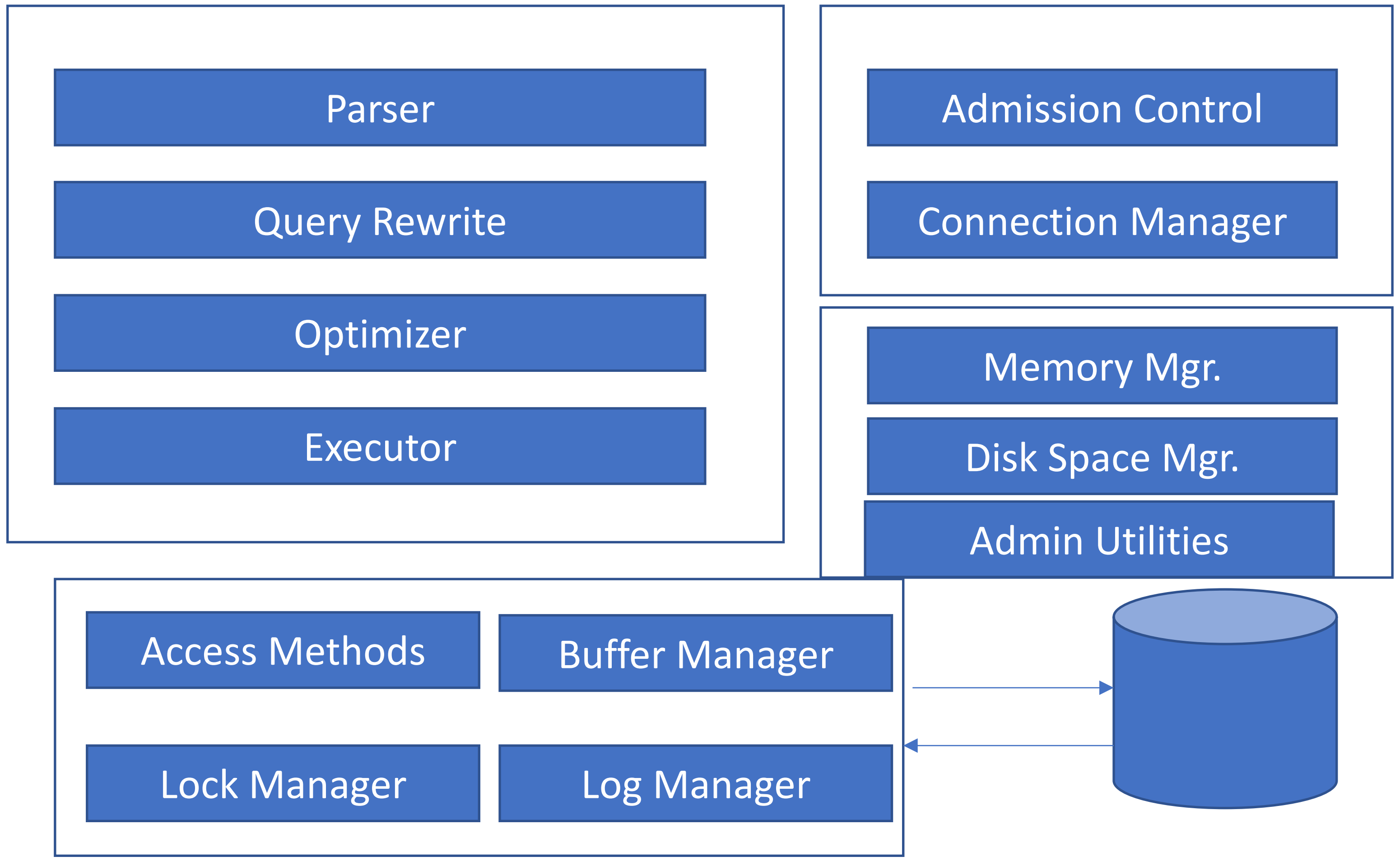
CSC553 Advanced Database Concepts

Tanu Malik

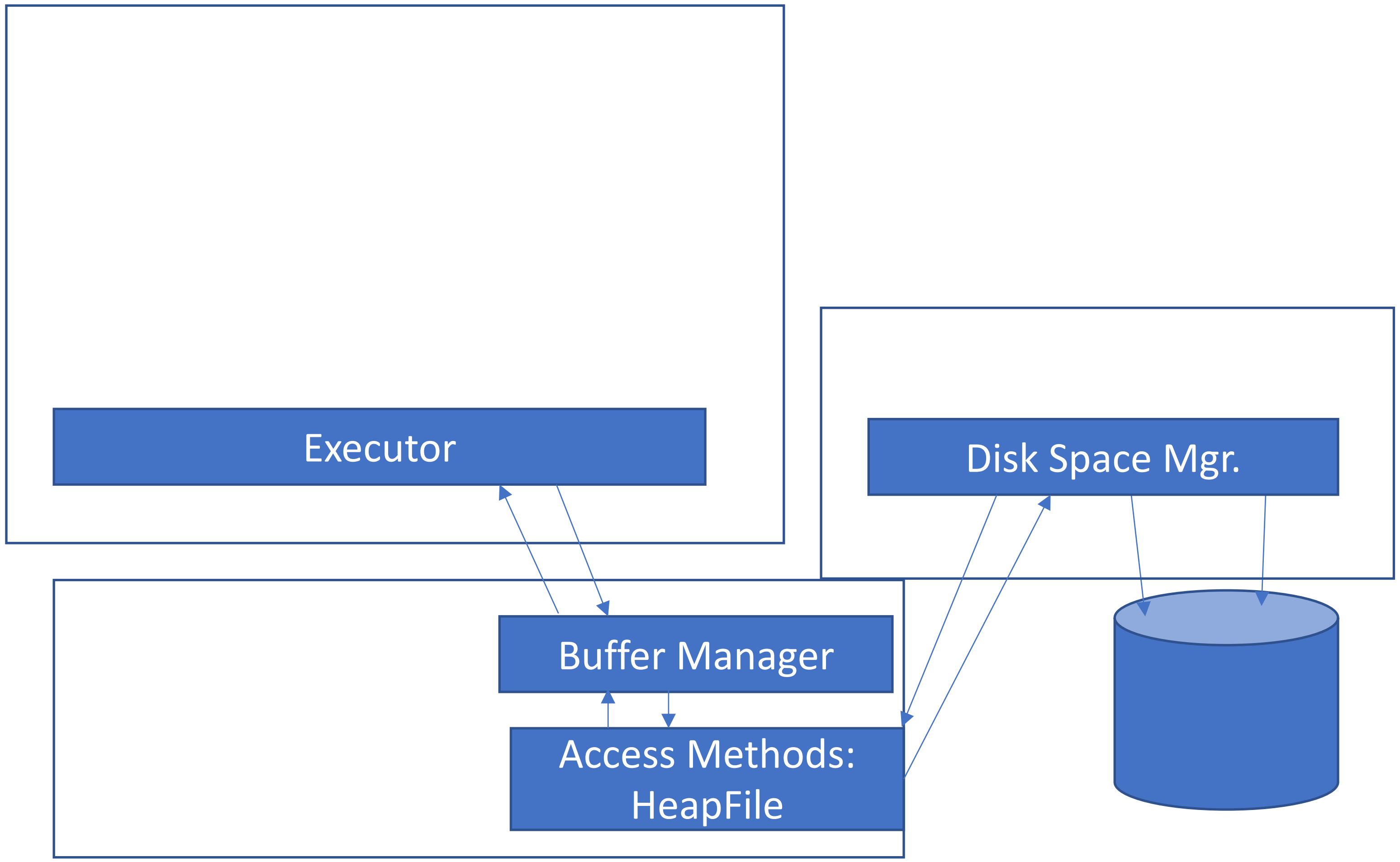
School of Computing

DePaul University

DBMS Architecture: Process Manager

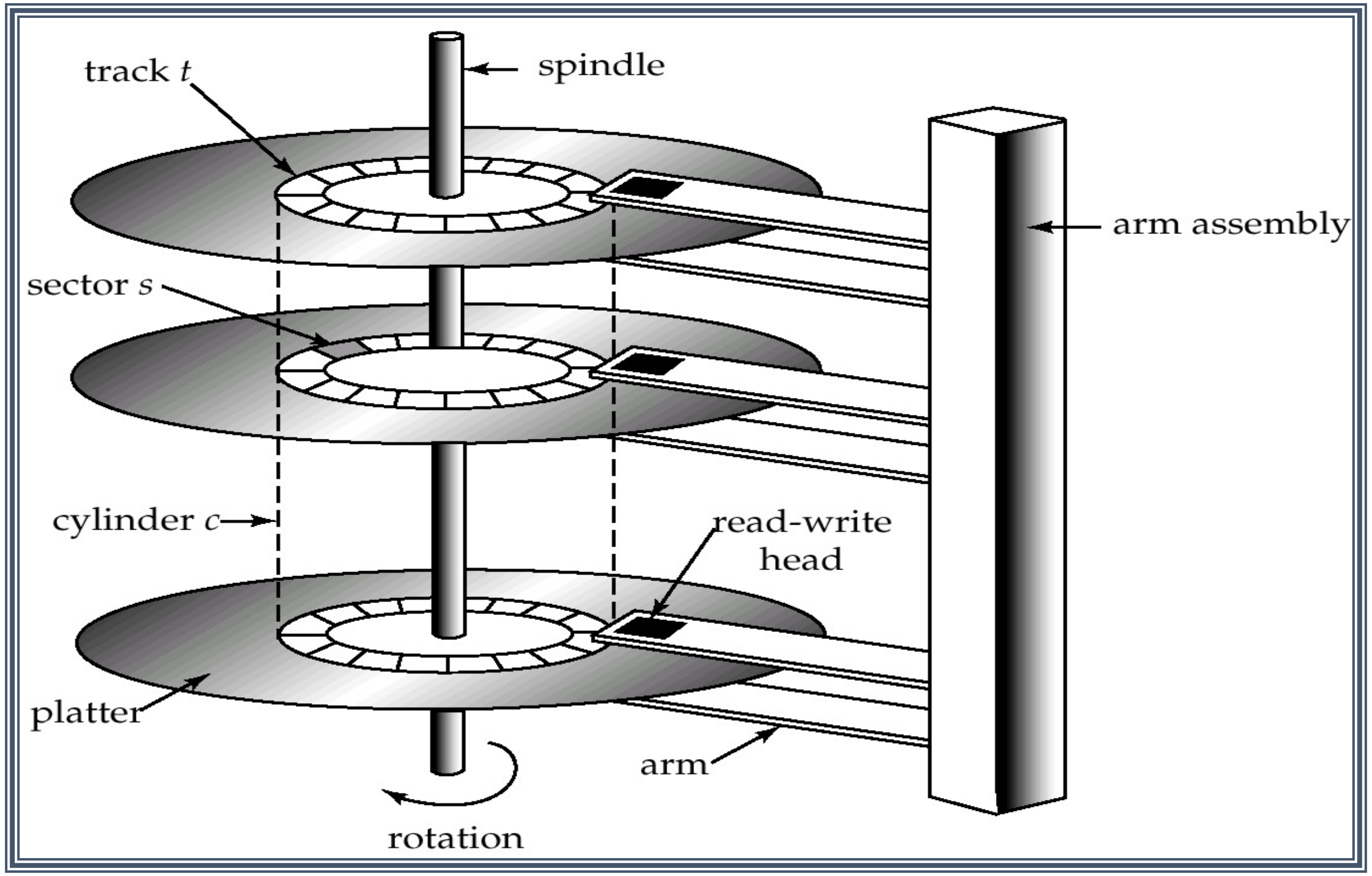


Storage Manager



- Process data with a simple query.
- Access methods: Organize data to support fast access to desired subsets of records.
- Buffer manager: Caches data in memory. Reads/writes data to/from disk as needed
- Disk-space manager: Allocates space on disk for files/access methods

Hard Disk Mechanism



Terminology

- **Sector** or **Block** – the smallest unit that can be read or written. Often 512 bytes.
- **Track** – all blocks that form a ring on a disk surface that can be read without moving the head.
- **Cylinder** – all tracks on all surfaces, one on top of another, that can be read without moving the head.

Disk Operation

To read (or write) data to the disk:

- The arm containing the read/write heads must be moved to the proper radius from the center.
- The system must wait for the data to rotate under the read head.
- The data is read as it passes under the read head.
- The data is checked and then passed to the I/O controller.

Disk times

- Seek time: Time taken to move the disk head to the track on which the desired block is located
- Rotational delay: Waiting time for the desired block to rotate under the disk head.
  - Time required for half a rotation on average
  - Usually less than seek time
- Transfer time: time to actually read or write the data in the block once the head is positioned.

Disk Time

- Reading or writing a disk block is an I/O operation.
- Time to read or write a block varies, depending on the location of the data:

Access time = seek time + rotational delay + transfer time

Database Storage

- Typically, each table/relation is stored in a separate *file* on the disk
- A file is a logical sequence of blocks.
- Each block consists of a collection of *records*, each corresponding to one row/tuple of the relational model.
- Each record consists of a collection of *fields*, each corresponding to one column/attribute defined in the CREATE TABLE statement.

Disk Manager

- Maps page number to disk block number.
- Must keep track of:
  - pages that are free (no data).
  - pages that have data.
- Operations: allocate pages, deallocate pages
- Design approaches: ?

OS Vs DB-native Disk Manager

- Which is better?

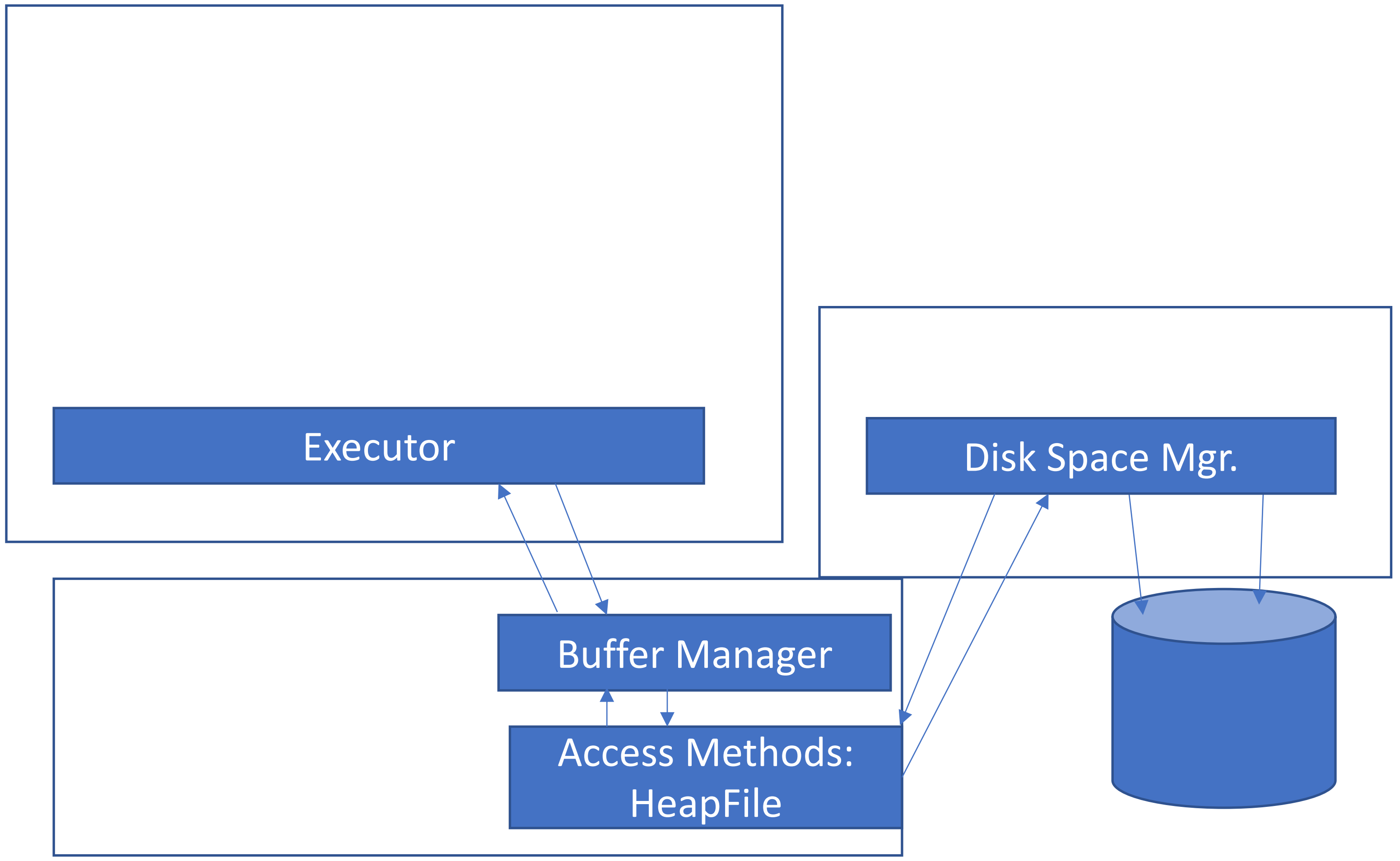
OS Vs DB-native Disk Manager

- Which is better?
  - Tied to specific OS interface (portability issues
  - File size limits; DB may want to implement files spanning disks as well
  - Most importantly, dictated by the Buffer Manager.

Buffer Manager

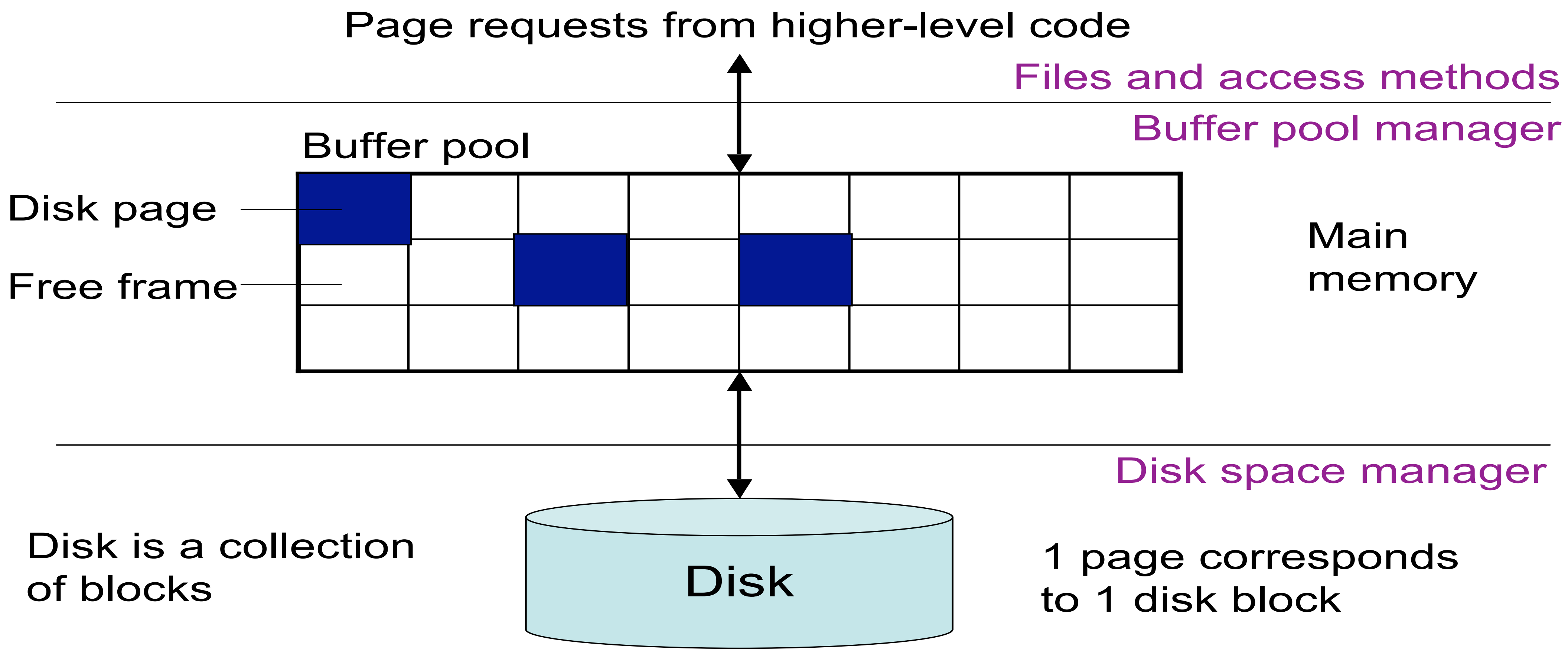
- Keep pages in a part of memory (buffer), read directly from there
- What happens if you bring a new page into buffer and buffer is full: you have to evict one page
- Replacement policy:
  - LRU : Least Recently Used (CLOCK)
  - MRU: Most Recently Used
  - Toss-immediate : remove a page if you know that you will not need it again
  - Pinning (for recovery, index based processing, etc)

Storage Manager

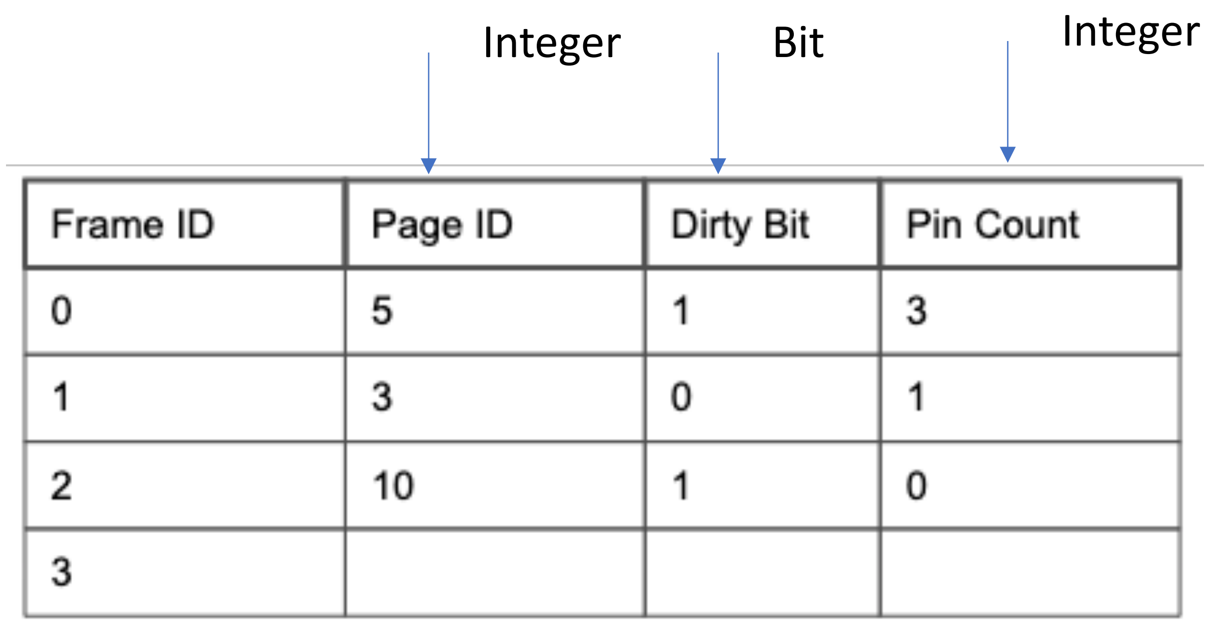


- Process data with a simple query.
- Access methods: Organize data to support fast access to desired subsets of records.
- Buffer manager: Caches data in memory. Reads/writes data to/from disk as needed
- Disk-space manager: Allocates space on disk for files/access methods

Buffer Manager



Data structures in Buffer Manager


Integer
Bit
Integer

Frame ID	Page ID	Dirty Bit	Pin Count
0	5	1	3
1	3	0	1
2	10	1	0
3			

- Frame ID that is uniquely associated with a memory address
- Page ID for determining which page of a table a frame currently contains
- Dirty Bit for verifying whether or not a page has been modified
- Pin Count for tracking the number of requestors currently using a page

Cache Replacement Policy

- If the page does not exist in the buffer pool and there is still space, the next empty frame is found and the page is read into that frame. The page’s pin count is set to 1 and the page’s memory address is returned.
- In the case where the page does not exist and there are no empty frames left, a replacement policy must be used to determine which page to evict.

Cache Replacement Policy

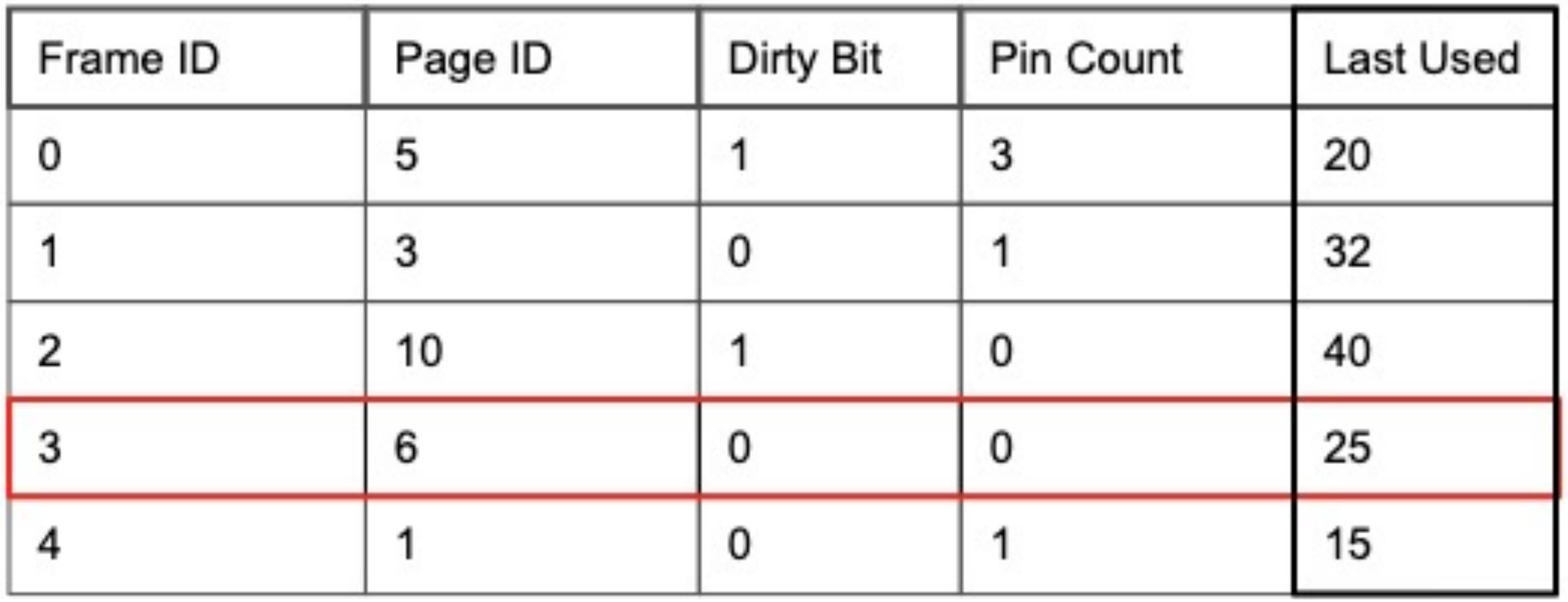
- Which next page will be read?
  - Cannot determine apriori
  - Depends on page access patterns
- What is an optimal policy?

Measure of a Cache Replacement Policy

- A **page hit** is when a requested page can be found in memory without having to go to disk.
- Each **page miss** incurs an additional IO cost.
- The hit rate for an access pattern is defined as : # of page hits / # of page accesses.

Least Recently Used

- Commonly used
- When new pages need to be read into a full buffer pool, the least recently used unpinned page which has pin count = 0 and the lowest value in the Last Used column is evicted.



Frame ID	Page ID	Dirty Bit	Pin Count	Last Used
0	5	1	3	20
1	3	0	1	32
2	10	1	0	40
3	6	0	0	25
4	1	0	1	15

Example

- Buffer size of 3
- 5,6,7,3,4,5,6,2,3,4,8,3,4,4

- Bad example of LRU?

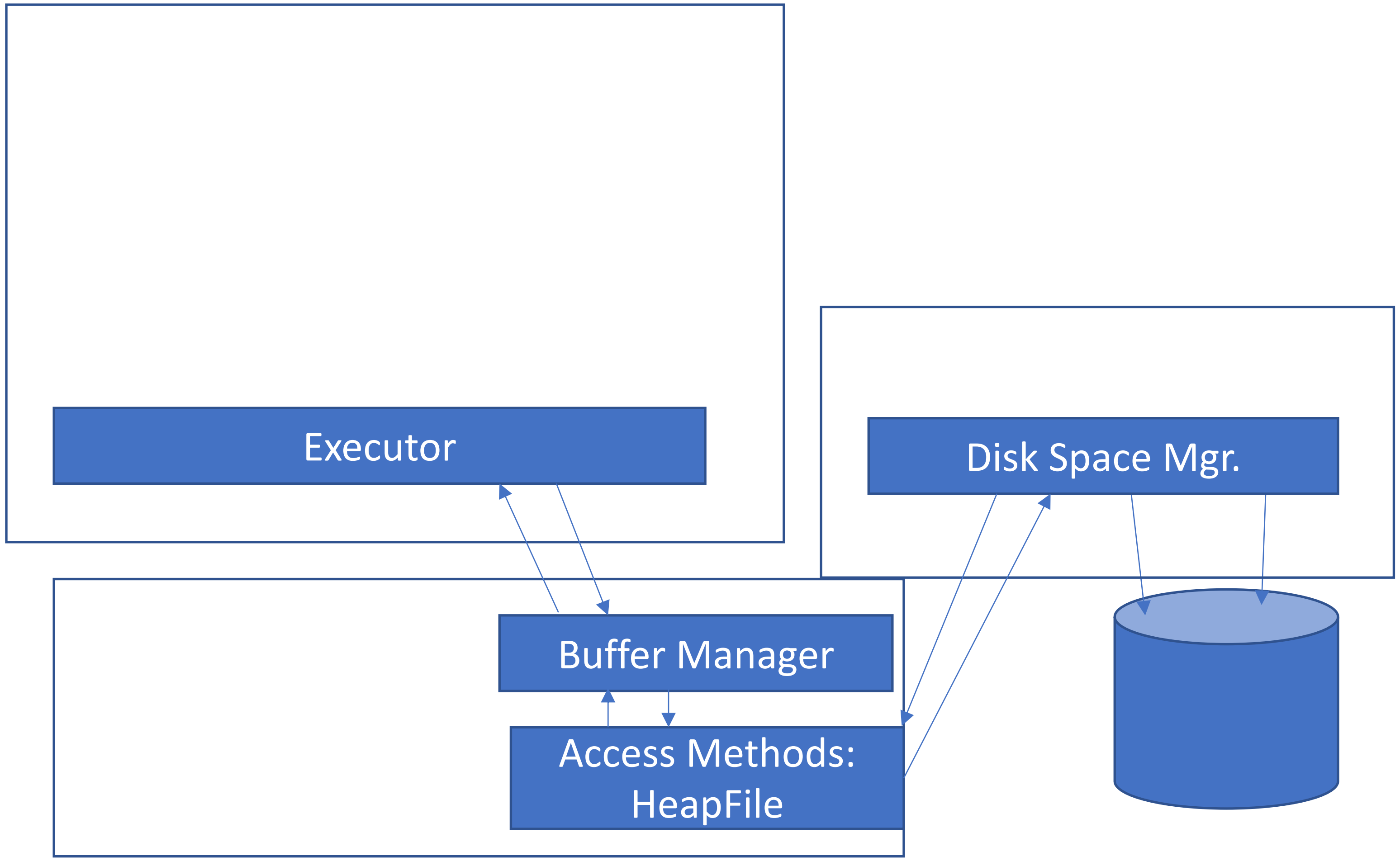
Cache Replacement Policy

- If the page does not exist in the buffer pool and there is still space, the next empty frame is found and the page is read into that frame. The page’s pin count is set to 1 and the page’s memory address is returned.
- In the case where the page does not exist and there are no empty frames left, a replacement policy must be used to determine which page to evict.
  - If the dirty bit is set write the evicted page to disk
- In either case, bring the new page (overwrite the existing page) and increment pin count

Few other considerations

- What if no page in buffer pool has pin count of 0 and a page not in the BufferManager is requested?
- What if different transactions attempt to modify the same page?
- Why not use OS buffer management?

Storage Manager



- Process data with a simple query.
- Access methods: Organize data to support fast access to desired subsets of records.
- Buffer manager: Caches data in memory. Reads/writes data to/from disk as needed
- Disk-space manager: Allocates space on disk for files/access methods

Unordered Files (Heap File)

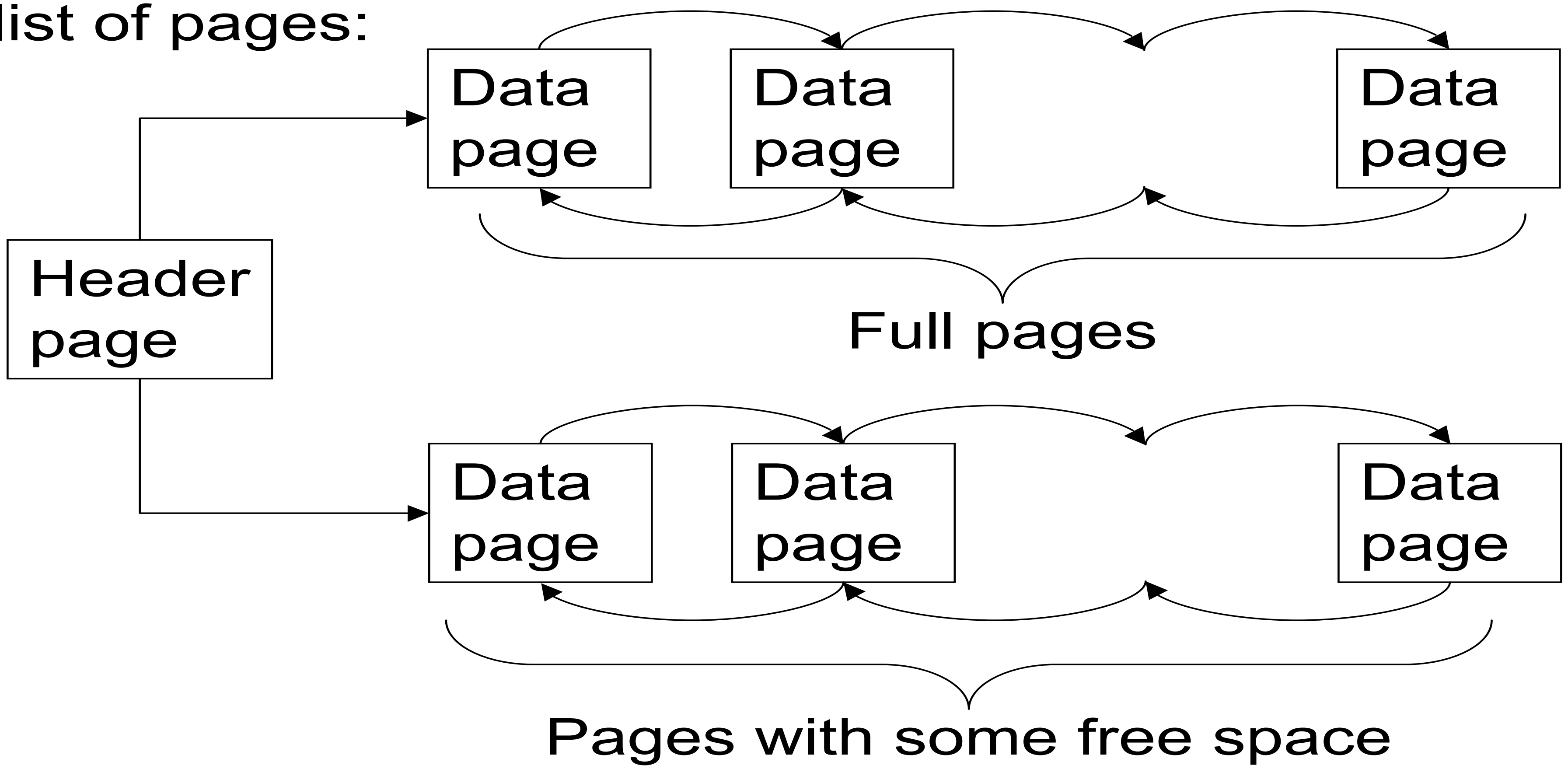
- No particular order maintained on the records
- Pages no control anyways.
- Unordered collection of records
- Each record in a heap file has a unique record identifier (rid)
- Typically, RID = (PageID, SlotNumber) <p,n>
  - Can identify disk address of page containing record by using rid

Unordered Files (Heap File)

- No particular order maintained on the records
  - Insertions done at the end of the file (O(1))
  - Deletions can be done efficiently provided you know the row to delete. Then (O(1))
    - move last element in file to replace deleted element
  - Searching for a record needs linear search (O(n))
    - n/2 records read on average, n in worst case
  - Updating a record may be costly also (O(n))
    - O(1) if you do not have to search for the record

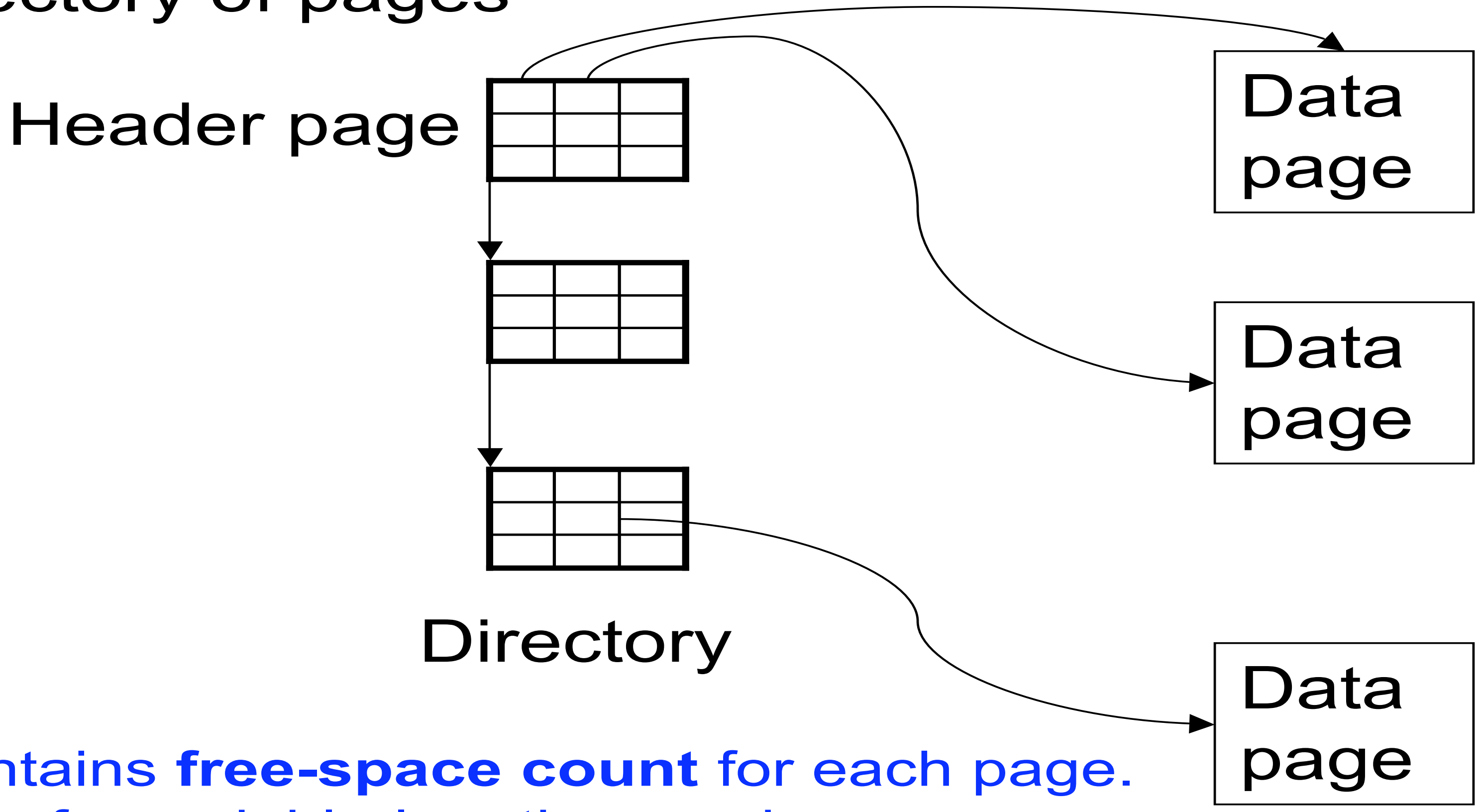
Heap File Implementation 1

Linked list of pages:



Heap File Implementation 2

Better: directory of pages



Directory contains **free-space count** for each page.
Faster inserts for variable-length records

Record Representation

- Fixed-Length Records
  - Example: Account(account-number char(10), branch-name char(20), balance real)

Each record is 38 bytes.

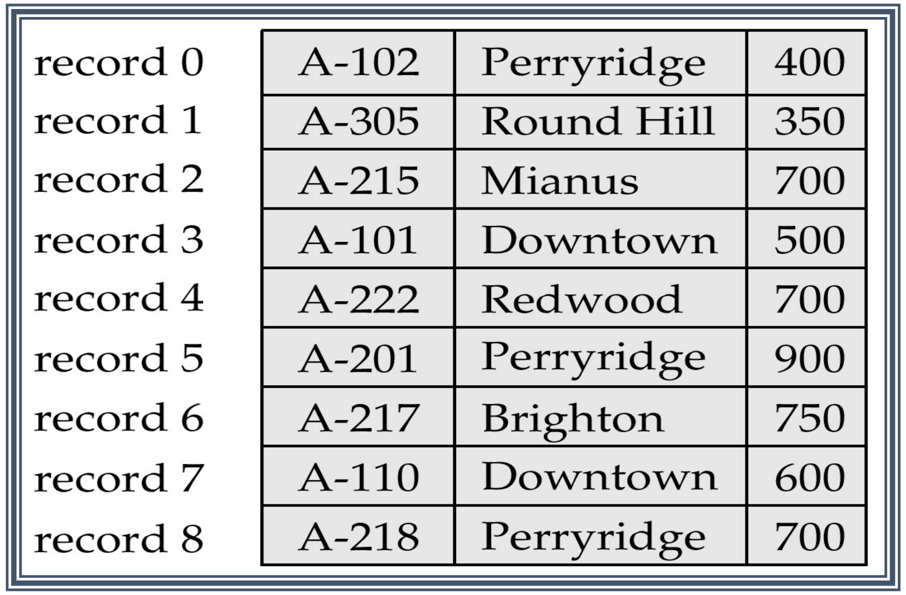
Store them sequentially, one after the other

Record1 at position 0,

Record2 at position 38,

Record3 at position 76, etc.

Table size/Compactness - (~350 bytes)



record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

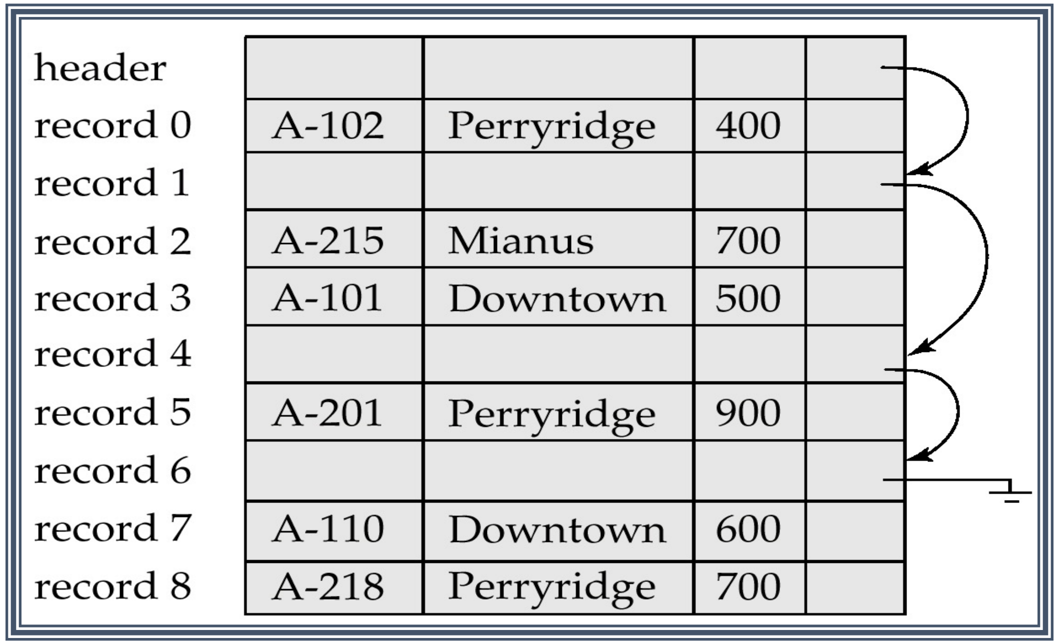
Fixed-Length Records

- Store record *i* starting from byte $n * (i - 1)$,
  where *n* is the size of each record.
- Record access is simple but records may cross blocks
  - Modification: do not let records cross block boundaries

- Insertion of record i: Add at the end
- Deletion of record i*: Two alternatives:*
  - move records:
    1. $i + 1, \ldots, n$ to $i, \ldots, n - 1$
    2. record *n* to *i*
  - do not move, but link free records on a *free list*

Free Lists

- A 2nd approach: FLRecords with Free Lists
- Store the address of the first deleted record in the file header.
- Use the first record to store the address of the second deleted record, and so on
- Can think of these stored addresses as pointers since they “point” to the location of a record.



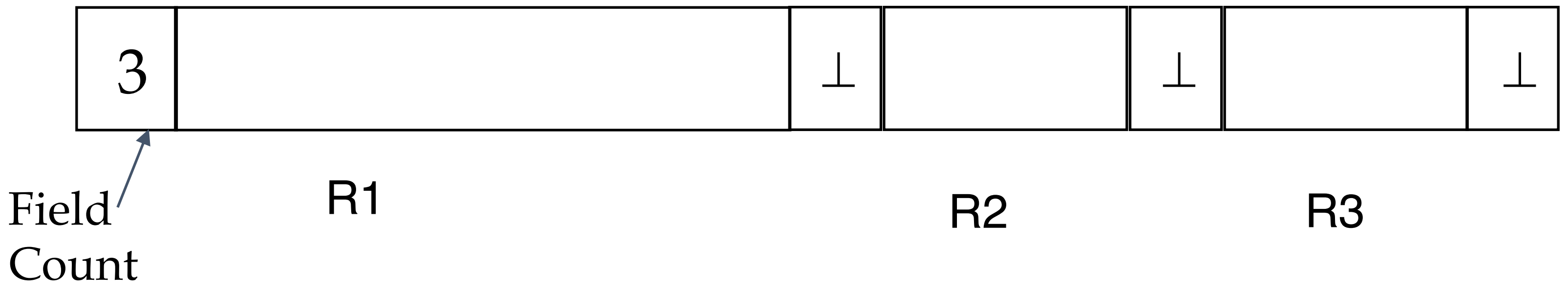
header				
record 0	A-102	Perryridge	400	
record 1				
record 2	A-215	Mianus	700	
record 3	A-101	Downtown	500	
record 4				
record 5	A-201	Perryridge	900	
record 6				
record 7	A-110	Downtown	600	
record 8	A-218	Perryridge	700	

Better handling of insert/delete

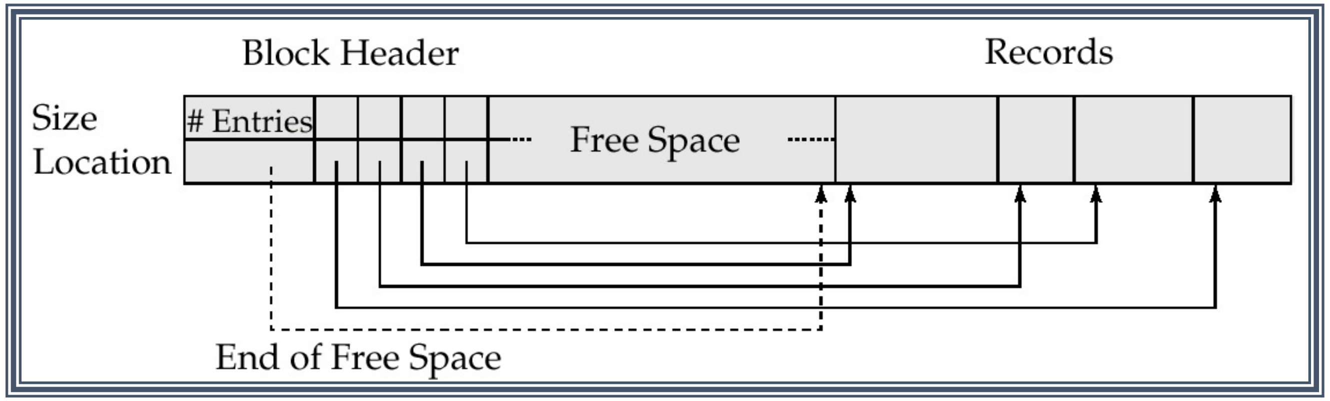
Less compact

Variable-Length Records

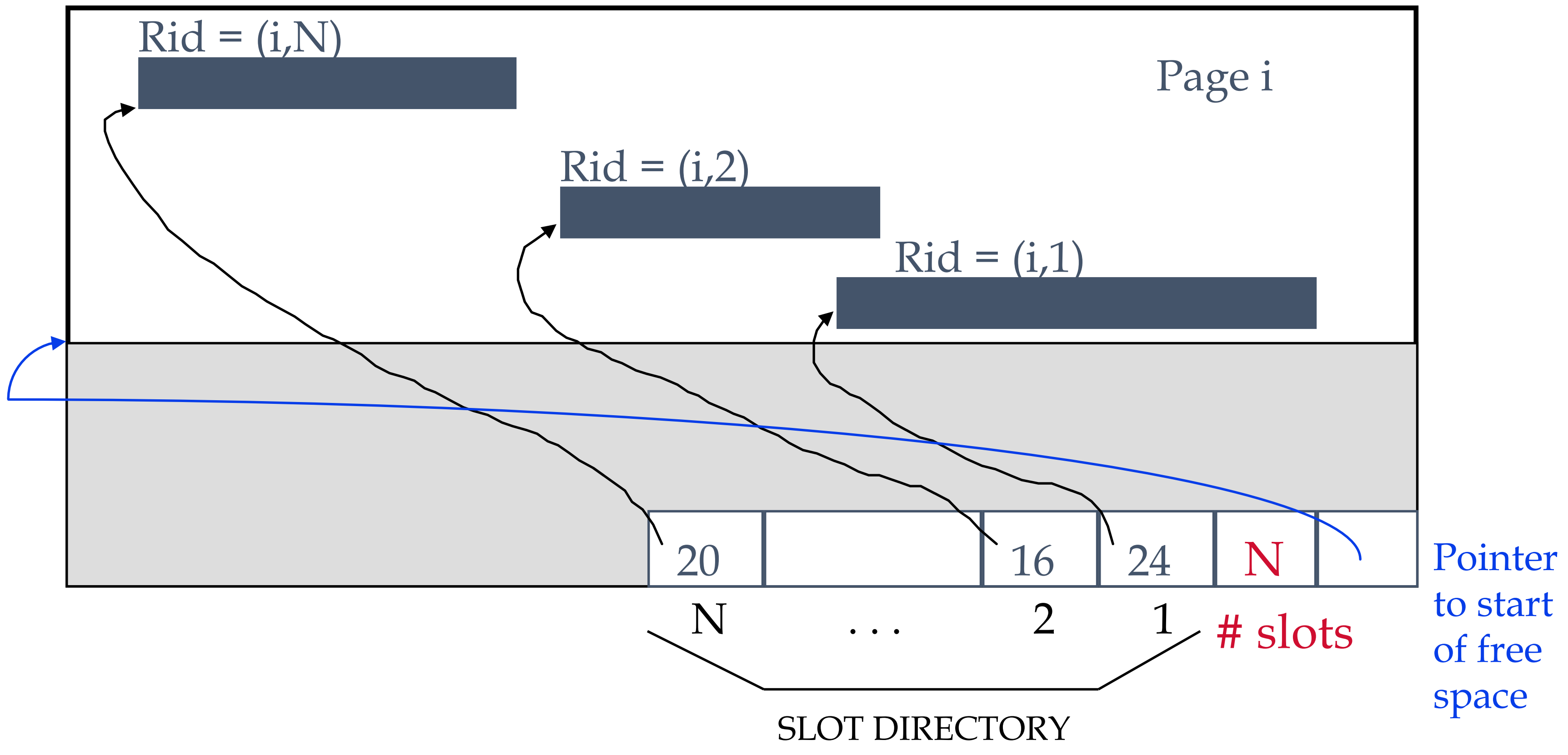
- 3rd approach: Variable-length records
  - Storage of multiple record types in a file.
  - Record types that allow variable lengths
- Byte string representation
  - Attach an *end-of-record* (⊥) control character to the end of each record
  - Difficulty with deletion (leaves holes)
  - Difficulty with growth



Variable-Length Records: Slotted Page Structure



- 4th approach VLRrecords-SP
- Slotted page header contains:
  - number of record entries
  - end of free space in the block
  - location and size of each record
- Records stored at the bottom of the page
- External tuple pointers point to record pointers: rec-id = <page-id, slot#>

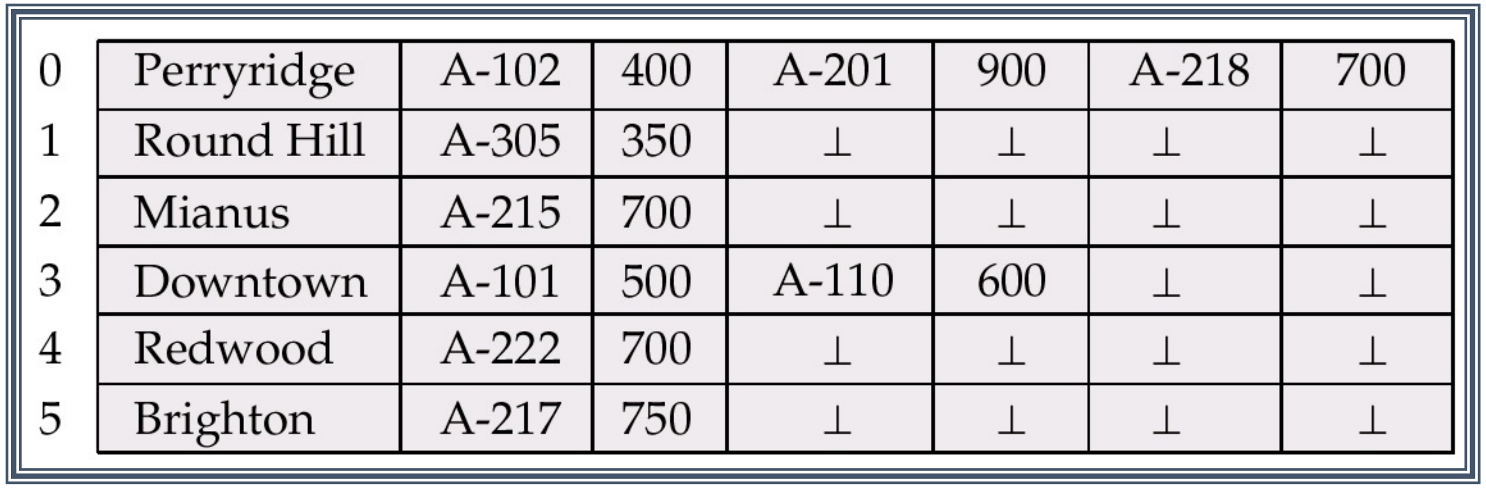


Insertion: 1) Use Free Space Pointer (FP) to find space and insert
2) Find available ptr in the directory (or create a new one)
3) adjust FP and number of records

Deletion ?

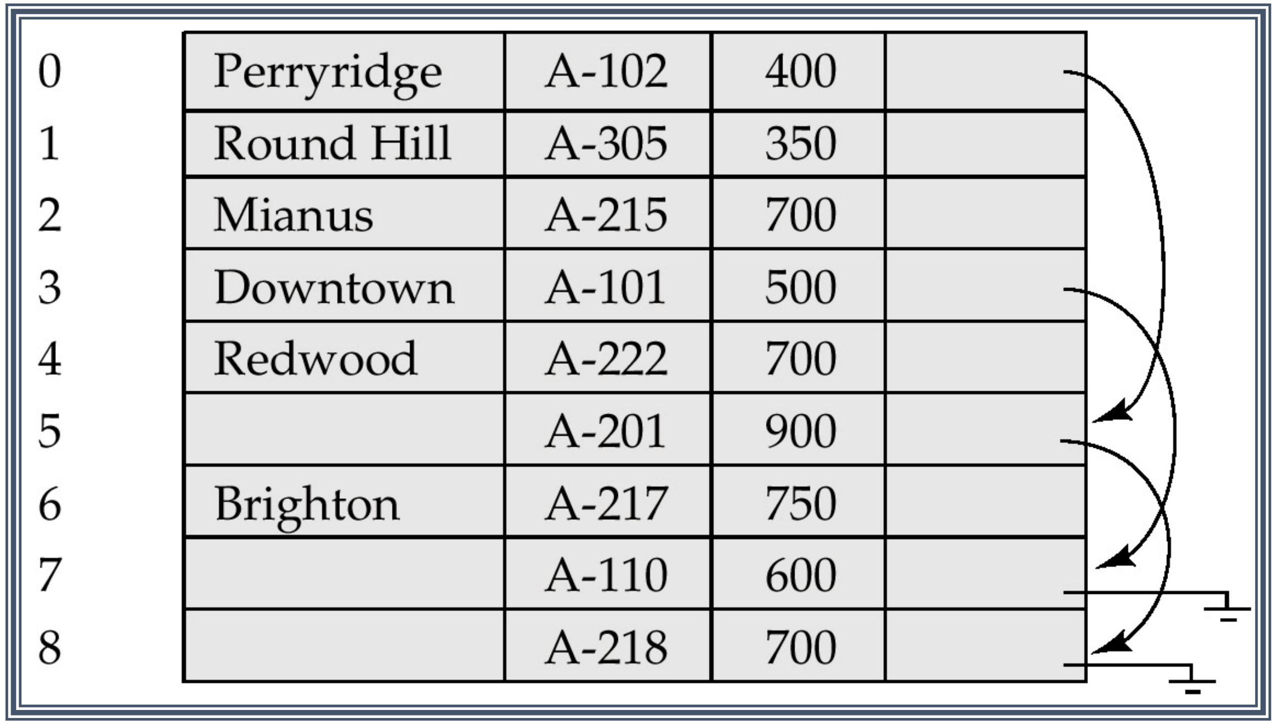
Variable-Length Records (Cont.)

- Fixed-length representation:
  - reserved space
  - pointers
- 5th approach: Fixed Limit Records (for VLRrecords)
- Reserved space – can use fixed-length records of a known maximum length; unused space in shorter records filled with a null or end-of-record symbol.



0	Perryridge	A-102	400	A-201	900	A-218	700
1	Round Hill	A-305	350	⊥	⊥	⊥	⊥
2	Mianus	A-215	700	⊥	⊥	⊥	⊥
3	Downtown	A-101	500	A-110	600	⊥	⊥
4	Redwood	A-222	700	⊥	⊥	⊥	⊥
5	Brighton	A-217	750	⊥	⊥	⊥	⊥

Pointer Method

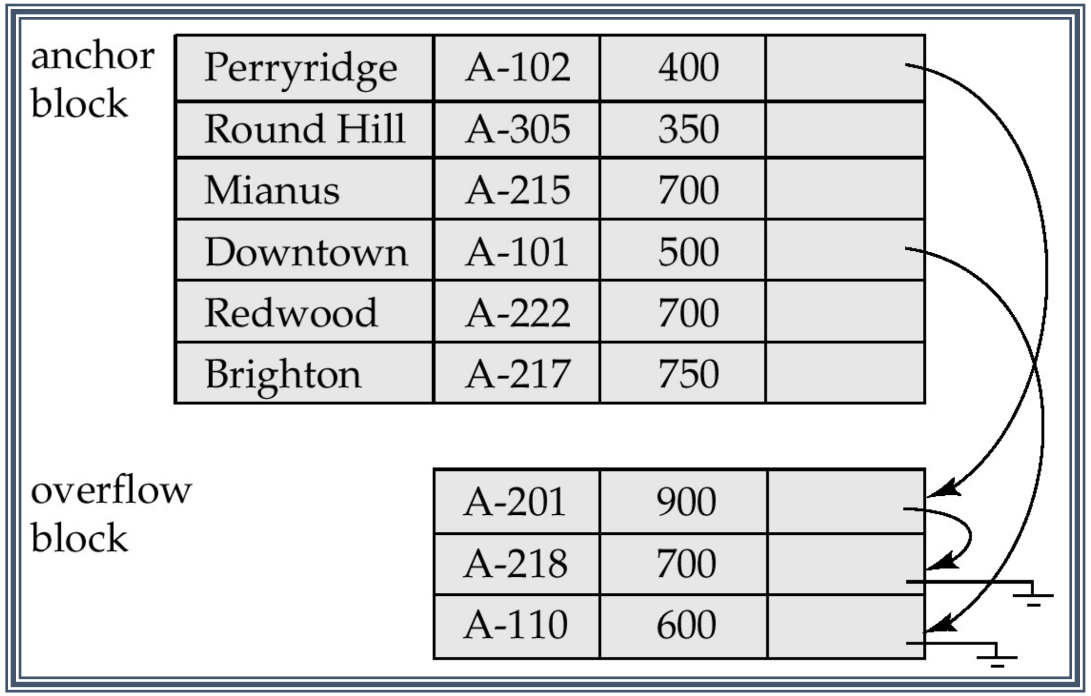


0	Perryridge	A-102	400	
1	Round Hill	A-305	350	
2	Mianus	A-215	700	
3	Downtown	A-101	500	
4	Redwood	A-222	700	
5		A-201	900	
6	Brighton	A-217	750	
7		A-110	600	
8		A-218	700	

- 6th approach: Pointer method
- Pointer method
  - A variable-length record is represented by a list of fixed-length records, chained together via pointers.
  - Can be used even if the maximum record length is not known

Pointer Method (Cont.)

- Disadvantage to pointer structure; space is wasted in all records except the first in a chain
- Solution is to allow two kinds of block in file:
  - Anchor block – contains the first records of chain
  - Overflow block – contains records other than those that are the first records of chains.



Perryridge	A-102	400	
Round Hill	A-305	350	
Mianus	A-215	700	
Downtown	A-101	500	
Redwood	A-222	700	
Brighton	A-217	750	

A-201	900	
A-218	700	
A-110	600	

Data Dictionary Storage

Data dictionary (also called system catalog) stores metadata: that is, *data about data,* such as:

- Information about relations
    - names of relations
    - names and types of attributes of each relation
    - names and definitions of views
    - integrity constraints
- User and accounting information, including passwords
- Statistical and descriptive data
    - number of tuples in each relation
- Physical file organization information
    - How relation is stored (sequential/hash/…)
    - Physical location of relation
        - operating system file name or
        - disk addresses of blocks containing records of the relation
- Information about indices

Data dictionary storage

- Stored as tables
- E-R diagram
  - Relations, attributes, domains
  - Each relation has name, some attributes
  - Each attribute has name, length and domain
  - Also, views, integrity constraints, indices
  - User info (authorizations, etc.)
  - statistics

Data Dictionary Storage (Cont.)

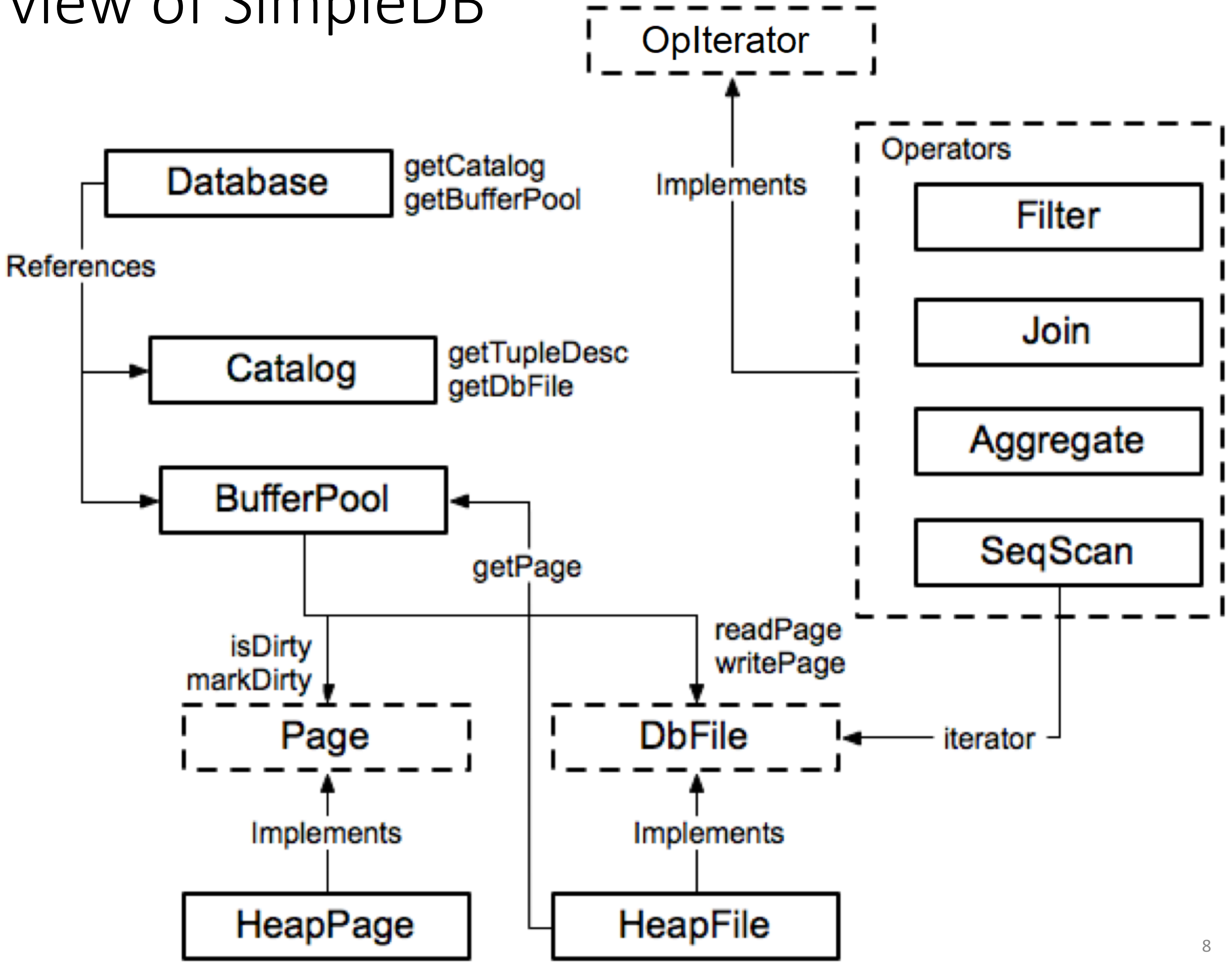
- A possible catalog representation:

Relation-metadata = (<u>relation-name</u>, number-of-attributes,
storage-organization, location)
Attribute-metadata = (<u>attribute-name, relation-name</u>, domain-type,
position, length)
User-metadata = (<u>user-name</u>, encrypted-password, group)
Index-metadata = (<u>index-name, relation-name</u>, index-type,
index-attributes)
View-metadata = (<u>view-name</u>, definition)

Example

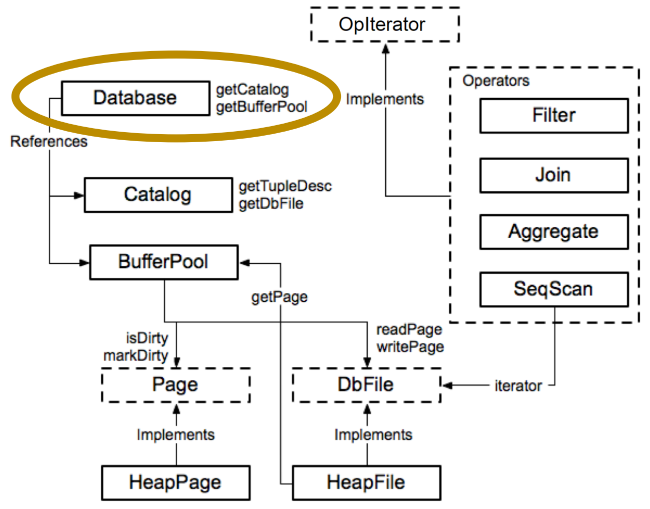
- Consider a disk with a sector size of 1024 bytes, 2000 tracks per surface, 50 sectors per track, five double-sided platters, and average seek time of 10 msec. Suppose that a file containing 100,000 records of 100 bytes each is to be stored on such a disk and that no record is allowed to span two blocks. Dis rotates at 5400 rpm.
- 1. How many records fit onto a block?
- 2. How many blocks are required to store the entire file?
- 3. If the file is arranged sequentially on disk, how many surfaces are needed?
- 4. How many records can you store worth 100 bytes?
- 5. Time to sequentially read?
- 6. Time to read each block in a random order? Assume that each block request incurs the average seek time and rotational delay.

Overview of SimpleDB

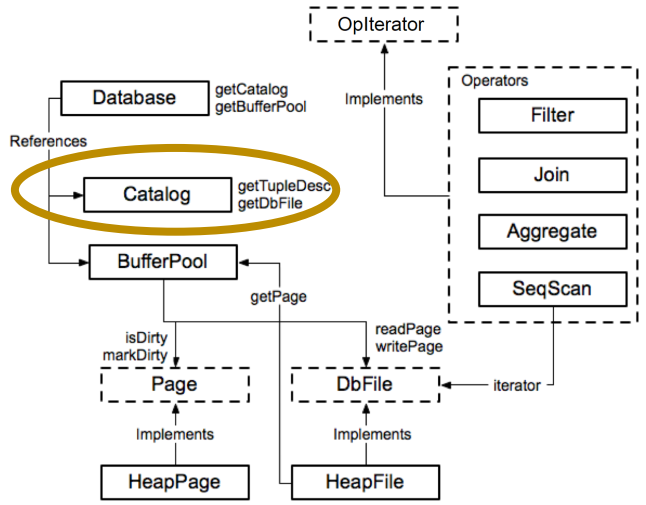


Database

- A single database
  - One schema
  - List of tables
- References to major components
  - Global instance of Catalog
  - Global instance of BufferPool

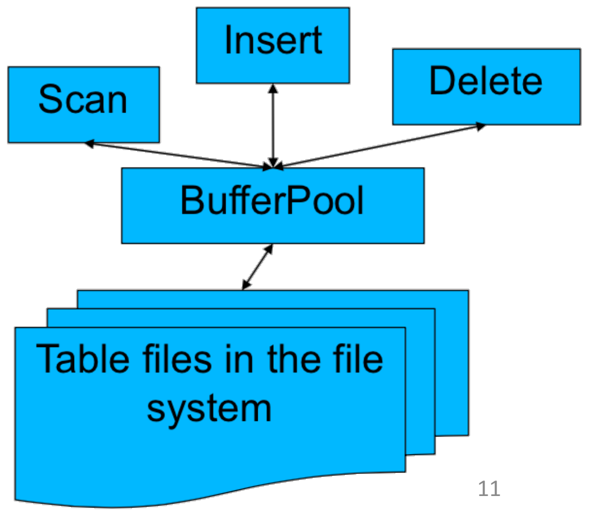


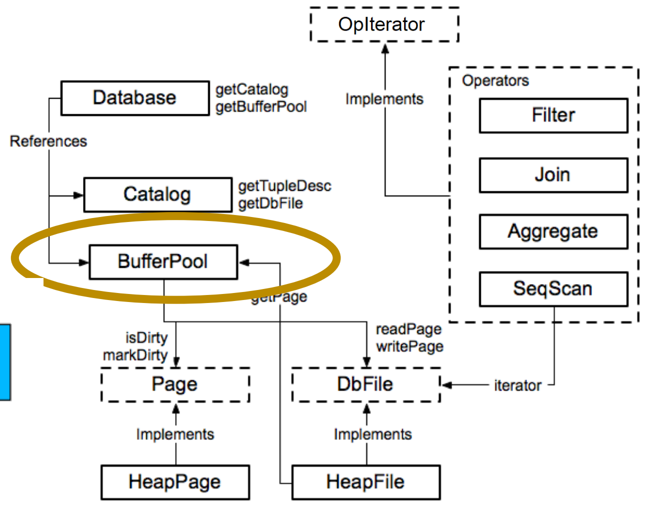
Catalog



- Stores metadata about tables in the database
  - void addTable(DbFile d, TupleDesc d)
  - DbFile getTable(int tableid)
  - TupleDesc getTupleDesc(int tableid) •...
- NOT persisted to disk
  - Catalog info is reloaded every time SimpleDB starts up

- The ONLY bridge between data-processing operators and actual data files
- Strict interface for physical independence!
- Data files are never accessed directly
- Later labs:
  - Locking for transactions
  - Flushing pages for recovery



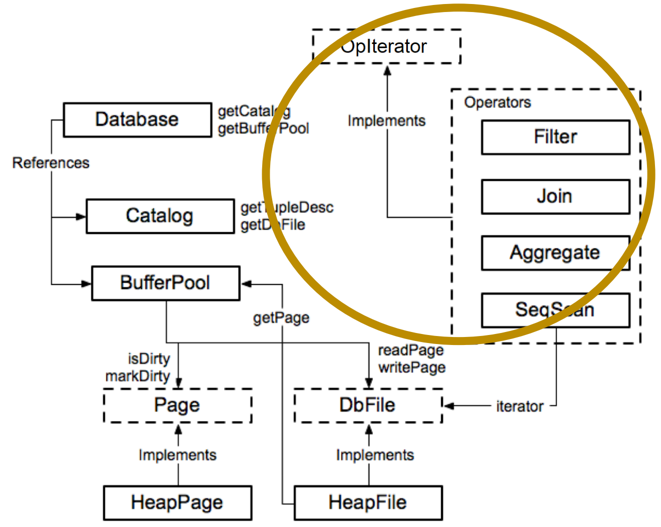


Data Types

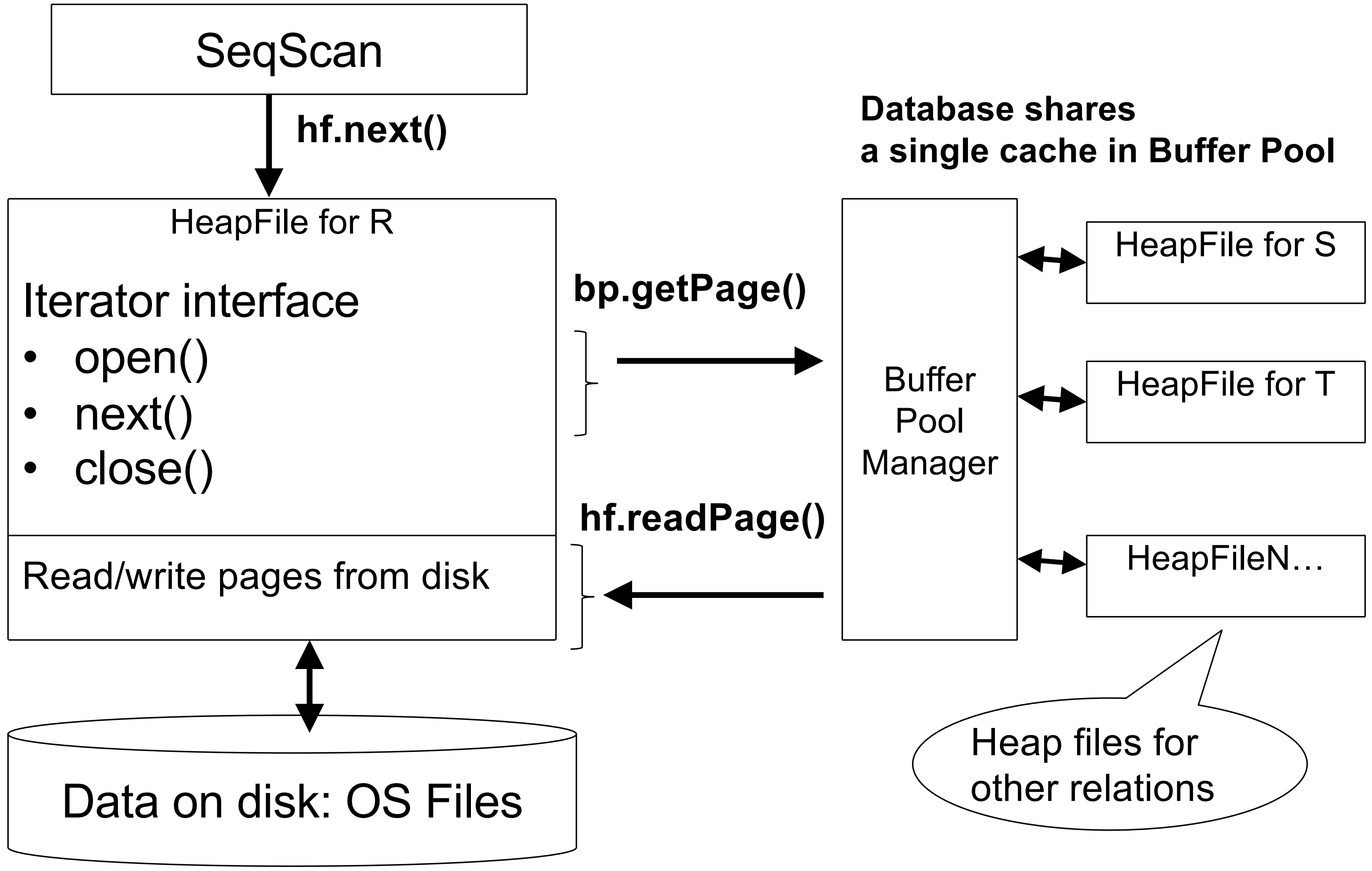
- Integer:
  - Type.INT_TYPE
  - 4 byte width
- Fixed-length Strings
  - Type.STRING_TYPE
  - 128 bytes long (Type.STRING_LEN)
  - Do not change this constant!

OpIterator

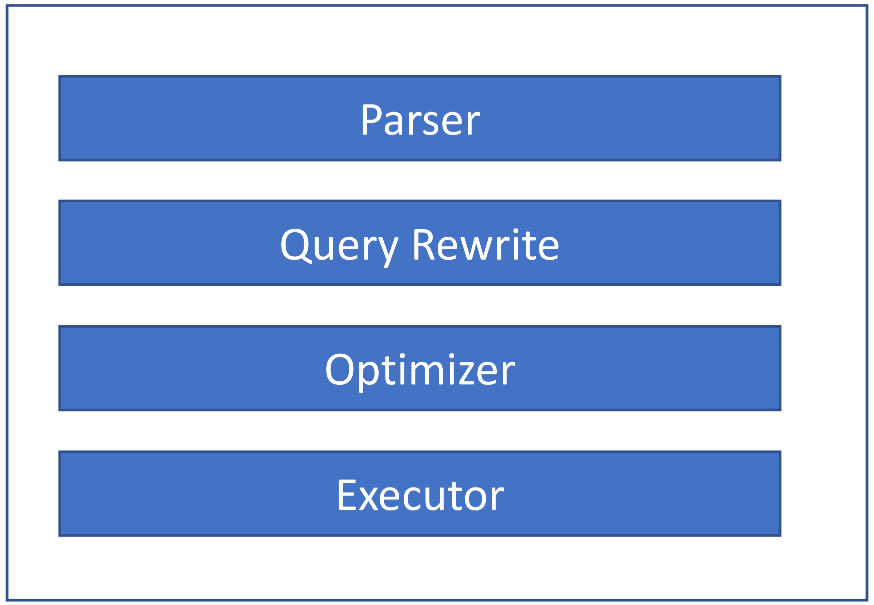
- Ancestor class for all operators
  - Join, Project, SeqScan, etc...
- Each operator has methods:
  - open(), close(), getTupleDesc(), hasNext(), next(), rewind()
- Iterator model: chain iterators together



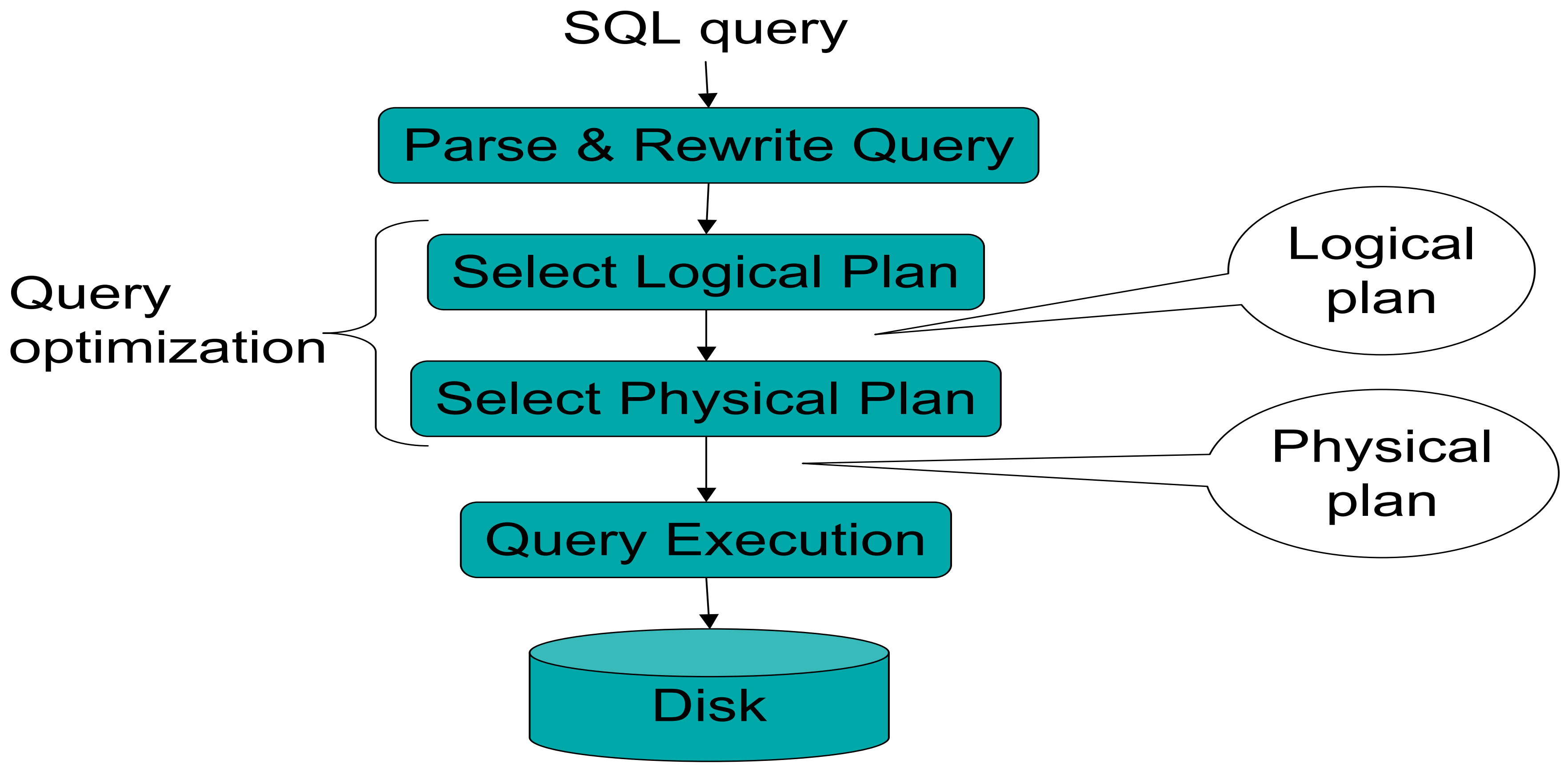
SimpleDB Architecture



Query Execution



Query Evaluation Steps



SQL Review

Relational Algebra

- Relational algebra (RA) is a query language for the relational model with a solid theoretical foundation.
- Relational algebra is not visible at the user interface level (not in any commercial RDBMS, at least).
- However, almost any RDBMS uses RA to represent queries internally (for query optimization and execution).
- Knowledge of relational algebra will help in understanding SQL and relational database systems in general.

Algebra equivalence

- In mathematics, an algebra is a
  - set (the carrier), and
  - operations that are closed with respect to the set.
  - Example: (N, {$*$, +}) forms an algebra.
- In case of RA,
  - the carrier is the set of all finite relations.

Bank Database Schema

Account		
bname	acct_no	balance

Branch		
bname	bcity	assets

Depositor	
cname	acct_no

Borrower	
cname	lno

Customer		
cname	cstreet	ccity

Loan		
bname	lno	amt

Bank Database

Account

bname	acct_no	balance
Downtown	A-101	500
Mianus	A-215	700
Perry	A-102	400
R.H.	A-305	350
Brighton	A-201	900
Redwood	A-222	700
Brighton	A-217	750

Depositor

cname	acct_no
Johnson	A-101
Smith	A-215
Hayes	A-102
Turner	A-305
Johnson	A-201
Jones	A-217
Lindsay	A-222

Customer

cname	cstreet	ccity
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stanford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stanford

Branch

bname	bcity	assets
Downtown	Brooklyn	9M
Redwood	Palo Alto	2.1M
Perry	Horseneck	1.7M
Mianus	Horseneck	0.4M
R.H.	Horseneck	8M
Pownel	Bennington	0.3M
N. Town	Rye	3.7M
Brighton	Brooklyn	7.1M

Borrower

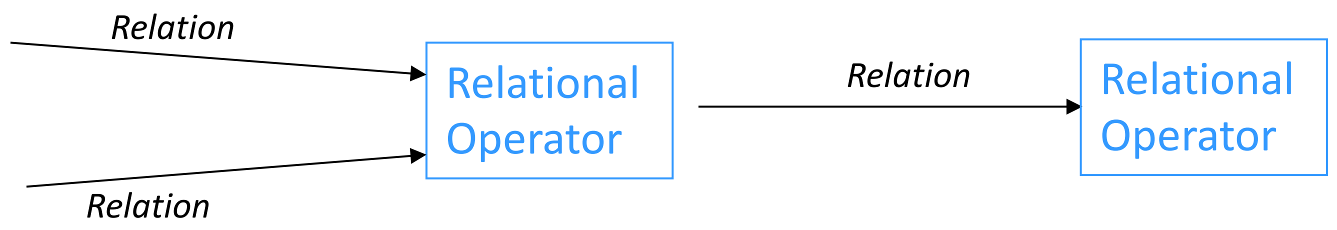
cname	lno
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Smith	L-11
Williams	L-17
Adams	L-16

Loan

bname	lno	amt
Downtown	L-17	1000
Redwood	L-23	2000
Perry	L-15	1500
Downtown	L-14	1500
Mianus	L-93	500
R.H.	L-11	900
Perry	L-16	1300

Relational Algebra

- Basic Operators
  1. select (σ)
  2. project (π)
  3. union (∪)
  4. set difference (–)
  5. cartesian product (×)
  6. rename (ρ)
- Closure Property



Select (σ)

Notation: $\sigma_{predicate}$ (*Relation*)

Relation: *Can be name of table or result of another query*

Predicate:

1. Simple
   - $attribute_1$ = $attribute_2$
   - attribute = constant value *(also: ≠, <, >, ≤, ≥)*

2. Complex
   - predicate AND predicate
   - predicate OR predicate
   - NOT predicate

Idea:

Select rows from a relation based on a predicate

Select (σ)

Notation: $\sigma_{predicate}(Relation)$

$\sigma_{\text{bcity = "Brooklyn"}}(\text{branch}) =$

bname	bcity	assets
Downtown	Brooklyn	9M
Brighton	Brooklyn	7.1M

$\sigma_{\text{assets > \$8M}}(\sigma_{\text{bcity = "Brooklyn"}}(\text{branch})) =$

bname	bcity	assets
Downtown	Brooklyn	9M

(same as $\sigma_{\text{assets > \$8M AND bcity = "Brooklyn"}}(\text{branch})$)

Project (π)

Notation: $\pi_{A1,\ \ldots,\ An}$ (*Relation*)

- Relation: name of a table or result of another query
- Each A_i is an attribute
- Idea: π selects columns (vs. σ which selects rows)

$\pi_{\text{cstreet, ccity}}$ (customer) $=$

cstreet	ccity
Main	Harrison
North	Rye
Park	Pittsfield
Putnam	Stanford
Nassau	Princeton
Spring	Pittsfield
Alma	Palo Alto
Sand Hill	Woodside
Senator	Brooklyn
Walnut	Stanford

Project (π)

$\pi_{\text{bcity}}(\sigma_{\text{assets} > 5M}(\text{branch})) =$

bcity
Brooklyn
Horseneck

Question: Does the result of Project always have the same number of tuples as its input?

Union (∪)

Notation: $\mathit{Relation}_1 \cup \mathit{Relation}_2$

R ∪ S valid only if:

1. *R, S have same number of columns (arity)*
2. *R, S corresponding columns have same domain (compatibility)*

Example:

$$(\pi_{\text{cname}}(\text{depositor})) \cup (\pi_{\text{cname}}(\text{borrower})) =$$

cname
Johnson
Smith
Hayes
Turner
Jones
Lindsay
Jackson
Curry
Williams
Adams

Schema:

Depositor	
cname	**acct_no**

Borrower	
cname	**lno**

Set Difference (–)

Notation: *Relation$_1$* - *Relation$_2$*

R - S valid only if:

1. *R, S have same number of columns (arity)*
2. *R, S corresponding columns have same domain (compatibility)*

Example:

$$(\pi_{bname}(\sigma_{amount \geq 1000}(loan))) - (\pi_{bname}(\sigma_{balance < 800}(account))) =$$

bname	lno	amount
Downtown	L-17	1000
Redwood	L-23	2000
Perry	L-15	1500
Downtown	L-14	1500
Perry	L-16	1300

–

bname	acct_no	balance
Mianus	A-215	700
Brighton	A-201	900
Redwood	A-222	700
Brighton	A-217	850

=

bname
Downtown
Perry

Cartesian Product (×)

Notation: $Relation_1 \times Relation_2$

R × S like cross product for mathematical relations:

- *every tuple of R appended to every tuple of S*

Example:

depositor × borrower =

depositor. cname	acct_no	borrower. cname	lno
Johnson	A-101	Jones	L-17
Johnson	A-101	Smith	L-23
Johnson	A-101	Hayes	L-15
Johnson	A-101	Jackson	L-14
Johnson	A-101	Curry	L-93
Johnson	A-101	Smith	L-11
Johnson	A-101	Williams	L-17
Johnson	A-101	Adams	L-16
Smith	A-215	Jones	L-17
…	…	…	…

How many tuples in the result?

*A: depositor (7) * borrower (8) = 56*

Rename (ρ)

Notation: $\rho_{\text{identifier}}$ (*Relation*)

renames a relation, or

Notation: $\rho_{\text{identifier}_0 (\text{identifier}_1, \ldots, \text{identifier}_n)}$ (*Relation*)

renames relation and columns of n-column relation

Use:

massage relations to make $\cup$, $-$ valid, or $\times$ more readable

Rename (ρ)

Notation: $\rho_{\text{identifier}_0 (\text{identifier}_1, \ldots, \text{identifier}_n)}$ (*Relation*)

Example:

$\rho_{\text{res (dcname, acctno, bcname, lno)}}$ (depositor × borrower) =

dccname	**acctno**	**bcname**	**lno**
Johnson	A-101	Jones	L-17
Johnson	A-101	Smith	L-23
Johnson	A-101	Hayes	L-15
Johnson	A-101	Jackson	L-14
Johnson	A-101	Curry	L-93
Johnson	A-101	Smith	L-11
Johnson	A-101	Williams	L-17
Johnson	A-101	Adams	L-16
Smith	A-215	Jones	L-17
…	…	…	…

Example Query in RA

- *Determine **lno** for loans that are for an amount that is larger than the amount of some other loan. (i.e. **lno** for all non-minimal loans)*

Can do in steps:

$\text{Temp}_1 \leftarrow \ldots$

$\text{Temp}_2 \leftarrow \ldots\ \text{Temp}_1\ \ldots$

…

Example Query in RA

1. Find the base data we need

$$\text{Temp}_1 \leftarrow \pi_{\text{lno,amt}}\ (\text{loan})$$

lno	amt
L-17	1000
L-23	2000
L-15	1500
L-14	1500
L-93	500
L-11	900
L-16	1300

2. Make a copy of (1)

$$\text{Temp}_2 \leftarrow \rho_{\text{Temp2 (lno2,amt2)}}\ (\text{Temp}_1)$$

lno2	amt2
L-17	1000
L-23	2000
L-15	1500
L-14	1500
L-93	500
L-11	900
L-16	1300

Example Query in RA

3. Take the cartesian product of 1 and 2

$$Temp_3 \leftarrow Temp_1 \times Temp_2$$

lno	amt	lno2	amt2
L-17	1000	L-17	1000
L-17	1000	L-23	2000
…	…	…	…
L-17	1000	L-16	1300
L-23	2000	L-17	1000
L-23	2000	L-23	2000
…	…	…	…
L-23	2000	L-16	1300
…	…	…	…

Example Query in RA

4. Select non-minimal loans

$\text{Temp}_4 \leftarrow \sigma_{\text{amt} > \text{amt2}} (\text{Temp}_3)$

5. Project on lno

$\text{Result} \leftarrow \pi_{\text{lno}} (\text{Temp}_4)$

… or, if you prefer…

- $\pi_{\text{lno}} (\sigma_{\text{amt} > \text{amt2}} (\pi_{\text{lno,amt}} (\text{loan}) \times (\rho_{\text{Temp2 (lno2,amt2)}} (\pi_{\text{lno,amt}} (\text{loan})))))$

Review

Express the following query in the RA:

Find the names of customers who have both accounts and loans

$T_1 \leftarrow \rho_{T1\ (cname2,\ lno)}\ (borrower)$

$T_2 \leftarrow depositor \times T_1$

$T_3 \leftarrow \sigma_{cname\ =\ cname2}\ (T_2)$

$Result \leftarrow \pi_{\ cname}\ (T_3)$

Above sequence of operators (ρ, ×, σ) very common.

Motivates additional (redundant) RA operators.

Relational Algebra

Redundant Operators

1. Natural Join ($\bowtie$)

2. Generalized Projection (π)
3. Outer Joins (⟗ ⟕ ⟖)
4. Update ($\leftarrow$) (we've already been using)

- *Redundant: Above can be expressed in terms of minimal RA*

  $\rightarrow$ *e.g. depositor* $\bowtie$ *borrower* =

  $\pi \ldots(\sigma\ldots(\text{depositor} \times \rho\ldots(\text{borrower})))$

- *Added for convenience*

Natural Join

Notation: $Relation_1 \bowtie Relation_2$

Idea: combines ρ, ×, σ

r

A	B	C	D
1	α	+	10
2	α	-	10
2	α	-	20
3	β	+	10

$\bowtie$

s

E	B	D
‘a’	α	10
‘a’	α	20
‘b’	β	10
‘c’	β	10

=

A	B	C	D	E
1	α	+	10	‘a’

depositor $\bowtie$ borrower

$$\equiv$$

$$\pi_{cname,acct\_no,lno}\,(\sigma_{cname=cname2}\,(depositor \times \rho_{t(cname2,lno)}\,(borrower)))$$

Generalized Projection

Notation: $\pi_{e_1,\ldots,e_n}(\mathit{Relation})$

$e_1, \ldots, e_n$ can include arithmetic expressions – not just attributes

Example

credit =

cname	limit	balance
Jones	5000	2000
Turner	3000	2500

Then…

$\pi_{\text{cname, limit - balance}}(\text{credit}) =$

cname	limit-balance
Jones	3000
Turner	500

Outer Joins

Motivation:

loan =

bname	lno	amt
Downtown	L-170	3000
Redwood	L-230	4000
Perry	L-260	1700

borrower =

cname	lno
Jones	L-170
Smith	L-230
Hayes	L-155

$\text{loan} ⟕ \text{borrower} =$

bname	lno	amt	cname
Downtown	L-170	3000	Jones
Redwood	L-230	4000	Smith

Join result loses…

→ *any record of Perry*
→ *any record of Hayes*

Outer Joins

loan =

bname	lno	amt
Downtown	L-170	3000
Redwood	L-230	4000
Perry	L-260	1700

borrower =

cname	lno
Jones	L-170
Smith	L-230
Hayes	L-155

1. Left Outer Join (⟕)

- *preserves all tuples in left relation*

loan ⟕ borrower =

bname	lno	amt	cname
Downtown	L-170	3000	Jones
Redwood	L-230	4000	Smith
Perry	L-260	1700	$\perp$

$\perp$ = NULL

Outer Joins

loan =

bname	lno	amt
Downtown	L-170	3000
Redwood	L-230	4000
Perry	L-260	1700

borrower =

cname	lno
Jones	L-170
Smith	L-230
Hayes	L-155

1. Left Outer Join ($⟕$)

- *preserves all tuples in left relation*

loan $⟕$ borrower =

bname	lno	amt	cname
Downtown	L-170	3000	Jones
Redwood	L-230	4000	Smith
Perry	L-260	1700	$\perp$

$\perp$ = NULL

$$R ⟕ S \quad \equiv \quad (R \bowtie S) \cup \big((R - \pi_{A,B}(R \bowtie S)) \times \{(C\text{:null})\}\big)$$

Outer Joins

loan =

bname	lno	amt
Downtown	L-170	3000
Redwood	L-230	4000
Perry	L-260	1700

borrower =

cname	lno
Jones	L-170
Smith	L-230
Hayes	L-155

2. Right Outer Join ($⟖$)

- *preserves all tuples in* <u>*right*</u> *relation*

loan $⟖$ borrower =

bname	lno	amt	cname
Downtown	L-170	3000	Jones
Redwood	L-230	4000	Smith
$\perp$	L-155	$\perp$	Hayes

$\perp$ = NULL

Outer Joins

loan =

bname	lno	amt
Downtown	L-170	3000
Redwood	L-230	4000
Perry	L-260	1700

borrower =

cname	lno
Jones	L-170
Smith	L-230
Hayes	L-155

3. Full Outer Join ($⟗$)

- *preserves all tuples in <u>both</u> relations*

loan $⟗$ borrower =

bname	lno	amt	cname
Downtown	L-170	3000	Jones
Redwood	L-230	4000	Smith
Perry	L-260	1700	$\perp$
$\perp$	L-155	$\perp$	Hayes

$\perp$ = NULL

Update

Notation: *Identifier* $\leftarrow$ *Query*

Common Uses:

1. Deletion: $r \leftarrow r - s$

   e.g., $\text{account} \leftarrow \text{account} - \sigma_{\text{bname=Perry}}(\text{account})$

   (deletes all Perry accounts)

2. Insertion: $r \leftarrow r \cup s$

   e.g., $\text{branch} \leftarrow \text{branch} \cup \{(\text{Waltham, Boston, 7M})\}$

   (inserts new branch with
   bname = Waltham, bcity = Boston, assets = 7M)

   e.g., $\text{depositor} \leftarrow \text{depositor} \cup (\rho_{\text{temp (cname,acct\_no)}}(\text{borrower}))$

   (adds all borrowers to depositors, treating lno's as acct_no's)

3. Update: $r \leftarrow \pi_{e1,\ldots,en}(r)$

   e.g., $\text{account} \leftarrow \pi_{\text{bname,acct\_no,bal*1.05}}(\text{account})$

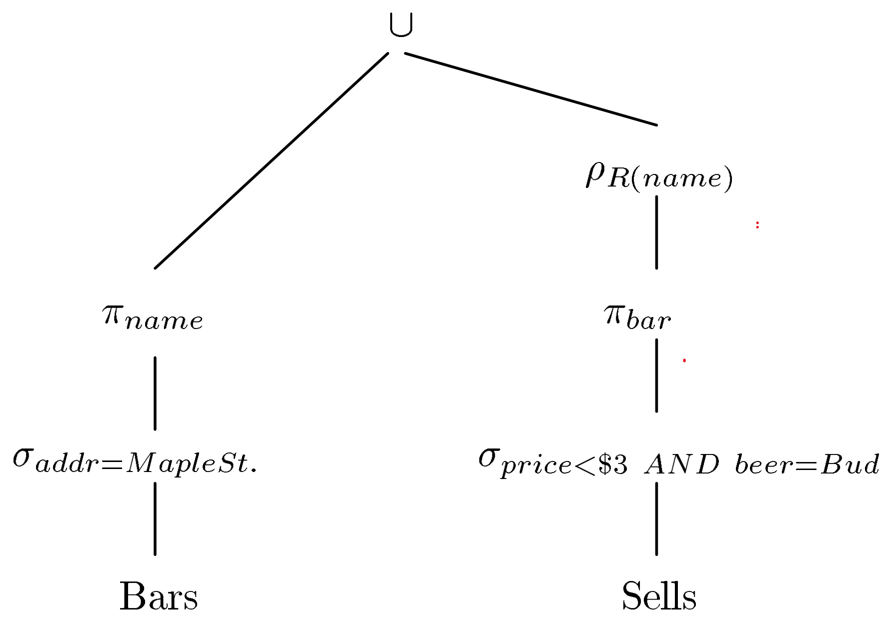
   (adds 5% interest to account balances)

Example 1

- Find the bars that are either on Maple street or sell Bud for less than \$3.
    - Sells(bar, beer, price)
    - Bars(name, addr)

Example 1

- Find the bars that are either on Maple street or sell Bud for less than \$3.
  - Sells(bar, beer, price)
  - Bars(name, addr)

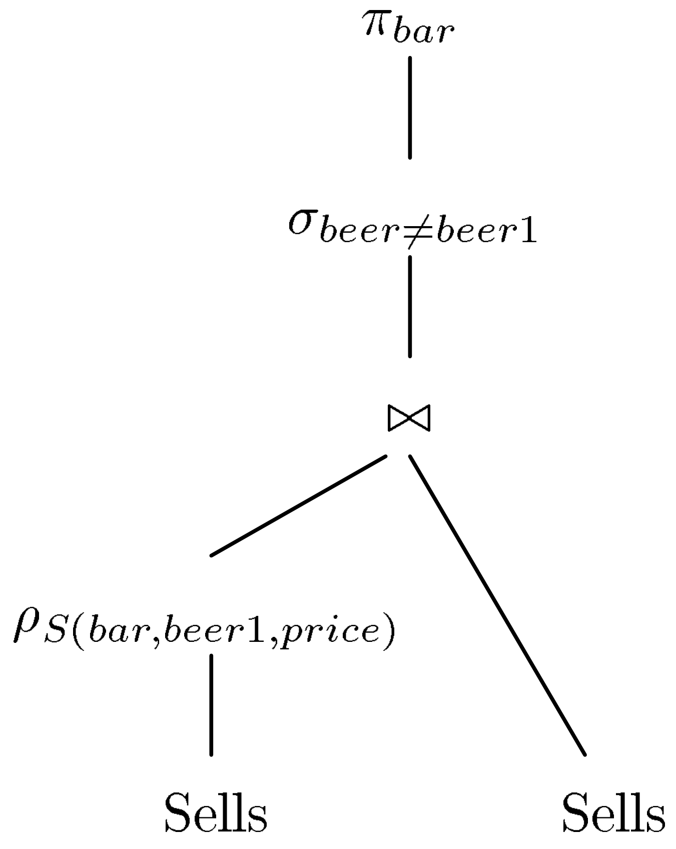


Example 2

- Find the bars that sell two different beers at the same price
    - Sells(bar, beer, price)

Example 2

- Find the bars that sell two different beers at the same price⍰
  - Sells(bar, beer, price)



Non RA operators

- Distinct (duplicate elimination)
- Order By (sort)
- Group By (aggregate)

Aggregate Functions and Operations

- An **aggregate function** takes a collection of values and returns a single value as a result.

avg: average value
min: minimum value
max: maximum value
sum: sum of values
count: number of values

- **Aggregate operation** in relational algebra

$${}_{G1,\, G2,\, \ldots,\, Gn}\, \mathcal{G}\, {}_{F1(A1),\, F2(A2), \ldots,\, Fn(An)}\, (E)$$

  - E is any relational-algebra expression
  - G_1, G_2 …, G_n is a list of attributes on which to group (can be empty)
  - Each F_i is an aggregate function
  - Each A_i is an attribute name

Aggregate Operation – Example

- Relation *r*:

A	*B*	*C*
α	α	7
α	β	7
β	β	3
β	β	10

$\mathcal{G}_{\text{sum(c)}}\text{(r)}$

sum-C
27

No grouping

Aggregate Operation – Example

- Relation *account* grouped by *branch-name*:

branch-name	*account-number*	*balance*
Perryridge	A-102	400
Perryridge	A-201	900
Brighton	A-217	750
Brighton	A-215	750
Redwood	A-222	700

$$_{branch\text{-}name}\,\mathcal{G}_{\,sum(balance)}\,(account)$$

branch-name	*sum(balance)*
Perryridge	1300
Brighton	1500
Redwood	700

Aggregate Functions (Cont.)

Result of aggregation does not have a name

- Can use rename operation to give it a name
- For convenience, we permit renaming as part of aggregate operation

$$_{\textit{branch-name}}\, \mathcal{G}\, _{\textit{sum(balance) as sum-balance}} (\textit{account})$$

Interator Interface

open()

next()

SeqScan

Operator at bottom of plan

open()

next()

In SimpleDB, SeqScan can find HeapFile in Catalog

Heap File Access Method

Offers iterator interface
open()
next()
close()
Knows how to read/write pages from disk

But if Heap File reads data directly from disk, it will not stay cached in Buffer Pool!

Iterator

- A group of four methods that allow a consumer of the result of physical operator to get the result one tuple at a time.

- Open(): starts the process of getting tuples
- hasNext(): determines if there is another tuple
- GetNext(): gets the next tuple and adjusts data structures to get the next tuple
- Close(): closes
- Which operator follows the Iterator interface: Table Scan Vs Sort

SQL to RA

- Product(pid, name, price)
- Purchase(pid, cid, store)
- Customer(cid, name, city)

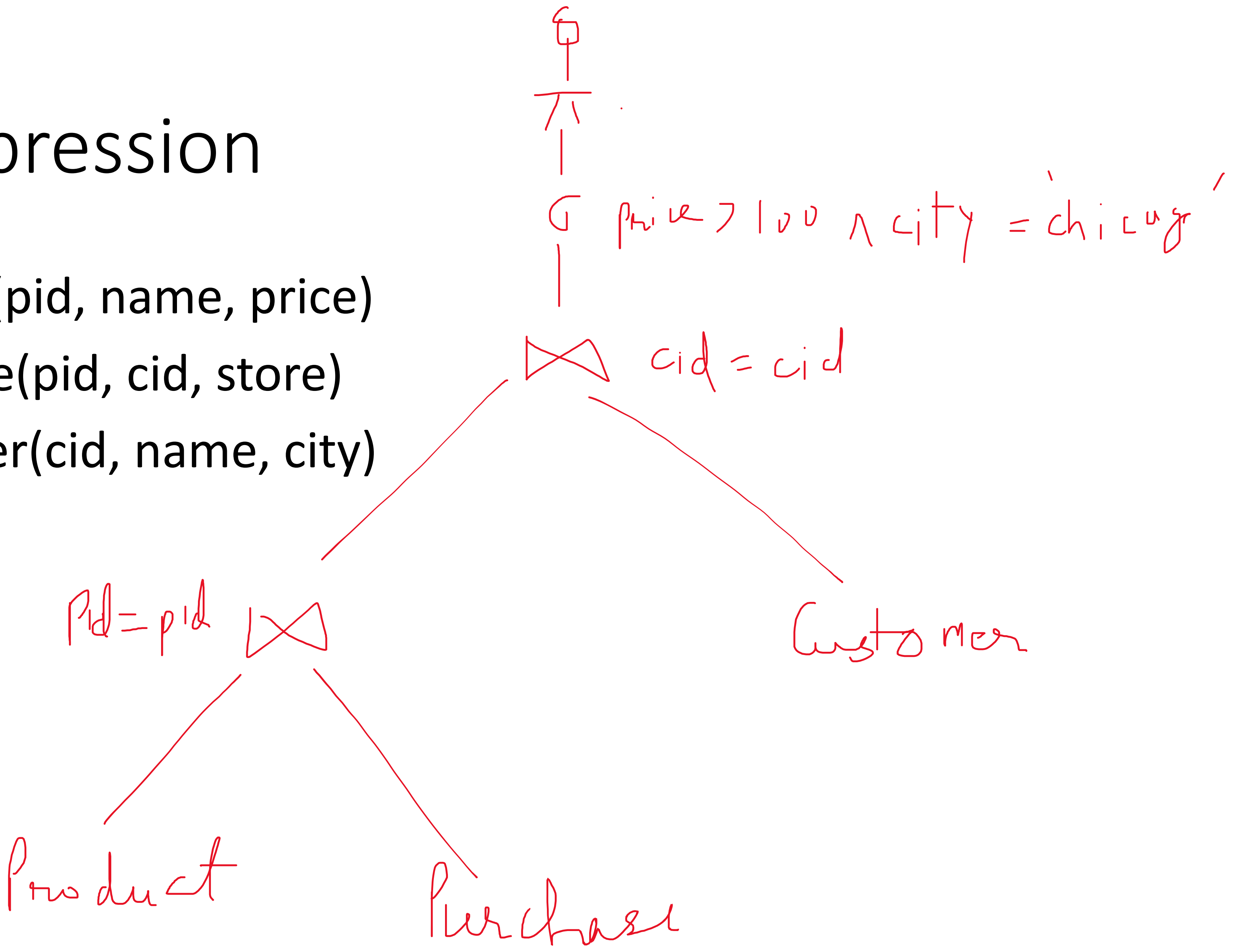
- Query:
- SELECT DISTINCT x.name, z.name

FROM Product x, Purchase y, Customer z

WHERE x.pid = y.pid AND y.cid = y.cid AND x.price > 100 AND z.city = 'Chicago'

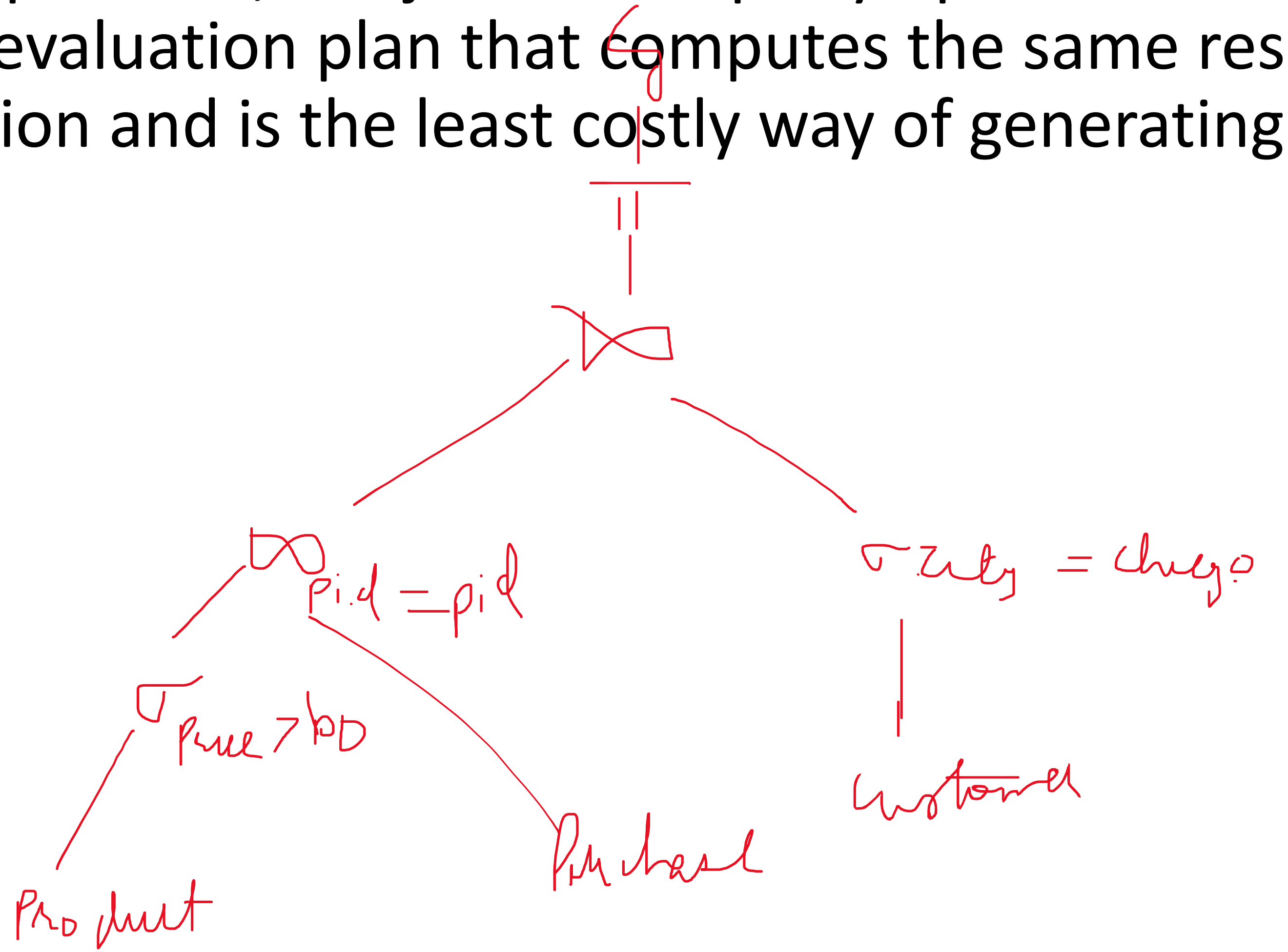
RA Expression

- Product(pid, name, price)
- Purchase(pid, cid, store)
- Customer(cid, name, city)



Query Execution

- Given a RA expression, the job of the query optimizer is to come up with a query evaluation plan that computes the same result as the given expression and is the least costly way of generating the result.



⋈
⋈ pid = pid
σ price > 100
σ city = chicago
Product
Purchase
Customer