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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Abbreviated Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>capacity</td>
<td>180 GB</td>
<td>$D_{\text{cap}}$</td>
</tr>
<tr>
<td>average latency time</td>
<td>5 ms</td>
<td>$D_{\text{lat}}$</td>
</tr>
<tr>
<td>sustained read rate</td>
<td>100 MB/s</td>
<td>$D_{\text{srr}}$</td>
</tr>
<tr>
<td>sustained write rate</td>
<td>100 MB/s</td>
<td>$D_{\text{swr}}$</td>
</tr>
</tbody>
</table>

The time a disk needs to read and transfer $n$ bytes is then approximated by $D_{\text{lat}} + \frac{n}{D_{\text{srr}}}$. 
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: $5 \text{ ms} + 1 \text{ s}$
- 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
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

$$n \times (a + p/s) = a + d/s$$

$$n = (a + d/s)/(a + p/s)$$

$$= (as + d)/(as + p)$$

$a$ 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.
Accessing the Data

Disk Drive

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

- $D_{\text{cyl}}$: total number of cylinders
- $D_{\text{track}}$: total number of tracks
- $D_{\text{sec}}$: total number of sectors
- $D_{\text{tpc}}$: number of tracks per cylinder (= number of surfaces)
- $D_{\text{cmd}}$: command interpretation time
- $D_{\text{rot}}$: time for a full rotation
- $D_{\text{rdsettle}}$: time for settle for read
- $D_{\text{wrsettle}}$: time for settle for write
- $D_{\text{hdswitch}}$: time for head switch
Disk Drive: Parameters (2)

- $D_{\text{zone}}$: 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 $i$ ($= D_{\text{rot}}/D_{\text{zspt}}(i)$)
Disk Drive: Parameters (3)

- $D_{\text{seekavg}}$: average seek costs
- $D_{\text{clim}}$: parameter for seek cost function
- $D_{\text{ca}}$: parameter for seek cost function
- $D_{\text{cb}}$: parameter for seek cost function
- $D_{\text{cc}}$: parameter for seek cost function
- $D_{\text{cd}}$: parameter for seek cost function

The cost of a seek of $d$ cylinders, $D_{\text{fseek}}(d)$, is defined as:

$$D_{\text{fseek}}(d) = \begin{cases} 
D_{\text{ca}} + D_{\text{cb}} \sqrt{d} & \text{if } d \leq D_{\text{clim}} \\
D_{\text{cc}} + D_{\text{cd}} d & \text{if } d > D_{\text{clim}}
\end{cases}$$

The rotation cost for $s$ sectors of zone $i$, $D_{\text{frot}}(s, i)$, is:

$$D_{\text{frot}}(s, i) = sD_{\text{zscan}}(i)$$
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, and has to stop at $s - 1$ of them, then $sD_{fseek}(C/s)$ is an upper bound for the seek time.
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 accesses in a buffer with 2 pages:

<table>
<thead>
<tr>
<th>page no</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read page 0, place it in buffer</td>
</tr>
<tr>
<td>1</td>
<td>read page 1, place it in buffer</td>
</tr>
<tr>
<td>0</td>
<td>fix page 0 in buffer</td>
</tr>
<tr>
<td>2</td>
<td>swap out a page (e.g. 1), read 2, place it in buffer</td>
</tr>
<tr>
<td>0</td>
<td>fix page 0 in buffer</td>
</tr>
<tr>
<td>3</td>
<td>swap out a page, read 3, place it in buffer</td>
</tr>
</tbody>
</table>

- replacement strategy is important
- 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 placed 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:

- Partition contains 1 Segment
- Segment consists of N Pages
- Page stores N Records
- Relation fragmented 1 Fragment
- Fragment contains M Relations
- Relation mapped M Tuples
- Record represents N Tuples
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 \textit{scan}
Scan Example

Using a relation scan \texttt{rscan}, the query

\begin{verbatim}
select  *  
from    Student
\end{verbatim}

can be answered by \texttt{rscan(Student)}: Assumption:

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

Then \texttt{fscan(Student)} and \texttt{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.
  (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

- $S_{\text{ext}}$: 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 $i$ ($= S_{\text{clast}}(i) - S_{\text{cfirst}}(i) + 1$)
- $S_{\text{sec}}$: 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:

---

fixed-length | size | variable-length | size | variable-length | size | variable-length

| fixed-length | offset | offset | offset | variable-length | variable-length |

---

codes | data

---

fixed-length | variable-length | strings

length and offset encoding

encoding for dictionary-based compression
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 \texttt{rscan} operation is rarely supported.
- instead: scans on segments (files).
- since a (data) segment is sometimes called \textit{file}, the correct plan for the above query is often denoted by \texttt{fscan(Student)}.

Several assumptions must hold:

- the \texttt{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 \texttt{fscan(Student)}, we could then simply use \texttt{Student} to denote leaf nodes in a query execution plan.
Attributes/Variables and their Binding

\texttt{select * from Student}

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

- to the physical record in the buffer holding the current tuple \textit{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

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

is incorrect (age is not bound!)

We have a choice:

- implicit attribute access
- make attribute accesses explicit
Explicit attribute access:

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

Advantage: makes attribute access costs explicit
Scan and Attribute Access (3)

Consider:

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

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

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

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 \textit{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:

\[ \text{Student}[s; \text{age} > 30, \text{name} \text{ like} \ '%m%' ] \]
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 one

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.
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

\[ \text{Dept}[d] \bowtie^n_{e.dno=d.dno} \sigma_{e.age>30 \land e.age<40}(\text{Emp}[d]). \]

Better:

\[ \text{Dept}[d] \bowtie^n_{e.dno=d.dno} \text{temp}(\sigma_{e.age>30 \land e.age<40}(\text{Emp}[d])). \]

Or:

1. \( R_{tmp} = \sigma_{e.age>30 \land e.age<40}(\text{Emp}[d]) \);
2. \( \text{Dept}[d] \bowtie^n_{e.dno=d.dno} R_{tmp}[e] \)
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:

```sql
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:

\[ R \bowtie S = \{ r \circ s | r \in R, s \in S(t) \}. \]

Translation of the above query:

\[ \chi_{b:\text{XTRY}(f.birthday)+100}(\sigma_{f.name=""Anton""}(\text{Friends}[f])) \bowtie \text{Primes}(c, b)[p] \]

where we assume that some global entity c holds the value of \text{CURRENT\_YEAR}. 
Table Functions (4)

The same for all friends:

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

Better:

```sql
select * 
from Friends f,
    TABLE(Primes(CURRENT_YEAR, (select max(birthday) from Friends) + 100)) as p
where p.prime ≥ f.birthday
```

At the algebraic level: this optimization requires some knowledge