Order Preserving Joins

- some query languages operatore on lists instead of sets/bags
- order of tuples matters
- examples: XPath/XQuery
- alternatives: either add sort operators or use order preserving operators

Here, we define order preserving operators, *list* $\longrightarrow$ *list*

- let L be a list
- $L[1]$ is the first entry in L
- $L[2 : |L|]$ are the remaining entries

Order Preserving Selection

We define the order preserving selection σ^L as follows:

$$\sigma^L_p(e) := \begin{cases} \epsilon & \text{if } e = \epsilon \\ < e[1] > \circ \sigma^L_p(e[2:|e|]) & \text{if } p(e[1]) \\ \sigma^L_p(e[2:|e|]) & \text{otherwise} \end{cases}$$

- filters like a normal selection
- preserves the relative ordering (guaranteed)

Order Preserving Cross Product

We define the order preserving cross product $\times^L$ as follows:

$$e_1 \times^L e_2 := \begin{cases} \epsilon & \text{if } e_1 = \epsilon \\ (e[1] \hat{\times}^L e_2) \circ (e_1[2 : |e_1] \times^L e_2) & \text{otherwise} \end{cases}$$

using the tuple/list product defined as:

$$t \hat{\times}^L e := \begin{cases} \epsilon & \text{if } e = \epsilon \\ < t \circ e[1] > \circ (t \hat{\times}^L e[2 : |e|]) & \text{otherwise} \end{cases}$$

- preserves the order of e_1
- order of e_2 is preserved for each e_1 group

Order Preserving Join

The definition of the order preserving join is analogous to the non-order preserving case:

$$e_1 \bowtie_p^L e_2 := \sigma_p^L(e_1 \times^L e_2)$$

- preserves order of e_1, order of e_2 relative to e_1

Equivalences

$$
\begin{array}{rcll}
\sigma^L_{p_1}(\sigma^L_{p_2}(e)) & \equiv & \sigma^L_{p_2}(\sigma^L_{p_1}(e)) & \\
\sigma^L_{p_1}(e_1 \bowtie^L_{p_2} e_2) & \equiv & \sigma^L_{p_1}(e_1) \bowtie^L_{p_2} e_2) & \text{if } \mathcal{F}(p_1) \subseteq \mathcal{A}(e_1) \\
\sigma^L_{p_2}(e_1 \bowtie^L_{p_2} e_2) & \equiv & e_1 \bowtie^L_{p_2} \sigma^L_{p_1}(e_2) & \text{if } \mathcal{F}(p_1) \subseteq \mathcal{A}(e_2) \\
e_1 \bowtie^L_{p_1}(e_2 \bowtie^L_{p_2} e_3) & \equiv & (e_1 \bowtie^L_{p_1} e_2) \bowtie^L_{p_2} e_3) & \text{if } \mathcal{F}(p_i) \subseteq \mathcal{A}(e_i) \cup \mathcal{A}(e_{i+1})
\end{array}
$$

- swap selections
- push selections down
- associativity

Commutativity

Consider the relations $R_1 =< [a:1],[a:2] >$ and $R_2 =< [b:1],[b:2] >$.
Then

$$
\begin{aligned}
R_1 \bowtie^L_{true} R_2 &= < [a:1,b:1],[a:1,b:2],[a:2,b:1],[a:2,b:2] > \\
R_2 \bowtie^L_{true} R_1 &= < [a:1,b:1],[a:2,b:1],[a:1,b:2],[a:2,b:2] >
\end{aligned}
$$

- the order preserving join is not commutative

Algorithm

- similar to matrix multiplication
- in addition: selection push down
- DP table is a $n \times n$ array (or rather 4 arrays)
- algorithm fills arrays p, s, c, t:
  - p: applicable predicates
  - s: statistics (cardinality, perhaps more)
  - c: costs
  - t: split position for larger plans
- plan is extracted from the arrays afterwards

Algorithm (2)

OrderPreservingJoins($R = \{R_1, \ldots, R_n\}$, P)
Input: a set of relations to be joined and a set of predicates
Output: fills p, s, c, t
for each $1 \leq i \leq n$ {
  $p[i, i]$ =predicates from P applicable to R_i
  $P = P \setminus p[i, i]$
  $s[i, i]$ =statistics for $\sigma_{p[i,i]}(R_i)$
  $c[i, i]$ =costs for $\sigma_{p[i,i]}(R_i)$
}

Algorithm (3)

for each $2 \leq l \leq n$ **ascending** {
  for each $1 \leq i \leq n - l + 1$ {
    $j = i + l - 1$
    $p[i,j]$=predicates from P applicable to $R_i, \ldots, R_j$
    $P = P \setminus p[i,j]$
    $s[i,j]$=statistics derived from $s[i,j-1]$ and $s[j,j]$ including $p[i,j]$
    $c[i,j]$=∞
    for each $i \leq k < j$ {
      $q = c[i,k] + c[k+1,j]$+costs for $s[i,k]$ and $s[k+1,j]$ and $p[i,j]$
      if $q < c[i,j]$ {
        c[i,j]=q
        t[i,j]=k
      }
    }
  }
}

Algorithm (4)

ExtractPlan($R = \{R_1, \ldots, R_n\}$,t,p)
Input: a set of relations, arrays t and p
Output: a bushy join tree
return ExtractPlanRec(R,t,p,1,n)

ExtractPlanRec($R = \{R_1, \ldots, R_n\}$,t,p,i,j)
if $i < j$ {
  T_1 =ExtractPlanRec(R,t,p,i,$t[i,j]$)
  T_2 =ExtractPlanRec(R,t,p,$t[i,j]+1$,j)
  return $T_1 \bowtie^{L}_{p[i,j]} T_2$
} **else** {
  return $\sigma_{p[i,j]} R_i$
}

4. Accessing the Data

In this chapter we go into some details:

- deep into the (runtime) system
- close to the hardware

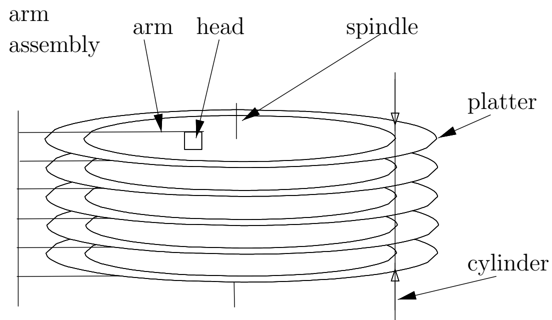
Goal:

- estimation and optimization of disk access costs

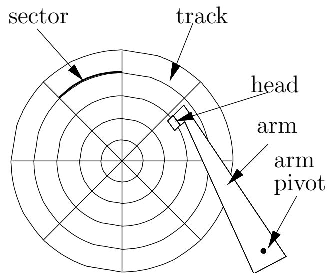
4. Accessing the Data (2)

- disk drives
- database buffer
- physical database organization
- physical algebra
- temporal relations and table functions
- indices
- counting the number of accesses
- disk drive costs
- selectivity estimations

Assembly



a. side view



b. top view

Zones

- outer tracks/sectors longer than inner ones
- highest density is fixed
- results in waste in outer sectors
- thus: cylinders organized into zones

Zones (2)

- every zone contains a fixed number of consecutive cylinders
- every cylinder in a zone has the same number of sectors per track
- outer zones have more sectors per track than inner zones
- since rotation speed is fixed: higher throughput on outer cylinders

Track Skew

Read all sectors of all tracks of some consecutive cylinders:

- read all sectors of one track
- switch to next track: small adjustment of head necessary
  called: *head switch*
- this causes tiny delay
- thus, if all tracks start at the same angular position then we miss the start of the first sector of the next track
- remedy: *track skew*

Cylinder Skew

Read all sectors of all tracks of some consecutive cylinders:

- read all sectors of all tracks of some cylinder
- switching to the next cylinder causes some delay
- again, we miss the start of the first sector, if the tracks start all start at the same angular position
- remedy: *cylinder skew*

Addressing Sectors

- physical Address: cylinder number, head (surface) number, sector number
- logical Address: LBN (logical block number)

LBN to Physical Address

Mapping:

Cylinder	Track	LBN	number of sectors per track
0	0	0	573
	1	573	573
...	...	...	...
	5	2865	573
1	0	3438	573
...	...	...	...
15041	0	35841845	253
...	...	...	...

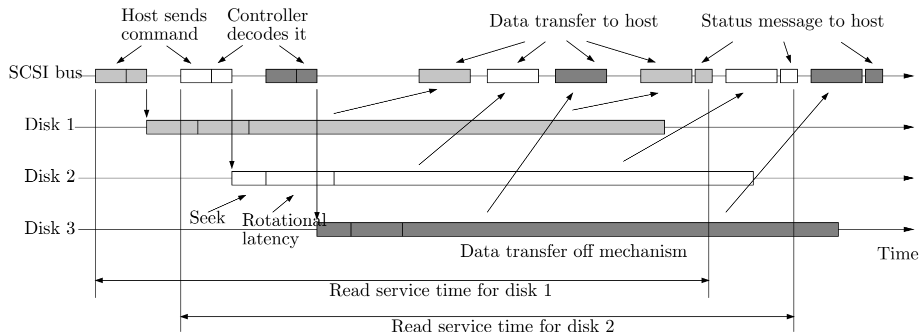
LBN to Physical Address (2)

This ideal view of the mapping is disturbed by *bad blocks*

- due to the high density, no perfect manufacturing is possible
- as a consequence *bad blocks* occur (sectors that cannot be used)
- reserve some blocks, tracks, cylinders for remapping bad blocks

Bad blocks may cause hickups during sequential reads

Reading/Writing a Block



Reading/Writing a Block (2)

1. the host sends the SCSI command.
2. the disk controller decodes the command and calculates the physical address.
3. during the seek the disk drive's arm is positioned such that the according head is correctly placed over the cylinder where the requested block resides. This step consists of several phases.
   3.1 the disk controler accelerates the arm.
   3.2 for long seeks, the arm moves with maximum velocity (coast).
   3.3 the disk controler slows down the arm.
   3.4 the disk arm settles for the desired location. The settle times differ for read and write requests. For reads, an aggressive strategy is used. If, after all, it turns out that the block could not be read correctly, we can just discard it. For writing, a more conservative strategy is in order.
4. the disk has to wait until the sector where the requested block resides comes under the head (rotation latency).
5. the disk reads the sector and transfers data to the host.
6. finally, it sends a status message.

Optimizing Round Trip Time

- caching
- read-ahead
- command queuing

Seek Time

A good approximation of the seek time where d cylinders have to be travelled is given by

$$seektime(d) = \begin{cases} c_1 + c_2\sqrt{d} & d \leq c_0 \\ c_3 + c_4 d & d > c_0 \end{cases}$$

where the constants c_i are disk specific. The constant c_0 indicates the maximum number cylinders where no coast takes place: seeking over a distance of more than c_0 cylinders results in a phase where the disk arm moves with maximum velocity.