

AQuery

A Database System for Order

Alberto Lerner (ENST-Paris / NYU)
joint work with Dennis Shasha (NYU)
lerner@cs.nyu.edu

July 2003

Motivation

The need for ordered data

- ◆ Queries in Finance, Biology, and Network Management depend on order.
- ◆ SQL 99 has extensions – the OLAP amendment – that incorporate order to the language but they are clumsy to use.

3-month moving average: the wrong way

month	sales	3-avg
1	100	100
2	120	110
3	140	120
4	140	133
5	130	136

```
SELECT t1.month,t1.sales,
       (t1.sales+t2.sales+t3.sales)/3
FROM   Sales t1, Sales t2, Sales t3
WHERE  t1.month - 1 = t2.month AND
       t1.month - 2 = t3.month
```

Problems?

- Join eliminates first two months!
- Do we really need a three-way join?
- Can the optimizer make it linear-time?

3-month moving average: the hard way

month	sales	3-avg
1	100	100
2	120	110
3	140	120
4	140	133
5	130	136

```
SELECT t1.month,t1.sales,
       (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 + t3.sales
END) / 3
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
```

Problems?

- “Write-once” query
- Three way join

3-month moving average: the OLAP way

month	sales	3-avg
1	100	100
2	120	110
3	140	120
4	140	133
5	130	136

```
SELECT  month,sales,
        avg(sales) OVER (ORDER BY month
                        ROWS BETWEEN
                        2 PRECEDING AND
                        CURRENT ROW)
FROM    Sales
```

Problems?

- OVER construct is confined to the SELECT clause
- Awkward syntax

Network Management Query

- ♦ Find duration and average length of packets of src-dst flows. A flow from src to dest ends after a 2-minute silence

Packets	src	dst	len	time
	s1	s2	250	1
	s1	s2	270	20
	s2	s1	330	47
	s1	s2	235	141
	s2	s1	280	150
	s2	s1	305	155

```
WITH
Prec AS (src,dst,len,time,ptime)
(SELECT  src,dst,len,time,min(time) OVER w
FROM    Packets
WINDOW w AS
        (PARTITION BY src,dst
         ORDER BY time
         ROWS BETWEEN 1 PRECEDING
         AND 1 PRECEDING)),
Flow AS (src,dst,len,time,flag)
(SELECT  src,dst,len,time,
        CASE
            WHEN time-ptime > 120 THEN 1
            ELSE 0
        FROM    Prec),
FlowID AS (src,dst,len,time,fID)
(SELECT  src,dst,len,time,sum(flag) OVER w
FROM    Flow
WINDOW w AS
        (ORDER BY src,dst, time
         ROWS UNBOUNDED PRECEDING))
SELECT  src,dst,count(*),avg(len)
FROM    FlowID
GROUP  BY src,dst,fID
```

Order in SQL:1999

- ◆ Inter-tuple operations require joins or additional query constructs - or both!
- ◆ Ordering can only be obtained in specific clauses (e.g., SELECT)

Bottom line:

- ◆ Queries become difficult to read
- ◆ Cost of execution is often non-linear (optimization of nested queries is still an open problem)

Idea

- ◆ Replace ordered tables (*arrables*) for tables in the data model
 - Whatever can be done on a table can be done on an *arrable*. Not vice-versa.
- ◆ Define order on a per-query basis
 - All query clauses can count on data ordering
- ◆ Maintain SQL flavor (upward compatibility to SQL 92) while allowing expressions based on order with no additional language constructs
- ◆ Exploit optimization techniques involving order

That's AQuery!

Moving average over Arrables

month	sales	3-avg
1	100	100
2	120	110
3	140	120
4	140	133
5	130	136

```
SELECT month,avgs(3,sales)
FROM Sales
ASSUMING ORDER month
```

- Arrable: a collection of named arrays, ordered by a column list

Moving average over Arrables

month	sales	3-avg
1	100	100
2	120	110
3	140	120
4	140	133
5	130	136

```
SELECT month,avgs(3,sales)
FROM Sales
ASSUMING ORDER month
```

- Arrable: a collection of named arrays, ordered by a column list
- Each query defines data ordering

Moving average over Arrables

month	sales	3-avg
1	100	100
2	120	110
3	140	120
4	140	133
5	130	136

```
SELECT month, avgs(3, sales)
FROM Sales
ASSUMING ORDER month
```

- Arrable: a collection of named arrays, ordered by a column list
- Each query defines data ordering
- Variables (e.g., month) are bound to an array, as opposed to a value

Moving average over Arrables

month	sales	3-avg
1	100	100
2	120	110
3	140	120
4	140	133
5	130	136

```
SELECT month, avgs(3, sales)
FROM Sales
ASSUMING ORDER month
```

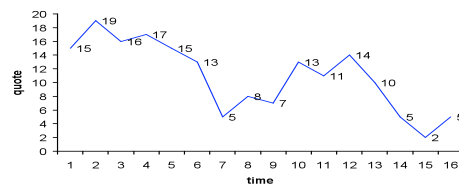
- Arrable: a collection of named arrays, ordered by a column list
- Each query defines data ordering
- Variables (e.g., month) are bound to an array, as opposed to a value
- Expression are mappings from arrays to array

Built-in Functions

	size-preserving	non size-preserving
order-dependent	prev, next avgs, prds, sums, mins, deltas, ratios, reverse, ...	drop, first, last
non order-dependent	rank, n-tile	min, max, avg, count

Emotive Query

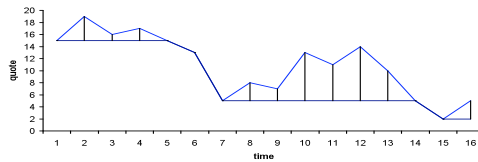
Find the best profit one could make by buying a stock and selling it later in the same day



price 15 19 16 17 15 13 5 8 7 13 11 14 10 5 2 5

Emotive Query

Find the best profit one could make by buying a stock and selling it later in the same day



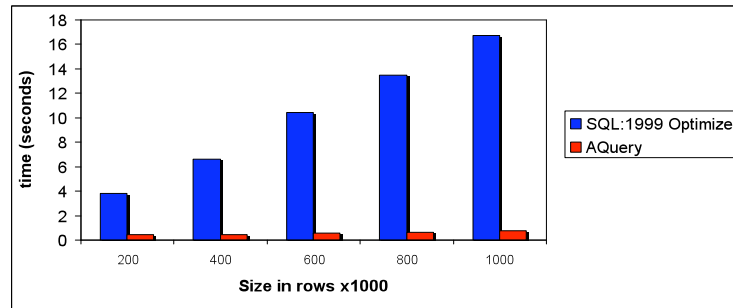
price	15	19	16	17	15	13	5	8	7	13	11	14	10	5	2	5
mins(price)	15	15	15	15	15	13	5	5	5	5	5	5	5	5	2	2
	0	4	1	2	0	0	0	3	2	8	6	9	0	0	0	3

Best-profit Query Comparison

[AQuery]	[SQL:1999]
SELECT max(price-mins(price))	SELECT max(rdif)
FROM ticks	FROM (SELECT ID,tradeDate,
ASSUMING timestamp	price - min(price)
WHERE ID="S"	OVER
AND tradeDate='1/10/03'	(PARTITION BY ID,
	tradeDate
	ORDER BY timestamp
	ROWS UNBOUNDED
	PRECEDING) AS rdif
	FROM Ticks) AS t1
	WHERE ID="S"
	AND tradeDate='1/10/03'

Optimizer doesn't push this selection. To get good performance, the query author has to rewrite it.

Best-profit Query Performance



Complex queries: Network Management Query Revisited

- ◆ Create a log of flow information. A flow from src to dest ends after a 2-minutes silence

Packets	pID	src	dest	len	time
...

```
SELECT  src, dst, count(*), avg(len)
FROM    Packets
        ASSUMING ORDER src, dst, time
GROUP   BY src, dst, sums (deltas(time) > 120)
```

Network Management Query in Pictures

Packets	src	dst	len	time
	s1	s2	250	1
	s1	s2	270	20
	s2	s1	330	47
	s1	s2	235	141
	s2	s1	280	150
	s2	s1	305	155

```
SELECT  src, dst, count(*), avg(len)
FROM    Packets
        ASSUMING ORDER src, dst, time
GROUP   BY src, dst, sums (deltas(time) > 120)
```

Network Management Query in Pictures

Packets	src	dst	len	time
	s1	s2	250	1
	s1	s2	270	20
	s2	s1	330	47
	s1	s2	235	141
	s2	s1	280	150
	s2	s1	305	155

Packets	src	dst	len	time
	s1	s2	250	1
	s1	s2	270	20
	s1	s2	235	141
	s2	s1	330	47
	s2	s1	280	150
	s2	s1	305	155

```
SELECT  src, dst, count(*), avg(len)
FROM    Packets
        ASSUMING ORDER src, dst, time
GROUP   BY src, dst, sums (deltas(time) > 120)
```

Network Management Query in Pictures

Packets	src	dst	len	time	c1
	s1	s2	250	1	F
	s1	s2	270	20	F
	s1	s2	235	141	T
	s2	s1	330	47	F
	s2	s1	280	150	F
	s2	s1	305	155	F

```
SELECT  src, dst, count(*), avg(len)
FROM    Packets
ASSUMING ORDER src, dst, time
GROUP   BY src, dst, sums (deltas(time) > 120)
```

Network Management Query in Pictures

Packets	src	dst	len	time	c1	c2
	s1	s2	250	1	F	0
	s1	s2	270	20	F	0
	s1	s2	235	141	T	1
	s2	s1	330	47	F	1
	s2	s1	280	150	F	1
	s2	s1	305	155	F	1

```
SELECT  src, dst, count(*), avg(len)
FROM    Packets
ASSUMING ORDER src, dst, time
GROUP   BY src, dst, sums (deltas(time) > 120)
```

Network Management Query in Pictures

Packets	src	dst	len	time	c1	c2
{	s1	s2	250	1	F	0
	s1	s2	270	20	F	0
{	s1	s2	235	141	T	1
	s2	s1	330	47	F	1
{	s2	s1	280	150	F	1
	s2	s1	305	155	F	1

```
SELECT  src, dst, count(*), avg(len)
FROM    Packets
ASSUMING ORDER src, dst, time
GROUP   BY src, dst, sums (deltas(time) > 120)
```

Network Management Query in Pictures

Packets	src	dst	len	time
{	s1	s2	250	1
	s1	s2	270	20
{	s1	s2	235	141
	s2	s1	330	47
{	s2	s1	280	150
	s2	s1	305	155

src	dst	len	time
s1	s2	250,270	1,20
s1	s2	235	141
s2	s1	330,280,305	47,150,155

```
SELECT  src, dst, count(*), avg(len)
FROM    Packets
ASSUMING ORDER src, dst, time
GROUP   BY src, dst, sums (deltas(time) > 120)
```

Network Management Query in Pictures

Packets	src	dst	len	time
{	s1	s2	250	1
	s1	s2	270	20
{	s1	s2	235	141
	s2	s1	330	47
{	s2	s1	280	150
	s2	s1	305	155

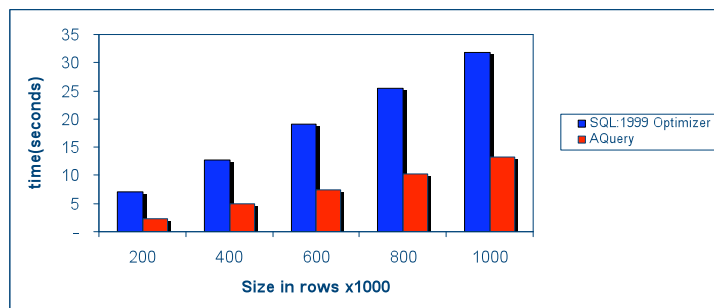
src	dst	len	time
s1	s2	250,270	1,20
s1	s2	235	141
s2	s1	330,280,305	47,150,155

src	dst	avg(len)	count(*)
s1	s2	260	2
s1	s2	235	1
s2	s1	305	3

```

SELECT  src, dst, count(*), avg(len)
FROM    Packets
ASSUMING ORDER src, dst, time
GROUP   BY src, dst, sums (deltas(time) > 120)
    
```

Network Management Query Performance



Order-aware Query Languages

- ◆ Relations, Sequences, and ordered-relations
 - SQL:1999
 - Sequin (Seshadri et al., 96)
 - SRQL (Ramakrishnan et al., 98)
 - Grouping in SQL (Chatziantoniou and Ross, 96)
- ◆ Array query languages
 - AQL (Libkin et al., 96)
 - AML (Marathe and Salem, 97)
 - RaSQL (Widmann and Baumann, 98)
 - KSQL (kx Systems)

Order-related Optimization Techniques

- ◆ Starburst's "glue" (Lohman 88) and Exodus/Volcano "Enforcers" (Graefe and McKeena, 93)
- ◆ DB2 Order optimization (Simmens et al., 96)
- ◆ Top-k query optimization (Carey and Kossman, 97; Bruno, Chaudhuri, and Gravano 02)
- ◆ Hash-based order-preserving join (Claussen et al., 01)
- ◆ Temporal query optimization addressing order and duplicates (Slivinskas et al., 01)

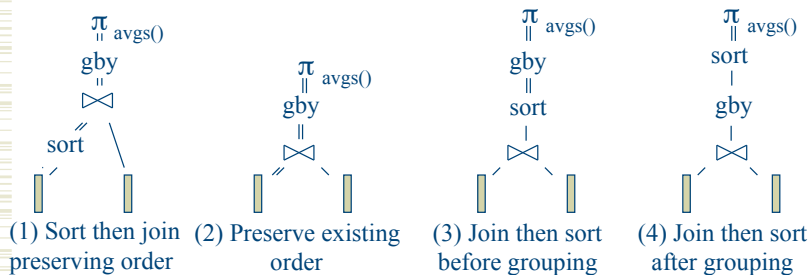
AQuery Optimization

- ◆ Optimization is cost based
- ◆ The main strategies are:
 - Define the extent of the order-preserving region of the plan, considering (correctness, obviously, and) the performance of variation of operators
 - Exploit algebraic equivalences
 - Apply efficient implementations of patterns of operators (e.g. “edge-by”)

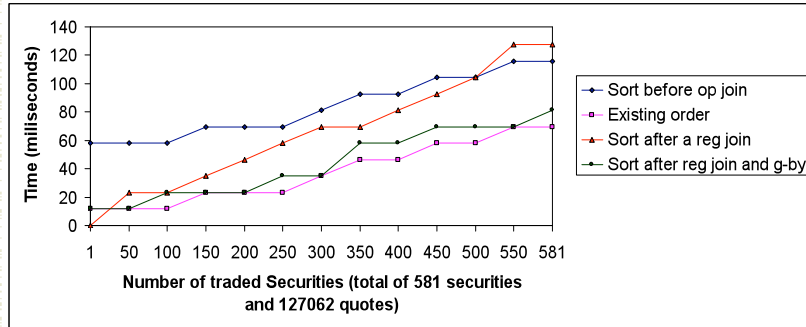
Interchange sorting + order preserving operators

```

SELECT ts.ID, avgs(10, hq.ClosePrice)
FROM   TradedStocks AS ts NATURAL JOIN
       HistoricQuotes AS hq
ASSUMING ORDER hq.TradeDate
GROUP BY Id
    
```



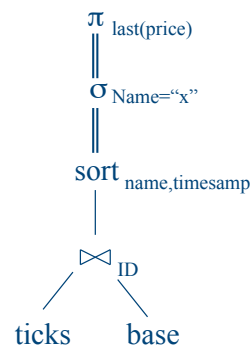
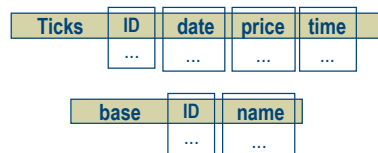
Performance depends on size



Last price for a name query

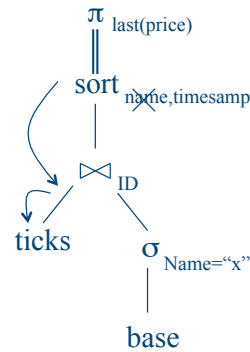
```

SELECT last(price)
FROM   ticks t,base b
      ASSUMING ORDER name,timestamp
WHERE  t.ID=b.ID
      AND name="x"
    
```



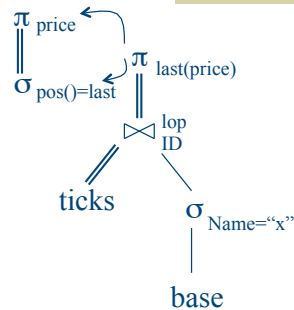
Last price for a name query

- ◆ The sort on name can be eliminated because there will be only one name
- ◆ Then, push sort
 - $\text{sort}_A(r_1 \bowtie r_2) \equiv_A \text{sort}_A(r_1) \bowtie^{\text{lop}} r_2$
 - $\text{sort}_A(r) \equiv_A r$



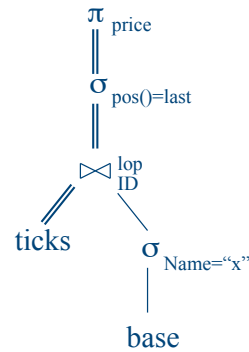
Last price for a name query

- ◆ The projection is carrying an implicit selection: $\text{last}(\text{price}) = \text{price}[n]$, where n is the last index of the price array
 - $\pi_{f(r.col[i])}(r) \equiv_{\text{order}(r)}$
 - $\pi_{f(r.col)}(\sigma_{\text{pos}()=i}(r))$



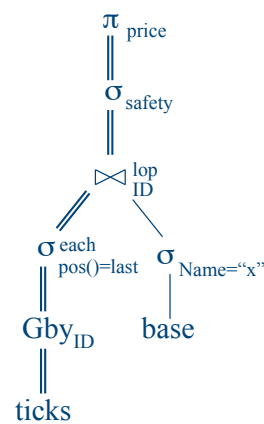
Last price for a name query

- ◆ But why join the entire relation if we are only using the last tuple?
- ◆ Can we somehow push the last selection down the join?

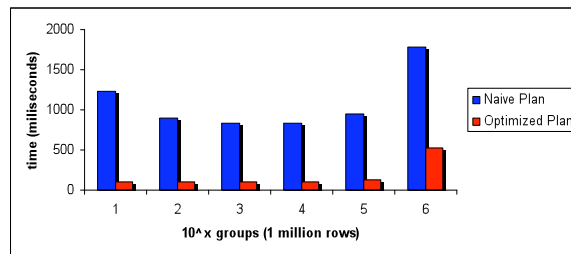


Last price for a name query

- ◆ We can take the last position of each ID on ticks to reduce cardinality, but we need to group by ticks.ID first
- ◆ But trading a join for a group by is usually a good deal?!
- ◆ One more step: make this an “edge by”



Performance



Conclusion

- ◆ AQuery declaratively incorporates order in a per-query basis
- ◆ Any clause can rely on order; expressions can be order-dependent
- ◆ Optimization possibilities are vast; performance improvements of an order of magnitude
- ◆ Applications to Finance, Network Management, ...

<http://www.cs.nyu.edu/~lerner>
(for additional references)