries is that often each resulting row is a combination of values of more than one input row. For example, consider a query to find the difference between each price and its previous value, assuming a time order. It needs to access two prices at once that are in distinct rows to calculate each pair's difference. *Row-oriented languages* such as SQL-92 can only iterate over only one row at a time, though. Thus they need to resort to either a self-join or an auxiliary construct to build a row that contains both prices. This operation has to be repeated for each pair.

In contrast, AQuery adopts a *column-oriented semantics* in that *variables are bound to entire arrays at a time*. Because variables in AQuery always refer to arrays, expressions always define mappings from a list of arrays to an array. For instance, the above pair-wise difference can be captured by a simple expression – 'price - prev(price)'. The function prev() over an array A is an array such that $\text{prev}_A[i] = A[i-1]$ if $i > 0$ and $A[0]$ if $i = 0$. For two arrays A and B such that $|A| = |B|$, minus (-) is element-wise subtraction.

The function prev() is a sample of the set of *vector-to-vector* functions that AQuery includes. These functions are classified according to their dependency on the input's array sort order and on the cardinality of the output they generate. For instance, prev() is *order-dependent* and *size-preserving*. The latter property indicates that it outputs vectors that have as many elements as the input array. Formally order-dependency can be defined as follows.

Definition 3.5 (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}, then $e(A_1, \cdots, A_m) and e(A_1^{perm}, \cdots, A_m^{perm})$ represent the same multiset. For example, avg(price) is order-independent. An expression that is not order-independent is *order-dependent*. For example, price - prev(price), is order-dependent. □

Other functions in the order-dependent, size-preserving category are the running aggregates. 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 $i = 0$. Running aggregates use this "s"-as-suffix pattern. 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 $0 \leq i < w - 1$.[2]

Another category of vector-to-vector functions are those that are order-dependent but not size-preserving. They reduce an array's cardinality and as such are called edge functions (i.e., they keep either the beginning or the end of an array). For instance, the first n positions of an array A, denoted first(n, A), is $\text{first}_{A,n} = A[0..n-1]$. Similarly, $\text{last}_{A,n} = A[|A| - n..|A| - 1]$.

The classic SQL aggregate functions (min, max, avg, count) can be seen as non-order-dependent, non-size-preserving vector-to-vector functions.

Example 3.1 – The combination of column-oriented semantics and array-typed expressions make it easier to write the best profit query (chapter 2). Recall that this query uses the Ticks arrable of Figure 3.1 to find what would be the best profit one could make by buying and selling a given stock in a given date. This query required a nested formulation when written in row-oriented languages, even order-aware ones. In AQuery, it can be written in a single block.

[**AQuery**] – Best Profit Query
```
SELECT max(price - mins(price))
FROM   Ticks
       ASSUMING ORDER timestamp
WHERE ID = "ACME" AND
       date = "05/11/2003"
```

The query should be read with a column-oriented mind-set. The FROM clause accesses the arrable Ticks and sorts it by timestamp order. In the WHERE clause, 'ID' and 'date' are both vectors; comparing each of them to scalars – ACME and

[2]Such a definition is commonplace in financial 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 as its argument.

05/11/2003, respectively – are valid array-typed expression that result in two booleans arrays. After they are combined, the resulting array maps each position of Ticks to true or false. Processing the WHERE clause means eliminating the false positions.

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 as is taking the max() of the resulting vector. □

3.3 Relational Manipulation of Arrables

The AQuery algebra supports the operators of the relational algebra. But here each operator takes array-typed expressions as arguments. If an expression is order-dependent, then the operator behaves in an order-preserving way. Otherwise the operator behaves in an order-cavalier way. We use the following order equivalence between arrays to define such behavior.

Definition 3.6 (Order-Equivalence) – Let r and s be arrables over the same set of attributes. Suppose that r is ordered by some attributes $X_1, \cdots, X_p$, and s by $Y_1, \cdots, Y_q$. Then r and s are *order-equivalent* with respect to attributes $B_1, \cdots, B_m$, denoted $r \equiv_{B_1,\cdots,B_m} s$, if the following conditions hold: (i) r and s are multiset-equivalent (i.e., there exists a permutation of rows P^1, P^2 such that $P^1(r) = P^2(s)$). (ii) $B_1, \cdots, B_m$ is a prefix of both $X_1, \cdots, X_p$ and $Y_1, \cdots, Y_q$.

When r and s are simply multiset-equivalent, we say that $r \equiv_{\{\}} s$. □

The order-cavalier variation of an operator is simply one that is multiset equivalent to its order-preserving variation. In the remaining of the section we define the order-preserving variations of the relational algebra operators.

3.3.1 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.[3]

```
projection(e,r)
1. s:= empty arrable having the same schema as e
2. for i = 0 to |r|-1
3.     append <e1[i],⋯,em[i]> to s
```

[3]Note: After considering different formalization notations including set comprehensions and lambda calculus we have decided to use a simple minded but (to us) clear loop formulation. In fact comprehensions would have worked well for many of the operators, but introduced problems for some such as certain variants of order-preserving joins

```
4. end for
5. output s
```

As mentioned before, if any e_i is order-dependent, the projection is said to be order-preserving, otherwise the projection is order-cavalier, denoted simply $\pi_e(r)$.

3.3.2 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.

```
selection(p,r)
1. s:= empty arrable having the same schema as r
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-cavalier. This may be interesting if we have a hash index for example.

Example 3.2 – Let e= max(price - mins(price)) and p= (ID = 'ACME') $\wedge$ (date = '05/11/2003'). The Best-Profit query can be translated to the AQuery algebra as follows,

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

□

3.3.3 Group By

Grouping in AQuery uses an arrable's facility to store array valued fields. Intuitively, grouping in AQuery partitions the operand arrable into disjoint sub-arrables that share the same group value. It then transforms each sub-arrable into a single row by replacing each non-grouped column (in the sub-array) by its equivalent array-typed value. For instance, the arrable Series in Figure 3.1 shows the effect of grouping the arrable Ticks in the same figure by ID and date.

Formally, let r be an arrable and $g = G_1, \cdots, G_m$ be a list of expressions over r's arrays such that $|G_1| = ... = |G_m| = |r|$. That is, to each r[i] there must exist a group characterized by g[i]. 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 having the same schema as g
2. s:= empty arrable having the same schema as r
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
```

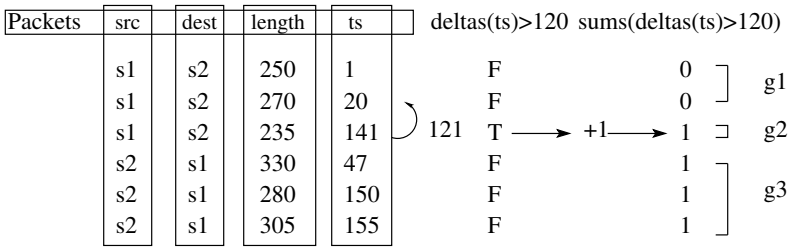
Step 13 above forms a single element list (or equivalently a vector). Step 8 concatenates to that list. The result is that fields may consist of vectors. As before, group-by is order-dependent if any of its grouping expressions are. Group-by can also have an order-cavalier variation in which the assembled arrays in fields may not be in the same order as in the original arrable.

Grouping in AQuery is independent of aggregation. To apply a function to each array-valued element of a column, AQuery provides an operator modifier called *each*. Formally, let the array A be a parameter (array) of a function F. The execution of F modified by 'each' is defined as follows

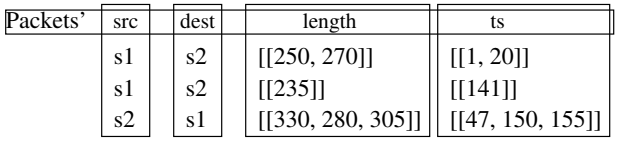
```
each(F, A)
1. B := empty array of the same type of F's result
2. for i = 0 to |A|-1
3.     append F(A[i]) to B
4. end for
5. output B
```

This definition can be naturally extended for cases where F takes more than one argument.

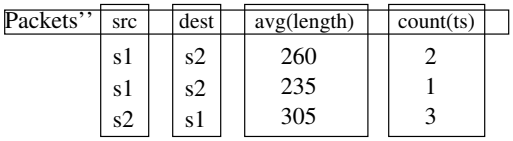
Example 3.3 – Consider the schema Packets(pID, src, dest, length, timestamp), where pID identifies a packet exchanged between a source (src) and a destination (dest) host. Length refers to the size of the packet and timestamp to the moment this packet was exchanged. A "flow" from a source s to a destination d ends whenever there is a 2-minute gap between consecutive packets from s to d [8]. Suppose a network administrator wants to know the count of packets and their average length within each flow. This query would need 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.



(a)



(b)



(c)

Figure 3.2: Intermediate arrables in the Newtork Management Query

In AQuery, such a grouping expression corresponds to the arrable 'src, dest, sums(deltas(timestamp)>120)'. 'deltas(col)' is the abbreviation of 'col - prev(col)'. Figure 3.2(a) shows how this expression is computed, supposing the Packets arrable is sorted over src, dest, and timestamp. The expression 'delta(timestamp)>120' finds for each packet whether it starts a new flow. Assuming that the boolean TRUE carries a value of 1, and FALSE of 0, the expression 'sums(deltas(timestamp) $>$ 120)' generates a unique flow identifier, when concatenated with src and dest.

The arrable we see in Figure 3.2(b) is the grouped one. Note that the columns of Packets that are not columns of g (the grouping expression) have arrays within fields. Because fields may be arrays (though not arrables), aggregate functions may apply over an entire column or over each field. In figure 3.2(c) we see that avg() was applied to each of the array-values of the column length.

The AQuery rendition is given below.

[**AQuery**] – Network Management Query
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, each(avg(),length), each(count(),timestamp) and g= src, dest, sums(deltas(timestamp) > 120), looks like the following. We mark with a corresponding superscript the operations that have components modified by each.

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

□

3.3.4 Flatten

Flatten generates a first normal form equivalent of an arrable that contains array-fields. It requires every row of the input arrable to be made of scalars, or of scalars and array-valued fields where the latter are of the same cardinality.

To define the flatten operation formally, assume that card() is a function that, given a row, returns the maximum cardinality of any of the row's elements. Further let r be an arrable made of arrays $A_1, \cdots, A_n$ and let there be a m such that $A_1, \cdots, A_m$, are vectors (contain only scalar elements) and $A_{m+1}, \cdots, A_n$ contain array fields as described earlier. Flatten over r could be defined as follows

```
flatten(r)
1. s:= empty arrable having the same schema as r
2. for i = 0 to |r|-1
3.     for j = 0 to card(r[i])
4.         append <r.A1[i], ···, r.Am[i], r.Am+1[i][j],···, r.An[i][j]>
5.     end for
6. end for
7. output s
```

Example 3.4 – Financial analysts often observe stock tendencies before making purchase decisions. Moving averages are capable of smoothing the volatile stock price curves and exposing underlying optimistic and pessimistic sentiment. For instance, whenever a short-term trend curve (a 5-day moving average) crosses above a longer-term one (21-day moving average) technical analysts would suspect the stock will move up soon.

The following query would be involved in this analysis.

[**AQuery**] – Crossing Averages Query
WITH
averages (ID, date, a21, a5) AS
(SELECT ID, date,

```
            avgs(21, price) as a21,
            avgs(5, price) as a5
    FROM    Ticks
            ASSUMING ORDER ID, timestamp
    GROUP   BY ID)
SELECT ID, date
FROM   FLATTEN(averages)
       ASSUMING ORDER ID, timestamp
WHERE  a21 > a5 AND
       prev(a21) <= prev(a5) AND
       prev(ID) = ID
```

This query finds the dates where the 21-day and the 5-day moving average for a given set of stocks cross. The WITH construct from AQuery was borrowed from SQL:1999. It defines a "local view" that can be referenced only in the FROM clauses of subsequent WITH queries or in the main query. Note that the view returns the averages as array fields for each ID and date (non-1NF arrable). The next step is simply to check crossings.

Let e= ID, date, avgs(21,price), avgs(5, price) and p= a21 < a5 $\wedge$ prev(a21) > prev(a5) $\wedge$ prev(ID) = ID. The crossing averages query is translated to the AQuery algebra as follows

$$r \leftarrow \pi_e^{op}(gby_{ID}^{op}(\text{sort}_{ID,timestamp}(\text{Ticks})))$$

$$\pi_{ID,date}^{op}(\sigma_p^{op}(flatten(r)))$$

□

3.3.5 Cross Product and Join

Cross-product ($\times$) in AQuery is order-cavalier and hence has the same definition as in the relational algebra. By contrast, joins may have several variations depending on whether and how the order of the input arrables is preserved. Let $r(A_1, \cdots, A_n)$ and $s(B_1, \cdots, B_m)$ be arrables. 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 with schema ⟨A1,···,An,B1,···,Bm⟩
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 ⟨A1[i],···,An[i],B1[j],···,Bm[j]⟩ to o
6.         end if
7.     end for
8. end for
9. output o
```

A query's order may require that only one of the join operand arrables' order be preserved. In that case a simpler order-dependent variation of the join can be used. Suppose that $r(A_1, \cdots, A_n)$ is the arrable for which order should be preserved. A *left order-preserving* join, $r \bowtie_p^{lop} s$, is one that is order-equivalent with respect only to $A_1, \cdots, An$ to a left-right order-preserving join of the same two arrables.

Example 3.5 – The arrable Portfolio(ID, tradedSince) ORDERED BY ID, stores information about the stocks that makes one analyst's portfolio. It is a subset of the stocks that appear in Ticks. If this analyst wanted to extract the ten last quotes for each stock that he or she traded, then the following query could be issued.

[**AQuery**] – Non-1NF Result Query
```
SELECT t.ID, last(10, price)
FROM   Ticks 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. As mentioned before, Cross-product in AQuery is order-cavalier. 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 price column to a maximum of the ten last positions of each price array. Letting e= ID, each(last(),10,price) and p= Trades.ID=Portfolio.ID, this query can be represented as follows. (Note that last() takes two arguments and therefore that is reflected on the syntax of the 'each' call.)

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

□

3.4 Positional Manipulation of Arrables

AQuery exploits the ordered nature of arrables so as to support the referencing of rows by their position. We describe two order-manipulation mechanisms that use this facility.

3.4.1 Querying with Arrable Indexing

Because expressions in AQuery are array-typed, it is only natural to allow indexing (i.e., access to an array's element given its position) in the language. Yet an access

to a non-existent position would cause a run-time error. AQuery avoids such a possibility by introducing the notion of safe index sequence generators (SISG). A SISG is an expression that is always evaluated to a valid sequence of indexes. For instance, the expression 'price[ODD]' uses the SISG 'ODD' to return all prices from position 1 until the last odd position in price in that context. Other SISG is EVEN, which works similarly to ODD but starts at 0 and uses only even positions. We use an example to introduce the SISG EVERY n.

Example 3.6 – Suppose a financial analyst wants to know the standard deviation of prices for ACME's stocks and whether it would be accurate to work with samples of its prices instead. The following query would return the standard deviation for the price column of Ticks, for samples at every 10th price, and at every 100th price. The function stddev() is a built-in one and calculates the standard deviation for a vector.

[**AQuery**] – Array Indexing Query
```
SELECT stddev(price), stddev(price[EVERY 10]), stddev(price[EVERY 100])
FROM   Ticks
       ASSUMING ORDER timestamp
WHERE ID = 'ACME'
```

□

3.4.2 Querying with Row Direct Addressing

After a query's FROM clause is evaluated, the resulting arrable implicitly gains an additional column (vector) called ROWID. This synthetic column can be referred to anywhere a regular column can.

Example 3.7 – Good candidate stocks for day-trading may be among the most early-traded stocks. To discover which stocks were quoted within the first thousand quotes of a given day and how many times, the following query may be issued.

[**AQuery**] – Row Direct Addressing Query
```
WITH
    OneDay AS
    (SELECT ID, price, timestamp
    FROM    Ticks
            ASSUMING ORDER timestamp
    WHERE   date = '05/11/2003')
SELECT ID, count(*)
FROM   OneDay
       ASSUMING ORDER timestamp
```

```
WHERE  ROWID < 1000
GROUP  BY ID
```

The WITH query filters out ticks that did not occur in the desired date. The main query's WHERE clause will eliminate all the rows having ROWIDs 1000 or greater, according to timestamp order. Note that this query requires two steps so as to make sure the date filter is executed before the ROWID one. □

3.5 Comparing AQuery to Other Order-Aware Languages

AQuery's renditions of order-dependent queries are usually more concise than those of row-oriented languages. To illustrate this point we present the SQL:1999 rendition of the Network Management query presented earlier. Recall that this query's goal is to break sequences of packets (sessions) between pairs of hosts down into "flows" and to calculate statistics of the latter. A flow between a pair of hosts ends – and a new one starts – whenever they stop communicating for a period of 120 seconds or more.

[**SQL:1999**] – Network Management Query

```
WITH
   Prec (src, dest, length, timestamp, ptime) AS
   (SELECT  src, dest, length, timestamp,
            min(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
```

Separating the flows requires checking the intervals between time-consequent packets for a given pair of hosts. Expressing this calculation in SQL:1999 is not entirely straightforward. 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.

By contrast, the combination of AQuery's column-orientation, underlying data model, and built-in support for order makes it easier to write the same query. For convenience, we repeat the query here.

[**AQuery**] – Network Management Query

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

For all queries presented in this chapter we found the same sort of structural discrepancies between AQuery and SQL:1999 renditions that the Network Management query presents. Ultimately, if a calculation depends on several row values at once, a row-oriented language needs an auxiliary construct to align those values in a row. Nevertheless, several row-oriented languages in the literature provided inspiring insights.

AQuery borrowed from SRQL the early introduction in a query of an order defining clause. AQuery differs from SRQL in that the latter has a row-oriented semantics. Therefore several expressions that are valid in AQuery are not so in SRQL. SRQL cannot handle the table equivalent of non-1NF arrables. We have shown that this feature was useful in order-dependent queries.

In contrast, SEQUIN can handle tables that have sequence-valued fields. But whenever a query involves fields of both the table and the sequence, SQL is used to deal with the former and SEQUIN with the latter. That can lead to somewhat difficult-to-read queries.

AQuery takes its main inspiration from KSQL: namely its arrable notion, and its column-oriented semantics. AQuery differs from KSQL by trying to preserve the SQL flavor to a much greater extent than KSQL, by the introductin of the ASSUMING ORDER clause to make the use of order declarative, and (though this is independent of the semantics) by using cost-based optimization.

3.6 Conclusion

AQuery was designed to support order-dependent queries without compromising on backwards compatibility to SQL-92 (modulo nested capabilities). AQuery's

clauses are the same as SQL's – ASSUMING ORDER is an optional clause – and support all expressions that SQL clauses do, even though the former is column-oriented and the latter is row-oriented.

AQuery mets the criteria defined earlier for order-aware languages. It has declarative order, semantically the operations preserve order and the order idioms may claim to be intuitive.

One characteristic makes AQuery particularly amenable to query optimization. It is rather simple to identify which expression in a query and thus which clauses are order-dependent. A simple type analysis can check whether the use of an order-dependent function makes an expression order-dependent. We will see in the next chapter how an optimizer can take advantage of that knowledge.

Chapter 4

AQuery Optimization

4.1 Introduction

A SQL query specifies how its result must look but it does not establish how to compute it. For instance, if a query's WHERE clause is a conjunction of two predicates (p_1 AND p_2), either predicate may be chosen to execute first or execution orders may be mixed. An optimizer decides so in a cost-conscious fashion, usually executing the most selective predicate first or the one having a useful index.

This optimization approach presupposes a query to have alternative query execution plans (QEPs). A first QEP comes naturally from translating the query's text into its algebraic equivalent using the order of operations given by the query. Other QEPs may be obtained by the application of *query transformations*, rearrangements of the operators in a QEP that do not alter the query's semantics. In our previous example, the optimizer could first use a transformation that broke a composite selection into two simple ones. It could then decide on any order, for there is a transformation that could commute this pair of selections [10].

AQuery optimization can be done in this transformational fashion. If a query does not contain any mention of order then it can be optimized exactly as SQL would be. If early ordering is used (ASSUMING ORDER clause) then transformations involving sort can be applied. In the literature, sort transformations were addressed in two distinct but complementary ways.

The transformations in [33] avoid redundant sorting work by either eliminating the sort altogether or by reducing the number of columns over which sort is done. To eliminate a sort over a table r with respect to column A_1 ($sort_{A_1}(r)$), A_1 must be found to be a prefix of the existing order of r's records. This is the case whenever an index (to be precise, an index that orders its key information such as a B-tree) clustered by A_1 is used to scan r, or whenever a predicate such as '$A_1 = value$' has been previously evaluated over r. To reduce a sort over r in respect to columns A_1 and A_2 to $sort_{A_1}(r)$, either A_1 must be a key of r or A_1

must functionally imply A_2.

Example 4.1 – Suppose Connections(host, port, client, timestamp) ORDERED BY timestamp is an arrable that stores the clients' addresses that accessed a network's services (port, host) and when did they do so. The following query fetches the clients that connected to host 'atlas' in timestamp order. (The use of ASSUMING ORDER here is merely illustrative; more realistic queries will follow this introduction.)

```
[AQuery]
SELECT client
FROM   Connections
       ASSUMING ORDER timestamp
WHERE  host = 'atlas'
```

The QEP derived directly from the query text is shown in Figure 4.1(a). We show plans in the usual diagrammatic way but introduce some auxiliary notation as follows. A single arc between a pair of operators means that the producer operator is outputting records in an order-cavalier fashion (i.e., in the most efficient or simple way possible, without guaranteeing any order). Double-arcs mean it is doing so in an order-preserving way. 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 4.1.

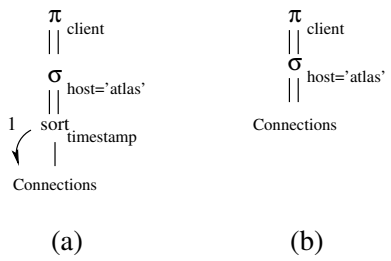


Figure 4.1: An initial QEP and the application of a sort elimination transformation

Note that operators after the sort are connected by double-arcs. This implies that they maintain the order the sort imposed. The sort can be eliminated because it matches the order defined for the arrable Connections, according to transformation 1 in Table 4.1. The resulting plan is shown in Figure 4.1(b). □

The other way order transformations were addressed in the literature was in a context in which sort and other operations interacted [35]. If relations were lists rather than sets of records and relational operations were carried in an order-preserving fashion, then sort and projection would be commutative. Similarly,

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

Table 4.1: Equivalences between sort and remaining algebra operators

sort and selection would be so, and sort could be pushed-down over a join – the complete list appears in [34]. These transformations all apply to AQuery as well.[1]

Example 4.2 – Take the same query and arrable as in the previous example, but this time let Connections be ORDERED BY host and timestamp. For convenience, the initial syntax-driven QEP is repeated in Figure 4.2(a). Note that the selection (host = 'atlas') can now benefit from the existing order (host, timestamp). Note also that by evaluating the selection first, fewer records will need to be sorted. The transformation 3 commutes a selection with a sort and when applied here it generates the QEP shown in Figure 4.2(b).

There is a way to further save the work involved in the sort. At this point, the order of records between the Connections scan and the selection is irrelevant (single arcs), as is the order between this latter and the sort. Changing the order of the records in a portion of the plan in which order is irrelevant does not impact the semantics. Thus the selection can be converted into an order-preserving one by transformation 4 so as to propagate the host, timestamp order. Since the selection output contains only records from host 'atlas' it can be said to be ordered by timestamp. This in turn makes the sort redundant. The latter

[1]However, [35] and [34] consider that every expression is an order-independent one. The implications of this are discussed in 4.3.

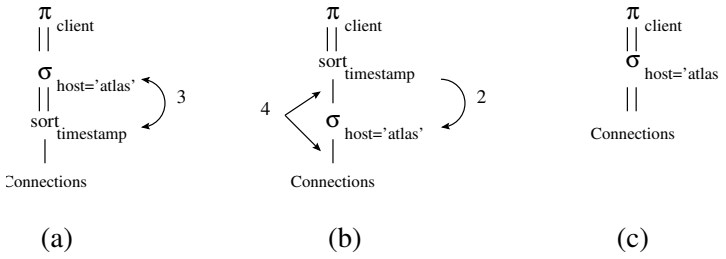


Figure 4.2: An initial QEP and the application of a selection push-down transformation

manipulation is captured by transformation 2 in Table 4.1 and the final plan is depicted by Figure 4.2(c). □

4.2 Optimization of Edge Selections

Optimization of AQuery goes beyond sort elimination and move-around. AQuery's order idioms are often built around the edge-functions (e.g., first(), last()), which allow rather aggressive optimizations as well. We introduce these new techniques through examples and evaluate their relative performance improvement.

4.2.1 Implicit Selections and Sort-Edge

Let us use the arrable Connections once again, this time ORDERED BY host. In an intrusion detection scenario an administrator may wish to find the last client that connected to a given server. In AQuery this query would look like the following.

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

The above query's initial plan is depicted in Figure 4.3(a). A regular selection such as $\sigma_{host='atlas'}$ can be pushed down over a sort [35]. Transformation 3 in Table 4.1 is a slight variation of that transformation but here order-preservation or lack thereof is made explicit. The advantages of this transformation are the same as example 4.2's: sort work reduction. But once more, the gains can go further.

The projection $\pi_{last(1,client)}$ includes an implicit selection, i.e., it is only interested in one client. This is a particularity of AQuery's column-oriented semantics – a projection over a function that itself performs a selection. The transformation 5 in Table 4.1 is a new transformation that replaces a projection with indexing

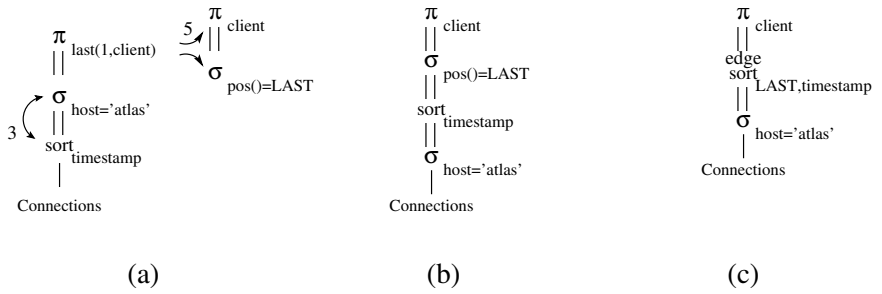


Figure 4.3: Implicit selection and sort-edge optimization

by a pure projection plus a selection of the desired positions. We say $\text{pos}(r) = i$ when we refer to the record $r[i]$. The special indexes for an arrable r, FIRST and LAST, are 0 and $|r| - 1$, respectively. 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 4.3(b).[2]

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.

AQuery implements the logical pattern $\sigma^{op}_{edge-condition}(sort(r))$ through a physical operator called sort-edge. It uses a modified heap-sort to keep the top (or bottom) n elements, as appropriate. This is similar to the approach used in [7] except that we modify the heap-sort to make it stable.[3]

The gains in performance provided by sort-edge are considerable. In Figure 4.4(a) we compare a sort-edge with a regular sort operation. The graph shows that for small slabs – 1 and 10 elements – sort-edge takes a small fraction of the time needed to sort the entire set.

The benefits of such work reductions reflect in the performance improvement of our example query. Figure 4.4(b) compares the performance of the three plans of shown in Figure 4.3 for a Connections arrable of 1 million rows divided across a varying number of hosts. The "naive" plan performs the entire sort and only then filters for host 'atlas'. The other plans start by executing the latter selection, which can benefit from existing order. Note that in the 1- and 10-distinct hosts scenarios a considerable number of rows have to be sorted and sort-edge delivers a much better performance than regular sort. For the remaining scenarios, very few rows survive the selection and both optimized plans perform equally well.

[2]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.

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

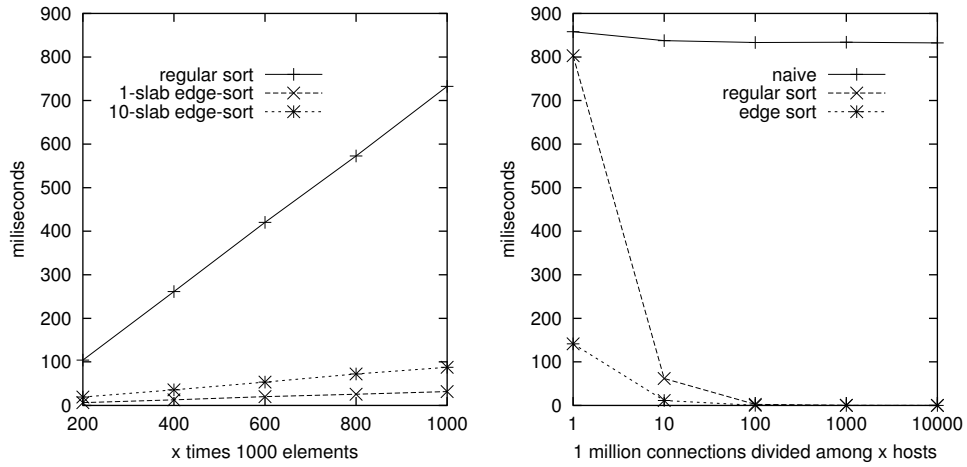


Figure 4.4: Efficiency of sort-edge technique

4.2.2 Sort Splitting

There are situations in which the arrable's existing order facilitates the evaluation of part of a query even though it does not match the query's assuming order. The sort splitting technique applies to this kind of scenario. Consider again the arrable Connections ORDERED BY host. The following query finds all the clients that connected to the last host to be accessed.

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

An initial plan for this query appears in Figure 4.5(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.

The sort-splitting technique says that 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 8 in table 4.1. The benefit of the semi-join is that we can now manipulate order on each of the semi-join's arguments independently.

Figure 4.5(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

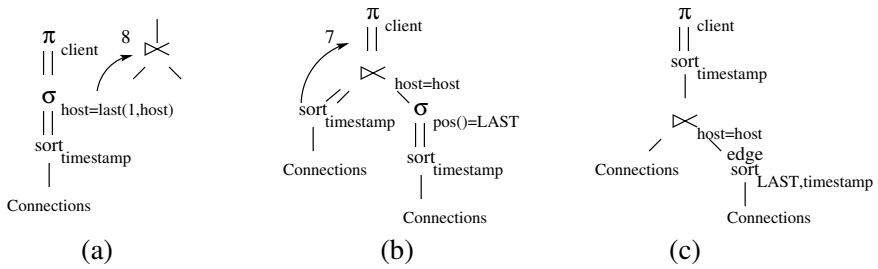


Figure 4.5: Sort-splitting optimization

thus defer it until after the join. The transformation 7 in table 4.1 commutes a semi-join and a sort. It states that under certain conditions sorting a semi-join is equivalent to sorting its left stream and then performing an order-preserving semi-join. The conditions hold here.

This transformation's impact here is two-fold. First, the evaluation of the semi-join predicate is facilitated by an existing order. Second, sorting over timestamp has to be done just over records generated by the semi-join. This is much cheaper than the original semi-join over the whole arrable. The resulting plan appears in Figure 4.5(c).

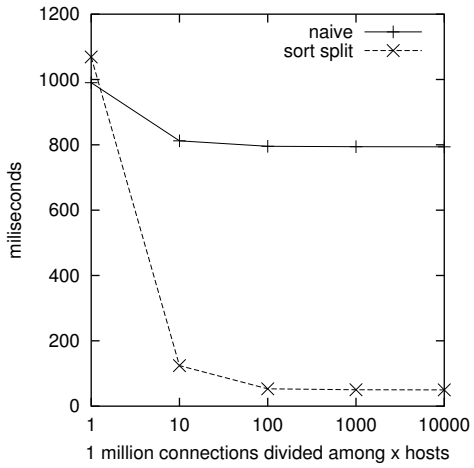


Figure 4.6: Efficiency of sort-splitting technique

Figure 4.6 shows the performance gains of applying the sort splitting technique to the example query. 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.

4.2.3 Early Edge Selection and Edgeby

Edge selections can often be performed very early in a query. This requires transformations that push edge selections all the way through operations such as joins.

Consider the arrable Ticks(ID, date, price, timestamp) ORDERED BY timestamp, which stores stock quotes, and the arrable Portfolio(ID, name, tradedSince) ORDERED BY ID that stores the subset of securities with which an analyst deals. Name is a unique identifier of securities in Portfolio, and so is ID. An analyst may want to retrieve the last price of a security by its name through the following query.

```
[AQuery]
SELECT last(1, price)
FROM   Ticks, Portfolio
       ASSUMING ORDER timestamp
WHERE  Ticks.ID=Portfolio.ID
       AND name = 'ACME'
```

An initial plan for this query is depicted in Figure 4.7(a). Note that in this plan the selection is carried after the join and the sort. It would be more advantageous to perform it earlier. A regular selection can be commuted with the sort by the application of transformation 3, as was done before. Because the selection would then be in a portion of the plan that is order-independent, it could then be pushed down over the join using the classic join-selection commutativity [10]. The result is seen in Figure 4.7(b).

Upon realization that the order of Ticks matches the ASSUMING ORDER of the query, the optimizer would try to eliminate the sorting completely. Transformation 6 in table 4.1 commutes a join with a sort while still keeping track of order. That is a slight variation of a transformation in [35] in which order-preservation is made explicit. As Trades is already ORDERED BY timestamp, that sort may be eliminated, as transformation 1 in Table 4.1 makes possible. The result is seen in Figure 4.7(c).

This query also contains a projection-with-selection, and again they can be broken apart. The consequent presence of the edge selection after the join suggests that it may be unnecessary to perform the join in its entirety. Portfolio.ID is a key and therefore it guarantees that each record in Ticks will match at most one record in Portfolio. (Foreign key joins are among the most frequent of equijoins.) Under these conditions we could push down this edge selection in the following way: For each ID in Trades, find its last record by grouping Trades by ID and selecting each last record. By applying the edge selection earlier, the query's join examines far fewer rows than before. The final selection would then pick the desired price. This is what transformation 9 in table 4.1 does. The final plan is shown in Figure 4.7(d).

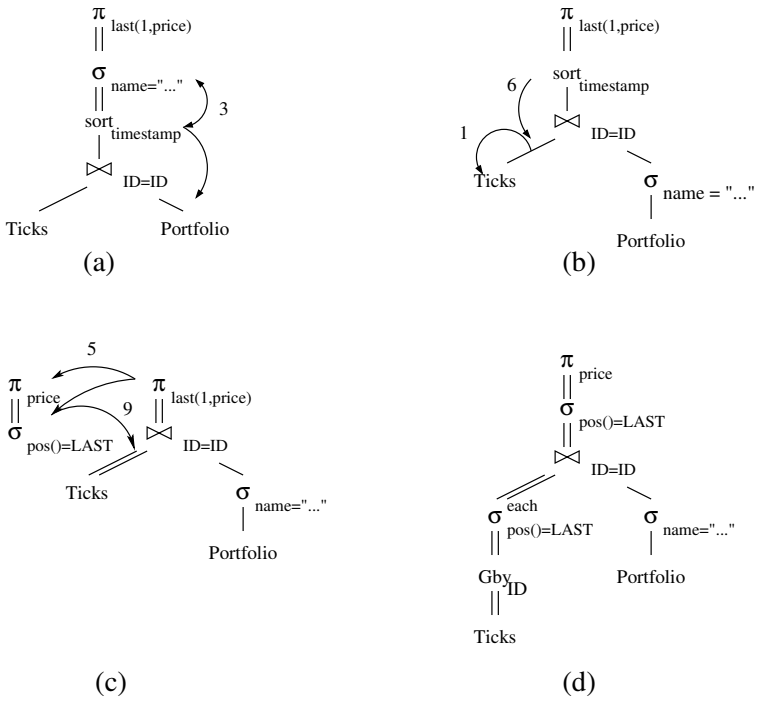
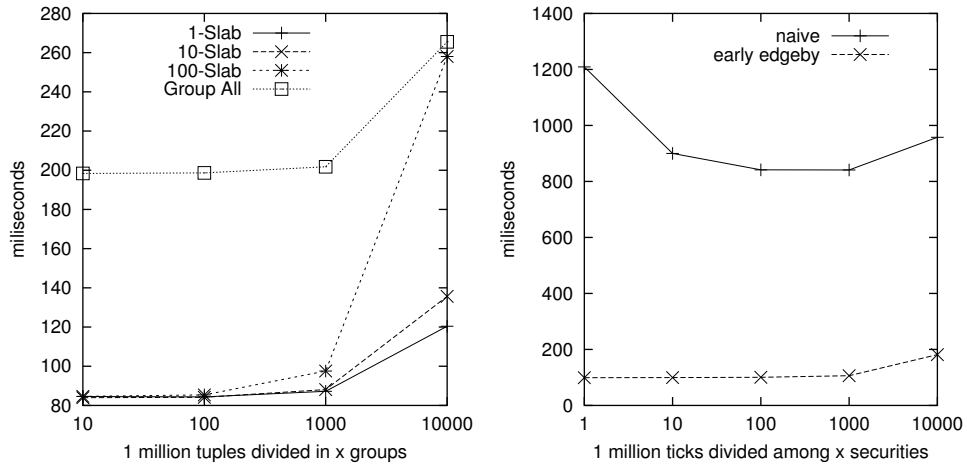


Figure 4.7: Early edgeby optimization

Replacing an edge-selection by a grouping operation and the very same edge-selection may look more expensive, but it isn't. 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 violate 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 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 4.8(a). We used the arrable Ticks 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 give 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 result of applying the early edgeby technique is shown in Figure 4.8(b).



(a) (b)

Figure 4.8: Efficiency of the early edgeby technique

The naive and the optimized plans for the example query can be seen. By applying a 1-slab 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.

4.2.4 Sort Embedding

If a GROUP BY operation follows a sort it may be advantageous to invert their order. Performing a single sort over the entire data is often more expensive than performing several sorts, each embedded within a group.

Consider again the arrable Ticks, this time with no determined ORDERED BY. (Often ticks arrive in a "near-timestamp" order.) The following query fetches the ten most recent prices for each security ID

```
[AQuery]
SELECT ID, last(10,price)
FROM   Ticks
       ASSUMING ORDER ID, timestamp
GROUP  BY ID
```

An initial plan for this query is shown in Figure 4.9(a). We can separate the implicit selection from the projection as we did before. The resulting plan appears in Figure 4.9(b).

It is possible to delay sort until after the GROUP BY ID is done. If delayed, sort would have to be applied only within each group. Moreover, for this particular query the smaller sorts would then be followed by edge selections – sort-edge would

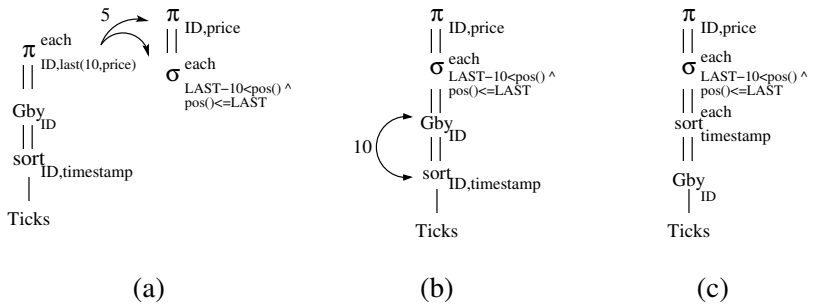


Figure 4.9: Sort-embedding optimization

apply. The transformation 10 in table 4.1 allows commuting 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); and (b) grouping must be over a prefix of sort's arguments. The result of this transformation is shown in Figure 4.9(c). Note how a double-arc connects group-by and sort-each, because this instance of group by is order generating.

Figure 4.10(a) characterizes the 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. Replacing one big sort by several smaller ones starts to payoff whenever more than 10 groups exist.

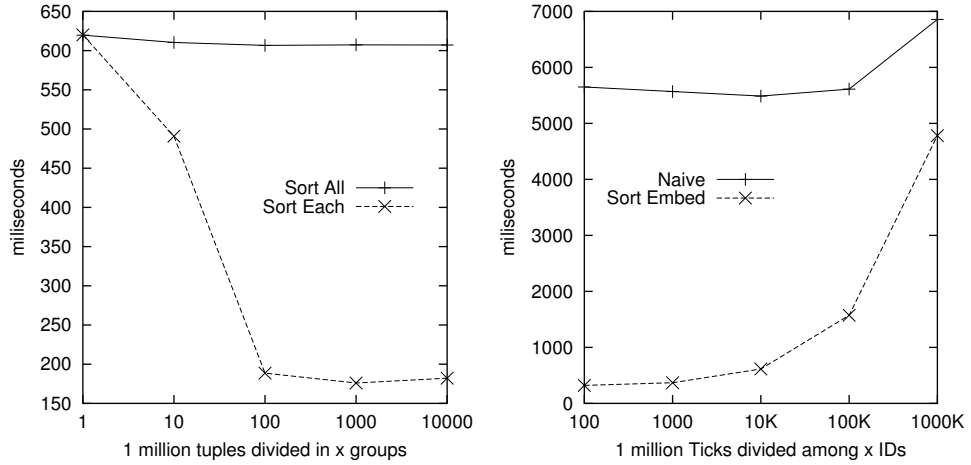


Figure 4.10: Efficiency of the sort embedding technique

The efficiency of sort-embedding reflects in the performance of the example query. Figure 4.10(b) shows the comparative performance of plans for the naive and optimized cases. 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.

4.3 Related Work

Order management is not normally fully integrated in a query's optimization process. Sort has traditionally been seen as a physical property to be included in a plan (if not specified by SQL's ORDER BY clause) only to support an efficient algorithm such as merge join. Mechanisms such as Starburst's "glue" [22] or Volcano's "enforcer" [12] made sure a sort step was added whenever an efficient algorithm required it. This approach to order management was shown to miss several optimization opportunities [33].

The authors in [33] suggested and implemented an order-management step in their optimization process. It greatly improved QEPs for queries that have order requirements due to the clauses ORDER BY, GROUP BY, or the DISTINCT modifier. The transformations 1 and 2 in table 4.1 come from their work. Yet this order management step did not consider interactions between sort and other operations – an important condition to perform order-management as an integrated aspect of query optimization.

To the best of our knowledge the authors in [35] were the first to promote sort to a logical operator and to suggest that order optimization could (i) be transformational and (ii) be considered at the same time as other transformations. AQuery follows this same idea.

While using several of the transformations in [35] – namely transformations 3, 6, and 7 in table 4.1 – AQuery has contributed a few of its own – transformations 5, 8, 9, and 10. The transformations suggested here go a step further by taking into account the fact that expressions in a query can be order-dependent. (In [35] expressions are all order-independent.) This allowed, for instance, to identify that two selections over order-dependent expressions do not commute, but they would had the expressions been otherwise. AQuery also differs from that work by treating edge selections as a case of order-dependent selections rather than as a new operator (called operator *top-k* in [35]).

In fact, edge selections were first described in [7]. A clause STOP AFTER was suggested that was capable of limiting the final cardinality of queries whose results were ordered (by an ORDER BY). The sort-edge operator presented here was motivated by a similar operator described there. There are two differences that distinguishes AQuery from that work, though. First, in [7] the optimization process for STOP AFTER was considered in a phase prior to – rather than

integrated with – the phase in which plan enumeration takes place. Therefore interaction between sort and other operators are not fully considered. Second, AQuery language allows edge selections to occur at any point of the query and not only at its end. We have shown, for instance, queries that applied edge selections within groups (e.g., last 10 quotes of each security in a Portfolio). We have also shown that new techniques such as sort embedding can be applied to optimize these queries.

A particularly inspiring integrated optimization technique appeared in [21]. The AQL optimizer can manipulate 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 splitting 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. A complete fusion of these ideas requires more exploration.

Chapter 5

System Design and Implementation

5.1 A Column-Oriented Execution Model

The AQuery system is a database management system that implements the arrable-based data model and that supports AQuery as the query interface. In developing a new database system, we wanted to investigate how the use of arrays as the basic data structure could work for performance. Our studies started with how to execute a query, once a query execution plan is obtained.

To carry out an execution plan a system has to run each of the plan's operators. There are some alternatives to do so. The most common way is to equip each operator with a *get-next-row* interface (getNext()) that returns one row of the operator's result at a time. This execution model is called *iterator-based* [11]. To obtain a plan's first resulting row the system calls the root operator's getNext() interface. In turn, the root operator calls the getNext() of the operator from which it consumes rows. This cascading effect propagates down the plan until a leaf operator reads an input's row. The row is passed back to the caller and it gets processed on its way up. Assuming it belongs to the query's result, it eventually arrives back at the root operator. The system starts the process over again until an end-of-rows is signaled.

The iterator model has several advantages, but its efficiency is poor especially because of poor cache. Cache inefficiency is due to the loading of unnecessary columns' data to perform an operation [1]. For instance, if a selection's predicate involved only one column, it would not need an entire row of values to evaluate the predicate. The unused fields occupy precious cache space.

In addition, the iterator model has an inherent overhead of a function call per row per operator. In a simple-predicate selection, a call's cost (parameter stacking and context saving) can be of the same order as the evaluation of the predicate itself.

A contrasting execution model is the *column-oriented* one found in the Monet database system [3]. Instead of handling a row at a time, the model assumes data is vertically partitioned in columns and manipulates these columns as units. For instance, a selection would look at the column involved in the predicate, and only at this, and would return the row IDs and that column's data that satisfy the predicate. This model presents better performance in modern architecture hardware – hierarchical memory and super-scalar CPU – than the iterator one [24].

The AQuery system uses a column-oriented model as well, but slightly more generally than Monet. The model maps easily to arrables which are naturally partitioned in arrays (columns). To perform the above selection, the AQuery system might materialize its results as Monet does. But this would involve performing memory copies of data. Alternatively, the AQuery system passes on the next operator the reference to the data it operated upon along with a collection of indexes to the still relevant rows. This avoids unnecessary materialization. We use the term *effective index array* to denote this collection of indexes resulting from one operator that may be passed along to subsequent operators. A running example of a plan demonstrates how this concept works.

Consider the arrable Trades, depicted in Figure 5.1, which stores prices at which stocks exchanged hands, and at which quantities (volume) and timestamps these transactions took place. The arrable Base, shown in the same figure, classifies stocks according to the type of business they conduct, using a Standard Industry Code (SIC). For visualization purposes we show the indexes of each arrable's rows in the left-hand column (small fonts under the arrables name). Consider further the following query that fetches the last quote of each stock in the COMPUTER industry.

```
[AQuery]
SELECT t.ID, last(price)
FROM   Trades t, Base b
       ASSUMING ORDER t.ID, ts
WHERE t.ID = b.ID AND
       SIC = "COMP"
GROUP  BY t.ID
```

The first step of a plan for this query is to initialize an effective index array. For clarity, let us assume temporarily that the arrables fit entirely in memory. The initial index array would contain all the existing indexes for both arrables, that is, 0..6 for arrable Trades and 0..2 for Base. This structure is depicted in Figure 5.2(a).

Following the query's text, the next operation in the plan would be the join of the two arrables. Because the predicate involves the columns Trades.ID and Base.ID, the join operation needs to access only these columns. The effective

Trades	ID	price	ts
0	ACME	12.05	1
1	XYWZ	42.35	2
2	ACME	12.04	3
3	EMCA	17.19	4
4	EMCA	17.20	5
5	ACME	12.02	6
6	XYWZ	42.37	7

Base	ID	SIC
0	ACME	COMP
1	XYWZ	AUTO
2	EMCA	COMP

Figure 5.1: Two example arrables and their rows indexes

Initialize index array

Trades	Base
0	0
1	1
2	2
3	
4	
5	
6	

(a)

Join of Trades and Base

Trades	Base
0	0
2	0
5	0
1	1
6	1
3	2
4	2

(b)

Sort by ID

Trades	Base
0	0
2	0
5	0
3	2
4	2
1	1
6	1

(c)

Filter for "COMP"

Trades	Base
0	0
2	0
5	0
3	2
4	2

(d)

Group by ID

Trades	Base
[0, 2, 5]	[0, 0, 0]
[3, 4]	[2, 2]

(e)

Materialize the result arrable

Result	ID	c1
0	ACME	12.02
1	EMCA	17.20

(f)

Figure 5.2: Effective index array during a query execution

index array depicted in Figure **??**(b) shows how this operation recorded which pair of positions of Trades and Base satisfied the join predicate.

Although irrelevant at this point of the query, the index array imposes an ordering for the rows of the intermediate result so far. Coincidentally, the first row of the intermediate result is a join of the first rows of the arrables, Trades[0] and Base[0]. The following row is the join of the third row of Trades, Trades[2], and the first row of Base, Base[0] – and so on. Note that the indexes used here are positions relative to the arrables. Data need not be stored in ascending index order (storage physical independence). What is important here is the ability to fetch a row by its position, a natural approach for arrays.

The ordering of rows in the current index array is not the one specified by the query's ASSUMING ORDER clause and thus a next step in a plan could be a sort. Recall that the ASSUMING ORDER is over Trades.ID and Trades.timestamp. Here again only those columns need to be accessed to compute the new index array. The sort would rearrange the indexes in a way to impose an ID and timestamp order. The Figure 5.2(c) shows the result of the operation.

Note that the sort took as input the result of the join operation. The permutation occurred on pairs of indexes of both arrables.

A selection may follow that eliminates the indexes that do not correspond to computer industry's stocks. The only column needed here is Base.SIC. The selection loads it and dereferences using the effective indexes that correspond to the Base arrable. Note that the Base.SIC column has three values in the original arrable. But here, the resulting data vector has six positions (0, 0, 0, 2, 2, 1), for a join and a sort have already been applied. The selection thus generates the effective index array seen in Figure 5.2(d).

Note that the selection eliminated the indexes that contained 1 in the Base portion of the index array.

The next operation is the GROUP BY. The first step here is to index the column Trades.ID by the portion of the index array corresponding to the arrable Trades. Then, the operation captures the notion of groups by putting the indexes of rows that belong together in a same array-valued field of the effective index array. Figure 5.2(e) shows the effect of GROUP BY.

The effective index array shows that the result of calculating the query up until the GROUP by has two rows. Moreover, each of the columns that was not grouped-by will have array-values fields. For instance, if one wished to fetch Trades.prices for the first group, one would load that column and use values at positions 0, 2, and 5. For the second group, positions 3 and 4 would be used instead.

Finally, the projection in the SELECT clause may be applied. A final projection means materialization of the results. Here it takes into consideration that ID was grouped-by. Only the first index of each group is used in the result (i.e., ID[0] and ID[3]). By contrast, the function last() is called once for each group

and is passed the corresponding price arrays for each group (price[0 2 5] for group 0, and price[3 4] for 1). It returns the values of price[5] and price[4] respectively. The resulting arrays are then assembled into an arrable, show in Figure 5.2(f).

The column-oriented execution model does not prevent a plan from using a constrained amount of memory. To understand why, we introduce the notion of *pipelining* among operators. A *non-blocking* operator (pipeline-friendly) is one that can decide on an answer without checking all the input rows. It therefore does not stop the flow of rows until it reads all the input. For instance, selection is a non-blocking operator. Sort is blocking.

To limit the memory a query plan uses, it suffices to adapt the execution of each operator type to the limitation. A non-blocking operator would chop the input columns to appropriately sized slices and would process one slice at a time. A blocking operator can use implementations based on external algorithms (e.g., sort on disk).

In summary, the execution model stores tuples of indexes instead of tuples of values. It dereferences those indexes on demand.

5.2 Implementing the Execution Model

The AQuery system implements the column-oriented execution model using an interpreted array-oriented language called K [19]. K is a modern descendant of APL [17], the reference language on array manipulation. In these languages, any data structure can be represented by arbitrarily shaped arrays, which can be manipulated by the composition of very simple but highly efficient array primitives. These primitives operate on the array level, i.e., they are mappings from an array (or a list thereof) to an array. Because AQuery was built on top of K, it was natural to represent AQuery's arrables and operators.

A K implementation of an arrable follows the latter's definition: a collection of named arrays. On disk, an arrable is mapped to its own sub-directory. Each component array is stored in a file named after the column it represents. In memory, an arrable is represented by a dictionary structure. Entries are named after the column they represent and each entry stores a column's array. Whenever a column needs to be accessed, the system loads the corresponding array-file from disk to virtual memory and adds an entry to the corresponding arrable's dictionary. When the array (column) is no longer needed the system eliminates the dictionary entry, freeing memory.

To be executed, a plan's operators are translated into a sequence of K instructions that follow a three-step template. In step one, the operator receives both the column handles it needs (e.g., the columns involved in a selection's predicate) and the current effective index array of the query. In step two, the operator accesses data (uses the index array to retrieve the still relevant columns' values) and

performs its calculation (evaluates the predicate over the values). In step three, it returns the updated index array (eliminates the rows that mapped to false).

In this context, the system translates a plan into a K function that puts together a sequence of such operators. The function takes handles to the arrables it manipulates and outputs an arrable. The Figure 5.3 shows the code generated for the following query.[1]

```
[AQuery]
SELECT last(price)
FROM   Trades
       ASSUMING ORDER ID,ts
WHERE ID= 'XYZW'
```

A query plan is an anonymous function, defined between the curly braces of lines 1 and 8. Immediately after its definition, the system calls the function passing a list of arrables (line 8). In this case, the function takes only a singleton list, a handle to the Trades arrable. The query plan starts by initializing its effective index array in line 2, with as many index vectors as there are input arrables. The code here is generic, in that it would have worked correctly if any other number of arrables were passed. The next step in that plan is to load data. Line 3 defines which column handles are going to be used; the variable 'cols' is thus initialized with handles for three columns. In line 4 the data for these columns are fetched from disk, resolving their address. A dictionary named 'r' is created to hold the Trades arrable's data in memory. Next, in line 5, the columns 'r.ID' and 'r.ts' are sorted in ascending order. That is what the K operators '<+' (flip and sort) are doing in that line. Note that all access to these variables is now indexed by the corresponding portion of the effective index array, eia[0]. As a result of the sort the index array is rearranged, as in 'eia:eia@....' Line 6 computes which positions of the contain "XYWZ" in their r.ID column. Again, the effective index array is rearranged accordingly. Finally, line 7 calls the in-line function last 'x[-1+#x]' and passes to it the price column at position that are still pertinent. Thus, only the prices of stock 'XYWZ' ordered by ID and timestamp are passed. The line 7 then creates the resulting arrable which contains a column called 'price' and the result of the last function.

5.3 From Text to Execution: the entire flow

Having discussed the execution engine's strategy, we are now ready to discuss the entire process from text to execution.

[1]Operations on K are denoted by symbols which makes the code extremely compact. Critics and even the language designer agree that K code looks like line noise to the uninitiated.

```
1.  {[t]                                                  / declare parameter
2.    eia:(#t)#_n                                        / initialize eia
3.    cols:`ID`price`ts                                  / list of needed columns
4.    r:@[_n;cols;:;{1:($t[0]),"/",$x}'cols]             / load data into dic r
5.    eia:eia@\:<+( r.ID[eia[0]];r.ts[eia[0]] )          / sort by ID,ts
6.    eia:eia@\:&r.ID[eia[0]]~``XYWZ''                   / selection over ID
7.    :.,(`price; {x[-1+#x]}[r.price[eia[0]]])           / compute last
8.  }[,`trades]
```

Figure 5.3: Example of a K plan

5.3.1 Parsing

The text first undergoes lexical and syntactical analysis, commonly referred to as "parsing." These phases make sure a query has valid syntax and generates an abstract syntax tree. The AQuery system parses a query using a LL(1) grammar and a recursive decent parser and generates tree that represents the query's text. This format is more amenable to the semantics step.

5.3.2 Semantics Step

Semantics checks in AQuery do more than simply verify that all the objects in a query – columns, arrables, functions – exist. AQuery semantics requires expressions to obey shape constraints, as discussed in Chapter 3. For instance, a search condition in the WHERE clause is a boolean expression whose cardinality is equal to that of the arrable resulting from the FROM clause. Therefore, each use of a non-size-preserving function in that clause must pass a semantics check.

To illustrate how shape verification is done, consider the expression 'col + first(10,col).' Addition is valid between two equally sized arrays, between two scalars, or between an array and a scalar. Therefore the above expression is invalid.

The AQuery system captures these shape constraints in tables such as Table 5.1 for the addition operation. Initial cardinality refers to the number of elements a column first had in the context being considered. For instance, in the FROM clause initial cardinality has as many elements as the table, in the WHERE as many element as the Cartesian product of the FROM clause generated, and so on. This notion is important because some clauses require initial cardinality (WHERE, GROUP BY, HAVING) while others don't (SELECT).

+	n	m	1	initial
n	n	error	n	error
m	error	m	m	error
1	n	m	1	initial
initial	error	error	initial	initial

Table 5.1: Cardinality of the addition operation

5.3.3 Relational Optimization Support

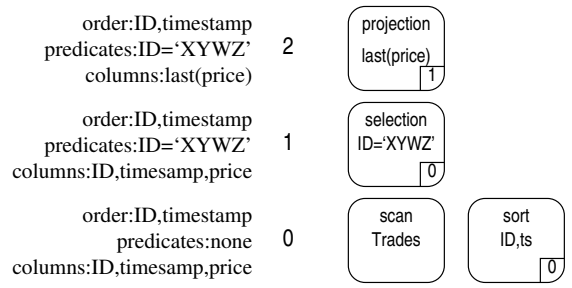
Once semantics correctness is verified, a query is translated into its algebraic format. The result is a tree in which nodes are operators from the arrable algebra described in chapter 3. This tree is a possible plan for the query, but only rarely a good one. Algebra equivalences (query transformations) may generate an equivalent plan with improved performance. These transformations are described in the relational literature [10] and new ones regarding order equivalence were introduced in chapter 4. Applying a transformation to a plan entails generating a new plan, which may share several nodes with the original one. Given the number of equivalent plans that even a small set of transformations can generate, support for plan manipulation is needed.

The AQuery system uses a data structure called MEMO [12] to store a forest of query plans in an efficient way. The main idea behind the MEMO is to group tree nodes into equivalence classes. Two nodes will belong to the same class if the result of processing the queries to which they belong up and including that node is semantically equivalent. A group is characterized by properties of the data their nodes generate. Examples of properties are: tables and columns accessed thus far, predicates applied, etc. In the AQuery system data ordering is a discriminating property for groups.

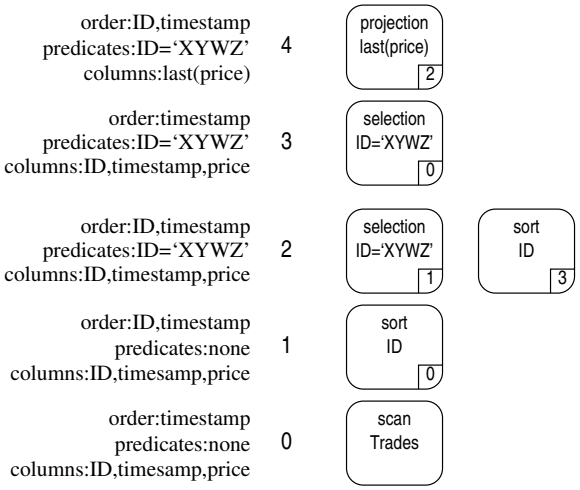
Figure 5.4 shows two representation of the MEMO structure for the query that fetches the last quote of the 'XYWZ' stock. Operations are represented as squares. An operation that takes the result of another one as input has the group of the latter indicated in the lower right corner. Groups of operations are numbered and a subset of the properties of each group is shown on the left-hand side of their MEMO. Recall that the query in question performs a scan on Trades to retrieve the ID, price, and timestamp columns; a sort over ID and timestamp; a filtering (selection) over ID; and, finally, a materialization (projection) of the application of last() over the resulting price vector. The configuration of the MEMO will depend on the arrable's order.

If the Trades arrable were ORDERED BY ID and timestamp, then the MEMO in figure 5.4(a) would result. The scan of Trades appears as the first operation in group 0. (We refer to it as operation 0.0.) The query also performs a sort, but because the arrable is already in a convenient order, the sort is redundant. Therefore, the sort over the result of the scan would not change the data's properties

and thus sort is placed in the same group as the scan. (It becomes operation 0.1.) Note that the nodes that point to group 0 may now choose between two alternate sub-plans. The subsequent operations change properties of the data and as such they are inserted into groups of their own. This MEMO represents two possible plans: <2.0 ← 1.0 ← 0.0> or <2.0 ← 1.0 ← 0.1 ← 0.0>.



(a)



(b)

Figure 5.4: Two possible MEMO configurations for a given query

If Trades were ORDERED BY timestamp alone, The MEMO depicted in Figure 5.4(b) would result. Sorting on ID and timestamp now generates a distinct result than that of the scanning of Trades. Therefore, the scan and that sort belong to different groups (0 and 1, respectively).

This query may benefit from the transformation that commutes a sort with a selection. That is, instead of sorting the outcome of group 0 and then filtering, it does the reverse. A new filter node that takes group 0 as input is inserted in the MEMO in a group 3, for no other group in the MEMO shares its properties. The node that sorts the output of the new filter is inserted in group 2 for it shares its

properties. That MEMO now encodes two possible plans: <4.0 ← 2.0 ← 1.0 ← 0.0> and <4.0 ← 2.1 ← 3.0 ← 0.0>.

At the time of this writing the system was still not applying query transformation automatically. An accurate cost-model is still missing. The system either generates a syntax-directed plan automatically or it generates plans in an interactive fashion where the system finds possible transformations to be applied and prompts a user to pick one until the user is satisfied with the plan.

5.3.4 'K'-Code Generation

After a plan is chosen, it is translated into K language instructions. The translation is made one node at time. Each type of node requires a different translation strategy. For instance, the code generated for a selection should evaluate its expression and filter the current index array accordingly. By contrast, the code generated for a sort should trigger a reordering of the index array. Moreover, nodes are sensitive to the context. The code generated for a projection changes depending on whether there is grouping or not. The latter's expressions have to be "eached" whereas the former's need not. The code generation module handles all these nuances.

The code generation module performs optimizations of its own. The most salient one is a limited but quite effective form of common sub-expression elimination in which column references appear more than once in a same expression. For instance, take the expression 'price - prev(price).' Accessing the still active positions of price twice is unnecessary. Upon detection of repeated column-references, it creates temporaries to store that result.

5.4 Conclusion

Performance is a major goal of the design of the AQuery system. Each technique in our framework plays a role in performance:

• Processing a query by maintaining its effective index array prevents unnecessary copies of data to be passed among operators. A selection, for instance. doesn't need to copy the actual selected tuples, but only to forward their positions to the next operation. A sort does not need to produce a new ordered copy of the data but only a permutation of positions. Ultimately, processing indexes as opposed to copies of data allows the system to defer materialization of results up until the end of the query.

• Column-wise representation of data avoids unnecessary disk accesses. It ensures that in every transfer only data that is actually going to be used – as opposed to entire rows with column data unused by a query – is fetched. This also benefits cache utilization, because irrelevant fields don't clutter cache lines.

• Evaluating expressions in a vector-oriented fashion promotes high reference locality. Vectors are contiguous in memory and most operations involve a sequential scan of a vector, often a contiguous one. Vector-orientation also allows calls to processing functions to be made only once for the entire vector as opposed to once for each element of the vector.

The AQuery system is a ongoing effort. We are currently investigating a query enumerating algorithm [14] that along with a suitable cost model would be able to apply the transformations automatically. We are also considering more sophisticated disk sub-system such as the one described in [38].

Chapter 6

Performance Analysis

6.1 Introduction

This chapter addresses the efficiency of AQuery in performing order-dependent queries. It does so by comparing the performance of the AQuery system against that of a commercial SQL:1999 one.[1] SQL:1999 [16] was chosen because its order features are implemented in at least two major DBMS products, and its syntax and semantics are thoroughly documented.

We have resorted to somewhat advanced SQL:1999 features to write the order-dependent queries we test here. To make the chapter more accessible to unfamiliar readers, we present a quick introduction to the "window" feature in the remainder of this section. The more advanced reader may want skip to Section 6.2.

One way to introduce a SQL:1999's window is as a function that maps each table's rows to a subset of this table. We will use the table T1 in figure 6.1 to show two running examples.

T1	c1	c2	c3
	a	2	10
	a	1	20
	b	2	30
	a	3	40
	b	1	50

Figure 6.1: A table to be used on a window definition

In our first example, let's assume one wanted to identify for each given row all the rows that preceded it. Let us assume an ordering based on column c2. To make the example more interesting, we impose a partitioning on column c1, that is, column c2 is ordered within partitions defined by column c1. To specify such a window in SQL:1999 one would write WINDOW w PARTITION BY c1 ORDER

[1] Due to license agreement restrictions, we omit the name of the product.

BY c2 ROWS UNBOUNDED PRECEDING. The last part of the window declaration says that the width of a window spans from the first row (of the partition) up until the row being considered. Figure 6.2 shows the calculation of such a window over table T1. In the figure, the window instance that corresponds to row i, according to the ordering given, is denoted w_i.

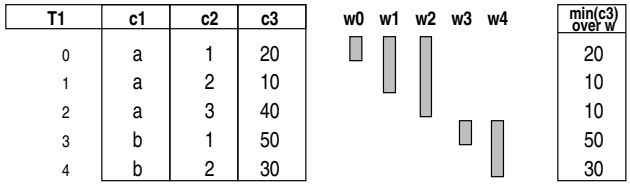


Figure 6.2: A running minimum window

Our example window is useful, for instance, to compute a running minimum. It suffices to apply the aggregate function min() to each of the window instances, as the Figure 6.2 shows. In a query, this calculation could be used as follows.[2]

[**SQL:1999**]
```
SELECT *, min(c3) OVER ( PARTITION BY c1
                         ORDER BY c2
                         ROWS UNBOUDED PRECEDING)
FROM   T1
```

A window may be used in the SELECT clause provided that the query applies an aggregate function over its instances.

In our second example, suppose one wanted to fetch just the previous row for each of T1's row rather than all the preceding rows. Let us use the same partitioning and ordering criteria as before. The difference is thus on the window width. To use a single-row wide window that contains a previous row, one would say in SQL:1999 WINDOW z PARTITION BY c1 ORDER BY c2 ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING. The Figure 6.3 shows the calculation of such a window over the table T1.

This window is useful, for instance, to obtain a column's previous value, the equivalent of prev() in AQuery. Note that even though the window instances contain only one row, a query must still specify an aggregating function to use the window. In this case min(), max(), and avg() would all have the same effect. Suppose one wants the previous element of each c3's value, the implementing SQL:1999 query is shown below.

[2]We are using the inline definition of the window here. In the full definition case, the WINDOW clause would be added at the end of the query and the expression would refer to the window name as in 'min(c3) OVER w.' As of this writing only the inline use of windows was supported, though.

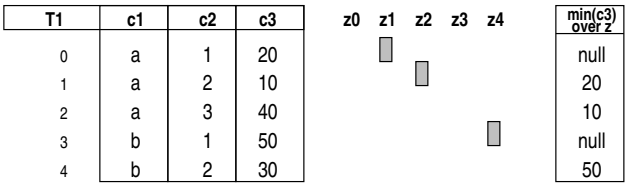


Figure 6.3: A previous row window

[**SQL:1999**]

```
SELECT *, min(c3) OVER ( PARTITION BY c1
                         ORDER BY c2
                         ROWS BETWEEN 1 PRECEDING
                         AND 1 PRECEDING)
FROM   T1
```

6.2 The Best Profit Query

The Best Profit query finds, for a given stock and a given date, the best profit one could obtain by buying it and then selling it later that same day. This query has been discussed previously in section 2.2 but for convenience, we again present the relevant aspects.

The query uses the arrable/table Ticks(ID, date, price, timestamp), where ID is the ticker of a traded security, timestamp identifies the date and time of a particular trade, date is a human-readable form of the day portion, and price is the price of the security. The table version of Ticks had indexes on timestamp (clustering), on ID (non-clustering), and on date (non-clustering). The arrable version was ORDERED BY timestamp.

The AQuery and SQL:1999 renditions of this query are the following.

[**AQuery**]

```
SELECT max(price - mins(price))
FROM   Ticks
       ASSUMING ORDER timestamp
WHERE ID = 'ACME' AND date = '05/11/2003'
```

[**SQL:1999**]

```
SELECT  max(running_diff)
FROM    (SELECT ID, date,
                price - min(price) OVER ( PARTITION BY ID,date
                                          ORDER BY timestamp
                                          ROWS UNBOUNDED PRECEDING)
                                          AS running_diff,
        FROM    Ticks ) AS t1
WHERE  ID = 'ACME' AND date = '05/11/2003'
```

The structure of the two renditions differs because in SQL:1999 an aggregate function (max()) cannot take a running aggregate (min(price) OVER $\cdots$) as an argument. Nevertheless, both renditions sort first (ASSUMING/ORDER BY timestamp) and then filter (ID='ACME' AND date='05/11/2003'). Note that an optimal plan would do the inverse – sorting is more expensive than filtering.

The plans generated for the two queries are shown in Figurefig:6plan1. The AQuery plan exploits the possibility of performing an early selection. Sort and selection are adjacent and can be commuted with a simple transformation (Table 4.1 transformation 3). Since there were indexes available to evaluate each of the selection's conjuncts, AQuery could use them both. Sort was done only on the reduced set of rows.

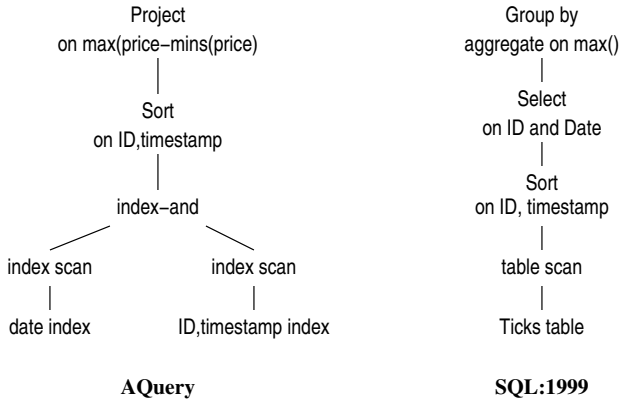


Figure 6.4: Plans for the best-profit query

By contrast, in the SQL:1999 query the sort and the selection are separated by a window operation. The SQL:1999 optimizer doesn't push the selection down. It could in this case since the selection doesn't impact the way groups are formed in the OVER clause; it just eliminates entire groups.[3] It remains though that pushing a selection over a projection that contains windowed functions (SELECT $\cdots$ price - min(price) OVER $\cdots$) requires considerable analysis. The SQL:1999 system we have analyzed chose none of the available indexes.

The difference in plans is noticed in the response times. The chart in Figure 6.5 shows the relative improvement of the AQuery's plan over the SQL:1999 optimizer's. We used Ticks arrables/tables with varying number of securities from 200 to 1000, and using 1000/ticks per security. The chart shows that AQuery results were between eight and twenty one times faster for the best-profit query.

This and the remaining experiments were conducted on a Pentium III-M 1.13Mhz with 1Gb of memory running Linux. The timings reported correspond

[3]If vendors use this thesis to improve their SQL:1999 systems, we will consider that a mark of success.

to wall clock timings. We were careful to allocate the same amount of memory for both optimizers and made sure execution used cold buffers (empty).

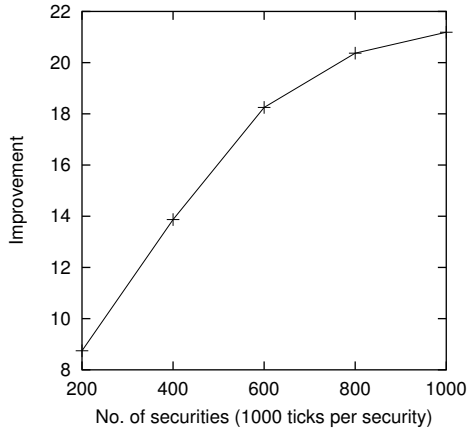


Figure 6.5: Best profit query relative improvement

The difference grows with the size of the input because the more securities used, the more the SQL:1999 plan sorts rows that will eventually be discarded.

6.3 Network Management Query

The Network Management query's goal is to break sequences of packets (sessions) between pairs of hosts down into "flows" and to compute statistics of the latter. A flow between a pair of hosts ends (and a new one starts) whenever they stop communicating for a period of 120 seconds or more. The schema involved in the query is 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. The table version of Packets had indexes over timestamp (clustered) and a composed one over source, destination, and timestamp (non-clustered). The arrable version was ORDERED BY timestamp. The AQuery rendition of this query was discussed in Example 3.3, but we repeat it here, along with its SQL:1999 counterpart.

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

[**SQL:1999**]
```
WITH
  Prec (src, dest, length, timestamp, ptime) AS
```

```
  (SELECT  src, dest, length, timestamp,
           min(timestamp) OVER
                    (PARTITION BY src,dest
                    ORDER BY timestamp
                    ROWS BETWEEN 1 PRECEDING
                    AND 1 PRECEDING)
  FROM     Packets),
  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
```

Expressing this calculation in SQL:1999 was not as straightforward as in AQuery, although the execution flow of both queries are quite similar – at least semantically. In SQL:1999, the first sub-query, Prec, creates a new column, ptime, containing the previous packet's timestamp within each source and destination partition. It uses a window that partitions by source and destination, sorts by timestamp, and uses a one-row width window instance. In AQuery, there is no partitioning but sort is done over source, destination, and timestamp. The Prec sub-query has the same effect of a prev(timestamp) in AQuery. (Recall that deltas(col) is equivalent to col - prev(col).)

Next, the SQL:1999 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). In the AQuery rendition, this is what the expression 'deltas(timestamp) $>$ 120' does.

The SQL:1999 query continues by calculating FlowID, which sums the flags cumulatively, creating an auxiliary flow ID, fID. Note that a new window is required here that does not fully agree with the preceding window definition. In AQuery, the sums() function does a similar task without resorting to any additional sort. The main query in SQL:1999 uses these results in exactly the same way as the SELECT clause of the AQuery rendition.

Once more the query structure impacted the quality of the plans found, which are shown in Figure 6.6. The main difference between the two plans is an extra

sort on the SQL:1999 one. Its optimizer added a sort by the same columns it is grouping by. By contrast, AQuery's group by is dependent on – and thus benefits from – the order enforced by the ASSUMING clause. The SQL:1999 optimizer did consider both windows to have the same ordering requirements, though, by sorting only once by source, destination, and timestamp. It is unclear from the SQL:1999 plan documentation how the ptime attribute is calculated.

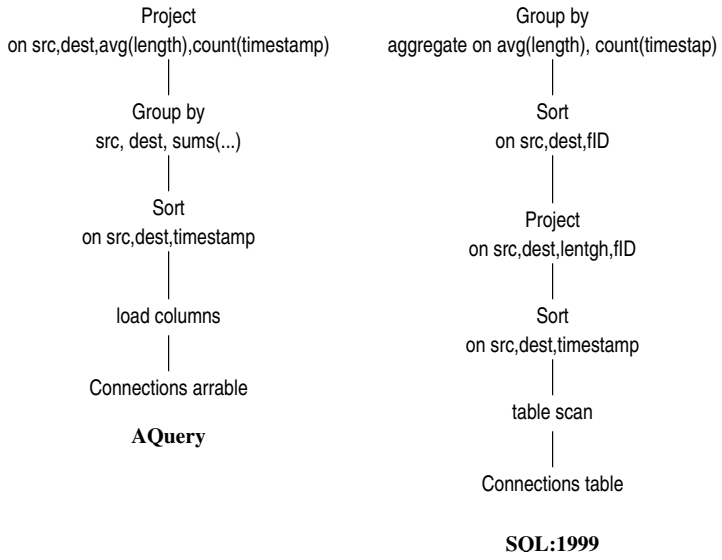


Figure 6.6: Plans for the network management query

AQuery took advantage of the existing order and used a stable algorithm to eliminate that column from the sort.

Here again the difference in the plans brought discrepant response times. The chart in Figure 6.7 shows the relative improvement of the AQuery plan over SQL:1999's. We used a Packets arrable/table with 100 sessions and varied the number of packets for each session from 2K to 10K. AQuery results were from a little less than 2.4 to 3 times faster.

A clear component of the worse response time is the extra sort but that alone cannot account for the entire time difference. We believe that the vertical-partitioning, array-processing approach of AQuery is playing an important role here. For one, processing a column at a time as opposed to a row at a time demands far less overhead. While in the latter case, each operator is called once every time an operator needs a row (iterator model), in AQuery there is only one call per operator (full columns are returned). For another, vertical partitioning and array processing combined highly favored locality of reference.

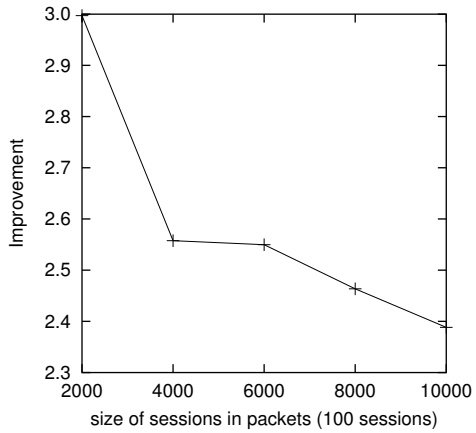


Figure 6.7: Network management query relative improvement

6.4 Conclusion

These experiments suggest basic flaws in the straightforward implementation of SQL:1999 for order queries. In the best-profit query, ordering was part of a more complex operation, a window, and therefore the optimizer missed the opportunity of pushing down a highly selective filtering operation through it. The moral here is that *if order implies complex structures (windows) then even trivial transformations (selection push-down) may require serious analysis.*

In the network management query, the ordering by source, destination, and flow ID is equivalent to that by source, destination, and timestamp. Flow ID grows monotonically with the latter. The optimizer missed that and inserted an extra sort step in the query. The moral here is that *the more complex the syntax, the harder it is to find order idioms.*

The experiments showed that AQuery's structural simplicity helped find much better plans. This translated into performance improvements often greater than an order of magnitude.

Chapter 7

Conclusion

7.1 Summary

A query in AQuery may determine the order through which it wishes to manipulate data. The query's clauses can exploit this order by the use of array-typed expressions and vector-to-vector functions. The use of order does not force a query to use a more complex structure than it normally would. As a consequence, queries that in other languages are hard to write become as natural in AQuery as SQL is for unordered queries.

The data ordering in a query is independent of the order in which data is stored. If these orderings match, then the query may execute more efficiently, but there is no semantic effect of the order. If these orderings don't match, then the sort work involved can often be diminished. AQuery's conciseness fosters recognition of common order idioms for which we have provided some effective optimization techniques. The techniques complement the vast body of knowledge in query transformations which mostly apply to AQuery as well.

The AQuery system is the validation of these ideas. It executes AQuery queries over arrables by breaking them down into a sequence of array primitives. We have used the system to successfully write several queries that occur in the finance and network management domains.

7.2 Ongoing Work

The motivational study we conducted in several application areas showed that ad-hoc order-dependent querying is virtually unavailable commercially. (SQL:1999 systems are known not to be widely used in finances, Biology, or network management.) We are responding to that need with a major effort for making AQuery public. We are building a database of regression tests that cover the entire language. Our ultimate goal is to have a solid system by the time of its first release.

As a parallel effort, we continue to investigate other query transformation for order dependent queries. Nevertheless, we are still experimenting on how to make the AQuery system apply these transformations automatically.

7.3 Future Work

Our current implementation carries several performance-improving operations such as edgeby or sort-edge. However, we have not yet touched other possibilities for gaining performance through parallelism or modern-architecture hardware exploitation (super-scalar CPUs and hierarchical memory). Our extensive use of arrays makes the latter seem particularly promising. Current super-scalar CPUs can use Single Instruction Multiple Data (SIMD) parallelism; we therefore can convert several of our array primitives to use this facility.

Finance is a domain where often order-dependent querying involves streaming data. AQuery showed a natural facility to express streams-based queries. We have however identified important queries in which the analysis of very recent data may be triggered by events on the head of the stream. This quasi-streaming approach in which one would need to backtrack to recent elements of a stream is an avenue that interests us.

Biological sequence databases for DNA or proteins make extensive use of order-dependent operations. The order idioms and the optimization techniques found here would still be valid, but the query language interface may need to be rethought. This is another avenue that we hope to pursue.

Bibliography

[1] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving Relations for Cache Performance. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 169–180, 2001.

[2] Peter Bauman, Andreas Dehmel, Paula Furtado, Roland Ritsch, and Norbert Widmann. The Multidimensional Database System RasDaMan. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 575–577, 1998.

[3] Peter A. Boncz. *Monet: A Next-Generation DBMS Kernel for Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, The Netherlands, 2002.

[4] Anthony J. Bonner and Giansalvatore Mecca. Sequences, Datalog, and Transducers. *Journal of Computer and System Sciences (JCSS)*, 57(3):234–259, 1998.

[5] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension Syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[6] Peter Buneman, Shamim Navqi, Val Tannen, and Limsoon Wong. Principles of Programming with Complex Objects and Collection Types. *Theoretical Compute Science*, 149(1):3–48, 1995.

[7] Michael J. Carey and Donald Kossmann. On Saying 'Enough Already!' in SQL. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 219–230, 1997.

[8] Chuck Cranor, Yuan Gao, Theodore Johnson, Vlaidslav Shkapenyuk, and Oliver Spatscheck. Gigascope: High Performance Network Monitoring with an SQL Interface. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 623–623, 2002.

[9] Marco de Vivo, Eddy Carrasco, Germinal Isern, and Gabriela O. de Vivo. A Review of Port Scanning Techniques. *ACM Computer Communications Review*, 29(2):41–48, 1999.

[10] Hector Garcia-Molina, Jeffrey Ullman, and Jennifer Widom. *Database System Implementation*. Prentice-Hall, 1999.

[11] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[12] Goetz Graefe and William McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 209–218, 1993.

[13] Torsten Grust. *Comprehending Queries*. PhD thesis, University of Konstanz, 1999.

[14] Yannis E. Ioannidis and Eugene Wong. Query Optimization by Simulated Annealing. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 9–22, 1987.

[15] ISO/IEC 9075. *Information Technology - Database Languages - SQL*, 1999.

[16] ISO/IEC 9075. *Amendment 1:2001 - Information Technology - Database Languages - SQL (SQL/OLAP)*, 2001.

[17] Kenneth E. Iverson. *A Programming Language*. Wiley, 1962.

[18] Kaippallimalil J. Jacob and Dennis Shasha. FinTime - A Financial Time Series Benchmark. *SIGMOD Record*, 28(4):42–48, 1999.

[19] KX Systems. *K Reference Manual*.

[20] KX Systems. *KSQL Reference Manual*.

[21] Leonid Libkin, Rona Machlin, and Limsoon Wong. A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 228–239, 1996.

[22] Guy M. Lohman. Grammar-like Functional Rules for Representing Query Optimization Alternatives. In *Proceeding of the 1988 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 18–27, 1988.

[23] David Maier and Bennet Vance. A Call to Order. In *Proceedings of the Twelfth ACM Symposium on Principles of Database Systems (PODS)*, pages 1–16, 1993.

[24] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing Database Architecture for the New Bottleneck: Memory Access. *The VLDB Journal*, 9(3):231–246, 2000.

[25] Arunprasad P. Marathe and Kenneth Salem. Query Processing Techniques for Arrays. *The VLDB Journal*, 11(1):68–91, 2001.

[26] Jim Melton. *Advanced SQL:1999 – Understanding Object-Relational and Other Advanced Features*. Morgan Kaufmann Publishers, 2002.

[27] Wilfred Ng. An Extension of the Relational Data Model to Incorporate Ordered Domains. *ACM Transactions on Database Systems (TODS)*, 26(3):344–383, 2001.

[28] Oracle. *Analytic SQL Features in Oracle 9i*, december 2001.

[29] Raghu Ramakrishnan, Donko Donjerkovic, Arvind Ranganathan, Kevin S. Beyer, and Muralidhar Krishnaprasad. SRQL: Sorted Relational Query Language. In *Proceedings 10th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 84–95, 1998.

[30] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. SEQ: A Model for Sequence Databases. In *Proceedings of the 11th Intenational Conference on Data Engineering (ICDE)*, pages 232–239, 1995.

[31] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The Design and Implementation of a Sequence Database System. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 99–110, 1996.

[32] Dennis Shasha. Time Series in Finances. Summer School in Extending Database Technologies in La Baule, France, 1999.

[33] David E. Simmen, Eugene J. Shekita, and Timothy Malkemus. Fundamental Techniques for Order Optimization. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 57–67, 1996.

[34] Giedrius Slivinskas. *A Middleware Approach to Temporal Query Processing*. PhD thesis, Aalborg University, Denmark, 2001.

[35] Giedrius Slivinskas, Christian S. Jensen, and Richard T. Snodgrass. Bringing Order to Query Optimization. *SIGMOD Record*, 31(2):5–14, 2002.

[36] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 204–215, 2002.

[37] Arthur Whitney and Dennis Shasha. Lots o' Ticks: Real-Time High Performance Time Series Queries on Billions of Trades and Quotes. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2001.

[38] Jingren Zhou and Kenneth A. Ross. A Multi-Resolution Block Storage Model for Database Design. In *To appear in the Proceedings of the 2003 International Database Engineering and Application Symposium (IDEAS)*, 2003.

[39] Yunyue Zhu and Dennis Shasha. StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 358–369, 2002.