4. Accessing the Data

In this chapter we go into some details:

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

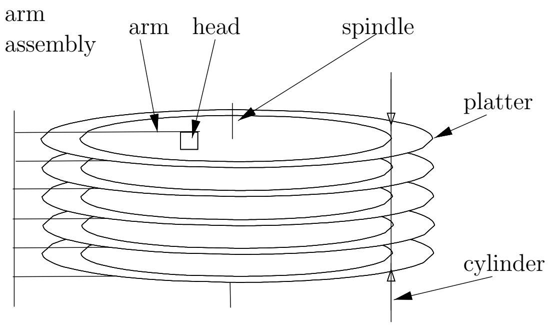
Goal:

- estimation and optimization of disk access costs

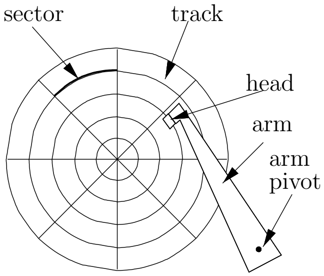
4. Accessing the Data (2)

- 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
...	...	...	...

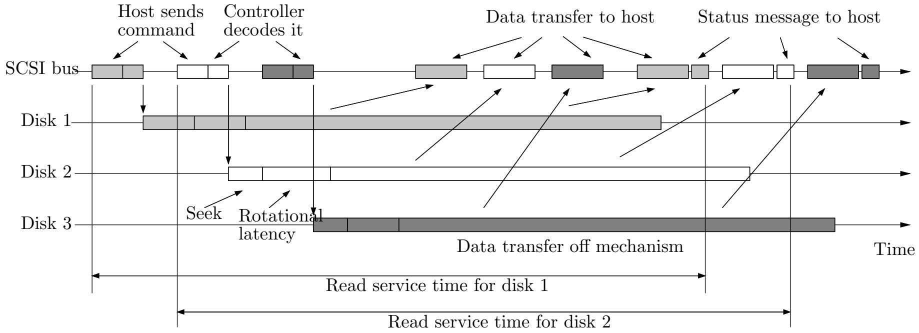
LBN to Physical Address (2)

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.

Cost model: initial thoughts

Disk access costs depend on

- the current position of the disk arm and
- the angular position of the platters

Both are not known at query compilation time
Consequence:

- estimating the costs of a single disk access at query compilation time may result in large estimation error

Better: costs of many accesses
Nonetheless: First Simplistic Cost Model to give a feeling for disk drive access costs

Simplistic Cost Model

We introduce some disk drive parameters for out simplistic cost model:

- average latency time: average time for positioning (seek+rotational delay)
    - use average access time for a single request
    - Estimation error can (on the average) be as “low” as 35%
- sustained read/write rate:
    - after positioning, rate at which data can be delivered using sequential read

Model 2004

A hypothetical disk (inspired by disks available in 2004) then has the following parameters:

Model 2004		
Parameter	Value	Abbreviated Name
capacity	180 GB	D_{cap}
average latency time	5 ms	D_{lat}
sustained read rate	100 MB/s	D_{srr}
sustained write rate	100 MB/s	D_{swr}

The time a disk needs to read and transfer n bytes is then approximated by $D_{\mathsf{lat}} + n/D_{\mathsf{srr}}$.

Sequential vs. Random I/O

Database management system developers distinguish between

- *sequential* I/O and
- *random* I/O.

In our simplistic cost model:

- for sequential I/O, there is only one positioning at the beginning and then, we can assume that data is read with the sustained read rate.
- for random I/O, one positioning for every unit of transfer—typically a page of say 8 KB—is assumed.

Simplistic Cost Model

Read 100 MB

- Sequential read: 5 ms + 1 s
- Random read (8K pages): 65 s

Simplistic Cost Model (2)

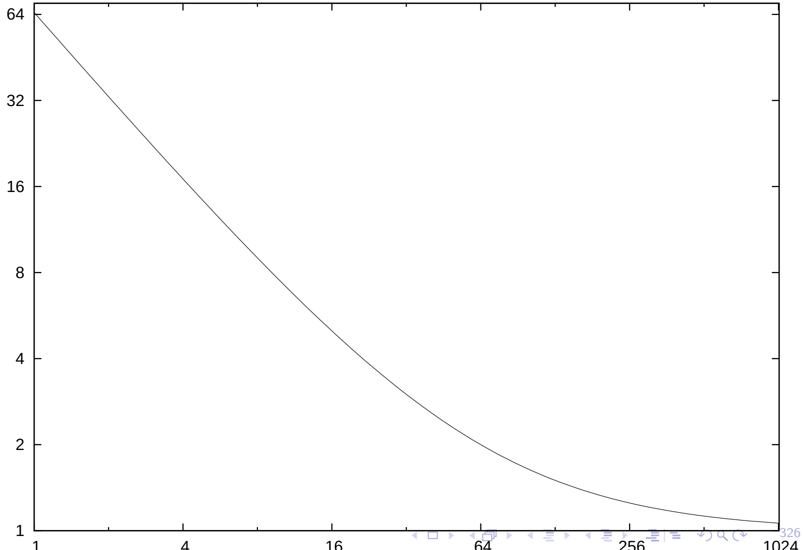
Problems:

- other applications
- other transactions
- other read operations in the same QEP

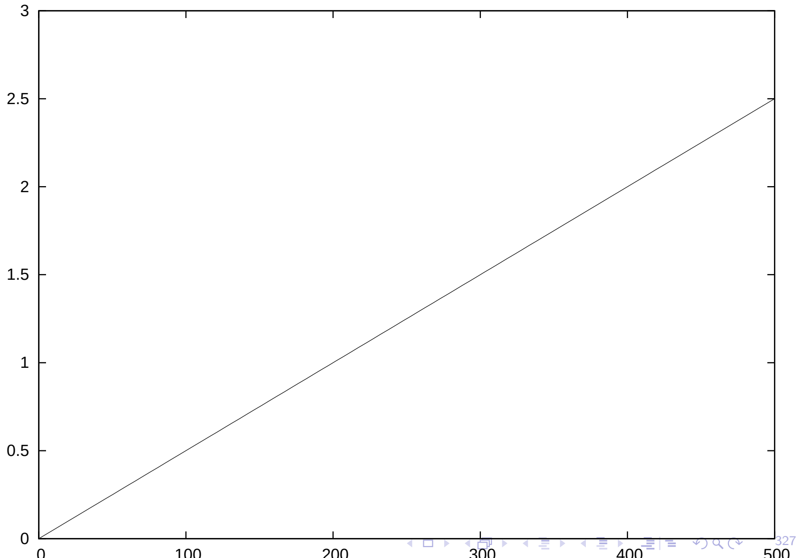
may request blocks from the same disk and move away the head(s) from the current position

Further: 100 MB sequential search poses problem to buffer manager

Time to Read 100 MB (x: number of 8 KB chunks)



Time to Read *n* Random Pages



Simplistic Cost Model (3)

100 MB can be stored on 12800 8 KB pages.
In our simplistic cost model, reading 200 pages randomly costs about the same as reading 100 MB sequentially.
That is, reading 1/64th of 100 MB randomly takes as long as reading the 100 MB sequentially.

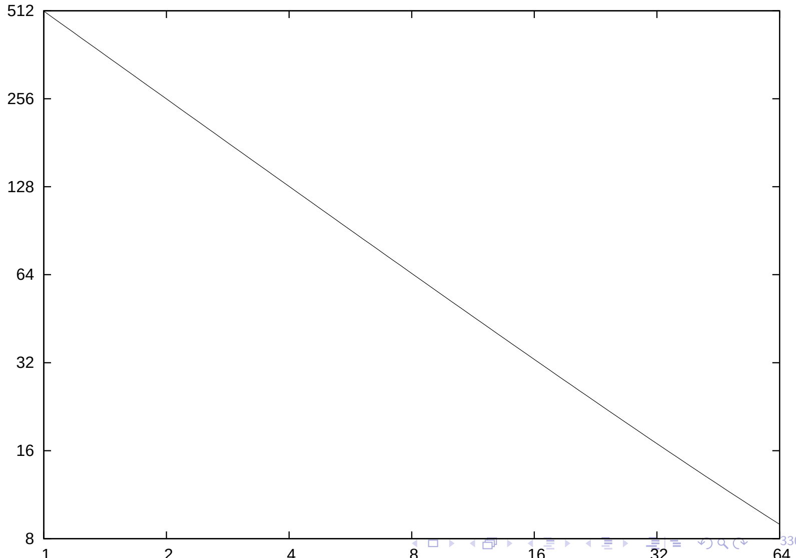
Simplistic Cost Model (4)

Let us denote by a the positioning time, s the sustained read rate, p the page size, and d some amount of consecutively stored bytes. Let us calculate the break even point

$$\begin{aligned} n * (a + p/s) &= a + d/s \\ n &= (a + d/s)/(a + p/s) \\ &= (as + d)/(as + p) \end{aligned}$$

a and s are disk parameters and, hence, fixed. For a fixed d, the break even point depends on the page size.
Next Figure: x-axis: is the page size p in multiples of 1 K; y-axis: $(d/p)/n$ for $d = 100$ MB.

Break Even Point (depending on page size)



Two Lessons Learned

- sequential read is much faster than random read
- the runtime system should secure sequential read

The latter point can be generalized:

- the runtime system of a database management system has, as far as query execution is concerned, two equally important tasks:
  - allow for efficient query evaluation plans and
  - allow for smooth, simple, and robust cost functions.

Measures to Achieve the Above

Typical measures on the database side are

- carefully chosen physical layout on disk
  (e.g. cylinder or track-aligned extents, clustering)
- disk scheduling, multi-page requests
- (asynchronous) prefetching,
- piggy-back scans,
- buffering (e.g. multiple buffers, replacement strategy) and last but not least
- efficient and robust algorithms for algebraic operators

Disk Drive: Parameters

D_{cyl}	total number of cylinders
D_{track}	total number of tracks
D_{sec}	total number of sectors
D_{tpc}	number of tracks per cylinder (= number of surfaces)

D_{cmd}	command interpretation time
D_{rot}	time for a full rotation
D_{rdsettle}	time for settle for read
D_{wrsettle}	time for settle for write
D_{hdswitch}	time for head switch

Disk Drive: Parameters (2)

D_{zone}	total number of zones
$D_{\text{zcyl}}(i)$	number of cylinders in zone i
$D_{\text{zspt}}(i)$	number of sectors per track in zone i
$D_{\text{zspc}}(i)$	number of sectors per cylinder in zone i $(= D_{\text{tpc}} D_{\text{zspt}}(i))$
$D_{\text{zscan}}(i)$	time to scan a sector in zone i $(= D_{\text{rot}}/Dzspti)$

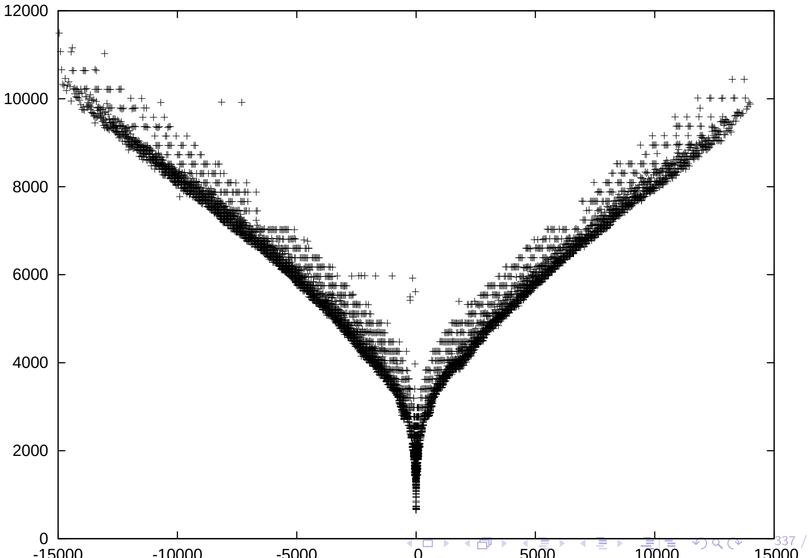
Disk Drive: Parameters (3)

D_{seekavg}	average seek costs
D_{clim}	parameter for seek cost function
D_{ca}	parameter for seek cost function
D_{cb}	parameter for seek cost function
D_{cc}	parameter for seek cost function
D_{cd}	parameter for seek cost function
$D_{\text{fseek}}(d)$	cost of a seek of d cylinders $D_{\text{fseek}}(d) = \begin{cases} D_{\text{ca}} + D_{\text{cb}}\sqrt{d} & \text{if } d \leq D_{\text{clim}} \\ D_{\text{cc}} + D_{\text{cd}}\,d & \text{if } d > D_{\text{clim}} \end{cases}$
$D_{\text{frot}}(s, i)$	rotation cost for s sectors of zone i $(= sD_{\text{zscan}}(i))$

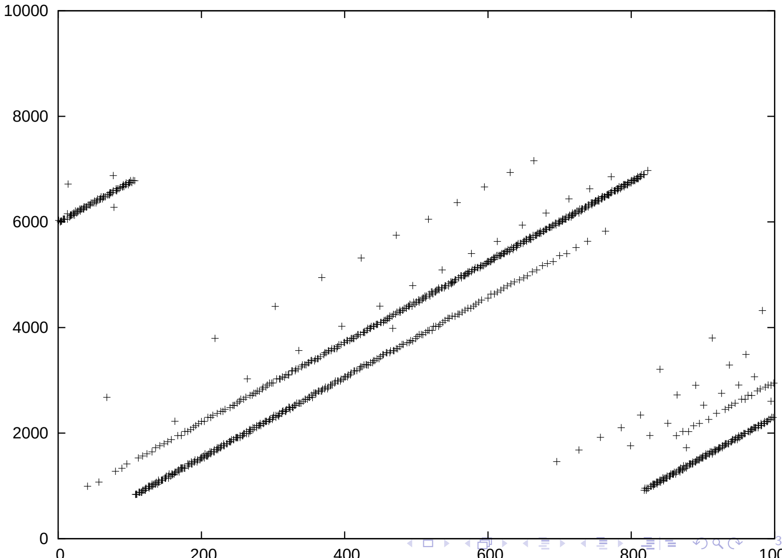
Extraction of Disk Drive Parameters

- documentation: often not sufficient
- mapping: interrogation via SCSI-Mapping command (disk drives lie)
- use benchmarking tools, e.g.:
  - Diskbench
  - Skippy (Microbenchmark)
  - Zoned

Seek Curve Measured with Diskbench



Skippy Benchmark Example



Interpretation of Skippy Results

- x-axis: distance (sectors)
- y-axis: time
- difference topmost/bottommost line: rotational latency
- difference two lowest ‘lines’: head switch time
- difference lowest ‘line’ topmost spots: cylinder switch time
- start lowest ‘line’: minimal time to media
- plus other parameters

Upper bound on Seek Time

Theorem (Qyang)

If the disk arm has to travel over a region of C cylinders, it is positioned on the first of the C cylinders, and has to stop at $s - 1$ of them, then $sD_{fseek}(C/s)$ is an upper bound for the seek time.

Database Buffer

The database buffer

1. is a finite piece of memory,
2. typically supports a limited number of different page sizes (mostly one or two),
3. is often fragmented into several buffer pools,
4. each having a replacement strategy (typically enhanced by hints).

Given the page identifier, the buffer frame is found by a hashtable lookup.
Accesses to the hash table and the buffer frame need to be synchronized.
Before accessing a page in the buffer, it must be fixed.
These points account for the fact that the costs of accessing a page in the buffer are therefore greater than zero.

Buffer Accesses

Consider page acceses in a buffer with 2 pages:

page no	action
0	read page 0, place it in buffer
1	read page 1, place it in buffer
0	fix page 0 in buffer
2	swap out a page (e.g. 1), read 2, place it in buffer
0	fix page 0 in buffer
3	swap out a page, read 3, place it in buffer
...	

- replacement strategy is imporant
- unfixes omitted

Replacement Strategies

Some popular replacement strategies:

- random
- fifo
- lru
- Q2

lru is very popular

Replacement Strategies - random

- when a new page slot is needed, remove a random other page from the buffer
- easy to implements, needs no additional memory
- but does not take the access patterns into account
- primarily used as base line
- suitable for analytic results

Replacement Strategies - fifo

- first in - first out
- remove the page that was place in the buffer first
- easy to implement, needs no/few additional memory
- but does not adapt very well do access patterns
- increasing buffer size may hurt it

Fifo Anomaly:

- access pattern: 3 2 1 0 3 2 4 3 2 1 0 4
- buffer sizes: 3 vs. 4

Replacement Strategies - lru

- least recently used
- remove the page that has not been accessed for longest time
- requires a priority queue/linked list
- adapt to access patterns, popular pages stay in memory
- but slow to remove pages

very popular replacement strategy

Replacement Strategies - 2Q

- two queues
- a fifo queue and a lru queue
- place pages first in fifo, if they are accessed again place them in lru
- gets rid of pages that are accessed only once fast
- superior to lru, example of a "real" replacement strategy

Replacement Strategies - Effect on the Cost Model

- replacement affects the costs
- cost model needs predictions, though
- very hard to do in general

Typical approaches:

- ignore buffer effects
- assume random replacement
- make use of known access characteristics

Physical Database Organization

The database organizes the physical storage in multiple layers:

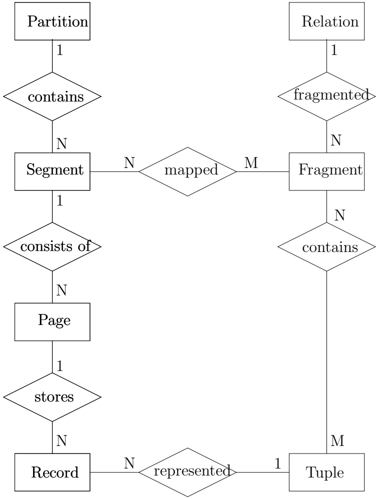
1. partition: sequence of pages (consecutive on disk)
2. extent: subsequence of a partition
3. segment (file): logical sequence of pages (implemented e.g. as set of extents)
4. record: sequence of bytes stored on a page

Note:

- partition/extent/page/record are physical structures
- a segment is a logical structure

Physical Storage of Relations

Mapping of a relation's tuples onto records stored on pages in segments:



Access to Database Items

- database item: something stored in DB
- database item can be set (bag, sequence) of items
- access to a database item then produces stream of smaller database items
- the operation that does so is called *scan*

Scan Example

Using a relation scan `rscan`, the query

select *
from Student

can be answered by `rscan(Student)`
(segments? extents?): Assumption:

- segment scans and each relation stored in one segment
- segment and relation name identical

Then `fscan(Student)` and `Student` denote scans of all tuples in a relation

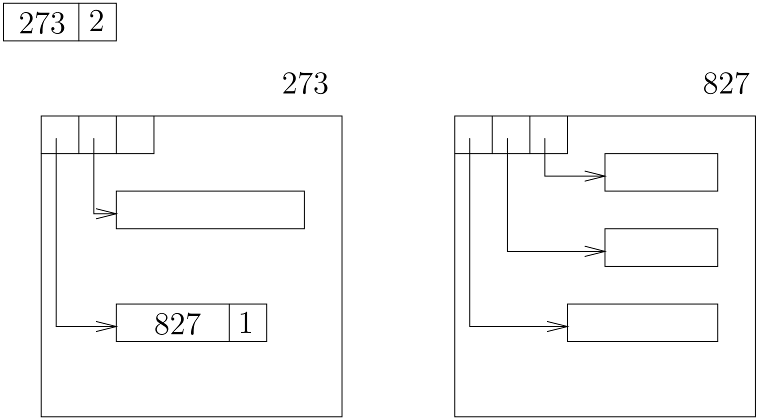
Model of a Segment

- for our cost model, we need a model of segments.
- we assume an extent-based segment implementation.
- every segment then is a sequence of extents.
- every extent can be described by a pair (F_j, L_j) containing its first and last cylinder.
  (For simplicity, we assume that extents span whole cylinders.)
- an extent may cross a zone boundary.
- hence: split extents to align them with zone boundaries.
- segment can be described by a sequence of triples (F_i, L_i, z_i) ordered on F_i where z_i is the zone number in which the extent lies.

Model of a Segment

S_{ext}	number of extents in the segment
$S_{\text{cfirst}}(i)$	first cylinder in extent i (F_i)
$S_{\text{clast}}(i)$	last cylinder in extent i (L_i)
$S_{\text{zone}}(i)$	zone of extent i (z_i)
$S_{\text{cpe}}(i)$	number of cylinders in extent i $(= S_{\text{clast}}(i) - S_{\text{cfirst}}(i) + 1)$
S_{sec}	total number of sectors in the segment $(= \sum_{i=1}^{S_{\text{ext}}} S_{\text{cpe}}(i) D_{\text{zspc}}(S_{\text{zone}}(i)))$

Slotted Page



- page is organized into areas (slots)
- slots point to data chunks
- slots may point to other pages

Tuple Identifier (TID)

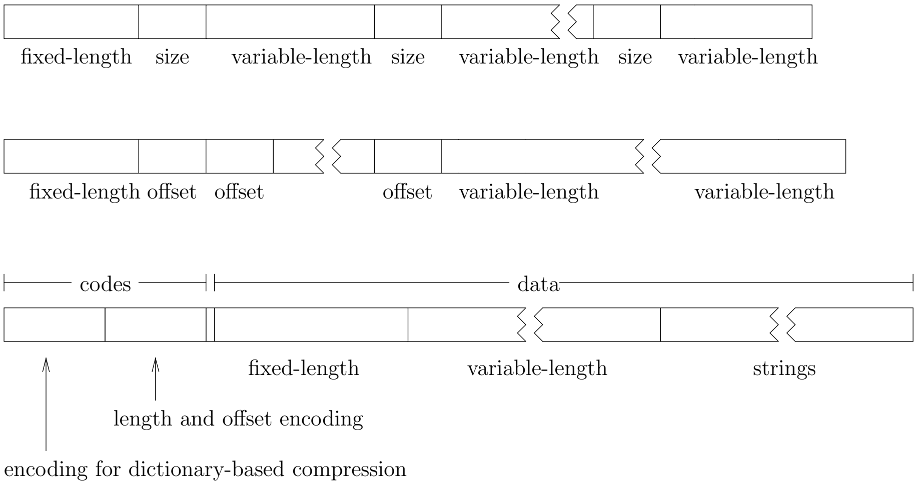
TID is conjunction of

- page identifier (e.g. partition/segment no, page no)
- slot number

TID sometimes called Row Identifier (RID)

Record Layout

Different layouts possible:



Record Layout (2)

Record layout is a compromise:

- space consumption vs. CPU
- data model specific properties: e.g. generalization
- versioning / easy schema migration
- record layout typically not trivial
- accessing an attribute value has non-zero cost

Physical Algebra

- building blocks for query execution
- implements the algorithms for query execution
- very generic, reusable components
- describes the general execution approach
- annotated with predicates etc. for query specific parts

Iterator Concept

The general interface of each operator is:

- open
- next
- close

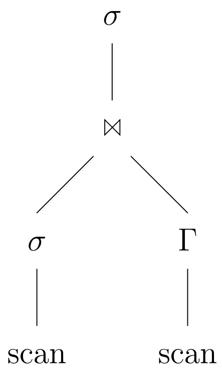
All physical algebraic operators are implemented as iterators.

- produce a stream of data items (tuples)

Implementations vary slightly for performance tuning (concept the same):

- `first`/`next` instead of `next`
- blocks of tuples instead of single tuples

Iterator Example



Note: all details (subscripts, implementations etc.) are omitted here

Pipelining

Pipelining is fundamental for the physical algebra:

- physical operators are iterators over the data
- they produce a stream of single tuples
- tuple stream if passed through other operators
- pipelining operators just pass the data through, they only filter or augment
- data is not copied or materialized
- very efficient processing

pipeline breakers disrupt this pipeline and materialize data:

- very expensive, can cause superfluous work
- sometimes cannot be avoided, though

Simple Scan

- a `rscan` operation is rarely supported.
- instead: scans on segments (files).
- since a (data) segment is sometimes called *file*, the correct plan for the above query is often denoted by `fscan(Student)`.

Several assumptions must hold:

- the `Student` relation is not fragmented, it is stored in a single segment,
- the name of this segment is the same as the relation name, and
- no tuples from other relations are stored in this segment.

Until otherwise stated, we assume that these assumptions hold.
Instead of `fscan(Student)`, we could then simply use `Student` to denote leaf nodes in a query execution plan.

Attributes/Variables and their Binding

select *
from Student

can be expressed as *Student*[*s*] instead of *Student*.
Result type: set of tuples with a single attribute *s*.
s is assumed to bind a pointer

- to the physical record in the buffer holding the current tuple *or*
- a pointer to the slot pointing to the record holding the current tuple

Building Block

- scan
- a leaf of a query execution plan

Leaf can be complex.
But: Plan generator does not try to reorder within building blocks
Nonetheless:

- building block organized around a single database item

If more than a single database item is involved: *access path*

Scan and Attribute Access

Strictly speaking, the plan

$$\sigma_{age>30}(Student[s])$$

is incorrect (age is not bound!)
We have a choice:

- implicit attribute access
- make attribute accesses explicit

Scan and Attribute Access (2)

Explicit attribute access:

$$\sigma_{s.age>30}(Student[s])$$

Advantage: makes attribute access costs explicit

Scan and Attribute Access (3)

Consider:

$$\sigma_{s.age>30 \wedge s.age<40}(Student[s])$$

Problem: accesses age twice

Scan and Attribute Access (4)

Map operator:

$$\chi_{a_1:e_1,\ldots,a_n:e_n}(e) := \{t \circ [a_1 : c_1, \ldots, a_n : c_n] | t \in e, c_i = e_i(t) \; \forall \; (1 \leq i \leq n)\}$$

Loading Attributes

The above problem can now be solved by

$$\sigma_{age>30 \land age<40}(\chi_{age:s.age}(Student[s])).$$

In general, it is beneficial to load attributes as late as possible. The latest point at which all attributes must be read from the page is typically just before a pipeline breaker.

Loading Attributes (2)

select name
from Student
where age $>$ 30

The plan

$$\Pi_n(\chi_{n:s.name}(\sigma_{a>30}(\chi_{a:s.age}(Student[s]))))$$

is better than

$$\Pi_n(\sigma a > 30(\chi_{n:s.name,a:s.age}(Student[s])))$$

Loading Attributes (3)

Alternative to this selective successive attribute access:

- scan has list of attributes to be projected (accessed, copied)
- predicate is applied before processing the projection list

Loading Attributes (4)

predicate evaluable on disk representation is called *SARGable* (search argument)

- boolean expression in simple predicates of the form $A\theta c$

If a predicate can be used for an index lookup: index SARGable
Other predicates: residual predicates

Loading Attributes (5)

$R[v; p]$ equivalent to $\sigma_p(R[v])$ but cheaper to evaluate

Remark

- if p is conjunct, order by $(f_i - 1)/c_i$

Example:

$$Student[s; age > 30, name\ like\ `\%m\%']$$

Loading Attributes and Pipeline Breakers

- attribute access not only for scans
- likewise all operators that materialize to disk
- most pipeline breakers
- projection and selection should always be integrated into pipeline breakers
- not that important for pipelining operators
- attribute access must happen before breaking the pipeline

Exception:

- RID join/semijoin techniques

Physical Operator - Selection

- consumes a tuple stream
- checks predicate on each tuple
- produces matching tuples

Characteristics:

- pipelining operator
- consumes no memory, causes no IO

Physical Operator - Nested Loop Join

- consumes two tuple streams
- for each tuple from one stream (trad: the left) consumes the whole other stream
- checks predicate on each pair
- produces matching tuples

Characteristics:

- pipelining operator
- consumes no memory, causes no IO (at least not directly)

Physical Operator - Blockwise Nested Loop Join

- consumes two tuple streams
- reads one stream (left) blockwise into memory, consumes the whole other stream for each block
- checks predicate on each pair of tuples
- produces matching tuples

Characteristics:

- pipeline breaker on the left stream
- consumes memory for the blocks, causes no IO (unusual for a pipeline breaker)

Variants (with hashing etc.) behave basically the same

Physical Operator - Sort Merge Join

We only consider the case that the input is already sorted (see *Sort*) and $1 : n$ or $1 : 1$.

- consumes two tuple streams
- skips uniformly through both streams
- checks predicate on each pair (implicitly)
- produces matching tuples

Characteristics:

- pipelining operator
- consumes no memory, causes no IO

Physical Operator - Grace Hash Join

- consumes two tuple streams
- reads one stream and splits it into partitions on disk
- the same of the other stream
- joins the partitions, produces matching tuples

Characteristics:

- full pipeline breaker
- consumes memory for one partition, writes/reads whole data at least once

IO behavior can be predicted relatively easily

Physical Operator - Hybrid Hash Join

- consumes two tuple streams
- reads one stream and splits it into partitions on disk. Tries to keep some partitions in memory
- reads the other stream, also splits it into partitions on disk, but already joins with partitions still in memory
- joins partitions on disk, produces matching tuples

Characteristics:

- (typically) full pipeline breaker. Might keep the pipeline for the second stream
- consumes memory for partitioning (size variable), might write/reads whole data

Behavior difficult to predict, might cause no IO, might write everything

Physical Operator - Sort

- consumes one input stream
- creates sorted runs, spools runs to disk, merges the runs
- produces sorted output stream

Characteristics:

- pipeline breaker
- consumes memory for one run, reads/write data log *n* times

Exact behavior depends on implementation, e.g. HeapSort might produce one run, while QuickSort produces fixed number of runs

Physical Operator - Sort Based Group By

We assume that the input is already sorted

- consumes one input stream
- aggregates the input directly
- produces an output tuple whenever the group by attribute changes

Characteristics:

- pipeline breaker (nearly pipelining, though)
- consumes memory for one tuple, causes no IO

Sometimes interleaved with sort (early aggregation)

Physical Operator - Hash Bases Group By

- consumes one input stream
- reads the stream, splits into partitions, writes partitions to disk (if needed)
- aggregates partitions, produces output tuples

Characteristics:

- pipeline breaker
- consumes memory for buffering (variable), might read/write the whole data
- two possibilities, similar to Grace Hash vs. Hybrid Hash

Variants with early aggregation etc.

Physical Operators - Others

Only mainstream operators included, some are missing:

- projection usually implicit
- duplicate elimination is a special kind of aggregation
- dependent join (nested loop, can be done somewhat differently)
- outer join/semi join/anti join etc. roughly similar to normal joins
- specialized operators for query languages: staircase join, twig join etc.
- their characteristics have to be known to the query optimizer

Temporal Relations

The query optimizer might introduce temporal relations:

- a "relations" just for the query
- allows for reusing intermediate results
- related: temporary views
- more efficient nested loop join
- materializes a subquery

Creating a temporary relation is an expensive operation therefore

- should be decided by the query optimizer
- but often done as rewrite
- typically breaks optimization in parts

Temporal Relations (2)

select e.name, d.name
from Emp e, Dept d
where e.age > 30 **and** e.age < 40 **and** e.dno = d.dno

can be evaluated by

$$Dept[d] \bowtie^{nl}_{e.dno=d.dno} \sigma_{e.age>30 \wedge e.age<40}(Emp[d]).$$

Better:

$$Dept[d] \bowtie^{nl}_{e.dno=d.dno} temp(\sigma_{e.age>30 \wedge e.age<40}(Emp[d])).$$

Or:

1. $R_{tmp} = \sigma_{e.age>30 \wedge e.age<40}(Emp[d]);$
2. $Dept[d] \bowtie^{nl}_{e.dno=d.dno} R_{tmp}[e]$

Table Functions

A *table function* is a function that returns a relation.
Example query:

select *
from TABLE(Primes(1,100)) as p

Translation:

$$Primes(1, 100)[p]$$

Looks the same as regular scan, but is of course computed differently.

Table Functions (2)

Special birthdays of Anton:

select *
from Friends f,
TABLE(Primes(
CURRENT_YEAR, EXTRACT(YEAR FROM f.birthday) + 100)) as p
where f.name = ‘Anton’

Note: The result of the table function depends on our friend Anton.
Translation: uses d-join

Table Functions (3)

Definition d-join:

$$R \rtimes\!\!\!\!\ltimes S = \{r \circ s | r \in R, s \in S(t)\}.$$

Translation of the above query:

$$\chi_{\text{b}:XTRY(f.birthday)+100}(\sigma_{f.name=''Anton''}(Friends[f])) \rtimes\!\!\!\!\ltimes Primes(c, b)[p]$$

where we assume that some global entity c holds the value of `CURRENT_YEAR`.

Table Functions (4)

The same for all friends:

select *
from Friends f,
TABLE(Primes(
CURRENT_YEAR, EXTRACT(YEAR FROM f.birthday) + 100)) as p

Better:

select *
from Friends f,
TABLE(Primes(
CURRENT_YEAR, (**select** max(birthday) **from** Friends) + 100)) as p
where p.prime $\leq$ EXTRACT(YEAR FROM f.birthday) + 100

At the algebraic level: this optimization requires some knowledge

Indices

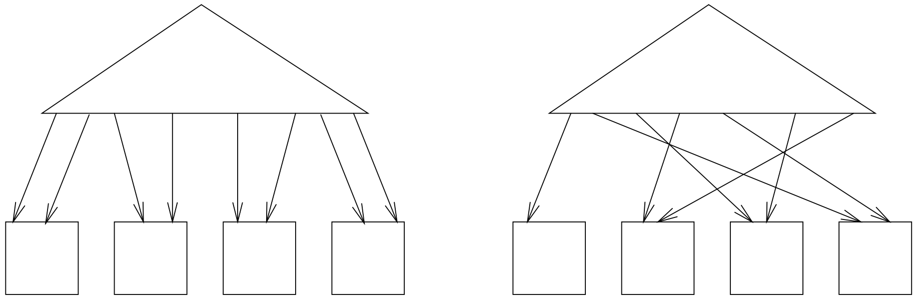
We consider B-Trees only

- key attributes: $a_1, \ldots, a_n$
- data attributes: $d_1, \ldots, d_m$
- Often: one special data attribute holding the TID of a tuple

Some notions:

- simple/complex key
- unique/non-unique index
- index-only relation (no TIDs available!)
- clustered/non-clustered index

Clustered vs. Non-Clustered B-Tree



- clustering is not always possible (or even desireable)

Single Index Access Path - Point Query

Exact match query:

select name
from Emp
where eno = 1077

Translation:

$$\Pi_{name}(\chi_{e:*x.tid,name:e.name}(Emp_{eno}[x; eno = 1077]))$$

Alternative translation using d-join:

$$\Pi_{name}(Emp_{eno}[x; eno = 1077] \ltimes \chi_{e:*.tid,name:e.name}(\Box))$$

(x: holds ptr to index entry; *: dereference TID, $\Box$ is a singleton scan)

Single Index Access Path - Range Query

Range query:

select name
from Emp
where age $\geq$ 25 **and** age $\leq$ 35

Translation:

$$\Pi_{name}(\chi_{e:*x.tid,name:e.name}(Emp_{age}[x; 25 \leq age; age \leq 35]))$$

(Start and Stop condition)

Single Index Access Path - Sequential I/O

Turning random I/O into sequential I/O:

$$\Pi_{name}(\chi_{e:*tid,name:e.name}(sort_{x.tid}(Emp_{age}[x; 25 \leq age; age \leq 35; tid])))$$

Note: explicit projection the TID attribute of the index within the index scan.

Single Index Access Path - Sorted Output

Query demanding ordered output:

select	name, age
from	Emp
where	age $\geq$ 25 **and** age $\leq$ 35
order by	age

Translation:

$$\Pi_{name,age}(\chi_{e:*x.tid,name:e.name}(Emp_{age}[x; 25 \leq age; age \leq 35]))$$

Note: output of index scan ordered on its key attributes
This order can be exploited in many ways: e.g.: subsequent merge join

Single Index Access Path - Sorted Output (2)

Turning random I/O into sequential I/O requires resort:

$\Pi_{name,age}(sort_{age}(\chi_{e:*tid,name:e.name}(sort_{tid}(Emp_{age}[x; 25 \leq age; age \leq 35; tid]$

Possible speedup of sort by dense numbering:

$$\begin{array}{l} \Pi_{name,age}(\\ \quad sort_{rank}(\\ \quad\quad \chi_{e:*tid,name:e.name}(\\ \quad\quad\quad sort_{tid}(\\ \quad\quad\quad\quad \chi_{rank:counter++}(\\ \quad\quad\quad\quad\quad Emp_{age}[x; 25 \leq age; age \leq 35; tid]))))) \end{array}$$

Single Index Access Path - Other Predicates

Some predicates not index sargable but still useful as residual predicates:

select name
from Emp
where age $\geq$ 25 **and** age $\leq$ 35 **and** age $\neq$ 30

Translation:

$$\Pi_{name}(\chi_{e:*x.tid,name:e.name}(Emp_{age}[x; 25 \leq age; age \leq 35; age \neq 30]))$$

Single Index Access Path - Other Predicates (2)

Non-inclusive bounds:

select name
from Emp
where age $>$ 25 **and** age $<$ 35

If supported by index:

$$\Pi_{name}(\chi_{e:*x.tid,name:e.name}(Emp_{age}[x; 25 < age; age < 35]))$$

If unsupported:

$$\Pi_{name}(\chi_{e:*x.tid,name:e.name}(Emp_{age}[x; 25 \leq age; age \leq 35; age \neq 25, age \neq 35$$

Especially for predicates on strings this might be expensive.

Single Index Access Path - Ranges

Start and stop conditions are optional:

select name
from Emp
where age $\geq$ 60

or

select name
from Emp
where age $\leq$ 20

Single Index Access Path - No Range

Full index scan also useful:

select count(*)
from Emp

Also works for sum/avg.
(notion: index only query)

Single Index Access Path - No Range (2)

Min/max even more efficient:

select min/max(salary)
from Emp

Single Index Access Path - No Range (3)

```
select  name
from    Emp
where   salary = (select  max(salary)
                  from    Emp)
```

Alternatives: one or two descents into the index.

Single Index Access Path - No Range (4)

Full index scan:

select	salary
from	Emp
order by	salary

Translation:

$$Emp_{salary}$$

Single Index Access Path - String Ranges

Predicate on string attribute:

select name, salary
from Emp
where name $\geq$ 'Maaa'

Start condition: $'Maaa' \leq name$

select name, salary
from Emp
where name **like** 'M%'

Start condition: $'M' \leq name$

Single Index Access Path

- an *access path* is a plan fragment with building blocks concerning a single database items.
- hence, every building block is an access path.
- above plans mostly touch two database items: a relation and an index on some attribute of that relation.
- if we say that an index concerns the relation that it indexes, such a fragment is an access path.
- for relational systems, the most general case of an access path uses several indices to retrieve the tuples of a single relation.
- we will see examples of these more complex access paths in the following section.
- a query that can be answered solely by accessing indexes is called an *index only query*.

Single Index Access Path - Complex Predicates

Query with `IN`:

select name
from Emp
where age in {28, 29, 31, 32}

Take min/max value for start/stop key plus one of the following as the residual predicate:

- $age = 28 \vee age = 29 \vee age = 31 \vee age = 32$
- $age \neq 30$

Single Index Access Path - Complex Predicates (2)

A case for the d-join:

select name
from Emp
where salary in {1111, 11111, 111111}

With $Sal = \{[s: 1111], [s: 11111], [s: 111111]\}$:

$$Sal[S] \ltimes \chi_{e:*tid, name:e.name}(Emp_{salary}[x; salary = S.s; tid])$$

- gap skipping/zig-zag skipping

Single Index Access Path - Compound Keys

In general an index can have a complex key comprising of key attributes $k_1, \ldots, k_n$ and data attributes $d_1, \ldots, d_m$.
Besides a full index scan, the index can be descended to directly search for the desired tuple(s):
If the search predicate is of the form

$$k_1 = c_1 \land k_2 = c_2 \land \ldots \land k_j = c_j$$

for some constants c_i and some $j <= n$, we can generate the start and stop condition

$$k_1 = c_1 \land \ldots \land k_j = c_j.$$

Single Index Access Path - Compound Keys

With ranges things become more complex and highly dependent on the implementation of the facilities of the B-Tree:

$$k_1 = c_1 \land k_2 \geq c_2 \land k_3 = c_3$$

Obviously, we can generate the start condition $k_1 = c_1 \land k_2 \geq c_2$ and the stop condition $k_1 = c_1$.
Here, we neglected the condition on k_3 which becomes a residual predicate.
However, with some care we can extend the start condition to
$k_1 = c_1 \land k_2 \geq c_2 \land k_3 = c_3$:
we only have to keep $k_3 = c_3$ as a residual predicate since for k_2 values larger than c_2 values different from c_3 can occur for k_3.

Single Index Access Path - Compound Keys (2)

If closed ranges are specified for a prefix of the key attributes as in

$$a_1 \leq k_1 \leq b_1 \wedge \ldots \wedge a_j \leq k_j \leq b_j$$

we can generate the start key $k_1 = a_1 \wedge \ldots \wedge k_j = a_j$, the stop key $k_1 = b_1 \wedge \ldots \wedge k_j = b_j$, and

$$a_2 \leq k_2 \leq b_2 \wedge \ldots \wedge a_j \leq k_j \leq b_j$$

as the residual predicate.
If for some search key attribute k_j the lower bound a_j is not specified, the start condition can not contain k_j and any k_{j+i}.
If for some search key attribute k_j the upper bound b_j is not specified, the stop condition can not contain k_j and any k_{j+i}.

Single Index Access Path - Improvements

Two further enhancements of the B-Tree functionality possibly allow for alternative start/stop conditions:

- The B-Tree implemenation allows to specify the order (ascending or descending) for each key attribute individually.
- The B-Tree implementation implements forward and backward scans

Single Index Access Path - Improvements (2)

Consider search predicate:

```
haircolor = 'blond' and height between 180 and 190
```

and index on

```
sex, haircolor, height
```

There are only the two values `male` and `female` available for `sex`.
Rewrite:

```
(sex = 'm' and haircolor = 'blond' and height
between 180 and 190) or (sex = 'f' and haircolor =
'blond' and height between 180 and 190)
```

Improvement: determine rewrite at query execution time in conjunction with gap skipping.

Multi Index Access Path - Example

Query:

select $*$
from Camera
where megapixel > 5 **and** distortion < 0.05
and noise < 0.01
zoomMin < 35 **and** zoomMax > 105

Indexes on all attributes

Multi Index Access Path - Example (2)

Translation:

$$
((((\\
Camera_{megapixel}[c; megapixel > 5; tid]\\
\cap\\
Camera_{distortion}[c; distortion < 0.05; tid])\\
\cap\\
Camera_{noise}[c; noise < 0.01; tid])\\
\cap\\
Camera_{zoomMin}[c; zoomMin < 35; tid])\\
\cap\\
Camera_{zoomMax}[c; zoomMax > 105; tid])
$$

Then dereference

- Notion: index and-ing/and merge (bitmap index)

Multi Index Access Path - Combining

Questions:

- In which order do we intersect the TID sets resulting from the index scans?
- Do we really apply all indexes before dereferencing the TIDs?

The answer to the latter question is clearly *"no"*, if the next index scan is more expensive than accessing the records in the current TID list.
It can be shown that the indexes in the cascade of intersections are ordered on increasing $(f_i - 1)/c_i$ terms where f_i is the selectivity of the index and c_i its access cost.
Further, we can stop as soon as accessing the original tuples in the base relation becomes cheaper than intersecting with another index and subsequently accessing the base relation.

Multi Index Access Path - Combining (2)

Index-oring (or merge):

select *
from Emp
where yearsOfEmployment $\geq$ 30
or age $\geq$ 65

Translation:

$Emp_{yearsOfEmployment}[c; yearsOfEmployment \geq 30; tid] \cup Emp_{age}[c; age \geq 65; ti$

Attention: duplicates
Optimal translation of complex boolean expressions? Factorization?

Multi Index Access Path - Combining (3)

Index differencing:

select *
from Emp
where yearsOfEmployment $\neq$ 10
and age $\geq$ 65

Translation:

$Emp_{age}[c; age \geq 65; tid] \setminus Emp_{yearsOfEmployment}[c; yearsOfEmployment = 10; ti$

Multi Index Access Path - Combining (3)

Non-restrictive index sargable predicates (more than half of the index has to be read):

select *
from Emp
where yearsOfEmployment $\leq$ 5
and age $\leq$ 60

Then

$Emp_{yearsOfEmployment}[c; yearsOfEmployment \leq 5; tid] \setminus Emp_{age}[c; age > 60; tid$

could be more efficient than

$Emp_{yearsOfEmployment}[c; yearsOfEmployment \leq 5; tid] \cap Emp_{age}[c; age \leq 60; tid$

Indices and Join

1. speed up joins by index exploitation
2. make join a general index processing operation

(intersection is similar to join (for sets))

Indices and Join (2)

Turn map

$$\chi_{e:*tid,name:e.name}(Emp_{salary}[x; 25 \leq age \leq 35; tid])$$

into d-join

$$Emp_{salary}[x; 25 \leq age \leq 35; tid] \ltimes \chi_{e:*tid,name:e.name}(\Box)$$

or even join

$$Emp_{salary}[x; 25 \leq age \leq 35] \bowtie_{x.tid=e.tid} Emp[e]$$

Variants: sorting at different places (by plan generator)

- pro: flexibility
- contra: large search space

Indices and Join (3)

Query:

select name,age
from Person
where name like 'R%' and age between 40 and 50

Translation:

$$\begin{aligned}&\Pi_{name,age}(\\ &\quad Emp_{age}[a; 40 \leq age \leq 50; TIDa, age]\\ &\quad\quad \bowtie_{TIDa=TIDn}\\ &\quad Emp_{name}[n; name \geq' R'; name <' S'; TIDn, name])\end{aligned}$$

Indices and Join (4)

The query

select $*$
from $Emp\ e,\ Dept\ d$
where $e.name =$ ‘$Maier$’ $and\ e.dno = d.dno$

can be directly translated to

$$\sigma_{e.name=''Maier''}(Emp[e]) \bowtie_{e.dno=d.dno} Dept[d]$$

Indices and Join (5)

If there are indexes on `Emp.name` and `Dept.dno`, we can replace $\sigma_{e.name=''Maier''}(Emp[e])$ by an index scan as we have seen previously:

$$\chi_{e:*x.tid}(Emp_{name}[x; name ='' \ Maier''])$$

Indices and Join (6)

With a d-join:

$$Emp_{name}[x; name ='' \ Maier''] \ltimes \chi_{e:*x.tid}(\Box)$$

Abbreviate $Emp_{name}[x; name ='' \ Maier'']$ by E_i
Abbreviate $\chi_{e:*x.tid}(\Box)$ by E_a.

Indices and Join (7)

Use index on `Dept.dno`:

$$E_i \ltimes E_a \ltimes Dept_{dno}[y; y.dno = dno]$$

Dereference TIDs (*index nested loop join*):

$$E_i \ltimes E_a \ltimes Dept_{dno}[y; y.dno = dno; dtid : y.tid] \ltimes \chi_{u:*dtid}(\square)$$

Abbreviate $Dept_{dno}[y; y.dno = dno; dtid : y.tid]$ by D_i
Abbreviate $\chi_{u:*dtid}(\square)$ by D_a
Fully abbreviated, the expression then becomes

$$E_i \ltimes E_a \ltimes D_i \ltimes D_a$$

Indices and Join - Performance Improvements

Optimizations: sorting the *outer* of a d-join is useful under several circumstances since it may

- turn random I/O into sequential I/O and/or
- avoid reading the same page twice.

In our example expression:

Indices and Join - Performance Improvements (2)

- We can sort the result of expression E_i on `TID` in order to turn random I/O into sequential I/O, if there are many employees named "Maier".
- We can sort the result of the expression $E_i \bowtie E_a$ on `dno` for two reasons:
  - If there are duplicates for `dno`, i.e. there are many employees named "Maier" in each department, then this guarantees that no index page (of the index `Dept.dno`) has to be read more than once.
  - If additionally `Dept.dno` is a clustered index or `Dept` is an index-only table contained in `Dept.dno` then large parts of the random I/O can be turned into sequential I/O.
  - If the result of the inner is materialized (see below), then only one result needs to be stored. Note that sorting is not necessary but grouping would suffice to avoid duplicate work.
- We can sort the result of the expression $E_i \bowtie E_a \bowtie D_i$ on `dtid` for the same reasons as mentioned above for sorting the result of E_i on `TID`.

Indices and Join - Temping the Inner

Typically, many employees will work in a single department and possibly several of them are called "Maier".
For everyone of them, we can be sure that there exists at most one department.
Let us assume that referential intregrity has been specified.
Then there exists exactly one department for every employee.
We have to find a way to rewrite the expression

$$E_i \bowtie E_a \bowtie Dept_{dno}[y; y.dno = dno; dtid : y.rid]$$

such that the mapping $dno \longrightarrow dtid$ is explicitly materialized (or, as one could also say, *cached*).

Indices and Join - Temping the Inner (2)

Use χ^{mat}:

$$E_i \bowtie E_a \bowtie \chi^{mat}_{tid:(Dept_{dno}[y;y.dno=dno]).tid}(\Box)$$

Indices and Join - Temping the Inner (3)

If we further assume that the outer $(E_i \bowtie E_a)$ is sorted on `dno`, then it suffices to remember only the TID for the latest `dno`.
We define the map operator $\chi^{mat,1}$ to do exactly this.
A more efficient plan could thus be

$$sort_{dno}(E_i \bowtie E_a) \bowtie \chi^{mat,1}_{dtid:(Dept_{dno}[y;y.dno=dno]).tid}(\square)$$

where, strictly speaking, sorting is not necessary: grouping would suffice.

Indices and Join - Temping the Inner (4)

Consider: $e_1 \ltimes e_2$
The free variables used in e_2 must be a subset of the variables (attributes) produced by e_1, i.e. $\mathcal{F}(e_2) \subseteq \mathcal{A}(e_1)$.
Even if e_1 does not contain duplicates, the projection of e_1 on $\mathcal{F}(e_2)$ may contain duplicates.
If so, materialization could pay off.
However, in general, for every binding of the variables $\mathcal{F}(e_2)$, the expression e_2 may produce several tuples.
This means that using χ^{mat} is not sufficient.

Indices and Join - Temping the Inner (5)

The query

select *
from Emp e, Wine w
where e.yearOfBirth = w.year

has the usual suspects as plans.
Assume we have only wines from a few years.
Then, it might make sense to consider the following alternative:

$$Wine[w] \ltimes \sigma_{e.yearOfBirth=w.year}(Emp[e])$$

Problem: scan `Emp` once for each `Wine` tuple
Duplicates in `Wine.year`: scan `Emp` only once per `Wine.year` value

Indices and Join - Temping the Inner (6)

The memox operator performs caching:

$$Wine[w] \ltimes memox(\sigma_{e.yearOfBirth=w.year}(Emp[e]))$$

Sorting still beneficial:

$$sort_{w.year}(Wine[w]) \ltimes memox^{1}(\sigma_{e.yearOfBirth=w.year}(Emp[e]))$$

Indices and Join - Temping the Inner (7)

Things can become even more efficient if there is an index on `Emp.yearOfBirth`:

$$sort_{w.year}(Wine[w]) \\ \rtimes memox^1(Emp_{yearOfBirth}[x; x.yearOfBirth = w.year] \rtimes \chi_{e:*(x.tid)}(\square))$$

Indices and Join - Temping the Inner (8)

Indexes on `Emp.yearOfBirth` and `Wine.year`.
Join result of index scans.
Since the index scan produces its output ordered on the key attributes, a simple merge join suffices (and we are back at the latter):

$$Emp_{yearOfBirth}[x] \bowtie^{merge}_{x.yearOfBirth=y.year} Wine_{year}[y]$$

Remarks on Access Path Generation

Side-ways information passing
Consider $R \bowtie_{R.a=S.b} S$

- min/max for restriction on other join argument
- full projection on join attributes (leads to semi-join)
- bitmap representation of the projection

From Cardinalities to Costs

Given: number of TIDs to dereference
Question: disk access costs?
Two step solution:

1. estimate number of pages to be accessed
2. estimate costs for accessing these pages

Parameters

Given a set of k TIDs after an index access:

How many pages do we have to access to dereference them?

Let R be the relation for which we have to retrieve the tuples. Then we use the following abbreviations

N	$\lvert R\rvert$	number of tuples in the relation R
m	$\lvert\lvert R\rvert\rvert$	number of pages on which tuples of R are stored
B	N/m	number of tuples per page
k		number of (distinct) TIDs for which tuples have to be retrieved

We assume that the tuples are uniformely distributed among the m pages. Then, each page stores $B = N/m$ tuples. B is called *blocking factor*.

Special Cases

Let us consider some border cases.
If $k > N - N/m$ or $m = 1$, then all pages are accessed.
If $k = 1$ then exactly one page is accessed.

General Case

The answer to the general question will be expressed in terms of

- *buckets* (pages in the above case) and
- *items* contained therein (tuples in the above case).

Later on, we will also use extents, cylinders, or tracks as buckets and tracks or sectors/blocks as items.

Different Settings

Outline:

1. random/direct access
   - 1.1 items uniformly distributed among the buckets
     - 1.1.1 request k distinct items
     - 1.1.2 request k non-distinct items
   - 1.2 non-uniform distribution of items among buckets
2. sequential access

Always: uniform access probability

Direct, Uniform, Distinct

Additional assumption:
The probability that we request a set with k items is

$$\frac{1}{\binom{N}{k}}$$

for all of the

$$\binom{N}{k}$$

possibilities to select a k-set.
[Every k-set is accessed with the same probability.]

Direct, Uniform, Distinct (2)

Theorem (Waters/Yao)

Consider m buckets with n items each. Then there is a total of $N = nm$ items. If we randomly select k distinct items from all items then the number of qualifying buckets is

$$\overline{\mathcal{Y}}_n^{N,m}(k) = m * \mathcal{Y}_n^N(k) \tag{17}$$

where $\mathcal{Y}_n^N(k)$ is the probability that a bucket contains at least one item.

Direct, Uniform, Distinct (3)

Theorem (Waters/Yao (cont.))

The probability is

$$\mathcal{Y}_n^N(k) = \begin{cases} [1-p] & k \leq N-n \\ 1 & k > N-n \end{cases}$$

where p is the probability that a bucket contains none of the k items. The following alternative expressions can be used to calculate p:

$$\begin{aligned} p &= \frac{\binom{N-n}{k}}{\binom{N}{k}} && (18) \\ &= \prod_{i=0}^{k-1} \frac{N-n-i}{N-i} && (19) \\ &= \prod_{i=0}^{n-1} \frac{N-k-i}{N-i} && (20) \end{aligned}$$

Direct, Uniform, Distinct (4)

Proof (1): The total number of possibilities to pick the k items from all N items is

$$\binom{N}{k}$$

The number of possibilities to pick k items from all items not contained in a fixed single bucket is

$$\binom{N-n}{k}$$

Hence, the probability p that a bucket does not qualify is

$$p = \binom{N-n}{k} / \binom{N}{k}$$

Direct, Uniform, Distinct (5)

Proof (2):

$$
\begin{aligned}
p &= \frac{\binom{N-n}{k}}{\binom{N}{k}} \\
&= \frac{(N-n)! \;\; k!(N-k)!}{k!((N-n)-k)! \;\; N!} \\
&= \prod_{i=0}^{k-1} \frac{N-n-i}{N-i}
\end{aligned}
$$

Direct, Uniform, Distinct (6)

Proof(3):

$$\begin{aligned}
p &= \frac{\binom{N-n}{k}}{\binom{N}{k}} \\
&= \frac{(N-n)! \;\; k!(N-k)!}{k!((N-n)-k)! \;\; N!} \\
&= \frac{(N-n)! \;\; (N-k)!}{N! \;\; ((N-k)-n)!} \\
&= \prod_{i=0}^{n-1} \frac{N-k-i}{N-i}
\end{aligned}$$

Direct, Uniform, Distinct (7)

Implementation remark:

> *The fraction $m = N/n$ may not be an integer.*
> *For these cases, it is advisable to have a Gamma-function based implementation of binomial coeffcients at hand*

Evaluation of Yao's formula is expensive. Approximations are more efficient to calculate.

Direct, Uniform, Distinct (8)

Special cases:

If	then $\mathcal{Y}_m^N(k) =$
$n = 1$	k/N
$n = N$	1
$k = 0$	0
$k = 1$	B/N
$k = N$	1

Direct, Uniform, Distinct (9)

Let N items be distributed over N buckets such that every bucket contains exactly one item.
Further let us be interested in a subset of m buckets ($1 \leq m \leq N$).
If we pick k items then the number of buckets within the subset of size m that qualify is

$$m\mathcal{Y}_1^N(k) = m\frac{k}{N} \tag{21}$$

qualify.

Direct, Uniform, Distinct (10)

Proof:

$$
\begin{aligned}
\mathcal{Y}_1^N(k) &= (1 - \frac{\binom{N-1}{k}}{\binom{N}{k}}) \\
&= (1 - \frac{\frac{(N-1)!}{k!((N-1)-k)!}}{\frac{N!}{k!(N-k)!}}) \\
&= (1 - \frac{(N-1)!k!(N-k)!}{N!k!((N-1)-k)!}) \\
&= (1 - \frac{N-k}{N}) \\
&= (\frac{N}{N} - \frac{N-k}{N}) \\
&= \frac{N-N+k}{N} \\
&= \frac{k}{N}
\end{aligned}
$$

Direct, Uniform, Distinct (11)

Approximation of Yao's formula (1):

$$p \approx (1 - k/N)^n$$

[Waters]

Direct, Uniform, Distinct (12)

Approximation of Yao's formula (2):
$\overline{\mathcal{Y}}_n^{N,m}(k)$ can be approximated by:

$$\begin{aligned} m * [\quad & (1-(1-1/m)^k)+ \\ & (1/(m^2 b) * k(k-1)/2 * (1-1/m)^{k-1})+ \\ & (1.5/(m^3 b^4) * k(k-1)(2k-1)/6 * (1-1/m)^{k-1}) \quad] \end{aligned}$$

[Whang, Wiederhold, Sagalowicz]

Direct, Uniform, Distinct (13)

Approximation of Yao's formula (3):

$$\overline{\mathcal{Y}}_n^{N,m}(k) \approx \begin{cases} k & \text{if } \; k < \frac{m}{2} \\ \frac{k+m}{3} & \text{if } \; \frac{m}{2} \leq k < 2m \\ m & \text{if } \; 2m \leq k \end{cases}$$

[Bernstein, Goodman, Wong, Reeve, Rothnie]

Direct, Uniform, Distinct (14)

Upper and lower bounds for p:

$$\begin{aligned} p_{\text{lower}} &= (1 - \frac{k}{N - \frac{n-1}{2}})^n \\ p_{\text{upper}} &= ((1 - \frac{k}{N}) * (1 - \frac{k}{N - n + 1}))^{n/2} \end{aligned}$$

for $n = N/m$.
Dihr and Saharia claim that the maximal difference resulting from the use of the lower and the upper bound to compute the number of page accesses is 0.224—far less than a single page access.

Direct, Uniform, Non-Distinct

Lemma

Let S be a set with $|S| = N$ elements. Then, the number of multisets with cardinality k containing only elements from S is

$$\binom{N+k-1}{k}$$

Proof: For a prove we just note that there is a bijection between the k-multisets and the k-subsets of a $N+k-1$-set.
We can go from a multiset to a set by f with

$$f(\{x_1 \leq \ldots \leq x_k\}) = \{x_1 + 0 < x_2 + 1 < \ldots < x_k + (k-1)\}$$

and from a set to a multiset via g with

$$g(\{x_1 < \ldots < x_k\}) = \{x_1 - 0 \leq x_2 - 1 \leq \ldots \leq x_k - (k-1)\}$$

Direct, Uniform, Non-Distinct (2)

Theorem (Cheung)

Consider m buckets with n items each. Then there is a total of $N = nm$ items. If we randomly select k not necessarily distinct items from all items, then the number of qualifying buckets is

$$\overline{Cheung}_n^{N,m}(k) = m * Cheung_n^N(k) \tag{22}$$

where

$$Cheung_n^N(k) = [1 - \tilde{p}] \tag{23}$$

Direct, Uniform, Non-Distinct (3)

Theorem (Cheung (cont.))

with the following equivalent expressions for $\tilde{p}$:

$$\tilde{p} = \frac{\binom{N-n+k-1}{k}}{\binom{N+k-1}{k}} \tag{24}$$

$$= \prod_{i=0}^{k-1} \frac{N-n+i}{N+i} \tag{25}$$

$$= \prod_{i=0}^{n-1} \frac{N-1-i}{N-1+k-i} \tag{26}$$

Direct, Uniform, Non-Distinct (4)

Proof(1):
Eq. 24 follows from the observation that the probability that some bucket does not contain any of the k possibly duplicate items is $\frac{\binom{N-n+k-1}{k}}{\binom{N+k-1}{k}}$.

Direct, Uniform, Non-Distinct (5)

Proof(2):
Eq. 25 follows from

$$\begin{aligned}
\tilde{p} &= \frac{\binom{N-n+k-1}{k}}{\binom{N+k-1}{k}} \\
&= \frac{(N-n+k-1)! \;\; k!((N+k-1)-k)!}{k!((N-n+k-1)-k)! \;\; (N+k-1)!} \\
&= \frac{(N-n-1+k)! \;\; (N-1)!}{(N-n-1)! \;\; (N-1+k)!} \\
&= \prod_{i=0}^{k-1} \frac{N-n+i}{N+i}
\end{aligned}$$

Direct, Uniform, Non-Distinct (6)

Proof(3):
Eq. 26 follows from

$$
\begin{aligned}
\tilde{p} &= \frac{\binom{N-n+k-1}{k}}{\binom{N+k-1}{k}} \\
&= \frac{(N-n+k-1)! \;\; k!((N+k-1)-k)!}{k!((N-n+k-1)-k)! \;\; (N+k-1)!} \\
&= \frac{(N+k-1-n)! \;\; (N-1)!}{(N+k-1)! \;\; (N-1-n)!} \\
&= \prod_{i=0}^{n-1} \frac{N-n+i}{N+k-n+i} \\
&= \prod_{i=0}^{n-1} \frac{N-1-i}{N-1+k-i}
\end{aligned}
$$

Direct, Uniform, Non-Distinct (7)

Approximation for $\tilde{p}$:

$$(1 - n/N)^k$$

[Cardenas]

Direct, Uniform, Non-Distinct (8)

Estimate for the number of distinct values in a bag:

Corollary

Let S be a k-multiset containing elements from an N-set T. Then the number of distinct items contained in S is

$$\mathcal{D}(N,k) = \frac{Nk}{N+k-1} \tag{27}$$

if the elements in T occur with the same probability in S.

Direct, Uniform, Non-Distinct (9)

Model switching:

$$\overline{\mathcal{Y}}_n^{N,m}(Distinct(N,k)) \approx \overline{\text{Cheung}}_n^{N,m}(k)$$

[for $n \geq 5$]

Direct, Non-Uniform, Distinct

So far:

1. every page contains the same number of records, and
2. every record is accessed with the same probability.

Now:

Model the distribution of items to buckets by m numbers n_i (for $1 \leq i \leq m$) if there are m buckets.
Each n_i equals the number of records in some bucket i.

Direct, Non-Uniform, Distinct (2)

The following theorem is a simple application of Yao's formula:

Theorem (Yao/Waters/Christodoulakis)

Assume a set of m buckets. Each bucket contains $n_j > 0$ items $(1 \leq j \leq m)$. The total number of items is $N = \sum_{j=1}^{m} n_j$. If we lookup k distinct items, then the probability that bucket j qualifies is

$$\mathcal{W}_{n_j}^{N}(k,j) = [1 - \frac{\binom{N-n_j}{k}}{\binom{N}{k}}] \quad (= \mathcal{Y}_{n_j}^{N}(k)) \tag{28}$$

and the expected number of qualifying buckets is

$$\overline{\mathcal{W}}_{n_j}^{N,m}(k) := \sum_{j=1}^{m} \mathcal{W}_{n_j}^{N}(k,j) \tag{29}$$

Direct, Non-Uniform, Distinct (3)

The product formulation in Eq. 20 of Theorem 2 results in a more efficient computation:

Corollary

If we lookup k distinct items, then the expected number of qualifying buckets is

$$\overline{\mathcal{W}}_{n_j}^{N,m}(k) = \sum_{j=1}^{m}(1 - p_j) \tag{30}$$

with

$$p_j = \begin{cases} \prod_{i=0}^{n_j-1} \frac{N-k-i}{N-i} & k \leq n_j \\ 0 & N - n_j < k \leq N \end{cases} \tag{31}$$

Direct, Non-Uniform, Distinct (4)

If we compute the p_j after we have sorted the n_j in ascending order, we can use the fact that

$$p_{j+1} = p_j * \prod_{i=n_j}^{n_{j+1}-1} \frac{N-k-i}{N-i}.$$

Direct, Non-Uniform, Distinct (5)

Many buckets: statistics too big. Better: Histograms

Corollary

For $1 \leq i \leq L$ *let there be* l_i *buckets containing* n_i *items. Then, the total number of buckets is* $m = \sum_{i=1}^{L} l_i$ *and the total number of items in all buckets is* $N = \sum_{i=1}^{L} l_i n_i$*. For* k *randomly selected items the number of qualifying buckets is*

$$\overline{\mathcal{W}}_{n_j}^{N,m}(k) = \sum_{i=1}^{L} l_i \mathcal{Y}_{n_j}^{N}(k) \tag{32}$$

Direct, Non-Uniform, Distinct (6)

Distribution function. The probability that $x \leq n_j$ items in a bucket j qualify, can be calculated as follows:

- The number of possibilities to select x items in bucket n_j is
$$\binom{n_j}{x}$$
- The number of possibilites to draw the remaining $k - x$ items from the other buckets is
$$\binom{N - n_j}{k - x}$$
- The total number of possibilities to distributed k items over the buckets is
$$\binom{N}{k}$$

This shows the following:

Direct, Non-Uniform, Distinct (7)

Theorem

Assume a set of m buckets. Each bucket contains $n_j > 0$ items $(1 \leq j \leq m)$. The total number of items is $N = \sum_{j=1}^{m} n_j$. If we lookup k distinct items, then the probability that x items in bucket j qualify is

$$\mathcal{X}_{n_j}^{N}(k,x) = \frac{\binom{n_j}{x} \quad \binom{N-n_j}{k-x}}{\binom{N}{k}} \tag{33}$$

Further, the expected number of qualifying items in bucket j is

$$\overline{\mathcal{X}}_{n_j}^{N,m}(k) = \sum_{x=0}^{\min(k,n_j)} x\mathcal{X}_{n_j}^{N}(k,x) \tag{34}$$

In standard statistics books the probability distribution $\mathcal{X}_{n_j}^{N}(k,x)$ is called *hypergeometric distribution*.

Direct, Non-Uniform, Distinct (8)

Let us consider the case where all n_j are equal to n. Then, we can calculate the average number of qualifying items in a bucket. With $y := \min(k, n)$ we have

$$\begin{aligned}
\overline{\mathcal{X}}^{N,m}_{n_j}(k) &= \sum_{x=0}^{\min(k,n)} x\mathcal{X}^{N}_{n}(k,x) \\
&= \sum_{x=1}^{\min(k,n)} x\mathcal{X}^{N}_{n}(k,x) \\
&= \frac{1}{\binom{N}{k}} \sum_{x=1}^{y} x \binom{n}{x} \binom{N-n}{k-x}
\end{aligned}$$

Direct, Non-Uniform, Distinct (9)

$$\begin{aligned}
\overline{\mathcal{X}}_{n_j}^{N,m}(k) &= \frac{1}{\binom{N}{k}} \sum_{x=1}^{y} x \binom{n}{x} \binom{N-n}{k-x} \\
&= \frac{1}{\binom{N}{k}} \sum_{x=1}^{y} \binom{x}{1} \binom{n}{x} \binom{N-n}{k-x} \\
&= \frac{1}{\binom{N}{k}} \sum_{x=1}^{y} \binom{n}{1} \binom{n-1}{x-1} \binom{N-n}{k-x} \\
&= \frac{\binom{n}{1}}{\binom{N}{k}} \sum_{x=0}^{y-1} \binom{n-1}{0+x} \binom{N-n}{(k-1)-x} \\
&= \ldots
\end{aligned}$$

(cont.)

Direct, Non-Uniform, Distinct (10)

$$\begin{aligned}
\overline{\mathcal{X}}_{n_j}^{N,m}(k) &= \ldots \\
&= \frac{\binom{n}{1}}{\binom{N}{k}} \binom{n-1+N-n}{0+k-1} \\
&= \frac{\binom{n}{1}}{\binom{N}{k}} \binom{N-1}{k-1} \\
&= n\frac{k}{N} = \frac{k}{m}
\end{aligned}$$

Direct, Non-Uniform, Distinct (11)

Let us consider the even more special case where every bucket contains a single item. That is, $N = m$ and $n_i = 1$. The probability that a bucket contains a qualifying item reduces to

$$\begin{aligned}\mathcal{X}_1^N(k,x) &= \frac{\binom{1}{x}\quad\binom{N-1}{k-1}}{\binom{N}{k}} \\ &= \frac{\binom{N-1}{k-1}}{\binom{N}{k}} \\ &= \frac{k}{N} \quad (= \frac{k}{m})\end{aligned}$$

Since x can then only be zero or one, the average number of qualifying items a bucket contains is also $\frac{k}{N}$.

Sequential: Vector of Bits

When estimating seek costs, we need to calculate the probability distribution for the distance between two subsequent qualifying cylinders. We model the situation as a bitvector of length B with b bits set to one. Then, B corresponds to the number of cylinders and a one indicates that a cylinder qualifies.
[Later: Vector of Buckets]

Sequential: Vector of Bits (2)

Theorem

Assume a bitvector of length B. Within it b ones are uniformly distributed. The remaining $B - b$ bits are zero. Then, the probability distribution of the number j of zeros

1. *between two consecutive ones,*
2. *before the first one, and*
3. *after the last one*

is given by

$$\mathcal{B}_b^B(j) = \frac{\binom{B-j-1}{b-1}}{\binom{B}{b}} \tag{35}$$

Sequential: Vector of Bits (3)

Proof:
To see why the formula holds, consider the total number of bitvectors having a one in position i followed by j zeros followed by a one.
This number is

$$\binom{B-j-2}{b-2}$$

We can chose $B-j-1$ positions for i.
The total number of bitvectors is

$$\binom{B}{b}$$

and each bitvector has $b-1$ sequences of the form that a one is followed by a sequence of zeros is followed by a one.

Sequential: Vector of Bits (4)

Hence,

$$\begin{aligned}\mathcal{B}_b^B(j) &= \frac{(B-j-1)\binom{B-j-2}{b-2}}{(b-1)\binom{B}{b}} \\ &= \frac{\binom{B-j-1}{b-1}}{\binom{B}{b}}\end{aligned}$$

Part (1) follows.
To prove (2), we count the number of bitvectors that start with j zeros before the first one.
There are $B-j-1$ positions left for the remaining $b-1$ ones.
Hence, the number of these bitvectors is $\binom{B-j-1}{b-1}$ and part (2) follows.
Part (3) follows by symmetry.

Sequential: Vector of Bits (5)

We can derive a less expensive way to calculate formula for $\mathcal{B}_b^B(j)$ as follows.
For $j = 0$, we have $\mathcal{B}_b^B(0) = \frac{b}{B}$.
If $j > 0$, then

$$\begin{aligned}
\mathcal{B}_b^B(j) &= \frac{\binom{B-j-1}{b-1}}{\binom{B}{b}} \\
&= \frac{\frac{(B-j-1)!}{(b-1)!((B-j-1)-(b-1))!}}{\frac{B!}{b!(B-b)!}} \\
&= \frac{(B-j-1)! \;\; b!(B-b)!}{(b-1)!((B-j-1)-(b-1))! \;\; B!}
\end{aligned}$$

Sequential: Vector of Bits (6)

$$\begin{aligned}
\mathcal{B}_b^B(j) &= \frac{(B-j-1)! \;\; b!(B-b)!}{(b-1)!((B-j-1)-(b-1))! \;\; B!} \\
&= b\frac{(B-j-1)! \;\; (B-b)!}{((B-j-1)-(b-1))! \;\; B!} \\
&= b\frac{(B-j-1)! \;\; (B-b)!}{(B-j-b)! \;\; B!} \\
&= \frac{b}{B-j}\,\frac{(B-j)! \;\; (B-b)!}{(B-b-j)! \;\; B!} \\
&= \frac{b}{B-j}\prod_{i=0}^{j-1}(1-\frac{b}{B-i})
\end{aligned}$$

This formula is useful when $\mathcal{B}_b^B(j)$ occurs in sums over j.

Sequential: Vector of Bits (7)

Corollary

Using the terminology of Theorem 8, the expected value for the number of zeros

1. *before the first one,*
2. *between two successive ones, and*
3. *after the last one*

is

$$\overline{\mathcal{B}}_b^B = \sum_{j=0}^{B-b} j\mathcal{B}_b^B(j) = \frac{B-b}{b+1} \tag{36}$$

Sequential: Vector of Bits (8)

Proof:

$$\begin{aligned}
\sum_{j=0}^{B-b} j\binom{B-j-1}{b-1} &= \sum_{j=0}^{B-b}(B-(B-j))\binom{B-j-1}{b-1} \\
&= B\sum_{j=0}^{B-b}\binom{B-j-1}{b-1} - \sum_{j=0}^{B-b}(B-j)\binom{B-j-1}{b-1} \\
&= B\sum_{j=0}^{B-b}\binom{b-1+j}{b-1} - b\sum_{j=0}^{B-b}\binom{B-j}{b} \\
&= B\sum_{j=0}^{B-b}\binom{b-1+j}{j} - b\sum_{j=0}^{B-b}\binom{b+j}{b}
\end{aligned}$$

Sequential: Vector of Bits (9)

$$
\begin{aligned}
\sum_{j=0}^{B-b} j\binom{B-j-1}{b-1} &= B\sum_{j=0}^{B-b}\binom{b-1+j}{j} - b\sum_{j=0}^{B-b}\binom{b+j}{b} \\
&= B\binom{(b-1)+(B-b)+1}{(b-1)+1} - b\binom{b+(B-b)+1}{b+1} \\
&= B\binom{B}{b} - b\binom{B+1}{b+1} \\
&= (B - b\frac{B+1}{b+1})\binom{B}{b}
\end{aligned}
$$

With

$$
\begin{aligned}
B - b\frac{B+1}{b+1} &= \frac{B(b+1)-(Bb+b)}{b+1} \\
&= \frac{B-b}{b+1}
\end{aligned}
$$

the claim follows

Sequential: Vector of Bits (10)

Corollary

Using the terminology of Theorem 8, the expected total number of bits from the first bit to the last one, both included, is

$$\overline{\mathcal{B}}_{tot}(B, b) = \frac{Bb + b}{b + 1} \tag{37}$$

Sequential: Vector of Bits (11)

Proof:
We subtract from B the average expected number of zeros between the last one and the last bit:

$$\begin{aligned} B - \frac{B-b}{b+1} &= \frac{B(b+1)}{b+1} - \frac{B-b}{b+1} \\ &= \frac{Bb+B-B+b}{b+1} \\ &= \frac{Bb+b}{b+1} \end{aligned}$$

Sequential: Vector of Bits (12)

Corollary

Using the terminology of Theorem 8, the number of bits from the first one and the last one, both included, is

$$\overline{\mathcal{B}}_{\textit{1-span}}(B,b)=\frac{Bb-B+2b}{b+1} \tag{38}$$

Sequential: Vector of Bits (13)

Proof (alternative 1):
Subtract from B the number of zeros at the beginning and the end:

$$\begin{aligned}
\overline{\mathcal{B}}_{\text{1-span}}(B, b) &= B - 2\frac{B-b}{b+1} \\
&= \frac{Bb + B - 2B + 2b}{b+1} \\
&= \frac{Bb - B + 2b}{b+1}
\end{aligned}$$

Sequential: Vector of Bits (14)

Proof (alternative 2):
Add the number of zeros between the first and the last one and the number of ones:

$$\begin{aligned}
\overline{\mathcal{B}}_{\text{1-span}}(B, b) &= (b-1)\overline{\mathcal{B}}_b^B + b \\
&= (b-1)\frac{B-b}{b+1} + \frac{b(b+1}{b+1} \\
&= \frac{Bb - b^2 - B + b + b^2 + b}{b+1} \\
&= \frac{Bb - B + 2b}{b+1}
\end{aligned}$$

Sequential: Applications for Bitvector Model

- If we look up one record in an array of B records and we search sequentially, how many array entries do we have to examine on average if the search is successful?
- Let a file consist of B consecutive cylinders. We search for k different keys all of which occur in the file. These k keys are distributed over b different cylinders. Of course, we can stop as soon as we have found the last key. What is the expected total distance the disk head has to travel if it is placed on the first cylinder of the file at the beginning of the search?
- Assume we have an array consisting of B different entries. We sequentially go through all entries of the array until we have found all the records for b different keys. We assume that the B entries in the array and the b keys are sorted. Further all b keys occur in the array. On the average, how many comparisons do we need to find all keys?

Sequential: Vector of Buckets

Theorem (Yao)

Consider a sequence of m buckets. For $1 \leq i \leq m$, let n_i be the number of items in a bucket i. Then there is a total of $N = \sum_{i=1}^{m} n_i$ items. Let $t_i = \sum_{l=0}^{i} n_i$ be the number of items in the first i buckets. If the buckets are searched sequentially, then the probability that j buckets that have to be examined until k distinct items have been found is

$$\mathcal{C}_{n_i}^{N,m}(k,j) = \frac{\binom{t_j}{k} - \binom{t_{j-1}}{k}}{\binom{N}{k}} \tag{39}$$

Thus, the expected number of buckets that need to be examined in order to retrieve k distinct items is

$$\overline{\mathcal{C}}_{n_i}^{N,m}(k) = \sum_{j=1}^{m} j\mathcal{C}_{n_i}^{N,m}(k,j) = m - \frac{\sum_{j=1}^{m} \binom{t_{j-1}}{k}}{\binom{N}{k}} \tag{40}$$

Sequential: Vector of Buckets (2)

The following theorem is very useful for deriving estimates for average sequential accesses under different models [Especially: the above theorem follows].

Theorem (Lang/Driscoll/Jou)

Consider a sequence of N items. For a batched search of k items, the expected number of accessed items is

$$A(N,k) = N - \sum_{i=1}^{N-1} Prob[Y \leq i] \tag{41}$$

where Y is a random variable for the last item in the sequence that occurs among the k items searched.

Disk Drive Costs for N Uniform Accesses

The goal of this section is to derive estimates for the costs (time) for retrieving N cache-missed sectors of a segment S from disk.
We assume that the N sectors are read in their physical order on disk.
This can be enforced by the DBMS, by the operating system's disk scheduling policy (SCAN policy), or by the disk drive controler.

Disk Drive Costs for N Uniform Accesses (2)

Remembering the description of disk drives, the total costs can be described as

$$C_{disk} = C_{cmd} + C_{seek} + C_{settle} + C_{rot} + C_{headswitch} \tag{42}$$

For brevity, we omitted the parameter N and the parameters describing the segment and the disk drive on which the segment resides.
Subsequently, we devote a (sometimes tiny) section to each summand.
Before that, we have to calculate the number of qualifying cylinders, tracks, and sectors.
These numbers will be used later on.

Number of Qualifying Cylinder

- N sectors are to be retrieved.
- We want to find the number of cylinders qualifying in extent i.
- S_{sec} denotes the total number of sectors our segment contains.
- Assume: The N sectors we want to retrieve are uniformly distributed among the S_{sec} sectors of the segment.
- $S_{\text{cpe}}(i) = L_i - F_i + 1$ denotes the number of cylinders of extent i.

Disk Costs: Number of Qualifying Cylinder

The number of qualifying cylinders in exent i is:

$S_{cpe}(i)$ ** (1 - Prob(a cylinder does not qualify))*

The probability that a cylinder does not qualify can be computed by deviding the total number of possibilities to chose the N sectors from sectors outside the cylinder by the total number of possibilities to chose N sectors from all S_{sec} sectors of the segment.
Hence, the number of qualifying cylinders in the considered extent is:

$$Q_c(i) = S_{\text{cpe}}(i)\mathcal{Y}^{S_{\text{sec}}}_{D_{\text{zspc}}(i)}(N) = S_{\text{cpe}}(i)(1 - \frac{\binom{S_{\text{sec}} - D_{\text{zspc}}(i)}{N}}{\binom{S_{\text{sec}}}{N}}) \qquad (43)$$

Number of Qualifying Tracks

Let us also calculate the number of qualifying tracks in a partion i.
It can be calculated by

$$S_{\text{cpe}}(i) D_{\text{tpc}}(1 - \text{Prob}(\texttt{a track does not qualify}))$$

The probability that a track does not qualify can be computed by dividing the number of ways to pick N sectors from sectors not belonging to a track divided by the number of possible ways to pick N sectors from all sectors:

$$Q_t(i) = S_{\text{cpe}}(i) D_{\text{tpc}} \mathcal{Y}^{S_{\text{sec}}}_{D_{\text{zspt}}(i)}(N) = S_{\text{cpe}}(i) D_{\text{tpc}}(1 - \frac{\binom{S_{\text{sec}} - D_{\text{zspt}}(i)}{N}}{\binom{S_{\text{sec}}}{N}}) \tag{44}$$

Number of Qualifying Tracks (2)

Just for fun, we calculate the number of qualifying sectors of an extent in zone i. It can be approximated by

$$Q_s(i) = S_{\text{cpe}}(i) D_{\text{zspc}}(i) \frac{N}{S_{\text{sec}}} \tag{45}$$

Since all $S_{\text{cpe}}(i)$ cylinders are in the same zone, they have the same number of sectors per track and we could also use Waters/Yao to approximate the number of qualifying cylinders by

$$Q_c(i) = \overline{\mathcal{Y}}^{S_{\text{cpe}}(i) D_{\text{zspc}}(S_{\text{zone}}(i)), S_{\text{cpe}}(i)}_{D_{\text{zspc}}(S_{\text{zone}}(i))}(Q_s(i)) \tag{46}$$

If $Q_s(i)$ is not too small (e.g. > 4).

Command Costs

The command costs C_{cmd} are easy to compute. Every read of a sector requires the execution of a command. Hence

$$C_{cmd} = ND_{\text{cmd}}$$

estimates the total command costs.

Seek Costs

- often the dominant part of the costs
- we look at several alternatives from less to more precise models

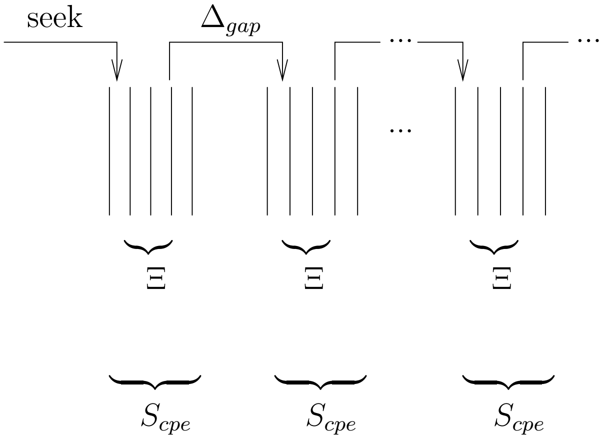
Seek Costs - Upper Bound

The first cylinder we have to visit requires a random seek with cost D_{seekavg}. (Truely upper bound: $D_{\text{fseek}}(D_{\text{cyl}} - 1)$)
After that, we have to visit the remaining $Q_c(i) - 1$ qualifying cylinders.
The segment spans a total of $S_{\text{clast}}(S_{\text{ext}}) - S_{\text{cfirst}}(1) + 1$ cylinders.
Let us assume that the first qualifying cylinder is the first cylinder and the last qualifying cylinder is the last cylinder of the segment.
Then, applying Qyang's Theorem 1 gives us the upper bound

$$C_{seek}(i) \leq (Q_c(i) - 1) D_{\text{fseek}}(\frac{S_{\text{clast}}(S_{\text{ext}}) - S_{\text{cfirst}}(1) + 1}{Q_c(i) - 1})$$

after we have found the first qualifying cylinder.

Seek Costs - Illustration



Seek Costs - Steps

Steps:

1. Estimate for $C_{seekgap}$
2. Estimates for $C_{seekext}(i)$

Seek Costs - Interextent Costs

The average seek cost for reaching the first qualifying cylinder is D_{seekavg}. How far within the first extent are we now? We use Corollary 4 to derive that the number of non-qualifying cylinders preceding the first qualifying one in some extent i is

$$\overline{\mathcal{B}}^{S_{\text{cpe}}(i)}_{Q_c(i)} = \frac{S_{\text{cpe}}(i) - Q_c(i)}{Q_c(i) + 1}.$$

The same is found for the number of non-qualifying cylinders following the last qualifying cylinder. Hence, for every gap between the last and the first qualifying cylinder of two extents i and $i + 1$, the disk arm has to travel the distance

$$\Delta_{\text{gap}}(i) := \overline{\mathcal{B}}^{S_{\text{cpe}}(i)}_{Q_c(i)} + S_{\text{cfirst}}(i+1) - S_{\text{clast}}(i) - 1 + \overline{\mathcal{B}}^{S_{\text{cpe}}(i+1)}_{Q_c(i+1)}$$

Using this, we get

$$C_{seekgap} = D_{\text{seekavg}} + \sum_{i=1}^{S_{\text{ext}}-1} D_{\text{fseek}}(\Delta_{\text{gap}}(i))$$

Seek Costs - Intraextent Costs (2)

Let us turn to $C_{seekext}(i)$. We first need the number of cylinders between the first and the last qualifying cylinder, both included, in extent i. It can be calculated using Corollary 6:

$$\Xi_{\text{ext}}(i) = \overline{\mathcal{B}}_{\text{1-span}}(S_{\text{cpe}}(i), Q_c(i))$$

Hence, $\Xi(i)$ is the minimal span of an extent that contains all qualifying cylinders.

Seek Costs - Intraextent Costs

Using $\Xi(i)$ and Qyang's Theorem 1, we can derive an upper bound for $C_{seekext}(i)$:

$$C_{seekext}(i) \leq (Q_c(i) - 1) D_{\text{fseek}}(\frac{\Xi(i)}{Q_c(i) - 1}) \tag{47}$$

Alternatively, we could formulate this as

$$C_{seekext}(i) \leq (Q_c(i) - 1) D_{\text{fseek}}(\overline{\mathcal{B}}^{S_{\text{cpe}}(i)}_{Q_c(i)}) \tag{48}$$

by applying Corollary 4.

Seek Costs - Intraextent Costs (2)

A seemingly more precise estimate for the expected seek cost within the qualifying cylinders of an extent is derived by using Theorem 8:

$$C_{seekext}(i) = Q_c(i) \sum_{j=0}^{S_{\mathrm{cpe}}(i)-Q_c(i)} D_{\mathrm{fseek}}(j+1)\mathcal{B}_{Q_c(i)}^{S_{\mathrm{cpe}}(i)}(j) \qquad (49)$$

Settle Costs

The average settle cost is easy to calculate. For every qualifying cylinder, one head settlement takes place:

$$C_{settle}(i) = Q_c(i) D_{\text{rdsettle}} \tag{50}$$

Rotational Delay

Let us turn to the rotational delay.
For some given track in zone i,
we want to read the $Q_t(i)$ qualifying sectors contained in it.
On average, we would expect that the read head is ready to start reading in the middle of some sector of a track.
If so, we have to wait for $\frac{1}{2}D_{\text{zscan}}(S_{\text{zone}}(i))$ before the first whole sector ocurs under the read head.
However, due to track and cylinder skew, this event does not occur after a head switch or a cylinder switch.
Instead of being overly precise here, we igore this half sector pass by time and assume we are always at the beginning of a sector.
This is also justified by the fact that we model the head switch time explicitly.

Rotational Delay (2)

Assume that the head is ready to read at the beginning of some sector of some track.
Then, in front of us is a — cyclic, which does not matter — bitvector of qualifying and non-qualifying sectors.
We can use Corollary 5 to estimate the total number of qualifying and non-qualifying sectors that have to pass under the head until all qualifying sectors have been seen.
The total rotational delay for the tracks of zone i is

$$C_{rot}(i) = Q_t(i) \;\; D_{\text{zscan}}(S_{\text{zone}}(i)) \;\; \overline{\mathcal{B}}_{\text{tot}}(D_{\text{zspt}}(S_{\text{zone}}(i)), Q_{\text{spt}}(i))$$

where $Q_{\text{spt}}(i) = \overline{\mathcal{W}}_1^{S_{\text{sec}}, D_{\text{zspt}}(S_{\text{zone}}(i))}(N) = D_{\text{zspt}}(S_{\text{zone}}(i)) \frac{N}{S_{\text{sec}}}$ is the expected number of qualifying sectors per track in extent i. In case $Q_{\text{spt}}(i) < 1$, we set $Q_{\text{spt}}(i) := 1$.

Rotational Delay (3)

A more precise model is derived as follows.
We sum up for all j the product of (1) the probability that j sectors in a track qualify and (2) the average number of sectors that have to be read if j sectors qualify.
This gives us the number of sectors that have to pass the head in order to read all qualifying sectors.
We only need to multiply this number by the time to scan a single sector and the number of qualifying tracks.
We can estimate (1) using Theorem 7. For (2) we again use Corollary 5.

$$\begin{aligned} C_{rot}(i) &= S_{\text{cpe}}(i) \;\; D_{\text{tpc}} \;\; D_{\text{zscan}}(S_{\text{zone}}(i)) \\ &\quad * \sum_{j=1}^{\min(N, D_{\text{zspt}}(S_{\text{zone}}(i)))} \mathcal{X}^{S_{\text{sec}}}_{D_{\text{zspt}}(S_{\text{zone}}(i))}(N, j) \;\; \overline{\mathcal{B}}_{\text{tot}}(D_{\text{zspt}}(S_{\text{zone}}(i)), j) \end{aligned}$$

Rotational Delay (4)

Yet another approach:
Split the total rotational delay into two components:

1. $C_{rotpass}(i)$ measures the time needed to skip unqualifying sectors
2. $C_{rotread}(i)$ that for scanning the qualifying sectors

Then

$$C_{rot} = \sum_{i=1}^{S_{\text{ext}}} C_{rotpass}(i) + C_{rotread}(i)$$

where the total transfer cost of the qualifying sectors can be estimated as

$$C_{rotread}(i) = Q_s(i) \;\; D_{\text{zscan}}(S_{\text{zone}}(i))$$

Rotational Delay (5)

Let us treat the first component ($C_{rotpass}(i)$).
Assume that j sectors of a track in extent i qualify.
The expected position on a track where the head is ready to read is the middle between two qualifying sectors.
Since the expected number of sectors between two qualifying sectors is $D_{\text{zspt}}(S_{\text{zone}}(i))/j$, the expected number of sectors scanned before the first qualifying sector comes under the head is

$$\frac{D_{\text{zspt}}(S_{\text{zone}}(i))}{2j}$$

Rotational Delay (6)

The expected positions of j qualifying sectors on the same track is such that the number non-qualifying sectors between two successively qualifying sectors is the same.
Hence, after having read a qualifying sector $\frac{D_{\text{zspt}}(S_{\text{zone}}(i))}{j}$ unqualifying sectors must be passed until the next qualifying sector shows up.
The total number of unqualifying sectors to be passed if j sectors qualify in a track of zone i is

$$N_s(j,i) = \frac{D_{\text{zspt}}(S_{\text{zone}}(i))}{2j} + (j-1)\frac{D_{\text{zspt}}(S_{\text{zone}}(i)) - j}{j}$$

Rotational Delay (7)

Using again Theorem 7, the expected rotational delay for the unqualifying sectors then is

$$\begin{aligned} C_{rotpass}(i) &= S_{\text{cpe}}(i) \quad D_{\text{tpc}} \quad D_{\text{zscan}}(S_{\text{zone}}(i)) \\ &* \sum_{j=1}^{\min(N, D_{\text{zspt}}(S_{\text{zone}}(i)))} \mathcal{X}^{S_{\text{sec}}}_{D_{\text{zspt}}(S_{\text{zone}}(i))}(N, j) N_s(j, i) \end{aligned}$$

Head Switch Costs

The average head switch cost is equal to the average number of head switches that occur times the average head switch cost.
The average number of head switch is equal to the number of tracks that qualify minus the number of cylinders that qualify since a head switch does not occur for the first track of each cylinder.
Summarizing

$$C_{headswitch} = \sum_{i=1}^{S_{\text{ext}}} (Q_t(i) - Q_c(i)) \quad D_{\text{hdswitch}} \tag{51}$$

where Q_t is the average number of tracks qualifying in an extent.

Discussion

We neglected many problems in our disk access model:

- partially filled cylinders,
- pages larger than a block,
- disk drive's cache,
- remapping of bad blocks,
- non-uniformly distributed accesses,
- clusteredness,
- and so on.

Whereas the first two items are easy to fix, the rest is not so easy.

Selectivity Estimations

- previous slides assume that we "know" how many tuples qualify
- but this has to be estimated somehow
- similar for join ordering algorithms etc.
- cardinalities (and thus selectivities) are fundamental for query optimization
- we will now look at deriving some estimations

Examples

SQL examples for typical selectivity problems:

- **select** *
  from rel r
  where r.a=10
- **select** *
  from rel r
  where r.b>2
- **select** *
  from rel1 r1,rel2 r2
  where r1.a=r2.b

The different problems require different approaches.

Heuristic Estimations

Some commonly used selectivity estimations:

predicate	selectivity	requirement
$A = c$	$1/\|D(A)\|$	if index on A
	$1/10$	otherwise
$A > c$	$(\max(A) - c)/(\max(A) - \min(A))$	if index on A, interpol.
	$1/3$	otherwise
$A_1 = A_2$	$1/\max(\|D(A_1)\|, \|(D(A_2)\|)$	if index on A_1 and A_2
	$1/\|D(A_1)\|$	if index on A_1 only
	$1/\|D(A_2)\|$	if index on A_2 only
	$1/10$	otherwise

Note: Without further statistics, $|D(A)|$ is typically only known (easily estimated) if A is a key or there is an index on A.

Using Histograms

- selectivity can be calculated easily by looking at the real data
- not feasible, therefore look at aggregated data
- histograms partition the data values into buckets

A histogram $H_A : B \to \mathbb{N}$ over a relation R partitions the domain of the aggregated attribute A into disjoint buckets B, such that

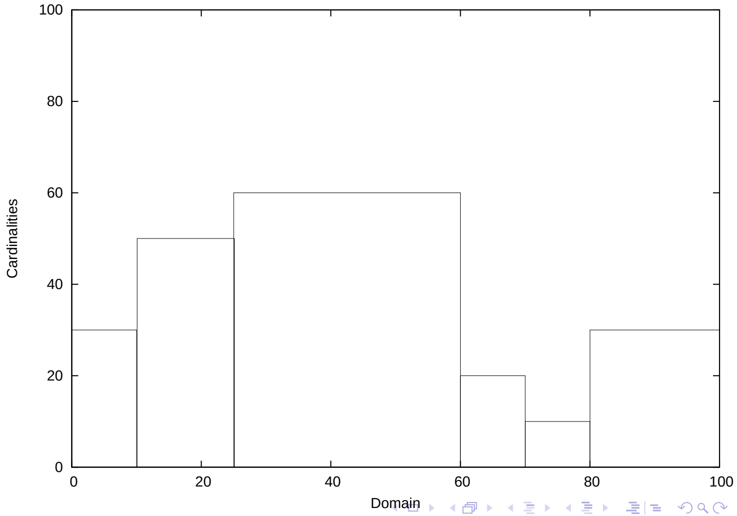
$$H_A(b) = |\{r | r \in R \wedge R.A \in b\}|$$

and thus $\sum_{b \in B} H_A(b) = |R|$.

Choosing B is very important, as we will see on the next slides.

Using Histograms (2)

A rough histogram might look like this:



Using Histograms (3)

Given a histogram, we can approximate the selectivities as follows:

$$A = c \qquad \frac{\sum_{b \in B: c \in b} H_A(b)}{\sum_{b \in B} H_A(b)}$$

$$A > c \qquad \frac{\sum_{b \in B: c \in b} \frac{\max(b) - c}{\max(b) - \min(b)} H_A(b) + \sum_{b \in B: \min(b) > c} H_A(b)}{\sum_{b \in B} H_A(b)}$$

$$A_1 = A_2 \qquad \frac{\sum_{b_1 \in B_1, b_2 \in B_2, b' = b_1 \cap b_2 : b' \neq \emptyset} \frac{\max(b') - \min(b')}{\max(b_1) - \min(b_1)} H_{A_1}(b_1) \frac{\max(b') - \min(b')}{\max(b_2) - \min(b_2)} H_{A_2}(b_2)}{\sum_{b_1 \in B_1} H_{A_1}(b_1) \sum_{b_2 \in B_2} H_{A_2}(b_2)}$$

Using Histograms - Remarks

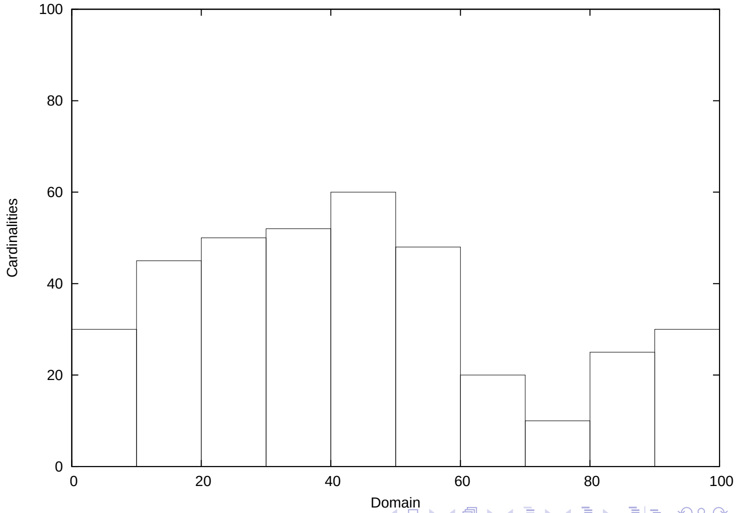
- estimations on previous slide can be improved
- in particular, the $A = c$ case is only a rough approximation
- requires more information
- if we interpret the histogram as a density function, $P(A = c) = 0$!
- a reasonable upper bound, though
- the $A > c$ case is more sound
- $A_1 = A_2$ assumes independence etc.

Building Histograms

- the buckets chosen greatly affect the overall quality
- histogram does not discern items within one bucket
- therefore: try to put items into different buckets
- how to choose the buckets?
- typical constraint: histogram size. n buckets (fixed)
- for a given set of data items, find a good histogram with n buckets
- additional constraint: data distribution is unknown (real data)

Building Histograms - Equiwidth

Partitions the domain into buckets with a fixed width



Building Histograms - Equiwidth (2)

Advantages:

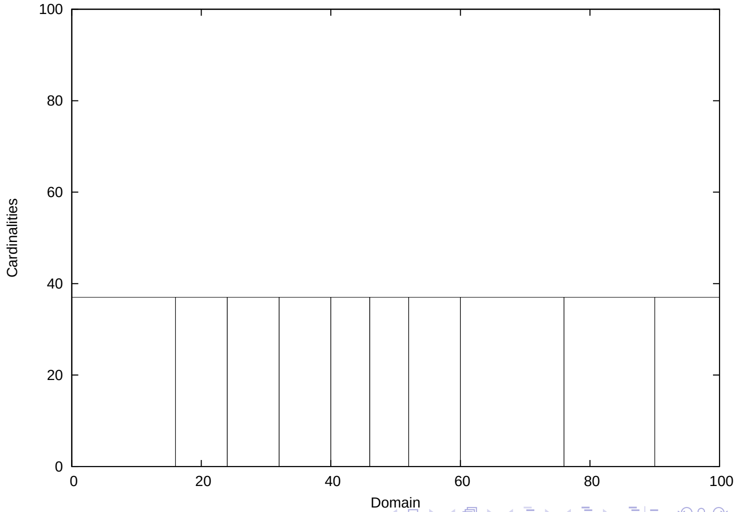
- easy to compute
- bucket boundaries can be computed (require no space)

Disadvantages:

- samples the domain uniformly
- does not handle skewed data well
- skew can lead to very uneven buckets
- greater estimation error in large buckets
- particular bad for zipf-like distributions

Building Histograms - Equidepth

Chooses the buckets to contain the same number of items



Building Histograms - Equidepth (2)

Advantages:

- adopts to data distribution
- reduces maximum error

Disadvantages:

- more involved (sort or similar)
- both boundaries and depth have to be stored (ties)

Very common histogram building technique

Building Histograms - Interpolation

- data is usually not completely random
- can we increase accuracy by interpolation?
- either within buckets (common) or instead of buckets (uncommon)
- histogram is a density function, not continuous, hard to interpolate
- use the equivalent distribution function instead
- very good for estimating $A > c$

Discussion

- estimations more complex in practice
- potentially different goals: maximum vs. average error
- histograms for derived values
- histogram convolution
- handling correlations
- multi-dimensional histograms
- cardinality estimators (sketches, MIPS etc.)