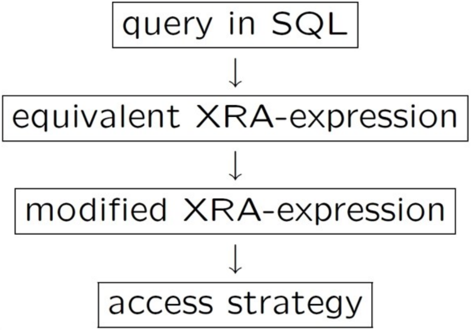
Query processing part 2
Algorithms

Hans Philippi

March 27, 2025

- We will not go into detail with respect to step 1
- The lecture on algebraic optimization deals with step 2
- We will now focus on step 3



Hardware characteristics of memory (2024)

	internal memory	SSD	hard disk
typical size	16-64 GB	1 TB	8 TB
access time	100 nsec	0.1 msec	8 msec
volatile	yes	no	no

- Disk IO is block (page) based; typical block size is 8 - 256 kB
- For our analysis, we suppose that tables are stored in an unordered collection of blocks
- Random access time can be minimized by indexing techniques
- Average access time can be enhanced by clustering

Hardware characteristics of memory

	internal memory	SSD	hard disk
typical size	16-64 GB	1 TB	8 TB
access time	100 nsec	0.1 msec	8 msec
volatile	yes	no	no

- To analyze performance of database access methods, we ignore internal memory access and only count IO (i.e. the number of disk accesses)
- ... although nowadays, analytical databases often are based on main memory storage techniques

Processing a selection

$$S := \sigma_p(R)$$

$$p : A_1 = c_1 \wedge A_2 = c_2 \wedge \ldots \wedge \; A_n = c_n$$

- Option 1: scan table R and apply predicate p to each tuple
- Option 2: if possible, use an index on one of the attributes A_i in p; check the retrieved tuples for the other selection requirements
- But what if R has more indices connected to $attr(p)$?
- Using more than one index and calculating intersections is an option, but more efficient solutions are available

Processing a selection

$$S := \sigma_p(R)$$

- Option 2: if possible, use an index on one of the attributes in p
- But what if R has more than one index connected to $attr(p)$?
- Selections on some attributes can be more selective than others
- Compare attributes *birthdate* (including year) and *weight* in kg for people
- The larger the number of values for an attribute, the higher the selectivity of the selection for that attribute

Table statistics

How to deal with statistics of the results of an algebraic operator?

- For each table R, we keep track of the number of tuples $T(R)$
- For each table R, we keep track of the number of blocks $B(R)$ on disk that R resides in
- In most cases, we know the tuple size in bytes, so we can estimate $B(R)$ from $T(R)$
- In general, a disk block will contain several tuples
- For each table R and each attribute A in $attr(R)$, we keep track of $V(R, A)$, i.e. the number of different values in $\pi_A(R)$

Table statistics: example

Bike			
bike_id	type	frame	gear
16531	xlite	aluminium	105
16647	xlite	carbon	ultegra
16648	xlite	carbon	dura-ace
23956	reveal	aluminium	105
23957	reveal	aluminium	ultegra

- $T(Bike) = 5$
- $B(Bike) = ?$, depends on ratio block size vs tuple size
- $V(Bike, type) = 2$; $V(Bike, frame) = 2$; $V(Bike, gear) = 3$
- $V(Bike, bike\_id) = 5$; equals $T(Bike)$, because $bike\_id$ is primary key

Processing a selection

$$S := \sigma_p(R)$$

- Option 2: if possible, use an index on one of the attributes in p
- But what if R has two indexed attributes: A and B?
- Choose the index on A if $V(R, A) \geq V(R, B)$, else otherwise

Result estimations

$$S := \sigma_{A=c}(R)$$

- $T(S) \approx T(R)/V(R, A)$
- $B(S)$ can be estimated from $T(S)$, given the ratio block size vs tuple size
- $V(S, A) = 1$
- Estimating $V(S, B)$ for other attributes B in $attr(S)$ is more tricky, but often unneccessary

Table statistics: histograms

$$S := \sigma_{A=c}(R)$$

- Compare $city = $ '*Amsterdam*' with $city = $ '*Giethoorn*'
- The distribution of $V(R, A)$ may be very uneven
- Refinement: histograms of value frequencies
- Example: attribute *weight* in kg for table *Patient* in a hospital

Patient: weight											
value	...	70	71	72	73	74	75	76	77	78	...
freq	...	1	3	4	7	5	3	4	2	2	...

Table statistics: histograms

Patient: weight										
value	...	70	71	72	73	74	75	76	77	78	...
freq	...	1	3	4	7	5	3	4	2	2	...

- Possible problem: size of statistics database
- Technique: interval histograms

Patient: weight				
value	...	71 - 74	75 - 78	...
freq	...	19	11	...

- Expected number of hits for *weight* $= 72$ equals 4.75
- Expected number of hits for *weight* $= 77$ equals 2.75

Table statistics: maintenance

Patient: weight										
value	...	70	71	72	73	74	75	76	77	78	...
freq	...	1	3	4	7	5	3	4	2	2	...

- Problem: maintenance of statistics database
- Observation: statistics do not need to be 100% correct
- Option: less frequent (partial) updating
- In case of extreme large databases, analyzing a small snapshot of the database in advance, to obtain statistics, is a possible technique

Join estimations

$$U := R \bowtie S$$

- Suppose we have one join attribute: A
- We will denote this situation by $U := R \bowtie_A S$
- Available statistics: $T(R), T(S), V(R, A), V(S, A)$
- Can we estimate $T(U)$?
- A good estimation for $T(U)$ is required when choosing between different join methods
- A good estimation for $T(U)$ is required when determining a join order for a join chain $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$

Join estimations

R

A	B	...
...	...	...
327	...	...
...	...	...

S

A	C	...
...	...	...
327	...	...
...	...	...
...	...	...
327	...	...
...	...	...

- Let us fix our attention on an arbitrary tuple t in R having A-value 327
- The estimated number of matching tuples for T in S equals $T(S)/V(S, A)$
- This results in $T(U) = T(R)T(S)/V(S, A)$

Join estimations

S

A	C	...
...	...	...
327	...	...
...	...	...

R

A	B	...
...	...	...
327	...	...
...	...	...
...	...	...
327	...	...
...	...	...

- Let us apply a modest feeling of symmetry
- The estimated number of matching tuples in R equals $T(R)/V(R,A)$
- This results in $T(U) = T(S)T(R)/V(R,A)$

Join estimations

$$U := R \bowtie_A S$$

- We have two estimations for $T(U)$
- $T(U) = T(R)T(S)/V(S, A)$
- $T(U) = T(R)T(S)/V(R, A)$
- We choose the minimum value of these two estimations
- Rationale: joins are in most cases asymmetric
- Special case: $R[A]$ is primary key and $S[A]$ is foreign key
- Then: $\pi_A S \subseteq \pi_A R$, $V(R, A) = T(R)$ and $T(U) = T(S)$

Join algorithms

$$U := R \bowtie_A S$$

- When analyzing performance, we will estimate the number of disk accessess (IO)
- Recall that an estimation of $T(R)$ also gives you an estimation of $B(R)$
- We have the horrible feeling that the number of IO's is proportional to $T(R)T(S)$ ...
- ... or at least to $B(R)B(S)$
- But we will see that $O(B(R) + B(S))$ is feasible!
- When comparing algorithms, we will ignore the IO of writing the final result table

Join algorithms

$$U := R \bowtie_A S$$

- General assumption: we have a main memory buffer size of M blocks to process joins
- ... although we silently suppose there is some extra buffer space for collecting output data
- We will discuss four join algorithms

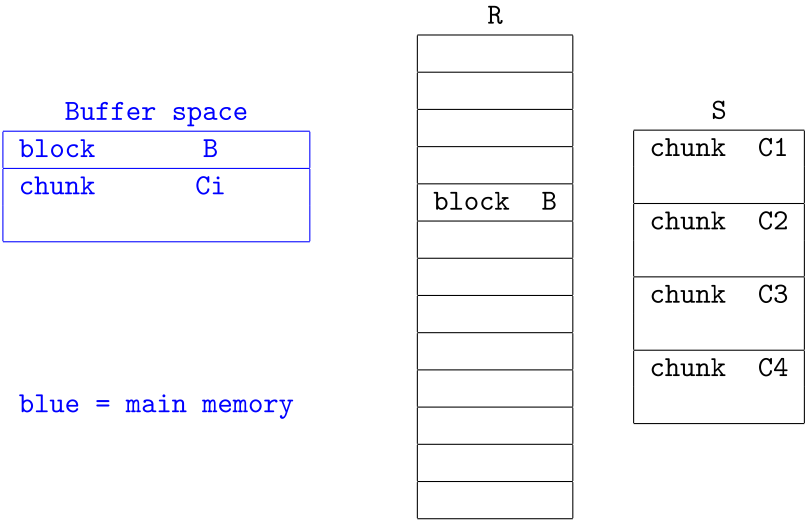
1. Block nested loop
2. Index nested loop
3. Sort-Merge join
4. Hash join

Join algorithms: Block nested loop

- Suppose S has the smallest number of blocks
- Split S in chunks, each of size $M - 1$ blocks (at most)

```
foreach chunk Ci of M-1 blocks of S {
  read Ci into main memory;
  foreach block B of R {
    read B into the free memory buffer;
    check all possible combinations
    of tuples t1 in chunk Ci and t2 in B2;
    if (t1.A = t2.A)
       write the join of these tuples to output;
  }
}
```

Join algorithms: Block nested loop



Block nested loop: analysis

- Each chunk C_i of S is read once; total IO for S is $B(S)$
- The number of times the outer loop runs: $\lceil B(S)/(M-1) \rceil$
- For each run of the outer loop, we need $B(R)$ disk accesses to scan R
- Total IO for both tables: $B(S) + \lceil B(S)/(M-1) \rceil * B(R)$
- Now suppose $B(S) < M$, then IO $= B(S) + B(R)$
- So if one operand of the join fits in main memory, block nested loop is optimal
- Note that we can improve this result if R and/or S is clustered on disk

Index nested loop

- Assumption: index on $S.A$

```
foreach block B of R {
  foreach tuple t in B {
    suppose t.A = a;
    use the index to find all t2 in S with t2.A = a;
    write the join of t with each t2 to output;
  }
}
```

Index nested loop: analysis

- $c =$ cost of index access (roughly 2 or 3 for B-tree)
- $\mu =$ average number of tuples found
- $\mu \approx T(S)/V(S, A)$
- $IO \approx B(R) + (c + \mu)T(R)$
- If A is primary key in S, $\mu = 1$
- This method might become interesting if the tuple size of R is large with respect to the block size of R

Sort-merge join

- The pseude code is given below
- An elaborate example will follow
- Note that any table R can be sorted in $IO = 4B(R)$

```
sort R on attribute A (if necessary);
sort S on attribute A (if necessary);
repeat {
  read the leading blocks from R and S
    containing the smallest common A-values;
  join the tuples in these blocks;
}
until R is finished or S is finished
```

Sort-merge join: example

Initial situation

R

A	B
a	13
b	27
a	94
c	33
c	56
c	29
c	83
b	76
a	39
...	

S

A	C
c	46
b	41
c	97
a	88
a	33
c	72
b	11
b	51
...	

Sort-merge join: example

After sorting on join attribute A

R

A	B
a	39
a	13
a	94
b	27
b	76
c	33
c	83
c	56
c	29
...	

S

A	C
a	33
a	88
b	51
b	14
b	11
c	97
c	46
c	72
...	

Sort-merge join: example

Copy leading blocks of both tables to buffer space

R

A	B
a	39
a	13
a	94
b	27
b	76
c	33
c	83
c	56
c	29
...	

S

A	C
a	33
a	88
b	51
b	14
b	11
c	97
c	46
c	72
...	

R

A	B
a	39
a	13
a	94
b	27

S

A	C
a	33
a	88
b	51
b	14

Sort-merge join: example

- Prepare partial join results in buffer space
- Partial join results are added tot Result table on disk

R

A	B
a	39
a	13
a	94
b	27
b	76
c	33
c	83
c	56
c	29
...	

S

A	C
a	33
a	88
b	51
b	14
b	11
c	97
c	46
c	72
...	

AddToResult

A	B	C
a	39	33
a	39	88
a	13	33
a	13	88
a	94	33
a	94	88

Sort-merge join: example

Copy new leading blocks of both tables to buffer space

R

A	B
a	39
a	13
a	94
b	27
b	76
c	33
c	83
c	56
c	29
...	

S

A	C
a	33
a	88
b	51
b	14
b	11
c	97
c	46
c	72
...	

R

A	B
b	27
b	76
c	33
c	83
c	56

S

A	C
b	51
b	14
b	11
c	97
c	46
c	72

Sort-merge join: example

- Prepare partial join results in buffer space (blue)
- Partial join results are added tot Result table on disk

R

A	B
a	39
a	13
a	94
b	27
b	76
c	33
c	83
c	56
c	29
...	

S

A	C
a	33
a	88
b	51
b	14
b	11
c	97
c	46
c	72
...	

AddToResult

A	B	C
b	27	51
b	27	14
b	27	11
b	76	51
b	76	14
b	76	11

Sort-merge join

- Note that the buffer space generally consists of a lot of megabytes or even gigabytes
- Due to the small example size, it is suggested we handle only one A-value in each iteration, but in reality, several A-values will be dealt with
- We do not count for the cost of the join in main memory
- In extreme cases, the amount of data dealing with one single join value may exceed the buffer space
- In that case, techniques inspired by two-phase external sorting can be applied

Sort-merge join: analysis

- Cost of sorting: $4(B(R) + B(S))$
- Two way merge scan: $B(R) + B(S)$
- Total cost: $5(B(R) + B(S))$
- Note that this join method can be integrated with the merge sort of both operands; in that case $IO = 3(B(R) + B(S))$
- Two phase merge sort algorithms are applicable as long as $B(R) + B(S) \leq M^2$
- M^2 is quite a lot

Hash join

- We will prepare hash buckets for both tables R and S
- Choose an appropriate size of M (number of buckets) for the hash buckets, based on table statistics
- Choose a hash function for the domain of A with codomain $0..M-1$
- Each bucket has a buffer window to collect hashed tuples
- Scan through R and send each tuple to the appropriate bucket
- Scan through S and send each tuple to the appropriate bucket
- After scanning R and S, load each corresponding couple of R and S buckets into main memory and determine the join results for the tuples in these buckets
- Hash join works only for equi join

Hash join: example

- Join $R[A, B]$ with $S[A, C]$
- hash function: $h(A) = A \; div \; 10$ [1]

R

A	B
12	a
3	b
29	c
7	d
13	e
12	f
27	g

S

A	C
29	h
7	i
12	j
8	k
28	l
12	m

[1] Not a very sophisticated hash function, but illustrative

- Create buckets for $h(A) = 0, 1, 2, ...$

R

A	B
12	a
3	b
29	c
7	d
13	e
12	f
27	g

S

A	C
29	h
7	i
12	j
8	k
28	l
12	m

R0

A	B
7	d
3	b

S0

A	C
7	i
8	k

R1

A	B
12	a
13	e
12	f

S1

A	C
12	j
12	m

R2

A	B
29	c
27	g

S2

A	C
28	l
29	h

Hash join: example

- Read buckets for $h(A) = 0$ into buffer space

R0

A	B
7	d
3	b

S0

A	C
7	i
8	k

R1

A	B
12	a
13	e
12	f

S1

A	C
12	j
12	m

R2

A	B
29	c
27	g

S2

A	C
28	l
29	h

R0

A	B
7	d
3	b

S0

A	C
7	i
8	k

Hash join: example

- Calculate joined tuples in buffer space

R0

A	B
7	d
3	b

S0

A	C
7	i
8	k

R1

A	B
12	a
13	e
12	f

S1

A	C
12	j
12	m

R2

A	B
29	c
27	g

S2

A	C
28	l
29	h

R0

A	B
7	d
3	b

S0

A	C
7	i
8	k

R0 $\bowtie$ S0

A	B	C
7	d	i

Hash join: example

- Write joined tuples to the result table (buffered)

R0

A	B
7	d
3	b

S0

A	C
7	i
8	k

R0 $\bowtie$ S0

A	B	C
7	d	i

Result

A	B	C
7	d	i

Hash join: example

- Read buckets for $h(A) = 1$ into buffer space

R0

A	B
7	d
3	b

S0

A	C
7	i
8	k

R1

A	B
12	a
13	e
12	f

S1

A	C
12	j
12	m

R2

A	B
29	c
27	g

S2

A	C
28	l
29	h

R1

A	B
12	a
13	e
12	f

S1

A	C
12	j
12	m

Hash join: example

- Calculate joined tuples in buffer space

R0

A	B
7	d
3	b

S0

A	C
7	i
8	k

R1

A	B
12	a
13	e
12	f

S1

A	C
12	j
12	m

R2

A	B
29	c
27	g

S2

A	C
28	l
29	h

R1

A	B
12	a
13	e
12	b

S1

A	C
12	j
12	m

R1 $\bowtie$ S1

A	B	C
12	a	j
12	a	m
12	b	j
12	a	m

Hash join: example

- Add joined tuples to the result table

R1

A	B
12	a
13	e
12	b

S1

A	C
12	j
12	m

R1 $\bowtie$ S1

A	B	C
12	a	j
12	a	m
12	b	j
12	b	m

Result

A	B	C
7	d	i
12	a	j
12	a	m
12	b	j
12	b	m

Hash join: example

- And repeat this for all buckets

R2

A	B
29	c
27	g

S2

A	C
28	l
29	h

R2 $\bowtie$ S2

A	B	C
29	c	h

Result

A	B	C
7	d	i
12	a	j
12	a	m
12	b	j
12	b	m
29	c	h

Hash join: analysis

- Note that the total size of the two hashed tables is roughly $B(R) + B(S)$
- The cost of scanning R: $IO = B(R)$
- The cost of filling the hash buckets for R: $IO \approx B(R)$
- The cost of scanning S: $IO = B(S)$
- The cost of filling the hash buckets for S: $IO \approx B(S)$
- Scanning all hash buckets to calculate resulting tuples: $IO \approx B(R) + B(S)$
- We do not count for the cost of the join in main memory
- We ignore writing the result
- Overall cost: $IO \approx 3(B(R) + B(S))$

Epilog: OLTP vs OLAP

- Note that we often distinguish bnetween two kinds of applications for database management systems: *OLTP* and *OLAP*
- OLTP stands for *online transaction processing*
- A transaction oriented DBMS (*production database*) typically supports processing high volumes of small updates (commerce, banking, reservation systems)
- Transactional integrity is of utmost importance
- Schema design for production databases focuses heavily on prevention of data redundancy and consistency (normalization)
- The amount of indices is limited to prevent update overhead, but some are essential for performance

OLTP vs OLAP

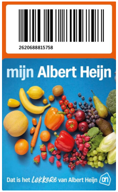
- Note that we often distinguish two kinds of applications for database management systems: *OLTP* and *OLAP*
- OLAP stands for *online analytical processing*
- An analysis oriented DBMS (*analytical database, data warehouse*) typically supports dealing with large and complex queries
- Analytical databases generally are created by taking snapshots from production databases
- These snapshots are extensively preprocessed
- Analytical databases typically are fixed for some time and read only
- Therefore, analytical databases generally do not support transaction processing

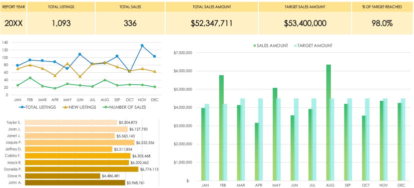
OLTP vs OLAP

- An analysis oriented DBMS (analytical database, data warehouse) typically supports dealing with large and complex queries
- Support by indices is abundant
- Given the read only behaviour, data redundancy can be applied where necessary (materialized views)
- Analytical databases often comprise historical data
- Main memory database technology is an interesting candidate for analytical databases

OLTP and OLAP: example

- A large, countrywide grocery store deals with millions of transactions a day
- An OLTP system supports all checkouts and payments
- In most cases, transactions are connected to a known client
- An OLAP system provides overviews of all sales regarding to a certain period
- Market analysts may identify trends with respect to specific products, product groups, time periods, price development, and so on ...





OLTP and OLAP: example

- Market analysts may find opportunities for client directed offerings
- OLAP supports *market basket analysis*: product X is often bought in combination with product Y
- Market basket analysis has been one of the earliest challenges of *data mining*
- Find groups of articles that are often bought together

Transaction ID	**itemset**
1	{wine, cheese, bread}
2	{cheese, bananas, wine }
3	{wine, strawberries, cheese}
4	{wine, cheese}
5	{diapers, beer, milk}