

COMP9311: DATABASE SYSTEMS

Term 1 2024

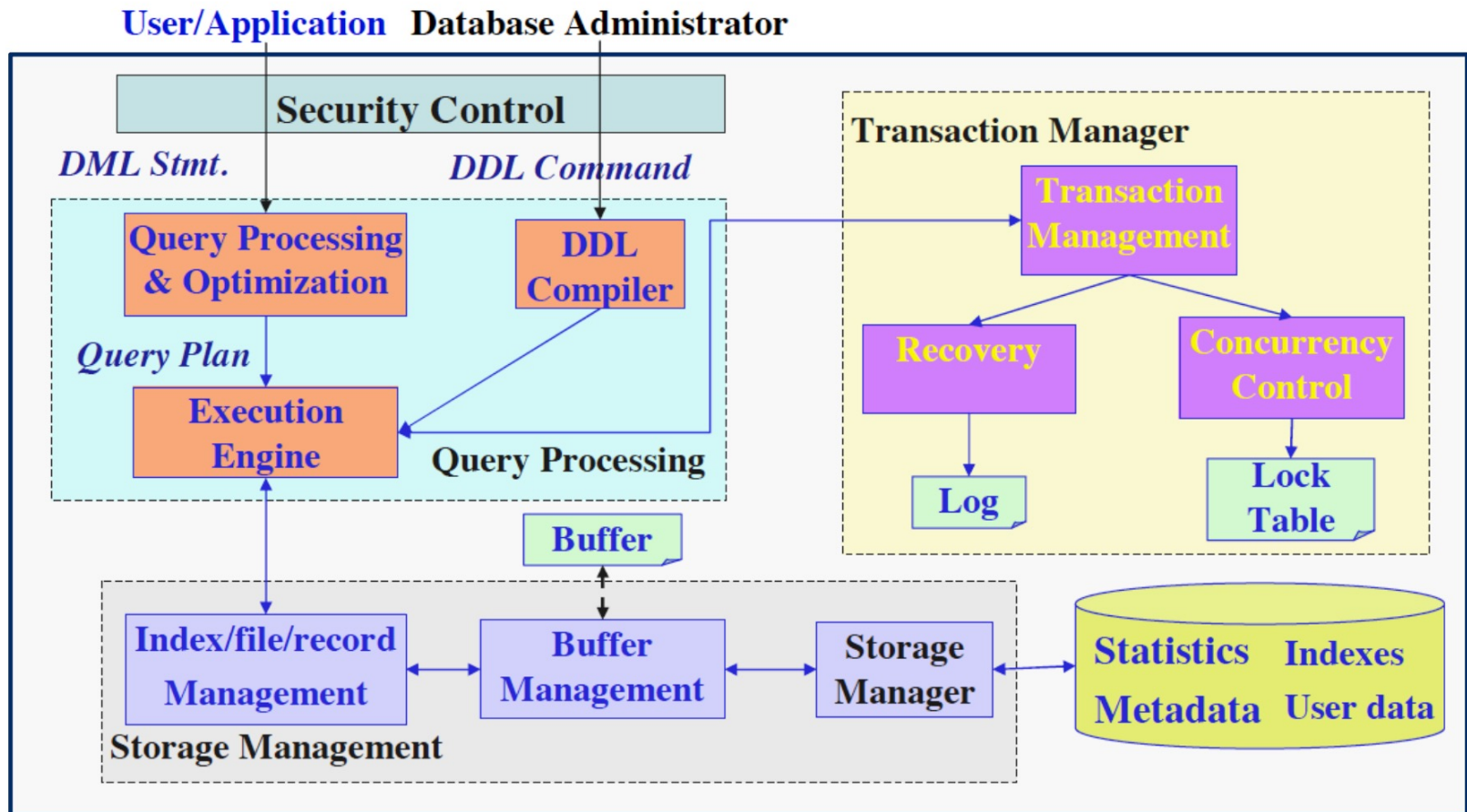
Week 8 – Storing Data: Disk, File and Index

By Xiaoyang Wang, CSE UNSW

*Disclaimer: the course materials are sourced from previous offerings of
COMP9311 and COMP3311*

A solid orange horizontal bar at the bottom of the slide.

Functional Components of DBMS



Storage Hierarchy

Primary Storage: main memory.

fast access, expensive.

Secondary storage: hard disk.

slower access, less expensive.

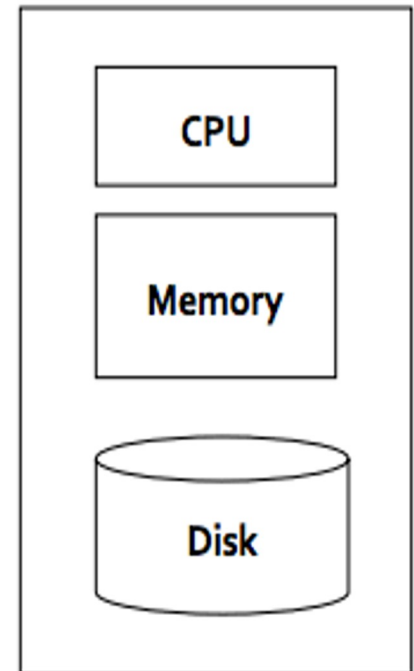
Tertiary storage: tapes, cd, etc.

slowest access, cheapest.

Primary Storage

Main memory:

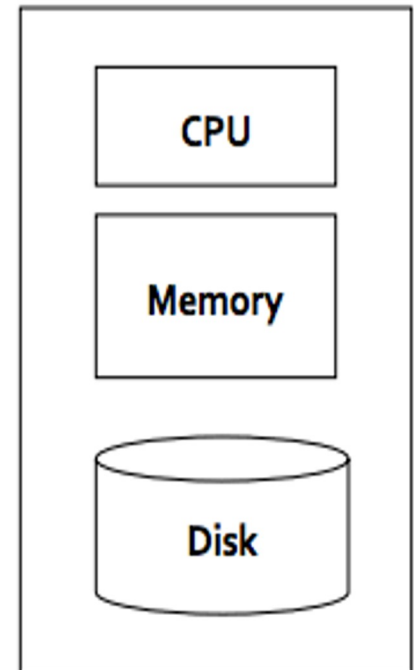
- Fast access (10s to 100s of nanoseconds; 1 nanosecond = 10^{-9} seconds)
- Generally, too small (or too expensive) to store the entire database
- **Volatile** — contents of main memory are usually lost if a power failure or system crash occurs.



Secondary Storage

Magnetic-disk

- Data is stored on spinning disk, and read/written magnetically
- Primary medium for the long-term storage of data; typically stores entire database.
- **Data must be moved from disk to main memory for access, and written back for storage**
Much slower access than main memory
- **Direct-access** – possible to read data on disk in any order.
- Survives power failures and system crashes
Recall: disk failure can destroy data, but is rare



Some Numbers

Event	Latency
1 CPU cycle	0.3 ns
Level 1 cache access	0.9 ns
Level 2 cache access	2.8 ns
Level 3 cache access	12.9 ns
Main memory access (DRAM, from CPU)	120 ns
Solid-state disk I/O (flash memory)	50-150 μ s
Rotational disk I/O	1-10 ms
Internet: San Francisco to New York	40 ms
Internet: San Francisco to United Kingdom	81 ms
Internet: San Francisco to Australia	183 ms
TCP packet retransmit	1-3 s

CPU cost vs I/O cost

The implementation Issues

There are two main costs, CPU cost and I/O (Input/Output) cost.

- CPU cost is to process data in main memory.
- I/O cost is to read/write data from/into disk.

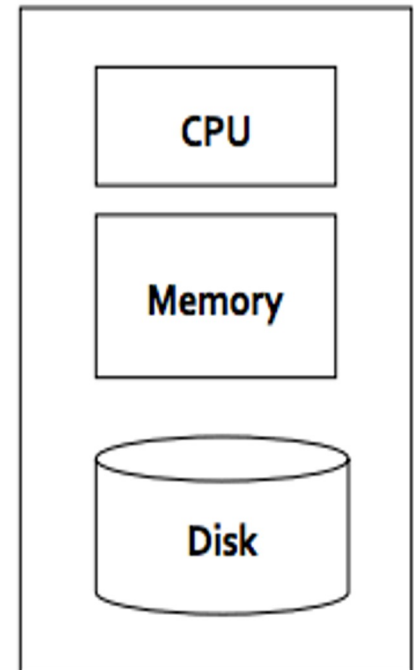
The dominating cost is I/O cost. For query processing in DBMS, CPU cost can be ignored.

The key issue is to reduce I/O cost.

- It is to reduce the number of I/O accesses.

What is I/O cost?

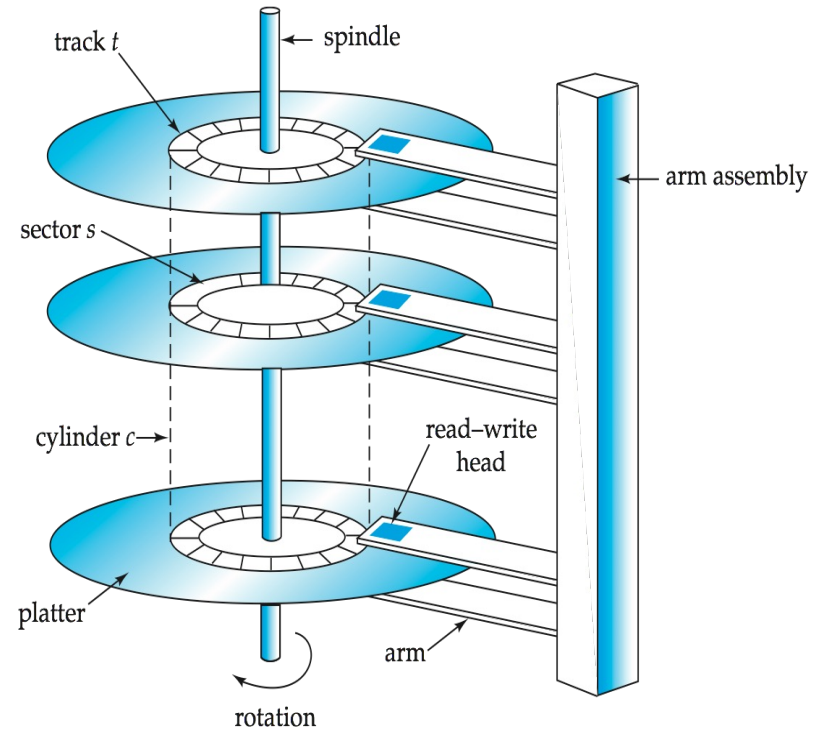
- A block (or page) to be read/written from/into disk is one I/O access (or one disk-block/page access).



OLD Magnetic Hard Disk

Characteristics of disks:

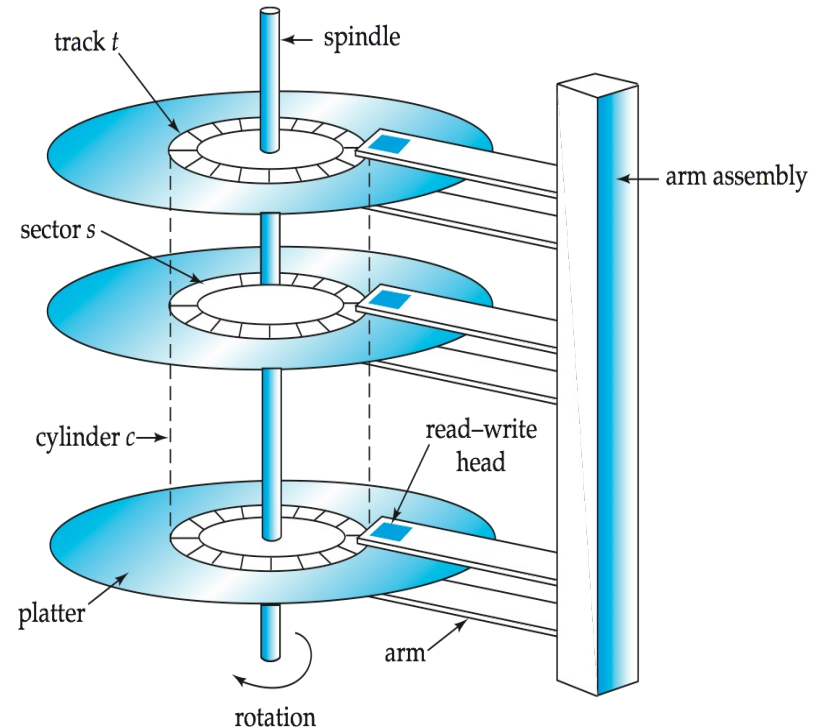
- collection of platters
- each platter = set of tracks
- each track = sequence of sectors (blocks)



NOTE: Diagram simplifies the structure of actual disk drives

OLD Magnetic Hard Disk

- Data must be in memory for the DBMS to operate on it.
- Smallest process unit is Block: If a single record in a block is needed, the entire block is transferred.



NOTE: Diagram simplifies the structure of actual disk drives

Disks

Access time includes:

- seek time (find the right track, e.g., 10msec)
- rotational delay (find the right sector, e.g., 5msec)
- transfer time (read/write block, e.g., 10μsec)

Random access is dominated by **seek time** and **rotational delay**

Disk Space Management

Improving Disk Access:

Use knowledge of data access patterns.

- E.g., two records often accessed together: put them in the same block (clustering)
- E.g., records scanned sequentially: place them in consecutive sectors on same track

Keep Track of Free Blocks

- Maintain a list of free blocks
- Use bitmap

Using OS File System to Manage Disk Space

- extend OS facilities, but not rely on the OS file system.
- (portability and scalability)

Storage Access

Data must be in memory for the DBMS to operate on it.

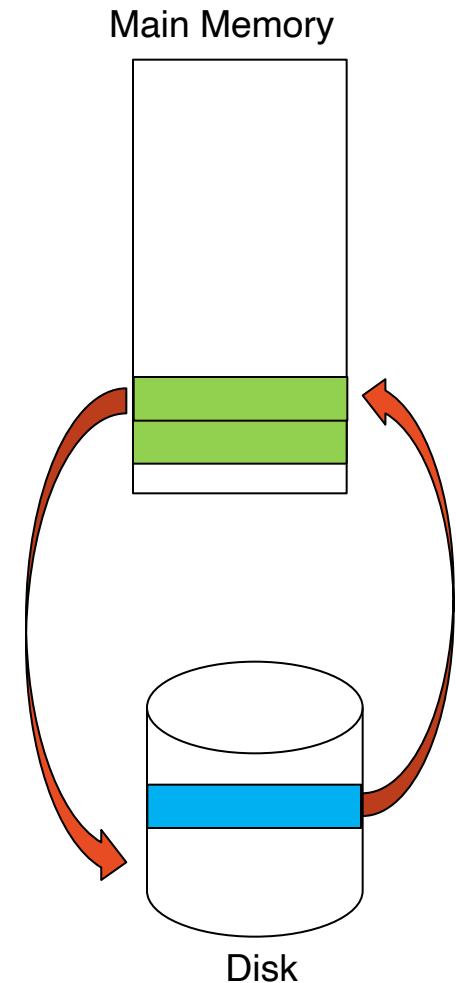
A database file is partitioned into fixed-length storage units called **blocks**. Blocks are units of both storage allocation and data transfer.

Database system seeks to **minimize the number of block transfers** between the disk and memory.

We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.

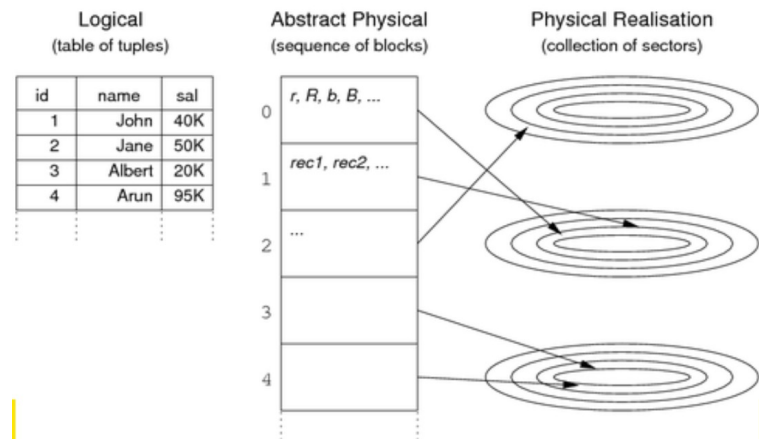
Buffer – portion of main memory available to store copies of disk blocks.

Buffer manager – subsystem responsible for allocating buffer space in main memory.

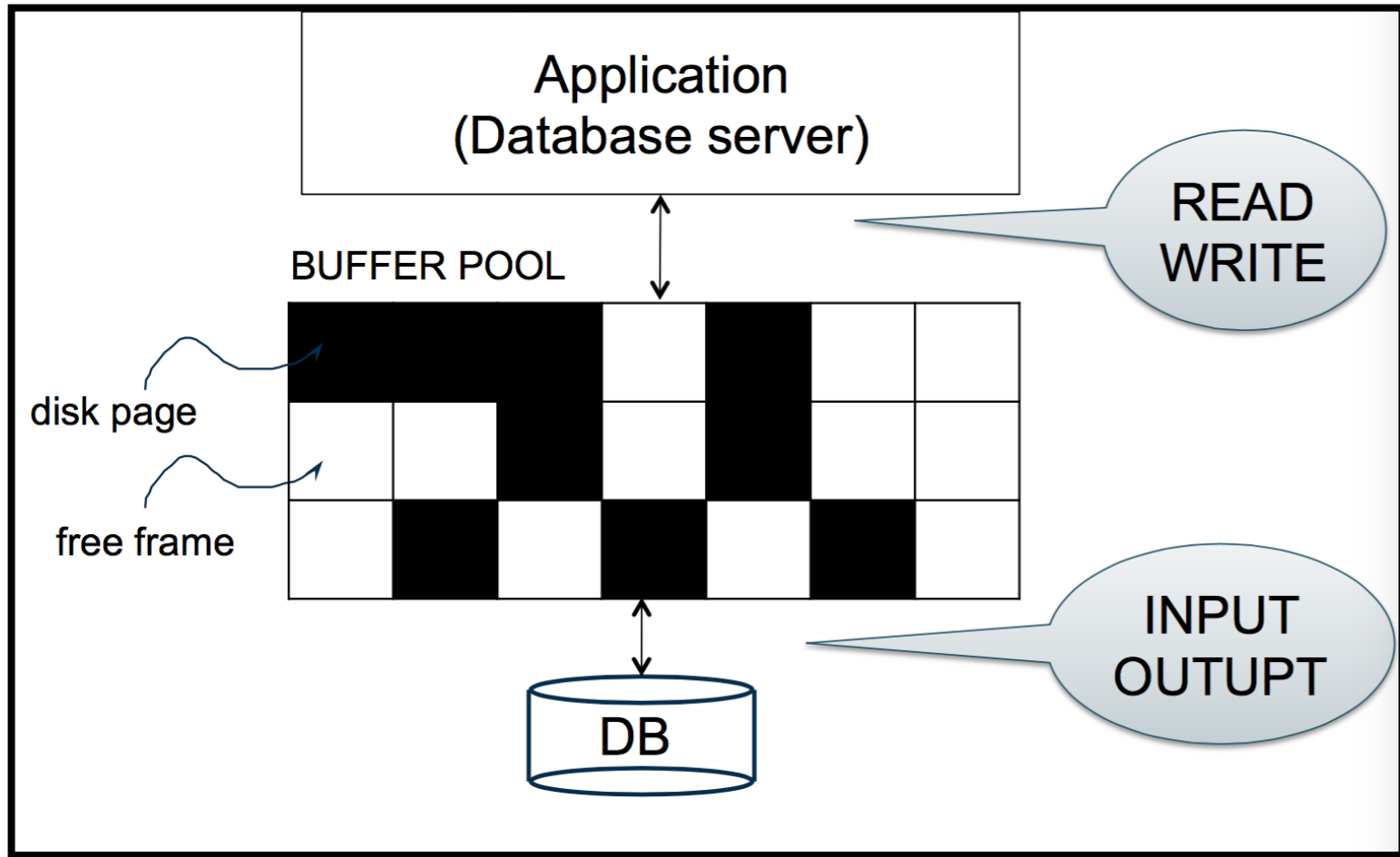


Disk-Block Access

- Smallest process unit is a **block**: If a single record in a block is needed, the entire block is transferred.
- Data are transferred between disk and main memory in **units** of blocks.
- A relation is stored as a **file** on **disk**.
- A file is a sequence of blocks, where a **block** is a fixed-length storage unit.
- A block is also called a **page**.



Buffer Management in a DBMS



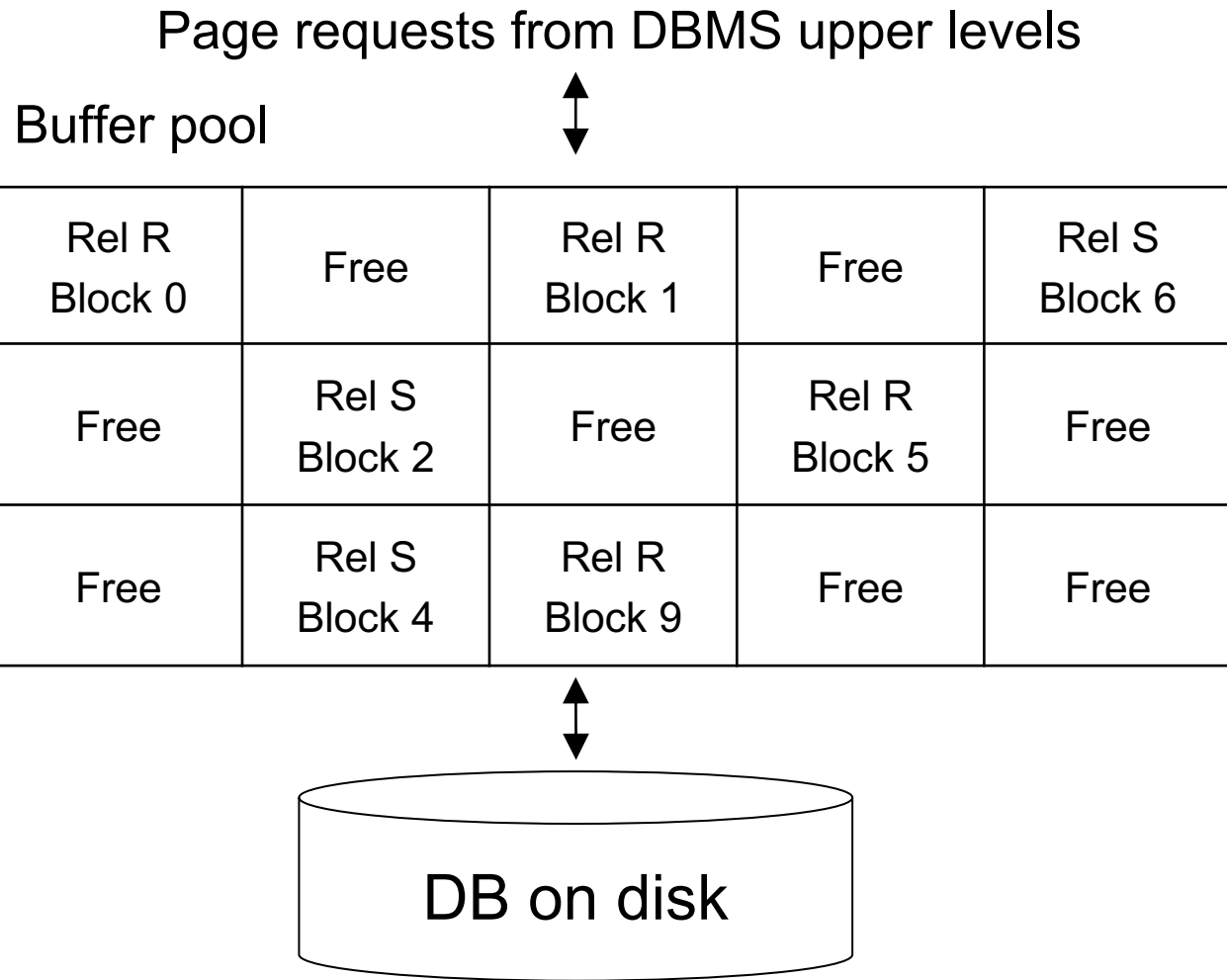
Buffer Management

Manages traffic between disk and memory by maintaining a **buffer pool** in main memory.

Buffer pool

- collection of page slots (frames) which can be filled with copies of disk block data.
- One page = 4096 Bytes = One block

Buffer Pool



Buffer Pool

The **request_block** operation

If block **is** already in buffer pool:

- no need to read it again
- use the copy there (unless write-locked)

If block **is not** in buffer pool yet:

- need to read from hard disk into a free frame
- if no free frames, need to remove block using a buffer replacement policy.

The **release_block** function indicates that block is no longer in use

- good candidate for removal / replacing

Buffer Pool

For each frame, we need to know:

- whether it is currently in use
- whether it has been modified since loading (dirty bit)
- how many transactions are currently using it (pin count)
- (maybe) time-stamp for most recent access

Buffer Replacement Policies

Least Recently Used (LRU)

- release the frame that has not been used for the longest period.
- intuitively appealing idea but can perform badly

First in First Out (FIFO)

- need to maintain a queue of frames
- enter tail of queue when read in

Most Recently Used (MRU):

- release the frame used most recently

Random

No one is guaranteed to be better than the others. Quite dependent on applications.

Example

Data pages: P1, P2, P3, P4

Queries:

Q1: read P1; Q2: read P2;

Q3: read P3; Q4: read P1;

Q5: read P2;

Buffer:

P1 Q1		
--------------	--	--

Example

Data pages: P1, P2, P3, P4

Queries:

Q1: read P1; Q2: read P2;

Q3: read P3; Q4: read P1;

Q5: read P2;

Buffer:

P1 Q1		
P1 Q1	P2 Q2	

Example

Data pages: P1, P2, P3, P4

Queries:

Q1: read P1; Q2: read P2;

Q3: read P3; Q4: read P1;

Q5: read P2;

Buffer:

P1 Q1		
P1 Q1	P2 Q2	
P1 Q1	P2 Q2	P3 Q3

Example

Data pages: P1, P2, P3, P4

Queries:

Q1: read P1; Q2: read P2;

Q3: read P3; Q4: read P1;

Q5: read P2;

Buffer:

P1 Q1		
P1 Q1	P2 Q2	
P1 Q1	P2 Q2	P3 Q3
P1 Q4	P2 Q2	P3 Q3

Example

Data pages: P1, P2, P3, P4

Queries:

Q1: read P1; Q2: read P2;

Q3: read P3; Q4: read P1;

Q5: read P2;

Buffer:

P1 Q1		
P1 Q1	P2 Q2	
P1 Q1	P2 Q2	P3 Q3
P1 Q4	P2 Q2	P3 Q3
P1 Q4	P2 Q5	P3 Q3

Example

Data pages: P1, P2, P3, P4

Queries:

Q1: read P1; Q2: read P2;

Q3: read P3; Q4: read P1;

Q5: read P2;

Buffer:

P1 Q1		
P1 Q1	P2 Q2	
P1 Q1	P2 Q2	P3 Q3
P1 Q4	P2 Q2	P3 Q3
P1 Q4	P2 Q5	P3 Q3

How about if Q6 read P4?

Using different buffer replacement policies

Example

Data pages: P1, P2, P3, P4

Queries:

Q1: read P1; Q2: read P2;

Q3: read P3; Q4: read P1;

Q5: read P2;

Buffer:

P1 Q1		
P1 Q1	P2 Q2	
P1 Q1	P2 Q2	P3 Q3
P1 Q4	P2 Q2	P3 Q3
P1 Q4	P2 Q5	P3 Q3
P1 Q4	P2 Q5	P4 Q6

How about if Q6 read P4?

Using different buffer replacement policies

LRU: Least recently used

Example

Data pages: P1, P2, P3, P4

Queries:

Q1: read P1; Q2: read P2;

Q3: read P3; Q4: read P1;

Q5: read P2;

Buffer:

P1 Q1		
P1 Q1	P2 Q2	
P1 Q1	P2 Q2	P3 Q3
P1 Q4	P2 Q2	P3 Q3
P1 Q4	P2 Q5	P3 Q3
P1 Q4	P4 Q6	P3 Q3

How about if Q6 read P4?

Using different buffer replacement policies

MRU: Most recently used

Example

Data pages: P1, P2, P3, P4

Queries:

Q1: read P1; Q2: read P2;

Q3: read P3; Q4: read P1;

Q5: read P2;

Buffer:

P1 Q1		
P1 Q1	P2 Q2	
P1 Q1	P2 Q2	P3 Q3
P1 Q4	P2 Q2	P3 Q3
P1 Q4	P2 Q5	P3 Q3
P4 Q6	P2 Q5	P3 Q3

How about if Q6 read P4?

Using different buffer replacement policies

FIFO: First in first out

Example

Data pages: P1, P2, P3, P4

Queries:

Q1: read P1; Q2: read P2;

Q3: read P3; Q4: read P1;

Q5: read P2;

Buffer:

P1 Q1		
P1 Q1	P2 Q2	
P1 Q1	P2 Q2	P3 Q3
P1 Q4	P2 Q2	P3 Q3
P1 Q4	P2 Q5	P3 Q3

How about if Q6 read P4?

Using different buffer replacement policies

Random: randomly choose one to replace

Cache Performance

Cache hits

- pages can be served by the cache

Cache misses

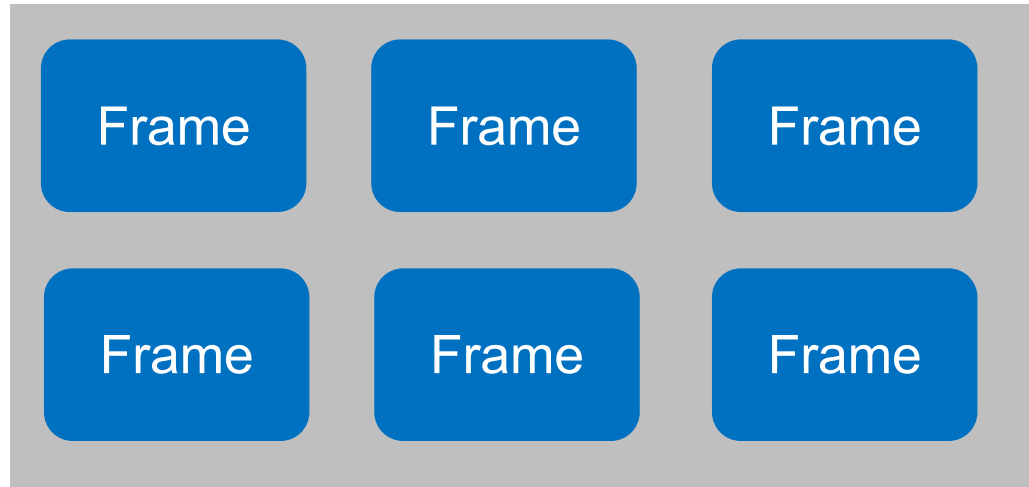
- pages have to be retrieved from the disk

Hit rate = $\# \text{cache hits} / (\# \text{cache hits} + \# \text{cache misses})$

Repeated Scan (LRU)

Cache hit : 0

Attempts : 0



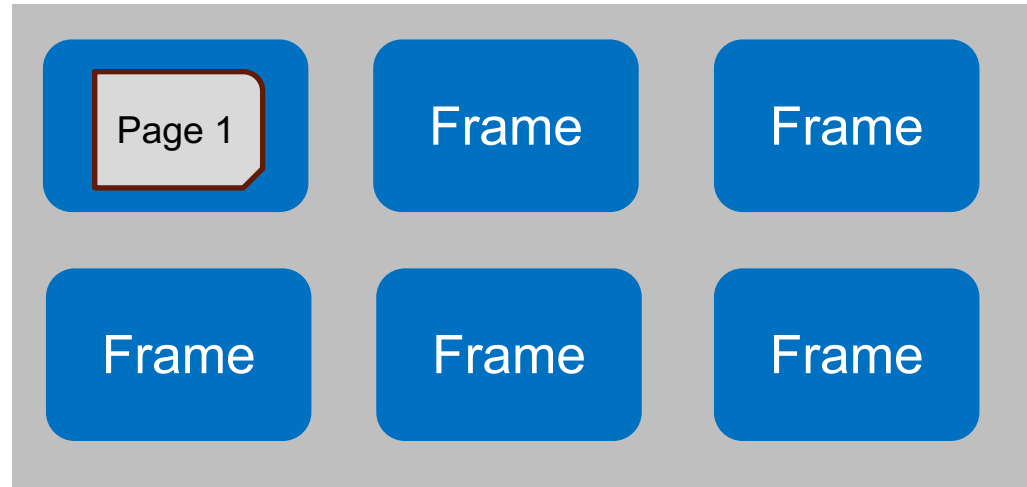
Disk Space Manager



Repeated Scan (LRU): Read Page 1

Cache hit : 0

Attempts : 1



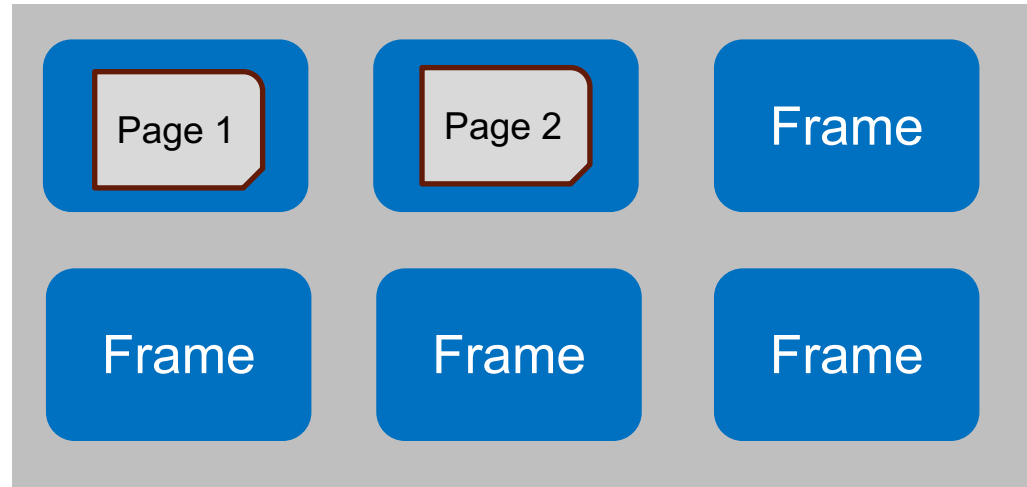
Disk Space Manager



Repeated Scan (LRU): Read Page 2

Cache hit : 0

Attempts : 2



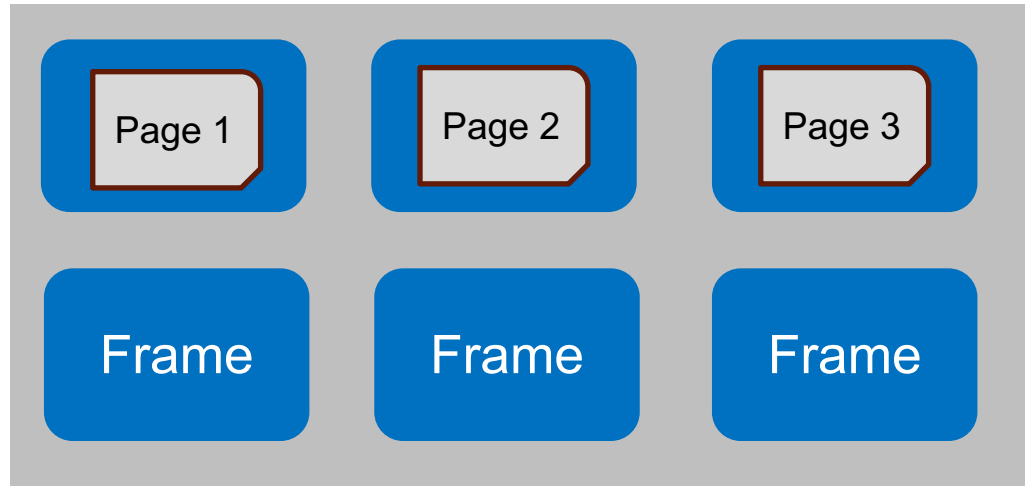
Disk Space Manager



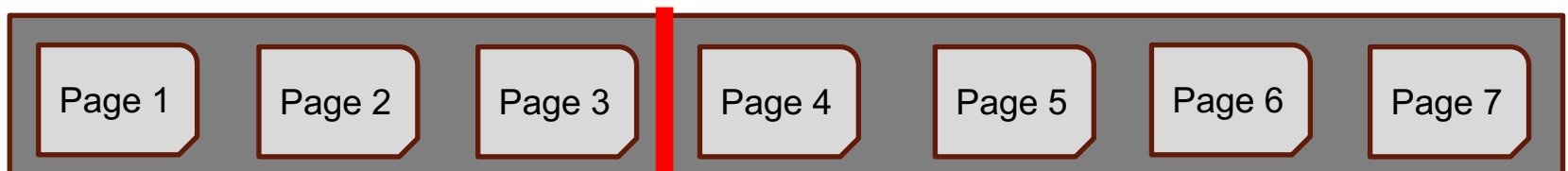
Repeated Scan (LRU): Read Page 3

Cache hit : 0

Attempts : 3



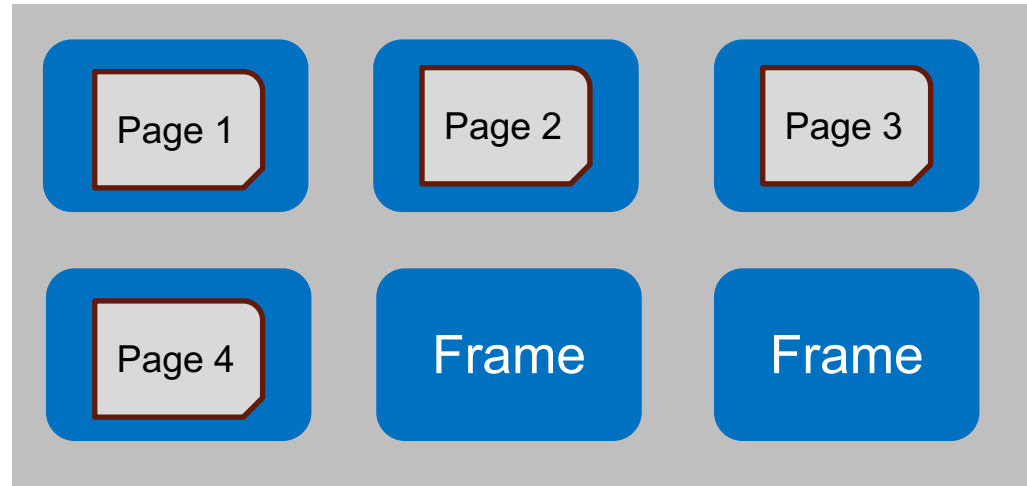
Disk Space Manager



Repeated Scan (LRU): Read Page 4

Cache hit : 0

Attempts : 4



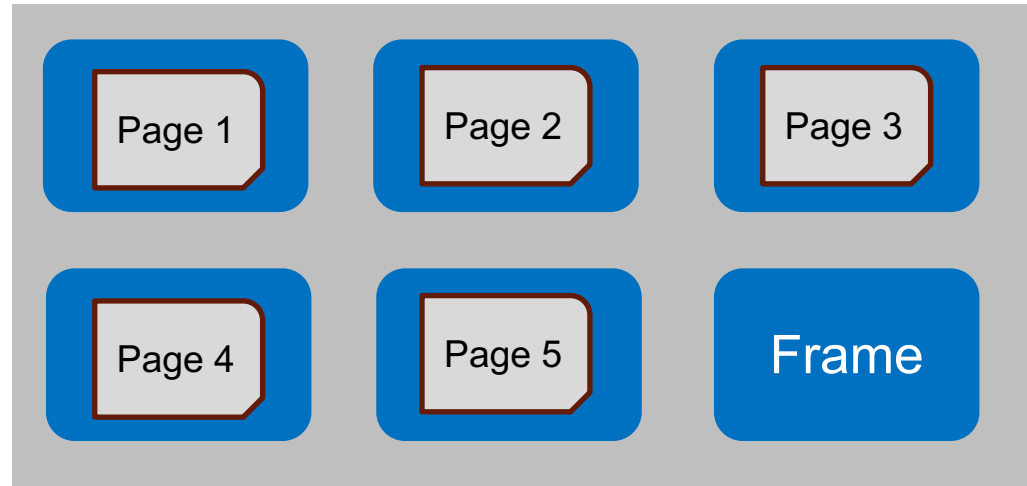
Disk Space Manager



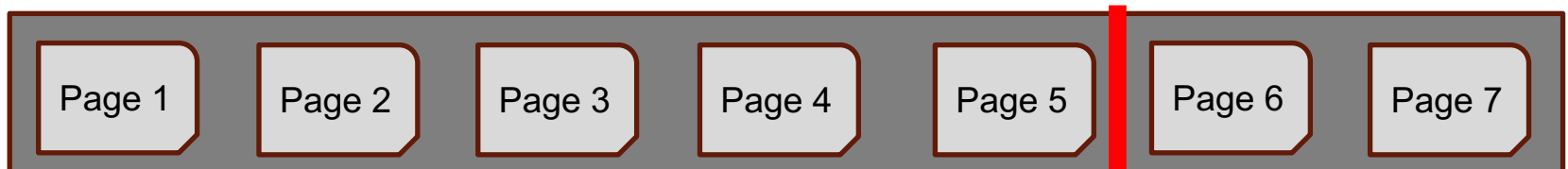
Repeated Scan (LRU): Read Page 5

Cache hit : 0

Attempts : 5



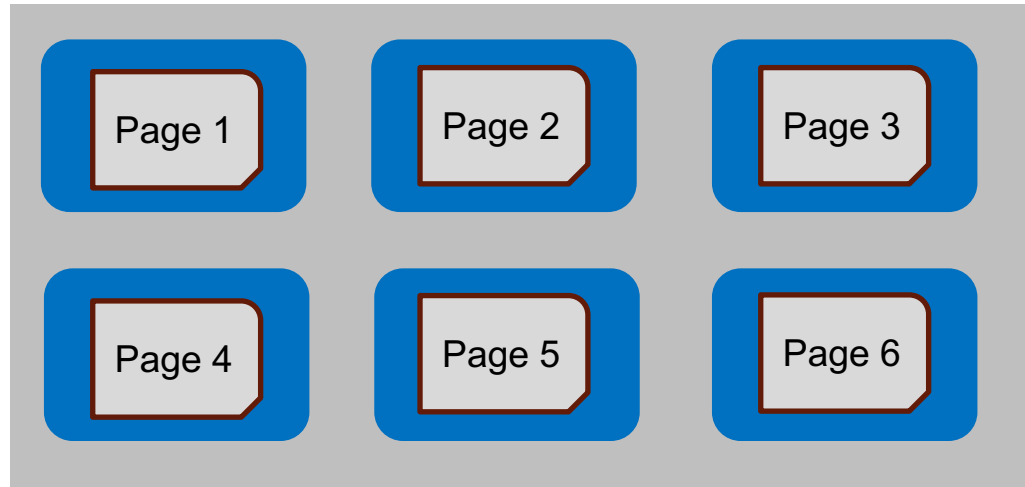
Disk Space Manager



Repeated Scan (LRU): Read Page 6

Cache hit : 0

Attempts : 6



So far, unavoidable cache misses.
Now the fun begins

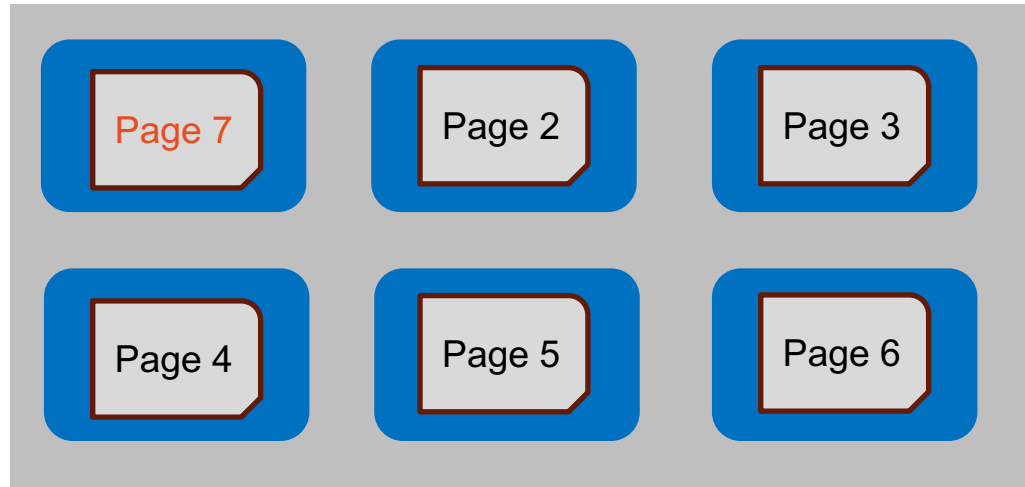
Disk Space Manager



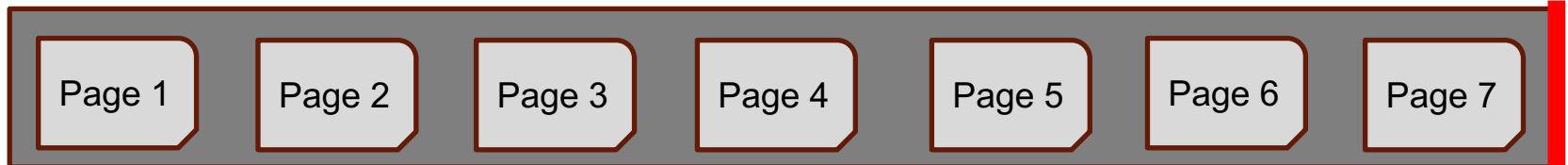
Repeated Scan (LRU): Read Page 7

Cache hit : 0

Attempts : 7



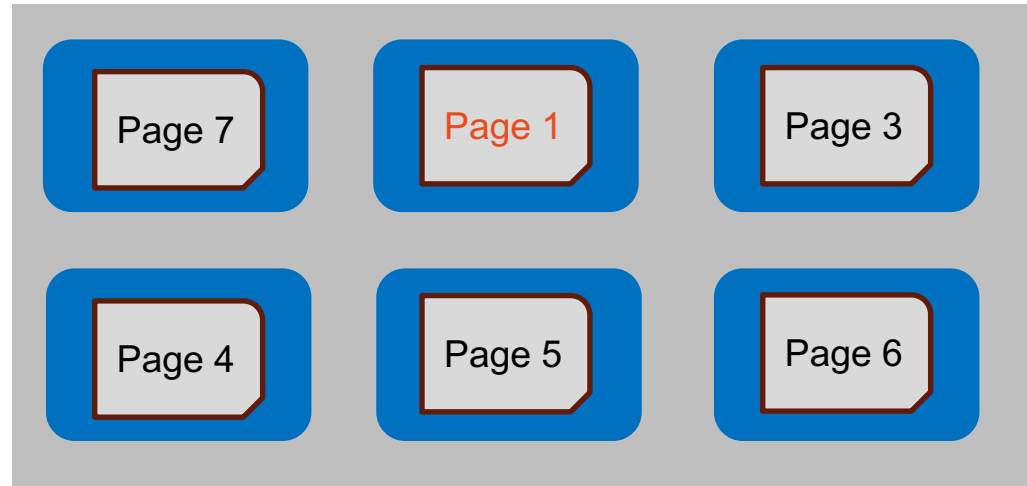
Disk Space Manager



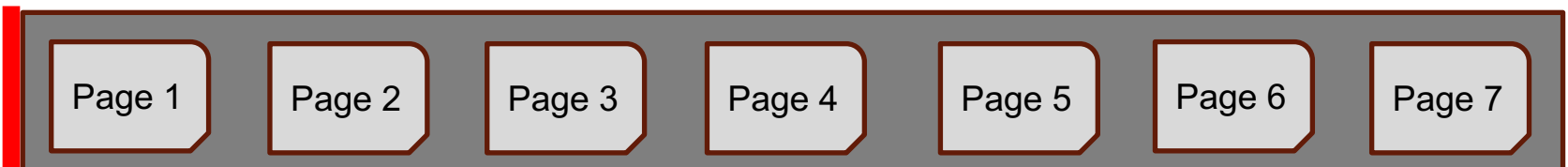
Repeated Scan (LRU): Read again

Cache hit : 0

Attempts : 8



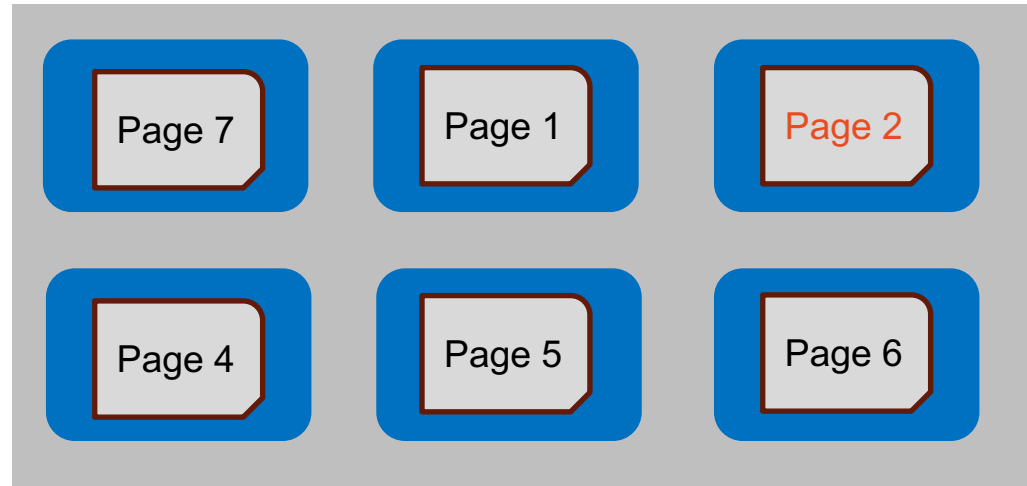
Disk Space Manager



Repeated Scan (LRU): Read again

Cache hit : 0

Attempts : 9



Get the picture? A worst-case scenario!

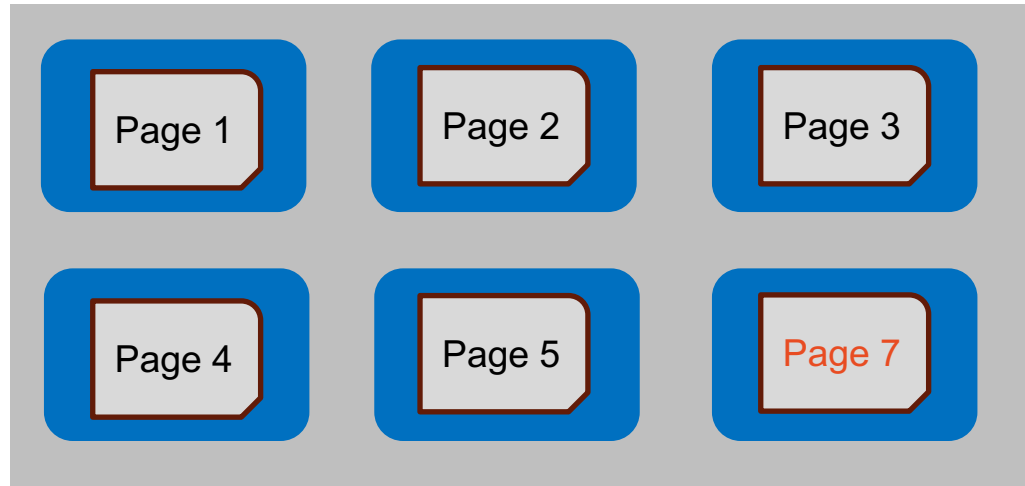
Disk Space Manager



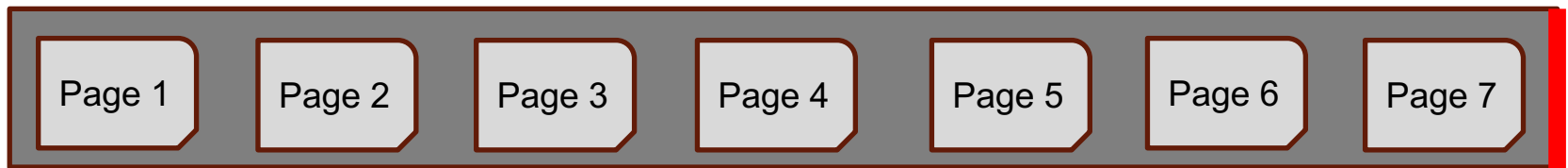
Repeated Scan (MRU): Read Page 7

Cache hit : 0

Attempts : 7



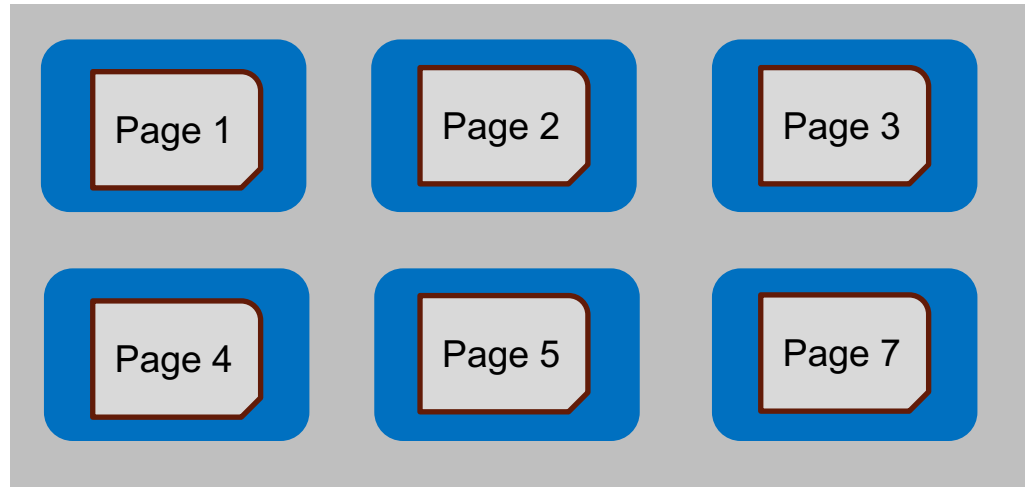
Disk Space Manager



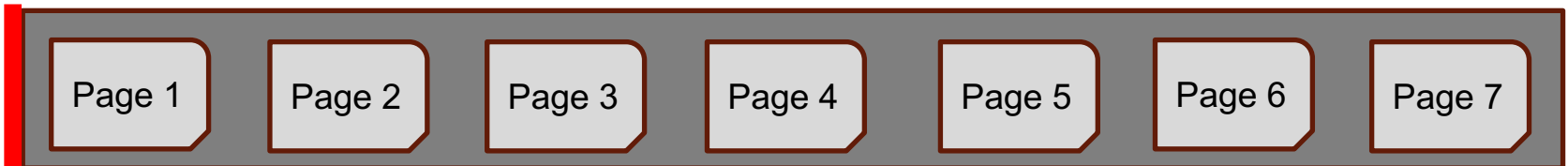
Repeated Scan (MRU): Read again

Cache hit : 1

Attempts : 8



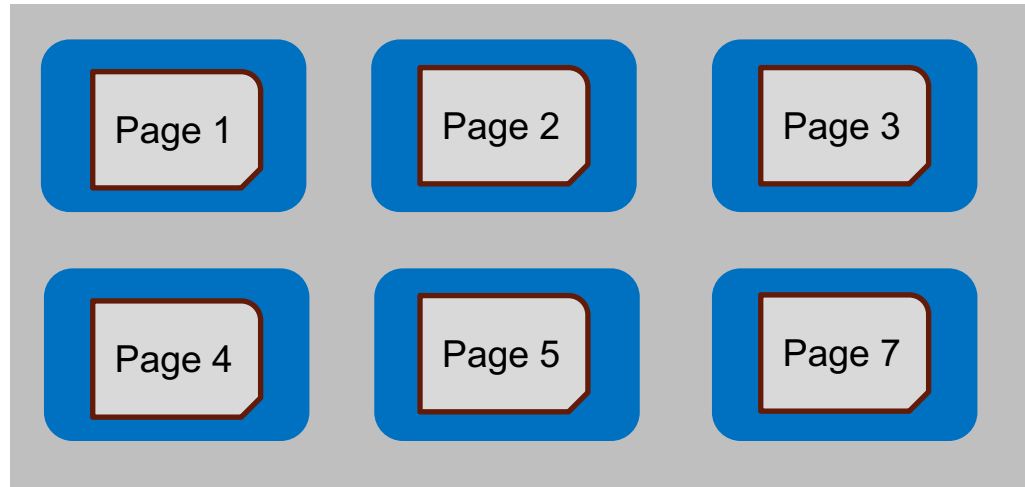
Disk Space Manager



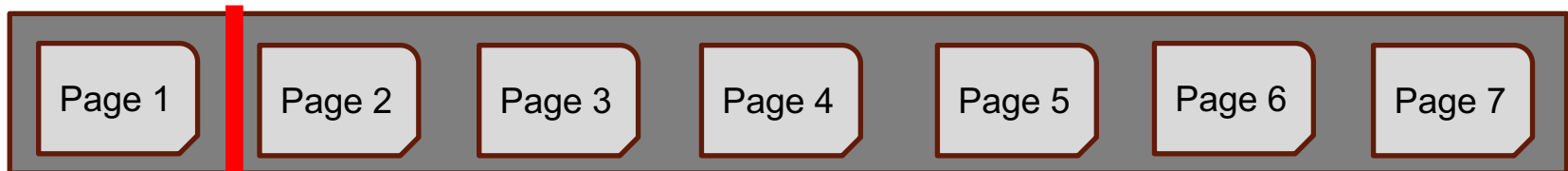
Repeated Scan (MRU): Read again

Cache hit : 2

Attempts : 9



Disk Space Manager



Compare LRU and MRU

When LRU and MRU both read Page 2 again

LRU:

Cache hit: 0

Attempts: 9

MRU:

Cache hit: 2

Attempts: 9

Sequential Flooding

LRU: We need to get in/out every page

- This is called Sequential Flooding

MRU: performs the best in this case (repeated scan)

Again, no replacement policy is guaranteed to be superior to the others. The choice often depends on specific applications and their requirements.

Block (Page) Formats

- **Block:** A block is a collection of slots.
- **Slot:** Each slot contains a record.
- **Record:** A record is identified by record_id: $rid = \langle \text{page id, slot number} \rangle$.

Question: How are records physically stored on disk?

Record Formats

Records are stored within fixed-length blocks.

Fixed-length: each field has a fixed length as well as the number of fields.

33357462	Neil Young	Musician	0277
4 bytes	40 bytes	20 bytes	4 bytes

- Easy for intra-block space management.
- Possible waste of space.

Variable-length: some field is of variable length.

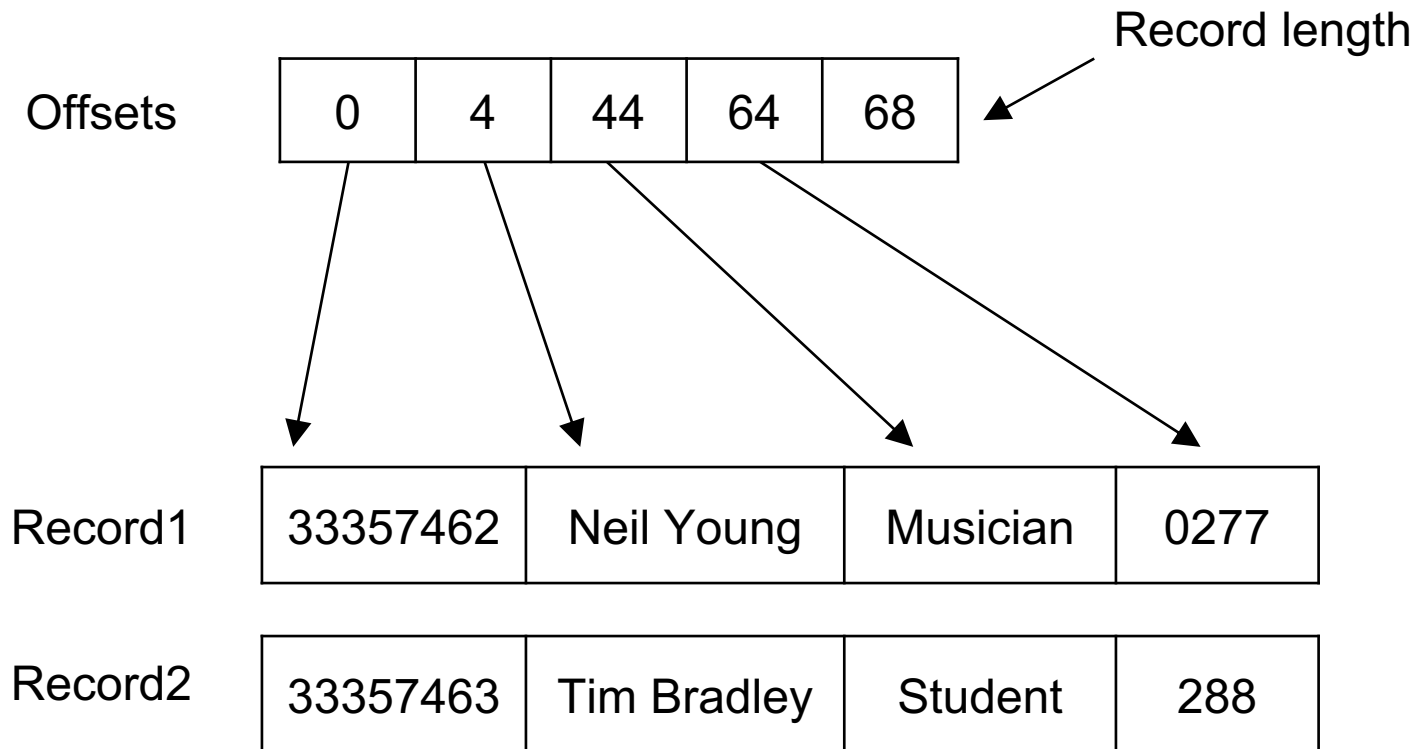
33357462	Neil Young	Musician	0277
4 bytes	10 bytes	8 bytes	4 bytes

- complicates intra-block space management
- does not waste (as much) space.

Fixed Length

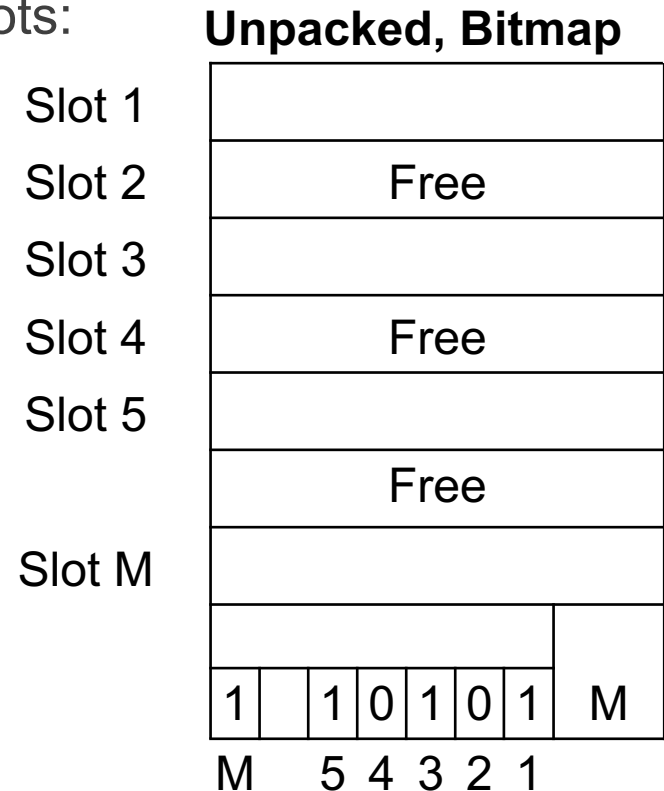
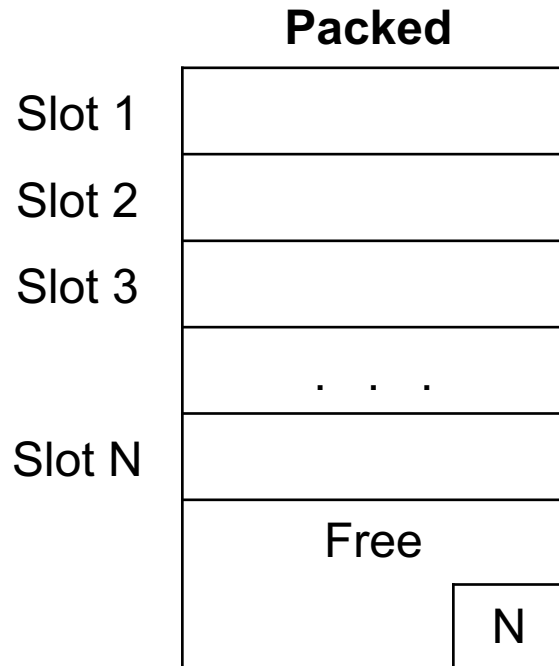
Encoding scheme for fixed-length records:

- length + offsets stored in header



Fixed-Length Records

For fixed-length records, use record slots:



Insertion: occupy first free slot; packed more efficient.

Deletion: (a) need to compact, (b) mark with 0; unpacked more efficient.

Variable-Length

Encoding schemes where attributes are stored **in order**.

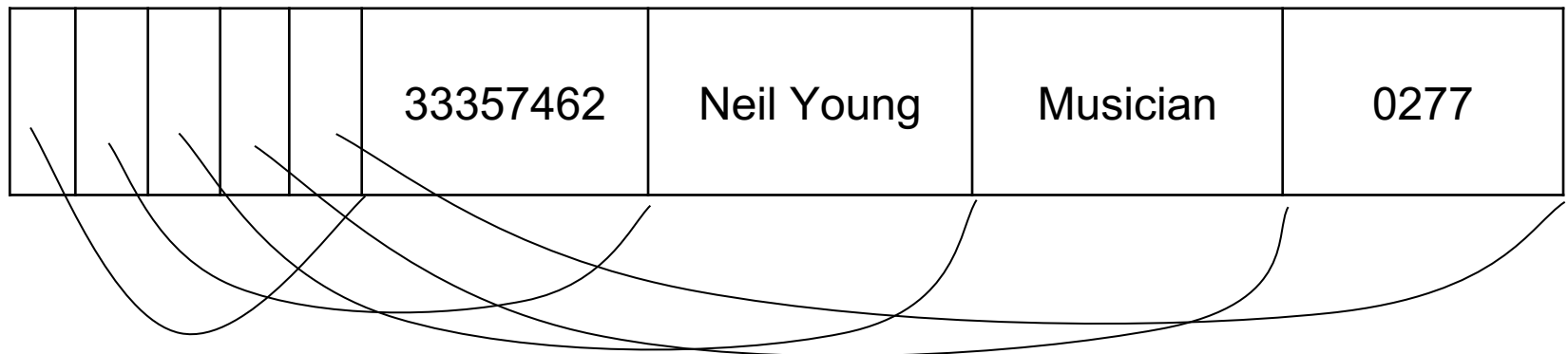
- Option 1: Prefix each field by length



- Option 2: Terminate fields by delimiter

33357462/Neil Young/Musician/0277/

- Option 3: Array of offsets



Variable-Length Records (1)

Another encoding scheme: attributes are not stored in order.

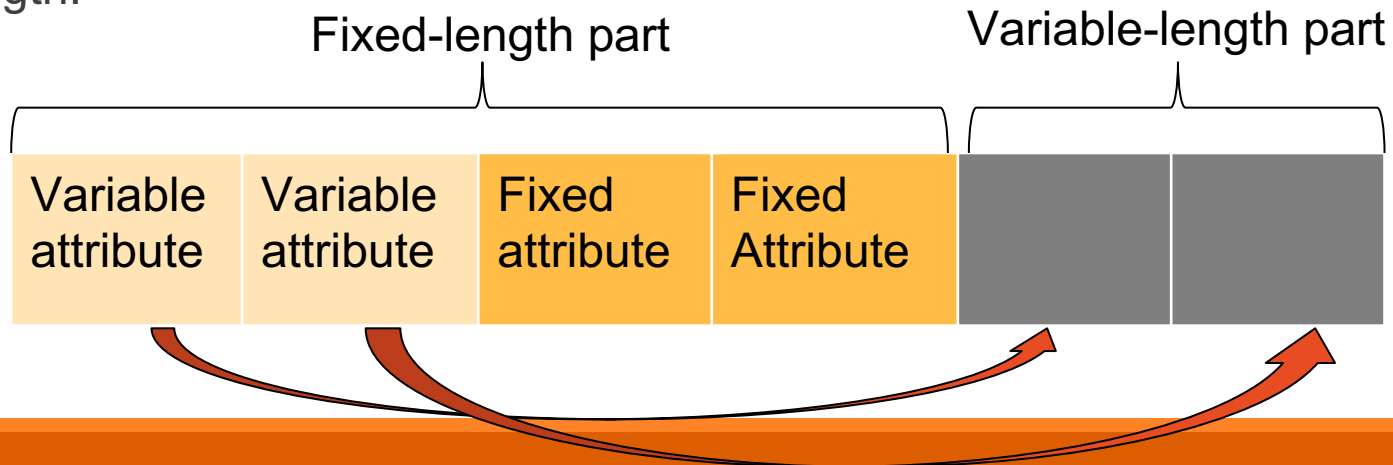
Fixed-length part followed by variable-length part.

- (b) The fixed-length part is to tell where we can find the data if it is a variable-length data field.
- (c) The variable-length part is to store the data.

Variable length attributes are represented by fixed size (**offset, length**) in the fixed-length part, and keep attribute values in the variable-length part.

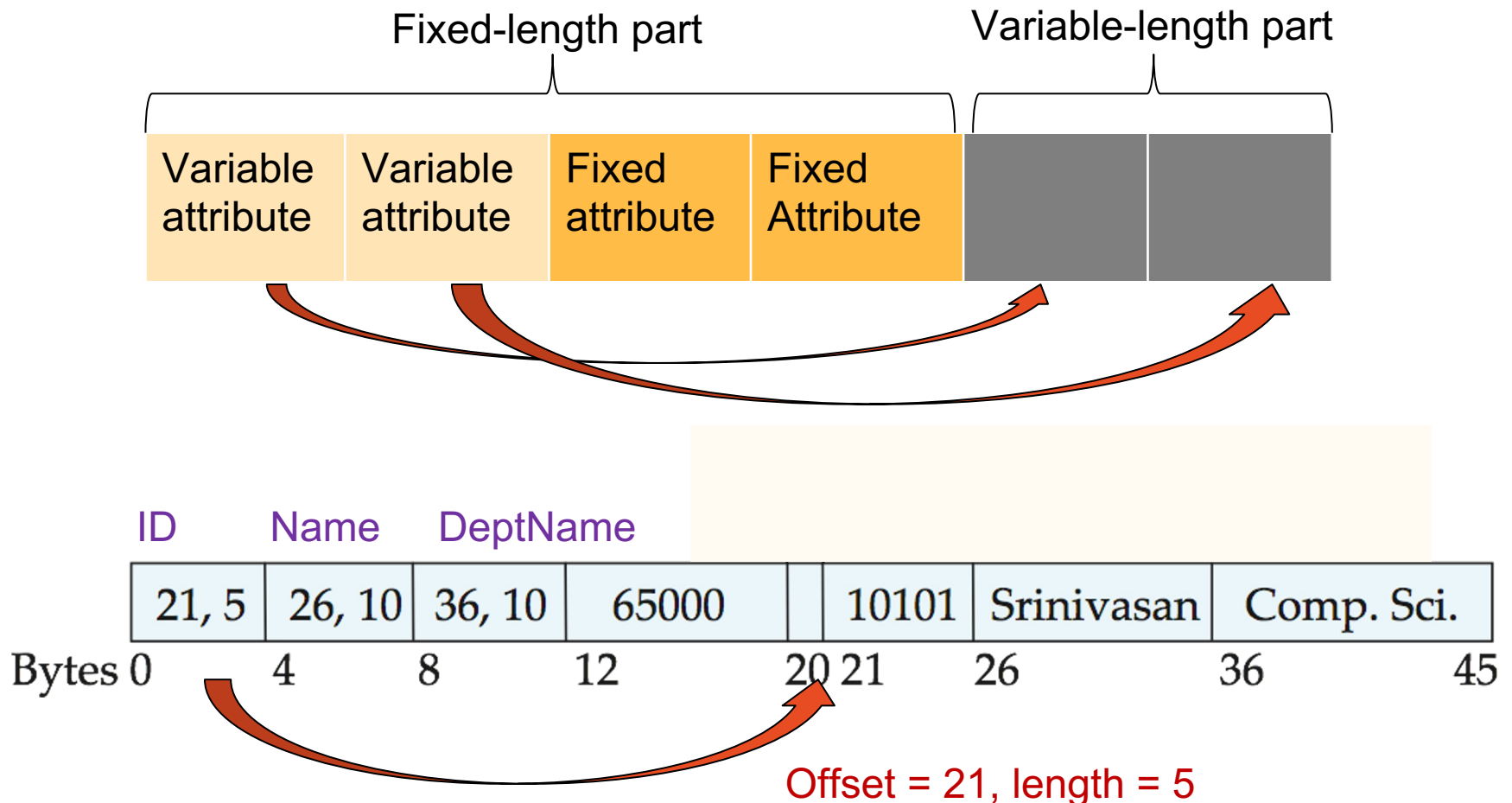
Fixed length attributes store attribute values in the fixed-length part.

Suppose there is a relation with 4 attributes: 2 fixed-length and 2 variable-length.



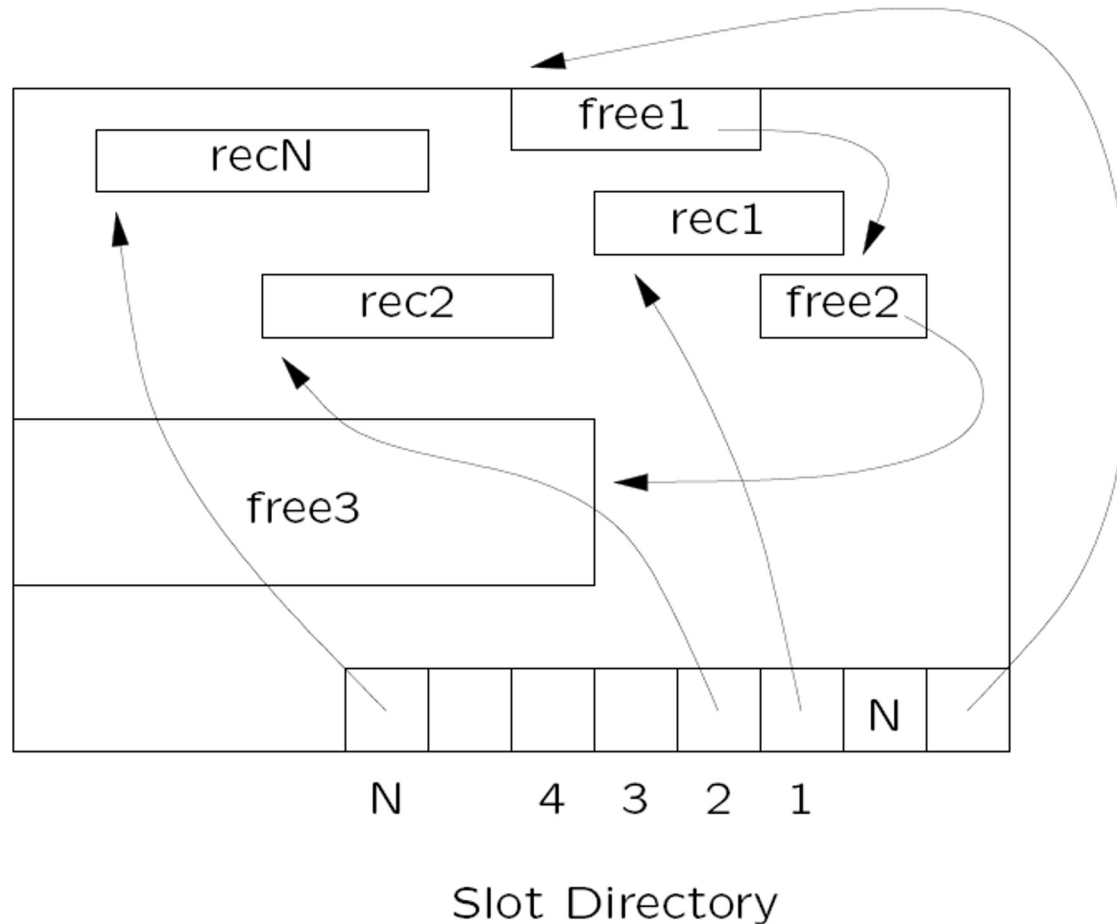
Variable-Length Records (2)

Example: a tuple of (ID, Name, DeptName, Salary) where the **first three** are variable length.



Variable-Length Records

Fragmented free space:



Notes

Reminder:

- The basic store unit on disk (in memory) is block (page)
- We will use page/block interchangeably.
- One page consists of multiple data records.

Indexes (basic concept)

Find all subcode belonging to the Law faculty (i.e., subcode = LAWS)

Basic strategy = scan ..., test, select
Not efficient ...

An “idea” of an index on a file on the search key ‘subcode’ may look like ...

LAND, {1}
ANAT, {2,19}
BENV, {...}
LAWS, {4,7, ...}

An index gives a short cut to the tuples that match the search key

An added cost for building/maintaining it ...

subcode	code	name	uoc	career
LAND	LAND1170	Design 1	4	UG
ANAT	ANAT2211	Histology 1	0	UG
CHEN	CHEN2062	Intro to Process Chemistry 2	3	UG
LAWS	LAWS2332	Law and Social Theory	8	UG
ECON	ECON4321	Economic History 4 Honours	48	UG
AHIS	AHIS1602	History 1	12	
LAWS	LAWS2425	Research Thesis	4	UG
ENVS	ENVS4503	Env Sci Hons (Geog) 18uoc	18	UG
SOLA	SOLA5058	Special Topic in PV	6	PG
BENV	BENV2704	Advanced Construction Systems	3	UG
ANAT	ANAT3141	Functional Anatomy 2	6	UG
BENV	BENV2205	Classical Architecture	3	UG
BENV	BENV2402	Design Modelling - Time Based	6	UG
BIOT	BIOT3081	Environmental Biotech	6	UG
BIOC	BIOC4103	Genetics 4 Honours Full-Time	24	UG
ECON	ECON4127	Thesis (Economics)	12	UG
ENVS	ENVS4404	Environmental Science 4 Chemis	24	UG
LAWS	LAWS2423	Research Thesis	8	UG
MUSC	MUSC2402	Professional Practices D	6	UG
REGS	REGS3756	Negotiation	8	UG
HPSC	HPSC5020	Supervised Reading Program	8	PG
CRIM	CRIM4000	Crim Honours (Research) F/T	24	UG
BIOC	BIOC4109	Genetics Honours (PT)	12	UG
CEIC	CEIC4105	Professional Electives	3	UG

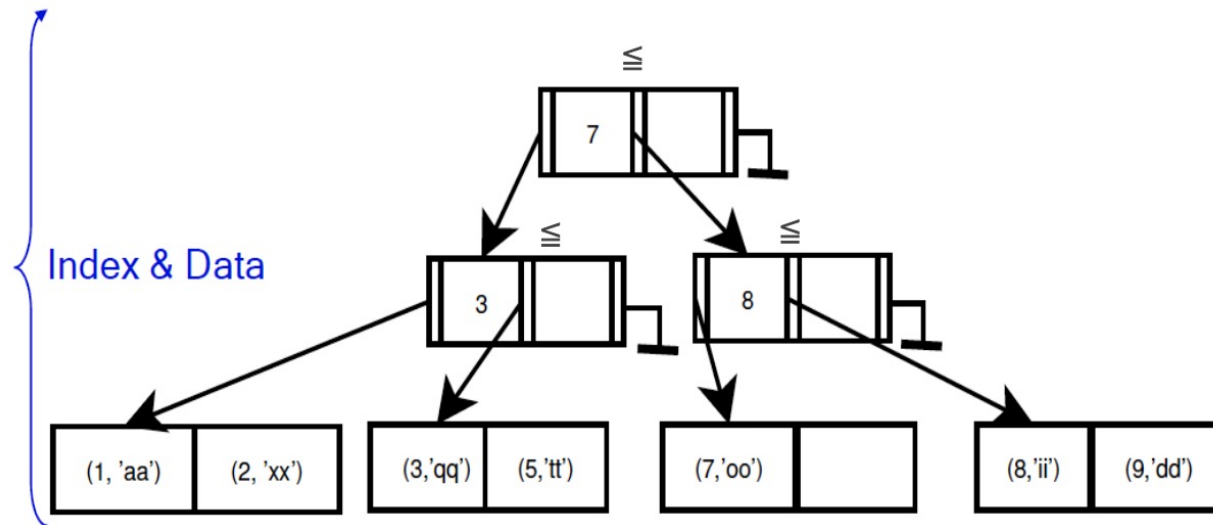
Indexes

An **index** on a file speeds up selections on the **search key fields** for the index

- Any subset of the fields of a relation can be the search key for an index on the relation
- Search key is not the same as key

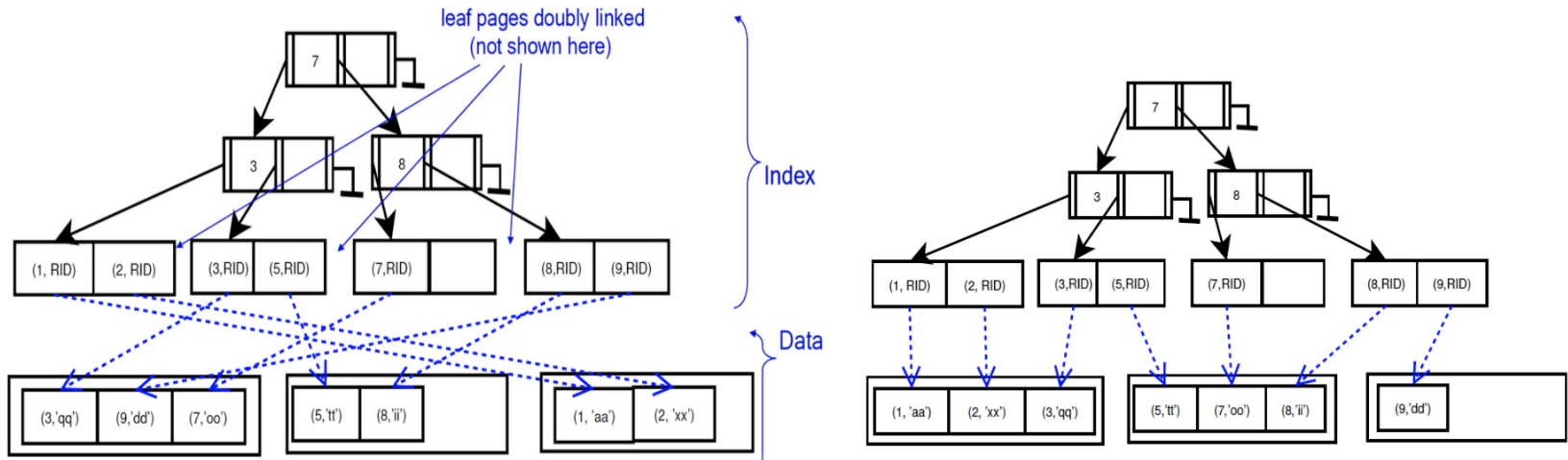
An index contains a collection of **data entries**, and supports efficient retrieval of all data entries K^* with a given key value K .

e.g., Data record in B+ Tree



A	B
1	aa
2	xx
3	qq
5	tt
7	oo
8	ii
9	dd

e.g., B+ Tree – (un)clustered



The leaves of the index contains a pointer to the data (single record)

You can build many such indexes on a file (different search keys) as the index is separated from the data

The underlying file that contains the records may or not be sorted by key ... when unsorted, the arrows (i.e., the pointers to the data) 'cross' each other, this is referred to as 'unclustered' index option (cf. clustered, on the right)

e.g., Hash index

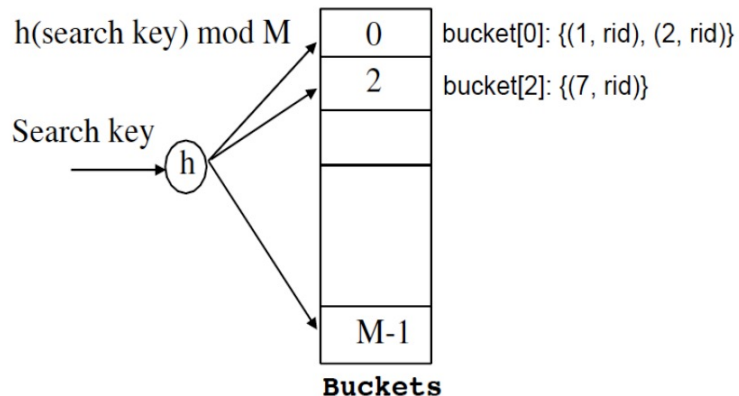
Index contains “buckets”, each bucket contains the index data entries ...

A hash function works on the search key and produces a number over the range of 0 ... M-1 (M is the number of buckets).

e.g., $h(K) = (a * K + b)$, where a, b are constant ... K is the search key.

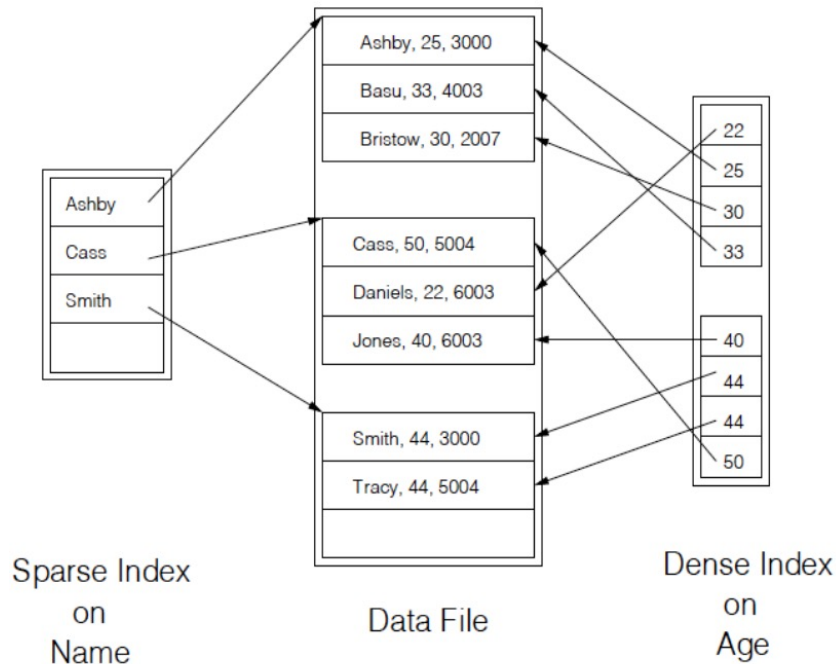
Fast to search (i.e., no traversing of tree nodes)

Best for equality searches, cannot support range searches.



(an approximate diagram of Hash Index)

Index can be dense or sparse



Regardless of the index structure ... An index can be dense or sparse

Dense, if the index contains at least one data entry per every value of the search key

- faster to search for a particular record
- high cost

Sparse, if only some values of the search key have data entries

- low cost
- slower to search (i.e., some scan required)

Summary on Index

Choice orthogonal to indexing technique used to locate data entries with a a given key value.

There may be several indexes on a given file of data records, each with a different search key.

Indexes can be classified as

- clustered vs. unclustered
- dense vs. sparse.

Differences have important consequences for utility/performance

DB Application Performance

In order to make DB applications efficient, we need to know:

- what operations on the data does the application require (which queries, updates, inserts and how frequently is each one performed)
- how these operations might be implemented in the DBMS (data structures and algorithms for select, project, join, sort, ...)
- how much each implementation will cost (in terms of the amount of data transferred between memory and disk)

and then, as much as the DBMS allows, "encourage" it to use the most efficient methods

DB Application Performance

Application programmer choices that affect the cost of executing a query...
how queries are expressed is important ... Generally speaking:

- a join is faster than subquery, especially if subquery is correlated
- avoid producing large intermediate tables then filtering
- avoid applying functions in where/group-by clauses

We could create indexes on tables

- index will speed-up filtering based on the search key attributes
- indexes generally only effective for equality or greater than/less than type search
- indexes have update-time and storage overheads (i.e., indexes could be costly)
 - only useful if filtering (i.e., reading) is much more frequent operations on that table than updates

Database Query Processing

Example: query to find “sales” people earning more than \$50K

```
select name from Employee
where salary > 50000 and
      empid in (select empid from WorksIn
                where dept = 'Sales')
```

Needs to examine all employees, even if not in Sales

A query optimiser might use the strategy (roughly ...)

```
SalesEmps = (select empid from WorksIn where dept='Sales')
foreach e in Employee {
    if (e.empid in SalesEmps && e.salary > 50000)
        add e to result set
}
```

Only examines Sales employees, and uses a simpler test

Query Processing

A query in SQL:

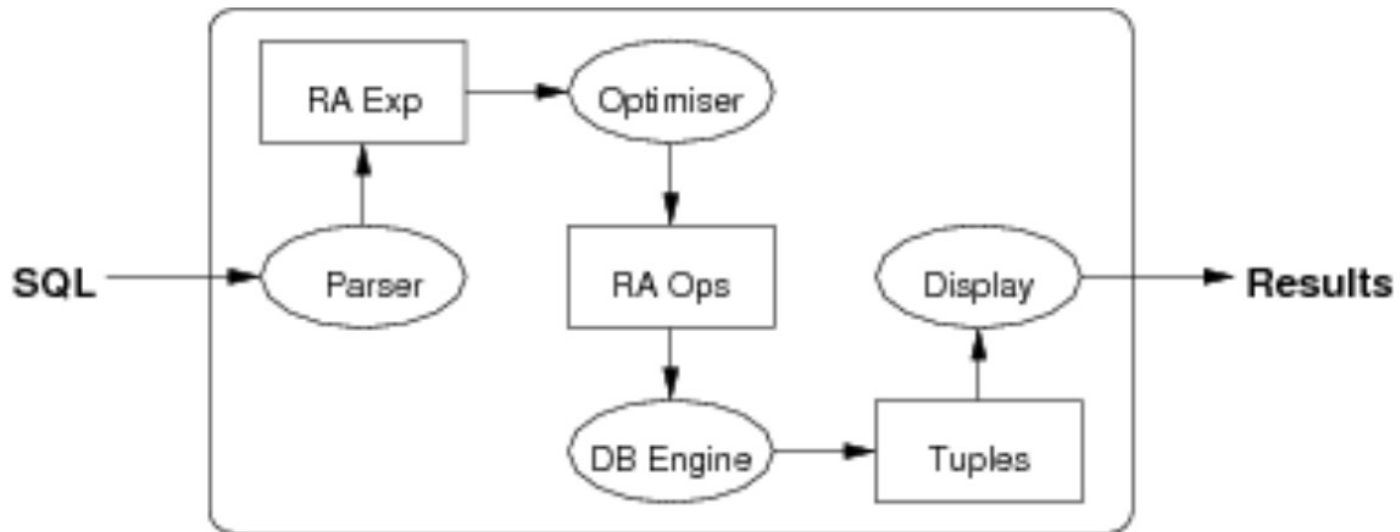
- states what answers are required
- says little about how they should be computed

A query evaluator & optimiser :

- takes a declarative description of the query in SQL
- parses the query into a relational algebra expression
- determines a plan for answering the query
- executes the plan via the database engine

Query Processing

mapping SQL to relational algebra (RA)



RA Expressions → Optimiser → concrete RA operations
(e.g., JOIN on empid) (e.g., HASH JOIN on empid)

Mapping SQL to RA Expression

A translation scheme would be something like:

- SELECT clause \rightarrow projection
- WHERE clause on single reln \rightarrow selection
- WHERE clause on two relns $R \rightarrow$ join

Example

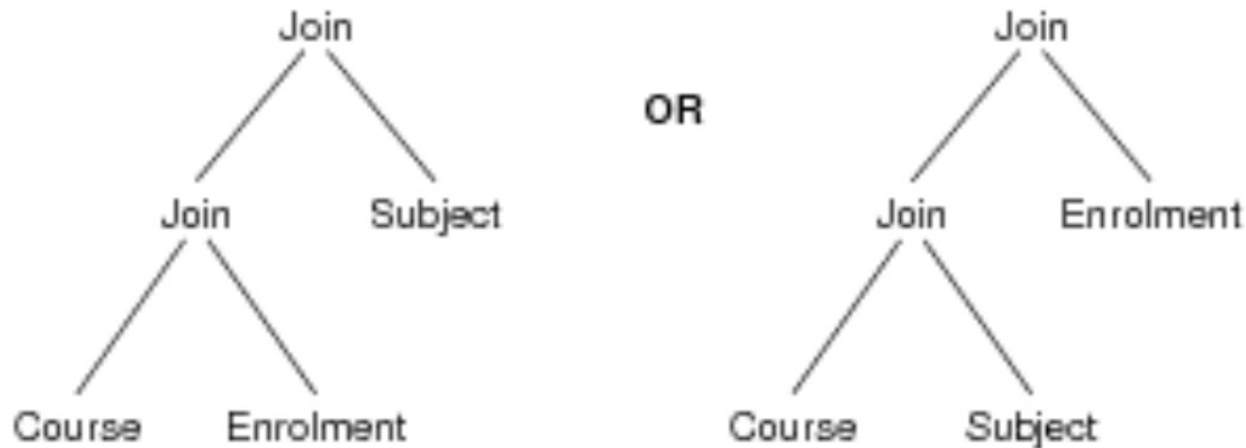
```
select s.name, e.course
from   Student s, Enrolment e
where  s.id = e.student and e.mark > 50;
```

Is translated into

Project [name,course] (Select [mark>50] (Join [id=student] (Student, Enrolment)))

e.g., Query Evaluation

The Join operations could be done (at least) two different ways



Which is better? ... The query optimiser works this out.

Note: for a join involving N tables, there are $O(N!)$ possible trees to consider ...

Query Optimisation Problem

The query optimiser start with an RA expression, then

- generates a set of equivalent expressions
- generates possible execution plans for each
- estimates cost of each plan, chooses cheapest

The cost of evaluating a query is determined by:

- size of relations (database relations and temporary relations)
- access mechanisms (indexing, hashing, sorting, join algorithms)
- size/number of main memory buffers (and replacement strategy)

Analysis of costs involves estimating:

- the size of intermediate results
- then, based on this, cost of disk storage accesses (i.e., I/O - page read/write)

Query Optimisation Problem

An execution plan is a sequence of relational operations

Consider execution plans for: $\sigma_c (R \bowtie_d S \bowtie_e T)$

```
tmp1    := hash_join[d](R,S)
tmp2    := sort_merge_join[e](tmp1,T)
result  := binary_search[c](tmp2)
```

or

```
tmp1    := sort_merge_join[e](S,T)
tmp2    := hash_join[d](R,tmp1)
result  := linear_search[c](tmp2)
```

or

```
tmp1    := btree_search[c](R)
tmp2    := hash_join[d](tmp1,S)
result  := sort_merge_join[e](tmp2)
```

All produce same result, but have different costs.

Tips: Performance Tuning

Tuning requires us to consider the following:

- which queries and transactions will be used?

(e.g., check balance for payment, display recent transaction history)

- how frequently does each query/transaction occur?

(e.g., 99% of transactions are EFTPOS payments; 1% are print balance)

- are there time constraints on queries/transactions?

(e.g., payment at EFTPOS terminals must be approved within 7 seconds)

- are there uniqueness constraints on any attributes?

(therefore, define index on attributes to speed up insertion uniqueness check)

- how frequently do updates occur?

(indexes slow down updates, because must update table and index)

Performance Tuning

Performance can be considered at two times:

during schema design

- typically towards the end of schema design process
- requires schema transformations such as **denormalisation**

after schema design

- requires adding extra data structures such as indexes

Indexes

Indexes provide efficient content-based access to tuples (i.e., through search keys).
Can build indexes on any (combination of) attributes.

Defining indexes (syntax):

```
CREATE INDEX index_name ON table_name ( attr1, attr2, ... )
```

e.g., `CREATE INDEX idx_address_phone ON address(phone);`

`CREATE INDEX` also allows us to specify

an access method (`USING` `btree`, `hash`, `rtree`, or `gist`)

e.g., `CREATE INDEX idx_address_phone ON address USING hash (phone);`

Indexes

Indexes can make a huge difference to query processing cost.

On the other hand, they introduce overheads (storage, updates).

Creating indexes to maximise performance benefits:

- apply to attributes used in equality, greater/less-than conditions, e.g.
 - `select * from Employee where id = 12345`
 - `select * from Employee where age > 60`
 - `select * from Employee where salary between 10000 and 20000`
- but only in queries that are frequently used
- and on tables that are not updated frequently

Indexes

Considerations in applying indexes:

- is an attribute used in frequent or expensive queries? (i.e., is it worth it?)
- should we create an index on a collection of attributes? (yes, if the collection is used in a frequent/expensive query)
- can we exploit a clustered index? (only one per table)
- should we use B-tree or Hash index?

```
-- use hashing for (unique) attributes in equality tests, e.g.  
select * from Employee where id = 12345
```

```
-- use B-tree for attributes in range tests, e.g.  
select * from Employee where age > 60
```

Query Tuning

Sometimes, a query can be re-phrased to affect performance:

- by helping the optimiser to make use of indexes
- by avoiding (unnecessary) operations that are expensive

Examples which may prevent optimiser from using indexes:

```
select name from Employee where salary/365 > 10.0
    -- fix by re-phrasing condition to (salary > 3650)
select name from Employee where name like '%ith%'
select name from Employee where birthday is null
    -- above two are difficult to "fix"
```

sometimes "OR" in condition prevents index from being used ... replace the or condition by a union of non-or clauses

```
select name from Employee where dept=1 or dept=2
vs
(select name from Employee where dept=1)
union
(select name from Employee where dept=2)
```

PostgreSQL Query Tuning: EXPLAIN

Select on indexed attribute

```
ass2=# explain select * from student where id=100250;  
               QUERY PLAN
```

```
-----  
Index Scan using student_pkey on student  (cost=0.00..5.94 rows=1 width=17)  
  Index Cond: (id = 100250)
```

```
ass2=# explain analyze select * from student where id=100250;  
               QUERY PLAN
```

```
-----  
Index Scan using student_pkey on student  (cost=0.00..5.94 rows=1 width=17)  
  (actual time=31.209..31.212 rows=1 loops=1)  
    Index Cond: (id = 100250)  
Total runtime: 31.252 ms
```

PostgreSQL Query Tuning: EXPLAIN

Select on non-indexed attribute

```
ass2=# explain select * from student where stype='local';  
               QUERY PLAN
```

```
-----  
Seq Scan on student  (cost=0.00..70.33 rows=18 width=17)  
  Filter: ((stype)::text = 'local'::text)
```

```
ass2=# explain analyze select * from student where stype='local';  
               QUERY PLAN
```

```
-----  
Seq Scan on student  (cost=0.00..70.33 rows=18 width=17)  
                    (actual time=0.061..4.784 rows=2512 loops=1)  
  Filter: ((stype)::text = 'local'::text)  
Total runtime: 7.554 ms
```

Learning Outcomes

- Buffer replacement policies: how does each policy work
- Record / Page management
- Index and query performance