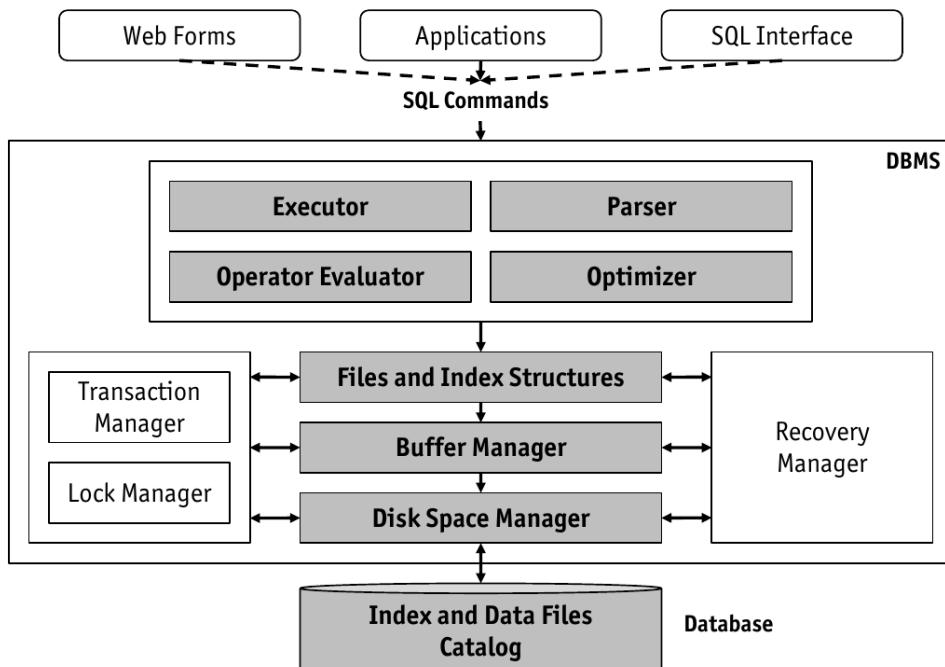


# DBMS Architecture



Topic	Exercise
<b>Introduction</b>	Disk Storage
<b>Disk and Buffer Management</b>	Disk Management
<b>File Management</b>	Buffer Management
<b>Tree-structured Indexing (I)</b>	
<b>Tree-structured Indexing (II)</b>	B+ Trees
<b>Hash-based Indexing</b>	Disk-based Hash Indexes
<b>Query Evaluation and Optimization</b>	
<b>External Sorting</b>	External Sorting
<b>Evaluating Relational Operators (I)</b>	
<b>Christmas Break</b>	
<b>Evaluating Relational Operators (II)</b>	Relational Operators
<b>Relational Algebra Equivalences</b>	
<b>Histograms</b>	Cost and Estimation
<b>Nested Queries</b>	Query Optimization
<b>Optimizer Architectures</b>	
<b>Physical Database Design and Tuning</b>	Flashback

# Contents

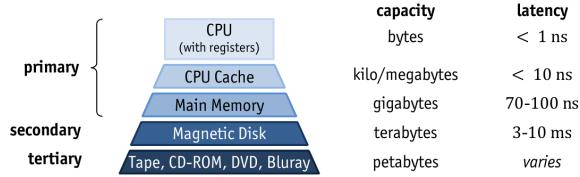
<b>1 Disk, Buffer &amp; File Management</b>	<b>3</b>
1.1 Hardware . . . . .	3
1.1.1 Memory Hierarchy: . . . . .	3
1.1.2 Magnetic Disks: . . . . .	3
1.1.3 RAID . . . . .	3
1.1.4 SSDs . . . . .	4
1.2 Disk Space Manager . . . . .	4
1.2.1 Tasks . . . . .	4
1.3 Buffer Manager . . . . .	4
1.3.1 Buffer Allocation policy . . . . .	4
1.3.2 Buffer Replacement policy . . . . .	4
1.4 Files and File-Index Structures . . . . .	5
1.4.1 Free Slot Management . . . . .	5
1.4.2 Record Insertion strategy . . . . .	5
1.4.3 Page Formats . . . . .	6
1.4.4 Record Formats . . . . .	6
1.4.5 Alternative Page Layouts . . . . .	6
1.4.6 Addressing Schemes . . . . .	7
1.5 File Organization . . . . .	7
<b>2 Hash- and Tree-based Indexes</b>	<b>9</b>
2.1 General Indexes . . . . .	9
2.2 Tree-based Indexes . . . . .	9
2.2.1 Indexed Sequential Access Method . . . . .	9
2.2.2 B+Tree . . . . .	9
2.3 Hash-based Indexes . . . . .	9
2.3.1 Static Hashing . . . . .	10
2.3.2 Extendible Hashing . . . . .	10
2.3.3 Linear Hashing . . . . .	10
<b>3 Query Evaluation</b>	<b>11</b>
3.1 Overview . . . . .	11
3.1.1 System Catalog . . . . .	11
3.1.2 Access Paths . . . . .	11
3.1.3 Communication and Scheduling . . . . .	11
3.2 External Sorting . . . . .	12
3.2.1 Two-Way Merge Sort . . . . .	12
3.2.2 External Merge Sort . . . . .	12
3.2.3 Optimizations . . . . .	12
3.3 Relational Operator Evaluation . . . . .	13
3.3.1 Join . . . . .	13
3.3.2 Selection . . . . .	14
3.3.3 Projection . . . . .	15
3.3.4 Set Operations & Aggregations . . . . .	15
3.3.5 Buffering and Pipelines . . . . .	15
<b>4 Query Optimization and Database Tuning</b>	<b>17</b>
4.1 Relational Algebra Equivalences & Rewriting . . . . .	17
4.1.1 Relational Algebra Equivalences . . . . .	17
4.1.2 Relational Algebra Rewriting . . . . .	17
4.2 Plan Enumeration . . . . .	17
4.2.1 Single-Relation Queries . . . . .	17
4.2.2 Multiple-Relation Queries . . . . .	17
4.2.3 Dynamic Programming . . . . .	17
4.3 Cardinality Estimation 30 Slides . . . . .	17
4.3.1 System Catalog . . . . .	17
4.3.2 Histograms . . . . .	17
4.4 Nested Subqueries: 30 Slides . . . . .	17

# Chapter 1

## Disk, Buffer & File Management

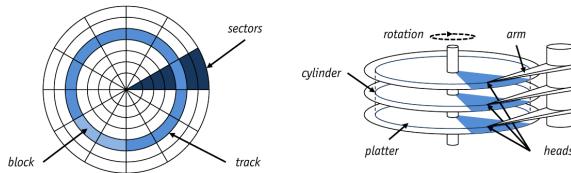
### 1.1 Hardware

#### 1.1.1 Memory Hierarchy:



Only data stored in non-volatile memory is persistent across shutdowns and the primary level of memory (address space) is not large enough to map the complete data-base, thus the need for magnetic disks, SSDs, tape and the like. The main goal of the primary memory level is to hide I/O latency by using faster memory to cache a subset of all necessary data. With increasing speed and decreasing size, the cost of the memory is approximately increasing by a factor of 100 per level.

#### 1.1.2 Magnetic Disks:



**Blocks:** Data is read and written to disk one block at a time. Size is multiple of the sector size.

**Sector:** A sector is a single part of a track

**Track:** A track is a ring on a platter

**Cylinder:** A cylinder is the set of all tracks with the same diameter

**Head:** The head is mounted at the arm which is controlled by a stepper motor to move from track to track as the disk rotates

The access time  $t$  is defined as

$$t = t_s + t_r + t_t$$

where

$t_s$  seek time (movement of disk head to desired track)

$t_r$  rotational delay (waiting time until the desired sector has rotated under the disk's head)

$t_t$  transfer time (time taken to read/write the block)

Sequential reads are much faster as one only needs to seek and to wait for the disk to rotate one time per sequential instead of on every access as in random I/O

Optimizations made by disk manufacturers:

- Track skewing: align sector of each track successively to avoid rotational delay in longer sequential scans
- Request scheduling: sort the disk requests so that minimal arm movement is needed
- zoning: outer tracks are longer than inner so can be divided into more tracks

⇒ Implies I/O focusedness as disks are major bottleneck and data needs to be in main memory to compute over it ⇒ Implies distance metric for records (i.e. same block, track, same cylinder, same disk, ...)

⇒ Clustering I/O ops vs. Declustered Storage

#### 1.1.3 RAID

Redundant Array of Independent Disks, provide redundancy to improve mean time to failure, improve performance using striping, hardware and software based implementations.

Raid Levels:

- 0 Striping  
Best Write performance, better read performance, no reliability improvements
- 1 Mirroring  
On-line backup, no performance improvements
- 10 Mirroring + Striping  
reasonable performance w extra reliability; write intensive workloads, small subsystems
- 2 ECC  
inferior to 3
- 3 Bit-Interleaved parity  
Appropriate for large continuous block requests, bad for many small requests
- 4 Block-Interleaved Parity  
Inferior to 5
- 5 Block-Interleaved Distributed Parity Good general purpose solution, best performance with redundancy for small and large r and large w ops
- 6 P+Q Parity  
highest level of reliability, like 5 with 2 parities



**Redundancy schemes:** Parity Groups: Disks or disk space may be grouped and parity may only be computed and stored per group

- ECC:  
Initialization: uses bit-level striping and Hamming code to compute ECC for data of n disks which is stored on n-1 additional disks  
Recovery: determine lost disk by using the n-1 extra disks and correct its content from one of those
- Parity Scheme  
Initialization: let  $i(n)$  be the number of bits set to 1 at position n on disk; n-th parity bit is set to 1 if  $i(n)$  is odd else 0  
Recovery: Let  $j(n)$  be the no. of bits set to 1 at pos n on the non-failed disks. If  $j(n)$  odd and parity bit 1 or  $j(n)$  even and parity bit is 0, n-th value on failed disk is 0, else 1

### 1.1.4 SSDs

Traits:

- Very low read latency ( $< 0.01\text{ms}$ )
- high transfer rate
- bad random write performance (worse than HDD)
- Page level read and write (only blocks, no single bits or bytes) e.g. 128 kB
- block-level delete e.g. 64 Pages = 1 block  $\Rightarrow$  major reason for bad random write performance: if internal fragmentation high: delete complete block reorganize pages and write the changes including addition

Outlook: SSDs and Storage Area Network.

## 1.2 Disk Space Manager

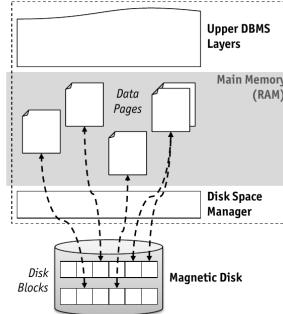
**A Page** is a disk block that has been brought into memory, thus pages and disk blocks equal in size. Sequences of pages map onto sequences of disk blocks. A Page is the unit of storage for all components in the system

### 1.2.1 Tasks

- Allocates/deallocates disk space/Pages; read/writes pages to disk
- track page locations (page no  $\leftrightarrow$  OS file + offset  $\leftrightarrow$  head, sector and track, disk)
- tracks free and used blocks
- does Segmentation (aka Table spaces, Partitions,...)

### Disk Space Manager

- Locality-preserving mapping
  - track page locations and block usage internally
  - page number  $\leftrightarrow$  physical location
- Abstract physical location
  - OS file name and offset within that file
  - head, sector, and track of a hard disk drive
  - tape number and offset for data stored in tape library
  - ...



tracking free and used blocks:

- Linked List of free blocks:
  1. pointer to first free block
  2. when block is freed pre/append it to free block list
  3. when block is needed take it from the list and adjust pointers
- Bitmap of free blocks
  1. reserve  $\text{ceil}(|blocks|/8)$  bytes
  2. interpret bitwise ( $0 = \text{free}$ ,  $1 = \text{used}$ )

**Segmentation/Partitioning** enables a DBMS to split data store into smaller units called e.g. partitions, segments, table spaces, storage pools, .... Those are then layered e.g. n tables per m table spaces per t segments of s storage pools each being one logical os volume.

## 1.3 Buffer Manager

The **BufferManager** mediates between external storage (Disk space manager) and main memory using the **BufferPool**. It is a designated area of main memory which is organized by the buffer manager. Each frame in the buffer pool has the same size as a page. pages are loaded into frames as needed using **pin(pageNo)** and evicted by a policy after the page got **unpin(pageNo, dirty)**. A page is only written if it's marked dirty using **unpin(pageNo, true)**. Thus all DB transactions need to be surrounded by pin and unpin.

### 1.3.1 Buffer Allocation policy

Problem: How to allocate buffer frames to each transaction?

Global One BufferPool for all transactions

Good overall usage of space, all TX are taken into account  
Transactions may hurt another by using up buffer frames and force others to reload their pages "extrenal page thrashing"

Local Treat all TX equal

TX cannot hurt each other  
bad overall usage of space, some TX may occupy vast amounts without using them while others starve "internal page thrashing"

Mainly mixed/hybrid aproaches perform best.

**Static allocation:** assign buffer budget once for each TX

**Dynamic allocation:** Adjust buffer budget according to some policy.

Examples:

- Local MRU  
Keep local LRU stack for each transaction  
keep global free list for not pinned pages

1. allocate a page from free list
2. allocate a page from the LRU stack of the requesting TX
3. allocate a page from TX with largest LRU stack

- Working Set Model

Avoid trashing by allocating just enough buffers to each TX observe page requests for a certain interval and deduce optimal budged from that based on the ratio of available pages

### 1.3.2 Buffer Replacement policy

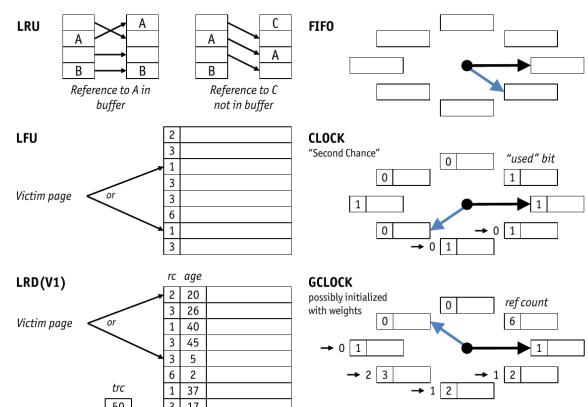
## Typical Buffer Replacement Policies

DBMS typically implement more than one replacement policy

- **FIFO** ("first in, first out")
- **LRU** ("least recently used"): evicts the page whose latest **unpin()** is longest ago
- **LRU-k**: like LRU, but evicts the  $k$  latest **unpin()** call, not just the latest
- **MRU** ("most recently used"): evicts the page that has been unpinned most recently
- **LFU** ("least frequently used")
- **LRD** ("least reference density")
- **CLOCK** ("second chance"): simulates LRU with less overhead (no LRU queue reorganization on every frame reference)
- **GCLOCK** ("generalized clock")
- **WS, HS** ("working set", "hot set")
- **Random**: evicts a random page

Criteria	Age of page in buffer		
	no	since last reference	total age
	none	Random	FIFO
References	last	LRU	CLOCK
	all	MRU	GCLOCK(V1)
	all	LFU	GCLOCK(V2)
	all	LRD	DGCLOCK
	all	LRD	LRD(V1)
	all	LRD	LRD(V2)

*Criteria matrix for victim selection*



## Details of LRD

- Record the following three parameters
  - $trc(t)$  total reference count of transaction  $t$
  - $age(p)$  value of  $trc(t)$  at the time of loading  $p$  into buffer
  - $rc(p)$  reference count of page  $p$
- Update these parameters during a transaction's page references ( $\text{pin}(pageNo)$  calls)
- Compute **mean reference density** of a page  $p$  at time  $t$  as
 
$$rd(p, t) := \frac{rc(p)}{trc(t) - age(p)}, \text{ where } trc(t) - rc(p) \geq 1$$
- Strategy** for victim selection
  - choose page with least reference density  $rd(p, t)$
  - many variants exist, e.g., for gradually disregarding old references

## Hot Set with Disjoint Page Sets

### Mode of Operation

- only those queries are activated, whose Hot Set buffer budget can be satisfied immediately
- queries with higher demand have to wait until their budget becomes available
- within its own buffer budget, each transaction applies a local LRU policy

### Properties

- no sharing** of buffered pages between transactions
- risk of **internal thrashing** when Hot Set estimates are wrong
- queries with large Hot Sets **block** following small queries (or, if bypassing is permitted, many small queries can lead to **starvation** of large queries)

## Hot Set with Non-Disjoint Page Set

### Mode of Operation

- queries allocate their budget stepwise, up to the size of their Hot Set
  - local LRU stacks are used for replacement
  - request for a page  $p$ 
    - if found in **own LRU stack**: update LRU stack
    - if found in **another transaction's LRU stack**: access page, but do not update the other LRU stack
    - if found in **free list**: push page on own LRU stack
  - call to **unpin(pageNo)**: push page onto free list stack
  - filling empty buffer frames: taken from the bottom of the free list stack
- As long as a page is in a local LRU stack, it cannot be replaced
- If a page drops out of a local LRU stack, it is pushed onto free list stack
- A page is replaced only if it reaches the bottom of the free list stack before some transaction pins it again

## Priority Hints

- Idea:** with **unpin(pageNo)**, a transaction gives one of two possible indications to the buffer manager
  - preferred page** managed in a transaction-local partition
  - ordinary page** managed in a global partition
- Strategy:** when a page needs to be replaced...
  - try to replace an ordinary page from the global partition using LRU
  - replace a preferred page of the requesting transaction using MRU
- Advantages**
  - much simpler than Hot Set, but similar performance
  - easy to deal with "too small" partitions

### Variant: Fixing and Hating Pages

Operator can **fix** a page if it may be useful in the near future (e.g., *nested-loop join*) or **hate** a page it will not access any time soon (e.g., *pages in a sequential scan*)

## Prefetching

- Buffer manager can try to **anticipate** page requests
  - asynchronously read ahead even if only a single page is requested
  - improve performance by overlapping CPU and I/O operations
- Prefetching techniques
  - prefetch lists: on-demand, asynchronous read-ahead**
    - e.g., when traversing the sequence set of an index, during a sequential scan of a relation
  - heuristic (speculative) prefetching**
    - e.g., sequential  $n$ -block look-ahead (cf. drive or controller buffers in hard disks), semantically determined supersets, index prefetch, ...

## 1.4 Files and File-Index Structures

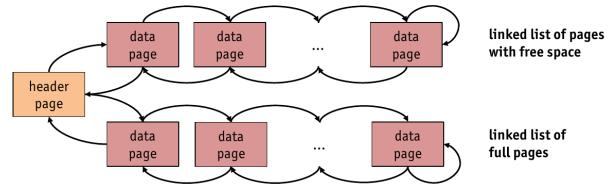
**RID** Each record has a unique identifier that is used like its address/pointer. Internally the file structure must be able to map a given

rid to the page and slot containing the record.  
 File  $\xrightarrow{\text{contains}}$  Pages  $\xrightarrow{\text{contains}}$  Records  $\xrightarrow{\text{contains}}$  Fields

### 1.4.1 Free Slot Management

File structure needs to keep track of pages with free space

## Linked List of Pages



### Operation $f \leftarrow \text{createFile}(n)$

- DBMS allocates a free page (called file **header page**) and stores an entry  $\langle n, \text{header page} \rangle$  to a known location on disk
- header page is initialized to point to two doubly linked lists of pages: one containing **full pages** and one containing **pages with free space**
- initially, both lists are **empty**

### Operation $rid \leftarrow \text{insertRecord}(f, r)$

- try to find a page  $p$  in the free list with space  $> |r|$
- should this fail ask the disk space manager to **allocate** a new page  $p$
- record  $r$  is **written** to page  $p$
- since generally  $|r| \ll |p|$ ,  $p$  will belong to the **list of pages with free space**
- a **unique rid** for  $r$  is computed and returned to the caller

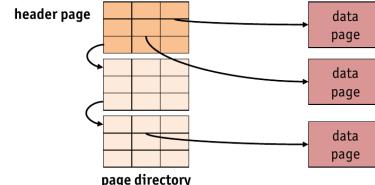
### Operation $\text{deleteRecord}(f, r)$

- may result in **moving the containing page** from the list of full pages to the list of pages with free space
- may even lead to **page deallocation** if the page is completely free after deletion

### Operation $\text{openScan}(f)$

- both page lists have to be traversed

## Directory of Pages



### Header page

- contains first page of a chain of **directory pages**
  - each entry in a directory page identifies a page of the file
  - $|page directory| \ll |data pages|$
- Free space management is also done through the directory
  - space vs. accuracy **trade-off**: from *open/closed* flag to exact information
  - for example, entries could be of the form  $\langle page\ addr\ p, nfree \rangle$ , where  $nfree$  indicates **actual amount of free space** (e.g., in bytes) on page  $p$
- Keeping **exact** counter value (e.g.,  $nfree$ ) during updates may produce a performance bottleneck in multi-user operations
  - each transaction, whose update changes the amount of available free space, needs to update this meta-information in the directory
  - locking (or some other form of synchronization) needs to be applied to avoid lost updates
  - frequent updates of the same record lead to "hot data item", introducing lock-wait queues and thus reducing degree of parallelism
- Keep **fuzzy information** on available free space in directory, e.g.,  $[nfree/8]$  (units of 8 bytes or some other granularity)
  - directory only needs to be updated for "large" changes in the available free space on a page
  - page itself contains the exact information (of course)

	Linked List	Directory
+	easy to implement	free space management more efficient
-	most pages in free space list	memory overhead

### 1.4.2 Record Insertion strategy

Standard Append, Best Fit, First Fit, Next Fit

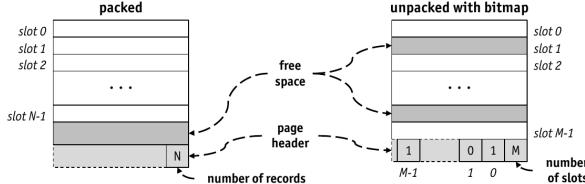
## Free Space Witness

1. classify pages into buckets
2. for each bucket remember any page that falls into that one as witness
3. only perform best/first/next fit if no witness page is recorded for the bucket. Else insert according to witness information
4. populate witness information e.g. as side effect while searching

### 1.4.3 Page Formats

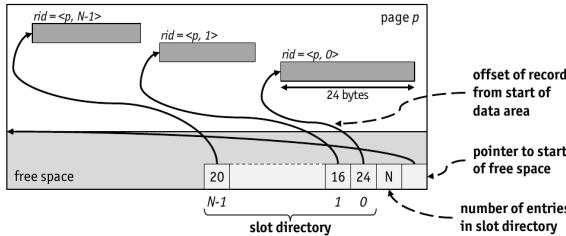
In the real world rids are PageNo + slotNo

## Free Slot Bitmap



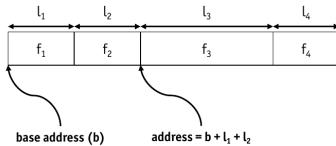
- Avoid record copying and therefore rid modifications
  - `deleteRecord(f, <p, n>)` simply needs to set bit  $n$  in bitmap to 0
  - no other rids affected

## Variable-Length Records



### 1.4.4 Record Formats

## Fixed-Length Fields



- Fixed-length record: each field has a **fixed length** and the number of fields is also **fixed**
  - fields can be stored **consecutively**
  - given the address of the record ( $b$ ), the address of a particular field can be calculated using information about **lengths of preceding fields** ( $l_i$ )
  - this information is available from the DBMS **system catalog**

## Variable-Length Fields

- Multiple variants exist to store records that contain variable-length fields
  1. use a special **delimiter symbol** (\$) to separate record fields: accessing field  $f_n$  requires a scan over the bytes of fields  $f_1 \dots f_{n-1}$

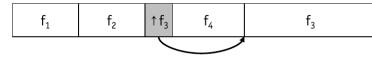


2. for a record of  $n$  fields, use an **array** of  $n+1$  offsets pointing into the record (the last array entry marks the end of field  $f_n$ )



## Record Formats

- Another popular record format to support variable-length fields is a **combination** of the delimiter and array approach
  - variable-length fields are stored **at the end of the record**
  - fixed-length fields and pointers to variable-length fields are stored sequentially, starting **at the beginning of the record**



- Note that it may even make sense to use variable-length records to store fixed-length records
  - support for null values (see above)
  - schema evolution, i.e., adding or removing columns

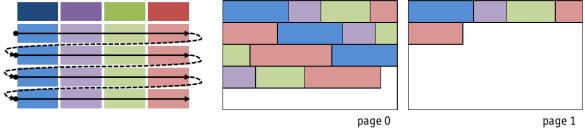
### 1.4.5 Alternative Page Layouts

Different layouts optimize different workloads:

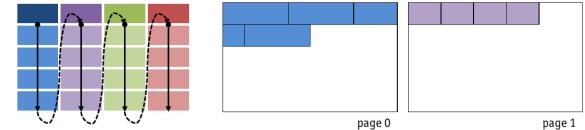
- row-major favors singular access of whole tuples (i.e. insert/update/delete, OLTP workloads)
- column-major favors table scans accessing only a subset of attributes (Scans/aggregations/analytics, OLAP workloads)
- optimally we want both ⇒ Hybrid layouts

**Row-store & column Store** aka **n-ary Storage Model (NSM)** and **Decomposition Storage Model (DSM)**

- So far, we populated data pages in **row-wise** order

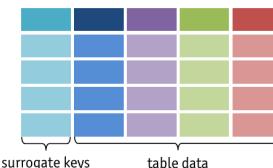


- We could also populate data pages in **column-wise** order

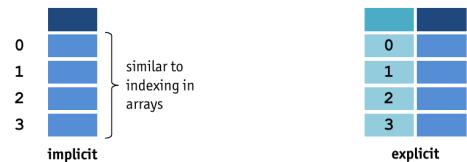


**Surrogate Keys** How to address in a column store

- One solution is to use **surrogate keys**



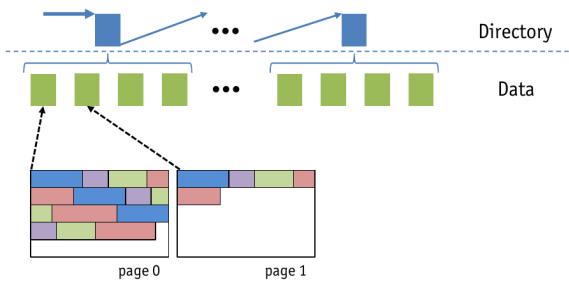
- Surrogate keys can be **implicit** or **explicit**



**Binary Association Tables** are tables broken into individual columns (DSM), using implicit surrogate keys to join.

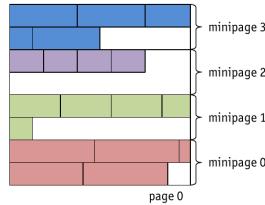
- early reconstruction: Access operator needs to know about columnar layout (and assemble the tuple as needed)
- late reconstruction: All operators need to support columnar layout

**Table Files** separates directory pages and data. Directory pages form a linked list while pages are listed below



#### Partition Attributes Across

- A hybrid approach is the **Partition Attributes Across (PAX)** layout
  - divide each page into **minipages**
  - group attributes into them
- On-disk performance of NSM
- Cache friendliness/compression characteristics of DSM

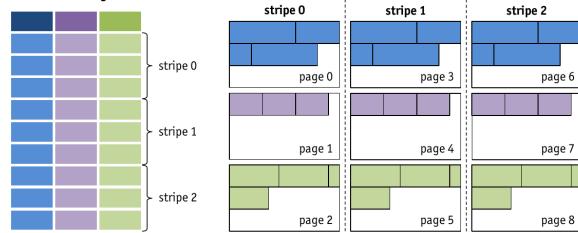


**Optimized Row Columnar** Stripe size is used as parameter to determine how row/column oriented the layout is:

- stripe size 1: row-major
- stripe size n: column major
- Another hybrid approach is the **Optimized Row Columnar (ORC)** file format used by `cstore_fdw` for PostgreSQL

#### ORC partitions records twice

- horizontally** into stripes
- vertically** into columns

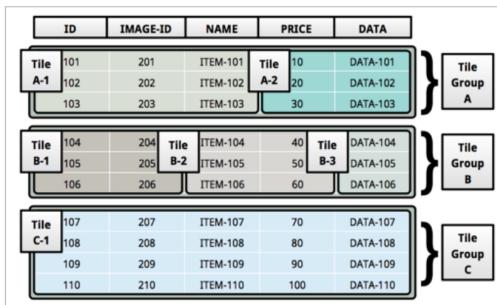


**Flexible Storage Model** Abstract physical from logical storage in **tile groups**.

Data is divided into two sets:

- Hot data: insert/update/delete heavy data stored in row-major
- Cold data: mostly Read data is stored column-major

#### Physical tiles (tile groups) are stored on disk



#### 1.4.6 Addressing Schemes

##### Direct Addressing

**Relative Byte Address (RBA)** uses a disk file as persistent address space, using byte offset as rid

- |   |  |
|---|--|
| + | very efficient to access pages and records |
| - | no stability wrt. moving records           |

**Page Pointers (PP)** uses disk page numbers as rid

- |   |   |
|---|---|
| + | very efficient to access page, locating a record within a page is also cheap (in-memory op) |
| - | only stable when moving records within a page   |

##### Indirect Addressing

**Logical Sequence Numbers (LSN)** assigns logical numbers to records and use address translation table to map to PP or RBA

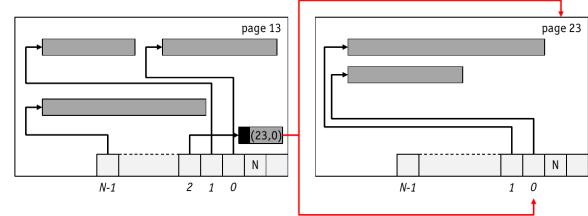
+	full stability
-	additional I/O op for table look-up (often in the buffer)

**Logical Sequence Numbers with Portable Page Pointers (LSN/PPP)** tries to save the look-up by using a PP. If the PP is not valid a table look-up is performed

+	full stability, if PPP valid saves one I/O
-	two additional I/O ops: one for loading the page and checking the PPP, one for the table look-up (often in the buffer)

**Tuple Identifier with forwarding pointer (TID/FP)** uses  $\langle \text{PageNo}, \text{SlotNo} \rangle$  pair as rid, where the slotNo is an index in page-local offset array. On relocation, leave forwarding address on origin page

+	full stability
-	extra I/O if relocated, extra space for the pointer



## 1.5 File Organization

### Cost Model :

Parameter	Description
b	number of pages per file
r	number of records per page
D	time to read a disk page
C	CPU time needed to process a record
H	CPU time taken to apply a function to a record e.g. comparison or hash

### Cost of Scan :

File Org	Description	est. Cost
Heap	read all pages, process each of the records per page	$b \cdot (D + r \cdot C)$
Sorted	Same as for heap files	$b \cdot (D + r \cdot C)$
Hashed	additional free space due to overflow chain avoidance	$(100/80) \cdot b \cdot (D + r \cdot C)$

### Cost of Search w. Equality :

File Org	Description	est. Cost
Heap	if equality test is on primary key, adds factor $\frac{1}{2}$	$b \cdot (D + r \cdot C)$ or $\frac{1}{2}b \cdot (D + r \cdot C)$
Sorted	Assuming equality test is on sort criterion, use bin search	$\log_2 b \cdot D \log_2 r \cdot C$
Hashed	Assuming equality test on hash attribute. Directly leads to the page containing the hit	$H + D + r \cdot C$ or $H + D + \frac{1}{2}r \cdot C$

### Cost of Search w. Range :

File Org	Description	est. Cost
Heap	Can appear everywhere = full scan	$b \cdot (D + r \cdot C)$
Sorted	Search for equality=lower and scan sequentially until the first record with A > upper	$\log_2 b \cdot D + \log_2 r \cdot C + \lfloor \frac{n}{r} \rfloor \cdot D + n \cdot C$
Hashed	Performs worst as additional space needs to be scanned	$(100/80) \cdot b \cdot (D + r \cdot C)$

**Cost of Insert :**

File Org	Description	est. Cost
Heap	Can be written to an arbitrary page, involves reading and writing the page	$2D + C)$
Sorted	Insert into a specific place and shift all subsequent	$\log_2 b \cdot D + \log_2 r \cdot C + \frac{1}{2} \cdot b \cdot (2 \cdot D + r \cdot C)$
Hashed	Write to the page that the hash fn indicates	$H + D + C + D$

**Cost of Delete :**

File Org	Description	est. Cost
Heap	Read, delete and write	$2D + C)$
Sorted	delete and shift all subsequent	$D + \frac{1}{2} \cdot b \cdot (2 \cdot D + r \cdot C)$
Hashed	Access by rid is faster than hashing so same as heap	$D + C + D$

**Summary :**

There is no single file organization that performs best overall. The choice can make serious differences. Indexes may provide all of the benefits with modest overhead.

# Chapter 2

## Hash- and Tree-based Indexes

### 2.1 General Indexes

Use an auxillary structure to provide support for certain functions. Variants: 1 resembles the sorted file on attribute A, thus only one such index should exist to avoid redundant storage of records. 2 and 3 use rids where 3 groups records that match a search key k.

1.  $\langle k, \langle \dots, A = k, \dots \rangle \rangle$
2.  $\langle k, \text{rid} \rangle$
3.  $\langle k, [\text{rid}_1, \dots] \rangle$

**Clustered & Unclustered Index:** An Index is clustered if the underlying file is sorted according to the indexed key. It's unclustered if the file is not sorted at all or on another key. So there may be at most one clustered index besides redundant indexes.

A **sparse index** is an index where only the smallest/largest key per page is stored instead of per record. Sparse Indexes can only be created if they are clustered indexes. If all records are present in the index it's called dense.

**Multi-Attribute Indexes** apply the previous techniques to a combination of attribute values. This enables the lookup of all combined attributes in common or a prefix of the multi-attribute key in case of a B+Tree index.

### 2.2 Tree-based Indexes

Containing only one record per page those auxiliary structures recurse until all data fit onto one page in terms of representation. Thus they are very useful for range selections.

**Fan-out:** The average number of children for a non-leaf node.

#### 2.2.1 Indexed Sequential Access Method

1. Sort the file on attribute A and store it
2. for each page maintain a pair  $\langle k_i, p_i \rangle$  containing the most extreme key wrt. a comparator of a certain Page i and a pointer to that page
3. use them as array to access the right page without scanning the page
  - One-level ISAM  $\Rightarrow$  Recurse on output to obtain Multi-level ISAM
  - ISAM is always static. Insert if space left or use overflow chains  $\Rightarrow$  search performance will decrease over time
  - since ISAM is static it **doesn't need to be locked**
  - costs for searching:  $\log_F N$

#### 2.2.2 B+Tree

- Similar to ISAM but dynamic wrt. updates  $\Rightarrow$  no overflow chains/remains balanced
- Search performance is also  $\log_F N$
- supports updates efficiently, guaranteed occupancy of 50
- non-leaves have same layout as in ISAM
- leaf nodes contain pointers to records (instead of ISAM: Pages)
- B+Tree index moves data records on split and merge, thus rid changes
  - 1.  $k_i^* = \langle k_i, \langle \text{attr}_1, \dots \rangle \rangle$
  - 2.  $k_i^* = \langle k_i, \text{rid} \rangle$
  - 3.  $k_i^* = \langle k_i, [\text{rid}_1, \dots] \rangle$
- occupancy rule might be relaxed
- duplicates are not supported, supported as normal values (affects search, checking the siblings) or using variant 3 grouped

**Searching:** As in ISAM, check the key and perform bin search until leaf is reached.

**Insert:** look for the right place, if not full insert. If full check siblings for redistribution (and update separator if necessary). If not possible split.

**Split:** Create new node, redistribute keys, take first value of the second leaf as new separator and propagate the split upwards.

**Delete:** Search leaf, delete value from it. If minimal occupancy below limit, check siblings for redistribution. If not possible, merge.

**Merge:** Merge with sibling node, delete key from parent level pointing to second leaf and propagate upwards

**Key Compression** uses prefixes or smaller data types to just approximate the actual values in the leafs. Another variant is to store common prefixes only once per node e.g. iri,o,r

**Bulk loading** If index is created, the tree is traversed  $|records|$  times. Most DBMS provide therefore a bulk tree loading utility.

1. for each key  $k$  in the data file, create a sorted list of pages of index leaf entries (does not imply sorting the data file itself on key  $k$  for variants 2,3; var 1 creates a clustered index)
2. allocate an empty root and let the first pointer  $p_0$  point to the first entry of the sorted list
3. for each leaf level node/list entry insert the index entry  $\langle p_n, \min(\text{val}(n)) \rangle$  into the rightmost index node above the leaf level

#### Invariants

- Order: d
- Occupancy:  $d - 1 < |\text{values}| < 2d + 1$ . Exception: root
- Fan-out: Non-leaf holding m keys has m+1 children
- Sorted Order nodes contain elements in comparator order, all children to the left are smaller wrt. comparator than the value in the parent and the subtrees to the right.
- Balanced: All leaf nodes are on the same level
- Height:  $\log_d N$

### 2.3 Hash-based Indexes

"Unbeatable for Equality operations", no support for range queries

#### Design of a hash function

1. **By division:** simply define

$$h(k) = k \bmod N$$

- this guarantees that range of  $h(k)$  to be  $[0, \dots, N - 1]$
- **prime numbers** work best for  $N$
- choosing  $N = 2^d$  for some  $d$  effectively considers the least  $d$  bits of  $k$  only

2. **By multiplication:** extract the fractional part of  $Z \cdot k$  (for a specific  $Z$ ) and multiply by hash table size  $N$

$$h(k) = [N \cdot (Z \cdot k - \lfloor Z \cdot k \rfloor)]$$

- the (inverse) **golden ratio**  $Z = \frac{2}{\sqrt{5}+1} \approx 0.6180339887$  is a good choice (according to D. E. Knuth, "Sorting and Searching")
- for  $Z = \frac{2}{2^w}$  and  $N = 2^d$  ( $w$  is the number of bits in a CPU word), we simply have  $h(k) = \text{msb}_d(Z \cdot k)$ , where  $\text{msb}_d(x)$  denotes the  $d$  **most significant bits** of  $x$  (e.g.,  $\text{msb}_3(42) = 5$ )

### Hash Function Family

Given an initial hash function  $h$  and an initial hash table size  $N$ , one approach to define such a family of hash functions  $h_0, h_1, h_2, \dots$  would be

$$h_{level}(k) = h(k) \bmod (2^{level} \cdot N)$$

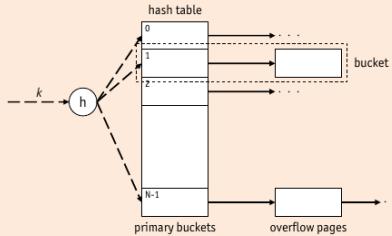
where  $level = 0, 1, 2, \dots$

### 2.3.1 Static Hashing

1. Allocate a fixed area of  $N$  successive disk pages **primary buckets**
2. for each bucket install a pointer to a chain of overflow pages (init to null)
3. Define a hash function  $h$  with range  $[0, N - 1]$ .
- search: 1 I/O
- insert & delete: 2 I/O
- Hash function should distribute evenly, overflow chains need distinct hash functions to avoid recursive collisions
- locking only required based on buckets/overflow chains

local depth  $d$ , global depth  $n$

### Static hash table



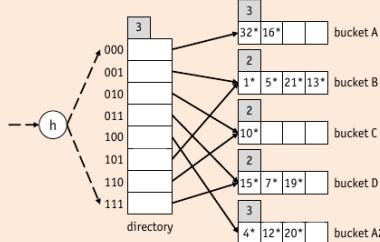
- A primary bucket and its chain of overflow pages is referred to as a **bucket**
- Each bucket contains index entries  $k^*$ , which can be implemented using any of the variants ①, ②, and ③

### 2.3.2 Extendible Hashing

Uses in-memory bucket directory to keep track of the actual primary buckets by adapting the hash function and the access to the buckets

#### Example: Insert a record with $h(k) = 20$

2. To address the new bucket, the directory needs to be **doubled** by simply copying its original pages (bucket pointer lookups now use  $\boxed{2} + 1 = \boxed{3}$  bits)
3. Let bucket pointer for  $100_2$  point to  $A_2$ , whereas the directory pointer for  $000_2$  still points to  $A$



**Searching:** buckets is an array of size  $2^{n-1}$  where each entry points to a corresponding bucket

1.  $n = \text{global depth}$
2.  $b = h(k) \& (2n - 1) // \text{mask last } n-1 \text{ bits}$
3.  $\text{bucket} = \text{buckets}[b]$

**Insert:** If there is free space in the corresponding bucket just insert, else

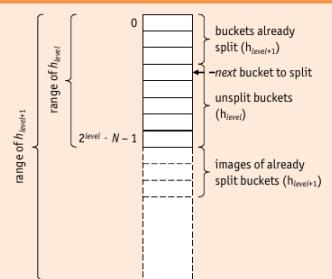
1. Split the bucket by creating a new bucket and use bit position  $d + 1$  to redistribute the entries where  $d$  is the local depth
2. if  $d + 1 > n$ ,  $n++$  and double the directory size. The hashing uses now  $d+1$  bits
3. let the old bucket be pointed to by  $0[...]$  and the new be pointed to by  $1[...]$
4. if no value goes to the new bucket an overflow chain is initialized

**Delete:** Analog to insertion

### 2.3.3 Linear Hashing

#### Rehashing

With every bucket split, next walks down the hash table. Therefore, hashing via  $h_{level}$  (search, insert, and delete) needs to take **current next position** into account.



$$h_{level}(k) \begin{cases} < \text{next: bucket already split, rehash: find record in bucket } h_{level+1}(k) \\ \geq \text{next: bucket not yet split, i.e., bucket found} \end{cases}$$

1. Init:  $\text{level} = 0, \text{next} = 0$
2. current hash fn =  $h_{level}$ , active hash buckets:  $[0, \dots, 2^{\text{level}} \cdot N]$
3. whenever a certain criterion is met, split the bucket which the next pointer references (e.g. % occupancy reached in a bucket, overflow chain grew longer than x, ...)

**Split** : All buckets with position  $i$  next have been already rehashed

1. Allocate new bucket and append it at position  $2^{\text{level}} \cdot N + \text{next}$
2. Redistribute entries in the bucket that next references by rehashing with  $h_{level+1}$
3.  $\text{next}++, \text{if next} > 2^{\text{level}} \cdot N - 1, \text{next} = 0; \text{level}++$

**Insert:** like in static hashing plus additional check for split criterion

**Delete:** like with static hashing plus if  $\text{bucket}[2^{\text{level}} \cdot N + \text{next}]$ .empty:

1. remove page pointed to by  $\text{bucket}[2^{\text{level}} \cdot N + \text{next}]$  from hash table
2.  $\text{next} -, \text{if next} \neq 0, \text{level} -, \text{next} = 2^{\text{level}} \cdot N + \text{next}$

# Chapter 3

## Query Evaluation

### 3.1 Overview

#### 3.1.1 System Catalog

size of buffer pool, page size, information and statistics about tables, views, indexes

Information stored in the system catalog																																																																																																																																	
<ul style="list-style-type: none"> <li><b>Table metadata</b> <ul style="list-style-type: none"> <li>table name, file name (or some identifier), file structure (e.g., heap file)</li> <li>attribute name and type of each attribute of the table</li> <li>index name of each index on the table</li> <li>integrity constraints (e.g., primary and foreign key constraints) on the table</li> </ul> </li> </ul>																																																																																																																																	
<ul style="list-style-type: none"> <li><b>Index metadata</b> <ul style="list-style-type: none"> <li>index name and structure (e.g., B+ tree)</li> <li>search key attributes</li> </ul> </li> </ul>																																																																																																																																	
<ul style="list-style-type: none"> <li><b>View metadata</b> <ul style="list-style-type: none"> <li>view name and definition</li> </ul> </li> </ul>																																																																																																																																	
Statistics																																																																																																																																	
<ul style="list-style-type: none"> <li><b>Table statistics</b> <ul style="list-style-type: none"> <li>cardinality: number of tuples <math>NTuples(R)</math> for each table <math>R</math></li> <li>size: number of pages <math>NPages(R)</math> for each table <math>R</math></li> </ul> </li> </ul>																																																																																																																																	
<ul style="list-style-type: none"> <li><b>Index statistics</b> <ul style="list-style-type: none"> <li>cardinality: number of distinct key values <math>NKeys(I)</math> for each index <math>I</math></li> <li>size: number of pages <math>INPages(I)</math> for each index (for a tree index <math>I</math>, <math>INPages(I)</math> denotes the number of leaf pages)</li> <li>height: number of non-leaf levels for each tree index <math>I</math></li> <li>range: minimum present key value <math>ILow(I)</math> and the maximum present key value <math>IHigh(I)</math> for each index <math>I</math></li> </ul> </li> </ul>																																																																																																																																	
Example																																																																																																																																	
<table border="1"> <thead> <tr> <th>Tables</th><th>name</th><th>file</th><th>#tuples</th><th>size</th></tr> </thead> <tbody> <tr><td>Tables</td><td>...</td><td>...</td><td>6</td><td>1</td></tr> <tr><td>Attributes</td><td>...</td><td>...</td><td>23</td><td>1</td></tr> <tr><td>Views</td><td>...</td><td>...</td><td>1</td><td>1</td></tr> <tr><td>Indexes</td><td>...</td><td>...</td><td>0</td><td>1</td></tr> <tr><td>Sailors</td><td>...</td><td>...</td><td>40,000</td><td>500</td></tr> <tr><td>Reserves</td><td>...</td><td>...</td><td>100,000</td><td>1000</td></tr> </tbody> </table> <table border="1"> <thead> <tr> <th>Views</th><th>name</th><th>text</th></tr> </thead> <tbody> <tr><td>Captains</td><td>SELECT * FROM Sailors WHERE...</td></tr> </tbody> </table> <table border="1"> <thead> <tr> <th>Indexes</th><th>name</th><th>file</th><th>type</th><th>#keys</th><th>size</th></tr> </thead> <tbody> <tr><td>Boats</td><td>...</td><td>B+Tree</td><td>100</td><td>1</td><td></td></tr> </tbody> </table>	Tables	name	file	#tuples	size	Tables	...	...	6	1	Attributes	...	...	23	1	Views	...	...	1	1	Indexes	...	...	0	1	Sailors	...	...	40,000	500	Reserves	...	...	100,000	1000	Views	name	text	Captains	SELECT * FROM Sailors WHERE...	Indexes	name	file	type	#keys	size	Boats	...	B+Tree	100	1		Attributes	<table border="1"> <thead> <tr> <th>name</th><th>table</th><th>type</th><th>pos</th></tr> </thead> <tbody> <tr><td>name</td><td>Tables</td><td>string</td><td>1</td></tr> <tr><td>file</td><td>Tables</td><td>string</td><td>2</td></tr> <tr><td>#tuples</td><td>Tables</td><td>integer</td><td>3</td></tr> <tr><td>size</td><td>Tables</td><td>integer</td><td>4</td></tr> <tr><td>name</td><td>Attributes</td><td>string</td><td>1</td></tr> <tr><td>table</td><td>Attributes</td><td>string</td><td>2</td></tr> <tr><td>type</td><td>Attributes</td><td>string</td><td>3</td></tr> <tr><td>pos</td><td>Attributes</td><td>integer</td><td>4</td></tr> <tr><td>I</td><td>I</td><td>I</td><td>I</td></tr> <tr><td>sid</td><td>Sailors</td><td>integer</td><td>1</td></tr> <tr><td>sname</td><td>Sailors</td><td>string</td><td>2</td></tr> <tr><td>rating</td><td>Sailors</td><td>integer</td><td>3</td></tr> <tr><td>age</td><td>Sailors</td><td>real</td><td>4</td></tr> <tr><td>sid</td><td>Reserves</td><td>integer</td><td>1</td></tr> <tr><td>bid</td><td>Reserves</td><td>integer</td><td>2</td></tr> <tr><td>day</td><td>Reserves</td><td>date</td><td>3</td></tr> <tr><td>rname</td><td>Reserves</td><td>string</td><td>4</td></tr> </tbody> </table>	name	table	type	pos	name	Tables	string	1	file	Tables	string	2	#tuples	Tables	integer	3	size	Tables	integer	4	name	Attributes	string	1	table	Attributes	string	2	type	Attributes	string	3	pos	Attributes	integer	4	I	I	I	I	sid	Sailors	integer	1	sname	Sailors	string	2	rating	Sailors	integer	3	age	Sailors	real	4	sid	Reserves	integer	1	bid	Reserves	integer	2	day	Reserves	date	3	rname	Reserves	string	4			
Tables	name	file	#tuples	size																																																																																																																													
Tables	...	...	6	1																																																																																																																													
Attributes	...	...	23	1																																																																																																																													
Views	...	...	1	1																																																																																																																													
Indexes	...	...	0	1																																																																																																																													
Sailors	...	...	40,000	500																																																																																																																													
Reserves	...	...	100,000	1000																																																																																																																													
Views	name	text																																																																																																																															
Captains	SELECT * FROM Sailors WHERE...																																																																																																																																
Indexes	name	file	type	#keys	size																																																																																																																												
Boats	...	B+Tree	100	1																																																																																																																													
name	table	type	pos																																																																																																																														
name	Tables	string	1																																																																																																																														
file	Tables	string	2																																																																																																																														
#tuples	Tables	integer	3																																																																																																																														
size	Tables	integer	4																																																																																																																														
name	Attributes	string	1																																																																																																																														
table	Attributes	string	2																																																																																																																														
type	Attributes	string	3																																																																																																																														
pos	Attributes	integer	4																																																																																																																														
I	I	I	I																																																																																																																														
sid	Sailors	integer	1																																																																																																																														
sname	Sailors	string	2																																																																																																																														
rating	Sailors	integer	3																																																																																																																														
age	Sailors	real	4																																																																																																																														
sid	Reserves	integer	1																																																																																																																														
bid	Reserves	integer	2																																																																																																																														
day	Reserves	date	3																																																																																																																														
rname	Reserves	string	4																																																																																																																														

#### 3.1.2 Access Paths

Possible access paths are

- file scan
- index with one matching selection condition
- index scan if all attributes required by the query are contained

If less than 5% are retrieved, a table scan is cheaper.

**Hash vs. B+Tree Index:** Hash Indexes match if selection contains equality on indexed attribute; B+Trees match if selection contains any condition on an attribute in the trees search prefix. If matches with an index were found in a CNF, those conjuncts are called **primary conjuncts**

The **Selectivity** of an access path is the number of index or data pages it retrieves. The most selective access path is the one that retrieves least pages.

**Reduction Factor:** The fraction of tuples that satisfy a certain conjunct. For several conjuncts this factor is approximated, making an independence assumption. For range queries uniformity is assumed: For a tree index the reduction factor is estimated using the system catalogue:  $\frac{High(T)-value}{High(T)-Low(T)}$

#### 3.1.3 Communication and Scheduling

**Pipelining** streams instead of writing out the intermediate result.

**Materialization** writes the intermediate result to disk

### Bracket Model



#### Bracket Template

- operating system and communication support
- central scheduler** can fit a single operator into the bracket on demand

#### Operators

- each operator **must** run its own thread, assumes all control within its thread, sends and receives data via networking procedures
- essentially relies on networking procedures for pacing and flow control
- each operator defines its own phases, synchronization points, and communication needs that must be known to the bracket implementation

### Iterator Model

- To simplify the code to coordinate plan execution, all operators implement a **uniform iterator interface**
  - hides **internal implementation details** of each operators
  - assumes **pipelined evaluation** of the query plan (see next slide)

#### Iterator interface

- open()**
  - initializes iterator by allocating input and output buffers
  - used to pass parameters that modify operator behavior, e.g., a selection condition
- next()**
  - calls **next** on each input operator
  - executes operator-specific code to process input tuples
  - places results in output buffer
  - updates iterator state to keep track of how much input has been consumed
- close()**
  - deallocates state information

### Iterator Model

Iterator	open()	next()	close()	State
print	open input	call <b>next()</b> on input, format item on screen	close input	
scan	open file	read next item	close file	open file descriptor
select	open input	call <b>next()</b> on input, until an item qualified	close input	
hash join without overflow resolution	allocate hash directory, open left build input, build hash table calling <b>next()</b> on build input, close build input, open right probe input	call <b>next()</b> in probe input until a match is found	close probe input, deallocate hash directory	hash directory
merge join without duplicates	open both inputs	get <b>next()</b> item from input with smaller key until a match is found	close both inputs	
sort	open input, build all initial run files calling <b>next()</b> on input, close input, merge run files until only one step is left	determine next output item, read new item from the correct run file	destroy remaining run files	merge heap, open file descriptors for run files

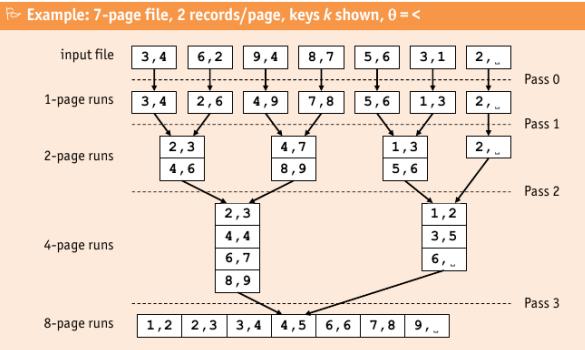
## 3.2 External Sorting

### 3.2.1 Two-Way Merge Sort

Sorts files of arbitrary size (here  $N = 2^k$ ) with three pages of buffer space in multiple passes, producing a certain number of sub-files called **runs**

- Pass 0:** Sorts each of the  $2^k$  input page individually in main memory, resulting in the same  $2^k$  runs
- subsequent passes:** Merge pairs of runs into larger runs. Pass  $n$  produces  $2^k - n$  runs
- pass  $k$  produces one overall sorted final run

Pass 0	
1.	Read $N$ pages, one page at a time
2.	Sort records, page-wise, in main memory
3.	Write sorted pages to disk (each page results in a <b>run</b> )
This pass requires <b>one page</b> of buffer space	
Pass 1	
(input: $N = 2^k$ unsorted pages, output: $2^k$ sorted pages)	
1.	Open two runs $r_1$ and $r_2$ from Pass 0 for reading
2.	Merge records from $r_1$ and $r_2$ , reading input page by page
3.	Write new two-page run to disk, page by page
This pass requires <b>three pages</b> of buffer space	
Pass $n$	
(input: $N = 2^{k-n+1}$ sorted runs, output: $2^{k-n}$ sorted runs)	
1.	Open two runs $r_1$ and $r_2$ from Pass $n-1$ for reading
2.	Merge records from $r_1$ and $r_2$ , reading input page by page
3.	Write new $2^n$ -page run to disk, page by page
This pass requires <b>three pages</b> of buffer space	



**Costs:** Each pass needs to read  $N$  pages, sort them in-memory and write them out again  $\Rightarrow 2N$  I/O ops per pass  
in total there are pass 0 and  $k$  subsequent pass  $\Rightarrow 1 + \log_2 N$  In Total Two-way Merge Sort costs

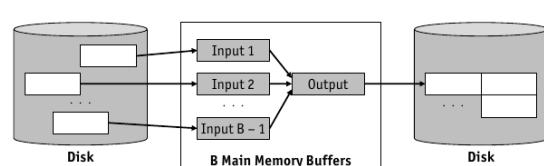
$$2N(1 + \log_2 N) \text{I/O ops}$$

### 3.2.2 External Merge Sort

like 2-way merge sort, but reduces the number of runs by using more buffer space to avoid creating one page runs in pass 0 and reduces the number of passes by merging more than two runs at a time.

## External Merge Sort

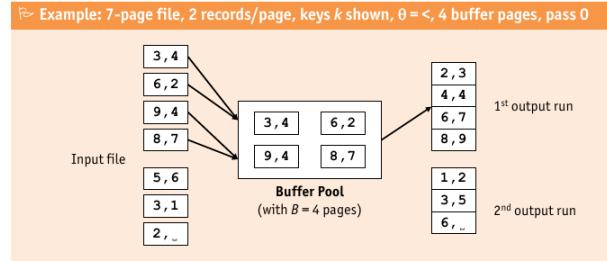
Pass 0	
1.	Read $N$ pages, $B$ pages at a time
2.	Sort records, page-wise, in main memory
3.	Write sorted pages to disk (resulting in $\lceil \frac{N}{B} \rceil$ runs)
This pass uses <b><math>B</math> pages</b> of buffer space	
Pass $n$	



- Suppose  $B$  pages are available in the buffer pool
  - $B$  pages can be read at a time during Pass 0 and sorted in memory
  - $B - 1$  pages can be merged at a time (leaving one page as a write buffer)

$N$	$B=3$	$B=5$	$B=9$	$B=17$	$B=129$	$B=257$
100	7	4	3	2	1	1
1000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

Number of Passes of External Merge Sort



**Costs:** As in two way: read, sort, write  $\Rightarrow 2N$   
In pass 0 only  $\lceil \frac{n}{B} \rceil$  are written thus only needs  $\lceil \log_{B-1} \lceil \frac{n}{B} \rceil \rceil$  where  $B-1$  pages are merged at the same time In Total Two-way Merge Sort costs

$$2N(1 + \lceil \log_{B-1} \lceil \frac{n}{B} \rceil \rceil) \text{I/O ops}$$

### 3.2.3 Optimizations

**Block-grouped I/O:** read blocks of  $b$  pages at once during merge. This decreases the I/O costs by factor  $b$  but decreases the fan-in thus increases the number of passes. In Total:

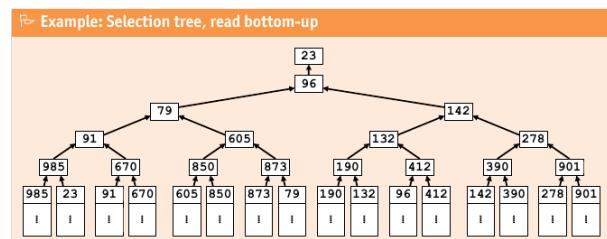
$$2N(1 + \lceil \log_{\lfloor \frac{B}{b} \rfloor - 1} \lceil \frac{N}{B} \rceil \rceil) \text{I/O ops}$$

$N$	$B=1000$	$B=5000$	$B=10,000$	$B=50,000$
100	1	1	1	1
1000	1	1	1	1
10,000	2	2	1	1
100,000	3	2	2	2
1,000,000	3	2	2	2
10,000,000	4	3	3	2
100,000,000	5	3	3	2
1,000,000,000	5	4	3	3

Number of Passes of External Merge Sort with Block Size  $b = 32$

### Tree of Losers

- The tree is initially filled with one tuple from each input by propagating them up the tree. The winner at each level is moved to the parent and we record if the left or right child won. At the end of this process the first element of the array contains the smallest tuple.
- Now we can remove the winner and write it to the output run.
- After that we have to refill the tree from below. Since all input runs are sorted, the tree currently contains the smallest tuple of all inputs except for the one the removed tuple came from. We can find that input by tracing the markers of who won/lost down the tree. Then we refill the tree with the next tuple from that input.
- If an input is drained, null can be used as the next tuple instead to mark an empty slot. We just have to make sure that null is compared as greater than any other tuple.
- When propagating the new tuple up the tree, we again adapt the winner/loser markers.
- Repeat steps ii) and iii) until the **winner** becomes null, at which point all input runs are drained and the merge is completed.



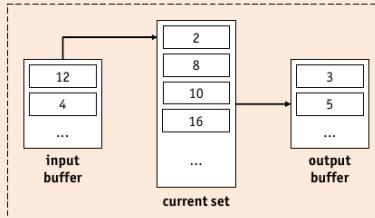
### Replacement Sort

#### Replacement sort

- Open an empty run file for writing.
- Load next page of file to be sorted into input buffer. If input file is exhausted, go to 4.
- While there is space in the current set, move a record from input buffer to current set (if the input buffer is empty, reload it at 2.)
- In current set, pick record  $r$  with smallest key value  $k$  such that  $k \geq k_{out}$ , where  $k_{out}$  is the maximum key value in output buffer (if output buffer is empty, define  $k_{out} = -\infty$ ). Move  $r$  to output buffer. If output buffer is full, append to current run.
- If all  $k$  in current set are  $< k_{out}$ , append output buffer to current run, close current run. Open new empty run file writing.
- If input file is exhausted, stop. Otherwise go to 3.

### Replacement sort

Assume a buffer pool with  $B$  pages. Two pages are dedicated **input** and **output** buffers. The remaining  $B - 2$  pages are called the **current set**.

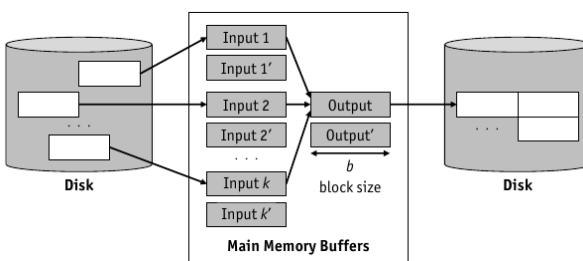


### Double Buffering

#### Double buffering

To avoid CPU waits, **double buffering** can be used.

1. create a second **shadow buffers** for each input and output buffer
2. CPU **switches** to the “double” buffer, once original buffers are empty/full
3. original buffer is reloaded by **asynchronously** initiating an I/O operation
4. CPU **switches** back to original buffer, once “double” buffer is empty/full, etc.



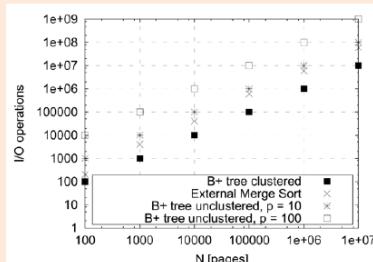
**B+ Tree for Sorting** Clustered B+Tree Index: just load the N pages as they are already sorted.

#### Expected page I/O operations for sorting

Let  $p$  denote the number of data records per page (typical values are  $p = 10, \dots, 1000$ ). The expected number of page I/O operations to sort using an unclustered B+tree index is therefore  $p \cdot N$  (worst case)

##### Assumptions in plot

- available buffer space for sorting is  $B = 257$  pages
- ignore I/O to traverse B+ tree as well as its sequence set



## 3.3 Relational Operator Evaluation

Different operator implementations exploit different physical properties:

- availability of indexes
- sortedness of input
- size of input file
- available buffer space
- buffer replacement policy, ...

A **logical Operator** is an operator used in the relational algebra. A **Physical Operator** is a specific variant of a logical operator. During query processing the **Query Optimizer** chooses which physical operator shall replace the logical operator based on the physical properties of the Relations intermediate results.

### 3.3.1 Join

Most basic variant is to calculate the cross product between the relations and apply selection according to the join predicate.

#### Nested Loops Join

- straight forward implementation of cross product and join equivalence
- needs only 3 buffer pages

### Nested loops join

```
function  $\bowtie^nl(R_1, R_2, R_{out}, p)$ 
   $in_1 \leftarrow openScan(R_1);$ 
   $out \leftarrow createFile(R_{out});$ 
  while ( $r_1 \leftarrow nextRecord(in_1)$ )  $\neq$  (EOF) do
     $in_2 \leftarrow openScan(R_2);$ 
    while ( $r_2 \leftarrow nextRecord(in_2)$ )  $\neq$  (EOF) do
      if  $p(r_1, r_2)$  then appendRecord(out,  $(r_1, r_2)$ );
    closeFile(out);
  end
```

#### Cost of $R_1 \bowtie_p^nl R_2$

access path	file scan ( <b>openScan</b> ) of $R_1$ and $R_2$
prerequisites	none ( $p$ arbitrary, $R_1$ and $R_2$ may be heap files)
I/O cost	$\underbrace{\ R_1\ }_{outer loop} + \underbrace{\ R_1\  \cdot \ R_2\ }_{inner loop}$

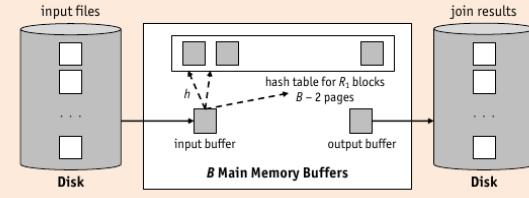
### Block Nested Loops Join

- Reads both relations in blocks of  $b_1, b_2$  pages respectively
- Costs:  $\lceil \|R_1\| / b_1 \rceil \cdot \lceil \|R_2\| / b_2 \rceil$

#### General block nested loops join

```
function  $\bowtie^{block-nl}(R_1, R_2, R_{out}, p)$ 
   $in_1 \leftarrow openScan(R_1);$ 
   $out \leftarrow createFile(R_{out});$ 
  foreach  $b_1$ -sized block in  $in_1$  do
     $in_2 \leftarrow openScan(R_2);$ 
    foreach  $b_2$ -sized block in  $in_2$  do
      for all matching in-memory tuples  $r_1 \in R_1$  block and  $r_2 \in R_2$  blocks,
        add  $(r_1, r_2)$  to the result out
    closeFile(out);
  end
```

#### Block nested loops join with hash table



#### Block nested loops join with hash table

```
function  $\bowtie^{block-nl}(R_1, R_2, R_{out}, p)$ 
   $in_1 \leftarrow openScan(R_1);$ 
   $out \leftarrow createFile(R_{out});$ 
  repeat
     $B \leftarrow \min(B - 2, \#remaining blocks in R_1);$  (do not read beyond (EOF) of R1)
    if  $B > 0$  then
      read  $B$ ' blocks of  $R_1$  into buffer, hash record  $r \in R_1$  to buffer page  $h(r.A_1) \bmod B'$ ;
       $in_2 \leftarrow openScan(R_2);$ 
      while ( $r_2 \leftarrow nextRecord(in_2)$ )  $\neq$  (EOF) do
        compare record  $r_2$  with records  $r_1$  stored in buffer page  $h(r_2.A_2) \bmod B'$ ;
        if  $r_1.A_1 = r_2.A_2$  then appendRecord(out,  $(r_1, r_2)$ );
      until  $B' < B - 2$ ;
      closeFile(out);
    end
```

### Index Nested Loops Join

- Uses index on inner Relation to avoid enumerating the cross product
- particularly useful when index is clustered and join is very selective

#### Index nested loops join

```
function  $\bowtie^{index-nl}(R_1, R_2, R_{out}, p)$ 
   $in_1 \leftarrow openScan(R_1);$ 
   $out \leftarrow createFile(R_{out});$ 
  while ( $r_1 \leftarrow nextRecord(in_1)$ )  $\neq$  (EOF) do
    probe index on  $R_2$  using (key value in)  $r_1$  to find matching tuples  $r_2 \in R_2$ ;
    appendRecord(out,  $(r_1, r_2)$ );
  closeFile(out);
end
```

#### Cost of $R_1 \bowtie_p^{index-nl} R_2$

access path	file scan ( <b>openScan</b> ) of $R_1$ , index access to $R_2$
prerequisites	index on $R_2$ that matches join predicate $p$
I/O cost	$\underbrace{\ R_1\ }_{outer loop} + \underbrace{\ R_1\  \cdot (\text{cost of one index access to } R_2)}_{inner loop}$

### Costs of an Index access :

- Hash Index:  $1.2 \begin{cases} 1.2 & \text{clustered} \\ n & \text{unclustered} \end{cases}$
- B+Tree:  $\log(|R_2|) \begin{cases} 1 & \text{clustered} \\ n & \text{unclustered} \end{cases}$

### Cost of $R_1 \bowtie_{A=B} R_2$

<b>access path</b>	file scan ( <b>openScan</b> ) of $R_1$ and $R_2$
<b>prerequisites</b>	p equality predicate $R_1.A = R_2.B$
<b>I/O cost</b>	$\ R_1\  + \ R_2\  + \ R_1\  + \ R_2\  + \ R_1\  + \ R_2\  = 3 \cdot (\ R_1\  + \ R_2\ )$
	$\underbrace{\text{read}}_{\text{partitioning phase}} \quad \underbrace{\text{write}}_{\text{probing phase}}$

### Sort Merge Join

- Uses sorting to partition both input
- best-case is optimal
- integration into external sort, block-grouped I/O, double buffering, replacement sort can be applied to optimize further
- output sorted on join attribute

#### Sort-merge join

```
function sort-merge ( $R_1, R_2, R_{out}, R_1.A = R_2.B$ )
    if  $R_1$  not sorted on A then sort it;      if  $R_2$  not sorted on B then sort it;
     $out \leftarrow \text{createFile}(R_{out})$ ;
     $in_1 \leftarrow \text{openScan}(R_1)$ ;            $in_2 \leftarrow \text{openScan}(R_2)$ ;
     $r_1 \leftarrow \text{nextRecord}(in_1)$ ;          $r_2 \leftarrow \text{nextRecord}(in_2)$ ;
    while  $r_1 \neq \langle EOF \rangle \wedge r_2 \neq \langle EOF \rangle$  do
        while  $r_1.A < r_2.B$  do  $r_1 \leftarrow \text{nextRecord}(in_1)$ ;
        while  $r_1.A > r_2.B$  do  $r_2 \leftarrow \text{nextRecord}(in_2)$ ;
         $r'_1 \leftarrow r_1$ ;                      (remember current position in  $R_2$ )
        while  $r_1.A < r'_2.B$  do  $r_1 \leftarrow \text{nextRecord}(in_1)$ ;          (all  $R_1$  tuples with the same A value)
         $r_2 \leftarrow r'_2$ ;                      (rewind  $r_2$  to  $r'_2$ )
        while  $r_1.A = r_2.B$  do
             $\text{appendRecord}(out, (r_1, r_2))$ ;
             $r_2 \leftarrow \text{nextRecord}(in_2)$ ;
             $r_1 \leftarrow \text{nextRecord}(in_1)$ ;
        closeFile(out);
    end
```

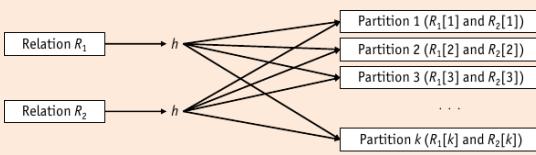
### Cost of $R_1 \bowtie_{A=B} \text{sort-merge } R_2$

<b>access path</b>	sorted file scan of $R_1$ and $R_2$
<b>prerequisites</b>	$p$ equality predicate $R_1.A = R_2.B$
<b>I/O cost</b>	cost of sorting $R_1$ and/or $R_2$ , if not sorted already, plus best case: $\ R_1\  + \ R_2\ $ worst case: $\ R_1\  \cdot \ R_2\ $

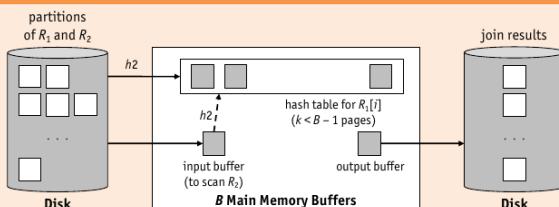
### Grace Hash Join

- works only for equality predicates
- Two Phases: Partitioning and matching/probing phase
- follows divide and conquer: partition and do per partition in-memory joins
- may be accelerated by using second hash function for probing as with the block NLJ
- Needs  $B > \sqrt{f \cdot \|R\|}$  Buffer pages where  $f$  is the factor of increase to maintain a hash table over the partition instead of the partition only, so that partition stays in memory

#### Partitioning phase of hash join



#### Probing phase of a hash join



#### Grace hash join

```
function hash-join ( $R_1, R_2, R_{out}, R_1.A = R_2.B$ )
     $in_1 \leftarrow \text{openScan}(R_1)$ ;           (partitioning phase)
    while ( $r_1 \leftarrow \text{nextRecord}(in_1)$ )  $\neq \langle EOF \rangle$  do
        add  $r_1$  to partition  $R_1[h(r_1.A)]$ ;
     $in_2 \leftarrow \text{openScan}(R_2)$ ;
    while ( $r_2 \leftarrow \text{nextRecord}(in_2)$ )  $\neq \langle EOF \rangle$  do
        add  $r_2$  to partition  $R_2[h(r_2.B)]$ ;
    out  $\leftarrow \text{createFile}(R_{out})$ ;
    foreach  $i \in 1, \dots, k$  do
        foreach tuple  $r_1 \in R_1[i]$  do          (build in-memory hash table for  $R_1[i]$ , using  $h2$ )
            insert  $r_1$  into hash table  $H$ , using  $h2(r_1.A)$ ;
        foreach tuple  $r_2 \in R_2[i]$  do          (scan  $R_2[i]$  and probe for matching  $R_1[i]$  tuples)
            probe  $H$  using  $h2(R_2.B)$  and append matching tuples  $(r_1, r_2)$  to  $out$ ;
            clear  $H$  to prepare for next partition;
    closeFile(out);
end
```

### 3.3.2 Selection

#### Definition

The **selectivity** (or **reduction factor**) or a predicate  $p$ , denoted by  $\text{sel}(p)$ , is the fraction of records in a relation  $R$  that satisfy the predicate  $p$ .

$$0 \leq \text{sel}(p) = \frac{|\sigma_p(R)|}{|R|} \leq 1$$

- Implemented using a combination of **iteration or indexing**
- may be applied on the fly in a pipelined plan
- Complex predicates may be expressed as conjuncts and disjuncts in terms of boolean logic
- three evaluation option for CNF terms:
  1. Single file scan
  2. single index that match a subset of the primary conjuncts, apply others on the fly
  3. multiple indexes each matching a subset of conjuncts, applying intersection over rid on the fly
- similar for DNF but union instead of intersection and only possible when all predicates are matched by index. Solution: **Bypass Selection**

#### No index, unsorted Data:

#### Selection

```
function  $\sigma(p, R_{in}, R_{out})$ 
    in  $\leftarrow \text{openScan}(R_{in})$ ;
    out  $\leftarrow \text{createFile}(R_{out})$ ;
    while ( $r \leftarrow \text{nextRecord}(in)$ )  $\neq \langle EOF \rangle$  do
        if  $p(r)$  then  $\text{appendRecord}(out, r)$ ;
    closeFile(out);
end
```

### Cost of $\sigma_p(R_{in})$ using a sequential scan

<b>access path</b>	file scan ( <b>openScan</b> ) of $R_{in}$
<b>prerequisites</b>	none ( $p$ arbitrary, $R_{in}$ may be a heap file)
<b>I/O cost</b>	$\ R_{in}\  + \text{sel}(p) \cdot \ R_{in}\ $
	$\underbrace{\text{input cost}}_{\text{input cost}} \quad \underbrace{\text{output cost}}_{\text{output cost}}$

#### No index, sorted on selection predicate Data:

#### Cost of $\sigma_p(R_{in})$ using binary search

<b>access path</b>	binary search, then sorted file scan of $R_{in}$
<b>prerequisites</b>	$R_{in}$ sorted on sort key $k$ that matches $p$
<b>I/O cost</b>	$\underbrace{\log_2 \ R_{in}\ }_{\text{input cost}} + \underbrace{\text{sel}(p) \cdot \ R_{in}\ }_{\text{sorted scan}} + \underbrace{\text{sel}(p) \cdot \ R_{in}\ }_{\text{output cost}}$

#### B+Tree index with predicate matching sort key

#### Implementing $\sigma_p(R_{in})$ using a B+ tree

- Descend the B+ tree to retrieve the first index entry that satisfies  $p$
- If the index is **clustered**, access that record on its page in  $R_{in}$  and continue to scan inside  $R_{in}$
- If the index is **unclustered** and  $\text{sel}(p)$  indicates a **large number of qualifying records**, it pays off to
  1. read all matching index entries  $k^* = \langle k, rid \rangle$  in the sequence set
  2. sort those entries on the  $rid$  field
  3. access the pages of  $R_{in}$  in sorted  $rid$  order

↳ Note that the lack of clustering is a minor issue if  $\text{sel}(p)$  is close to 0

### Cost of $\sigma_p(R_{in})$ using a clustered B+ tree index

<b>access path</b>	access of B+ tree on $R_{in}$ , then sequence set scan
<b>prerequisites</b>	clustered B+ tree on $R_{in}$ with key $k$ that matches $p$
<b>I/O cost</b>	$\underbrace{\approx 3 + \text{sel}(p) \cdot \ R_{in}\ }_{\text{B+ tree access}} + \underbrace{\text{sel}(p) \cdot \ R_{in}\ }_{\text{sorted scan}} + \underbrace{\text{sel}(p) \cdot \ R_{in}\ }_{\text{output cost}}$

## Hash Index

### Cost of $\sigma_p(R_{in})$ using a hash index

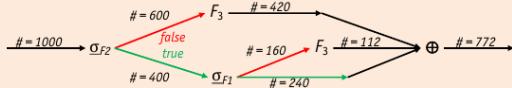
<b>access path</b>	hash index on $R_{in}$
<b>prerequisites</b>	$R_{in}$ hashed on key $k$ , $p$ has a term $k = c$
<b>I/O cost</b>	$\approx 1.2 + \text{sel}(p) \cdot \ R_{in}\ $

B+ tree access      output cost

**Bypass selection** avoids expensive and unselective selections by applying the most selective and the cheap selections first. I.E. the goal is to eliminate tuples early and avoid duplicates.

1. Convert selection condition to CNF
2. apply most selective/cheapest predicate first and safe disjunct tuples (true/false on predicate  $p_1$ )
3. repeat step 2 until all conjuncts are applied (when a conjunct contains disjuncts, just chain them)

### Example



Mean cost per tuple ( $\oplus$  disjoint union):  $C_2 + (1 - s_2) \cdot C_3 + s_2 \cdot (C_1 + (1 - s_1) \cdot C_3) = 40.6$

Note that many variations are possible, e.g., for tuning in parallel environments

### 3.3.3 Projection

- removes unwanted attributes and eliminates duplicates
  - implemented using iteration or partitioning
  - without duplicate elimination can be pipelined, with needs to be materialized
1. Do a FileScan,  $\forall r \in \text{File}$ : cut off unneeded attributes and append to the output
  2. do duplicate elimination as follows:

### Projection based on sorting

```
function πsort(l, Rin, Rout)
  in ← openScan(Rin);
  out ← createFile(Rtmp);
  while (r ← nextRecord(in)) ≠ (EOF) do
    r' ← r with any field not listed in l cut off;
    appendRecord(out, r');
    closeFile(out);
  external-merge-sort(Rtmp, Rtmp, 0);
  in ← openScan("run_*_0");
  out ← createFile(Rout);
  lastr ← {};
  while (r ← nextRecord(in)) ≠ (EOF) do
    if r ≠ lastr then
      appendRecord(out, r);
    lastr ← r;
  closeFile(out);
end
```

### Marriage of sorting and projection with duplicate elimination

Step ① (projection) and Step ② (duplicate elimination) can be integrated into the passes performed by the external merge sort algorithm.

#### Pass 0

1. read  $B$  pages at a time, projecting unwanted attributes out
2. use in-memory sort to sort the records of these  $B$  pages
3. write a sorted run of  $B$  internally sorted pages out to disk

#### Pass 1, ..., $n$

1. select  $B - 1$  runs from previous pass, read a page from each run
2. perform a  $(B - 1)$ -way merge, eliminating duplicates
3. use the  $B$ -th page as an output buffer

### Cost of $\pi^{\text{sort}}(R_{in})$ using sorting for duplicate elimination

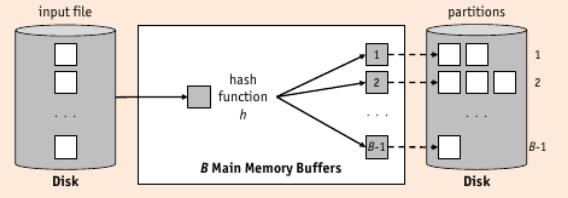
<b>access path</b>	file scan (openScan) of $R_{in}$
<b>prerequisites</b>	none ( $B$ available buffer pages)
<b>I/O cost</b>	$\ R_{in}\  + \ R_{tmp}\  + 2 \cdot \ R_{tmp}\  \cdot (\log_{B-1} \ R_{tmp}\  / B)$

projection      duplicate elimination

### Partitioning phase

1. Allocate all  $B$  buffer pages: one page will be the **input buffer**, the remaining  $B - 1$  pages will be used as **hash buckets**.
2. Read the file  $R_{in}$ , page by page: for each record  $r$ , **project out** the attributes not listed in list  $l$ .
3. For each such record, apply hash function  $h_1(r) = h(r) \bmod (B - 1)$ , which depends on **all remaining fields** of  $r$ , and store  $r$  in hash bucket  $h_1(r)$ .
4. If the hash bucket is **full**, write it to disk, i.e., the overflow chain of a bucket resides on disk

### Partitioning phase



### Duplicate elimination phase

1. For each partition, read each partition page by page (possibly in parallel), using the same buffer layout as before.
2. To each record, apply a hash function  $h_2$  ( $h_2 \neq h_1$ ) to all record fields.
3. Only if two records collide with respect to  $h_2$ , check if  $r = r'$  and, if so, discard  $r'$ .
4. After the entire partition has been read in, append all hash buckets to the result file, which will be free of duplicates.

### Cost of $\pi^{\text{hash}}(R_{in})$ using hashing for duplicate elimination

<b>access path</b>	file scan (openScan) of $R_{in}$
<b>prerequisites</b>	none ( $B$ available buffer pages)
<b>I/O cost</b>	$\ R_{in}\  + \ R_{tmp}\  + \ R_{tmp}\  + \ R_{tmp}\ $

projection      duplicate elimination

### 3.3.4 Set Operations & Aggregations

**Set Operations:** Intersection and Cross Product are implemented as special cases of join (equality on all attributes join for intersection and true for the cross product). Union and difference are implemented as special case selection (union: mainly duplicate elimination, difference: variation of duplicate elimination)

### Sorting for union and difference

Implementation of  $R \cup S$

1. sort both  $R$  and  $S$  using the combination of **all** fields
2. scan the sorted  $R$  and  $S$  in parallel and merge them, eliminating duplicates

As with projection, the implementation of difference can be integrated with the external sort operator.

Implementation of  $R - S$  is similar. During the merging pass tuples of  $R$  are only written to the result after checking that they **do not appear** in  $S$ .

### Hashing for union and difference

Implementation of  $R \cup S$

1. partition both  $R$  and  $S$  using a hash function  $h(\cdot)$  over the combination of **all** fields
2. process each partition  $i$  as follows
  - build an in-memory hash table, using hash function  $h_2(\cdot) \neq h(\cdot)$ , for  $S[i]$
  - scan  $R[i]$ ; for each tuple probe the hash table for  $S[i]$ ; if the tuple is in the hash table, discard it; otherwise, add it to the table
  - write out the hash table; clear it to prepare for the next partition

Implementation of  $R - S$  is similar, but processes the partition differently. After building an in-memory hash table for  $S[i]$ ,  $R[i]$  is scanned and  $S[i]$  is probed for every tuple in  $R[i]$ . If the tuple is **not in the table**, it is written to the result.

**Aggregations** There are 4 approaches: basic, using sorting, using hashing (last both only for aggregation and grouping), index only plans (if index contains all attributes needed for aggregation, group by if prefix matches index key and index is b+tree).

Basic Algorithm:

1. Scan relation, maintaining running information
2. compute aggregate using running information when finished

Using sorting (Aggregation and Grouping only):

1. Sort relation, record running information for each distinct value group and compute aggregation when sorting has finished

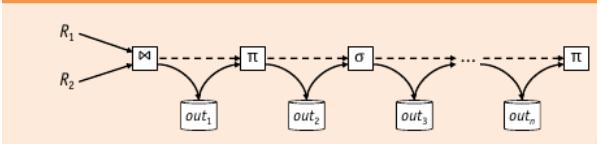
Using hashing (Aggregation and Grouping only):

1. build hash table on grouping attribute and maintain running information, compute aggreg

### 3.3.5 Buffering and Pipelines

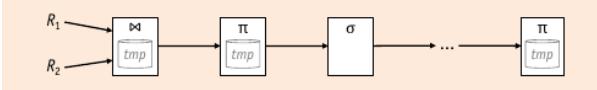
- concurrently executed operators share the buffer pool
- unclustered indexes fill the buffer pool rapidly and make pinning new pages hard to predict.
- repeated pattern of access can be speed up or be slowed down by a certain replacement policy
- all pseudo code by now assumed Materialization
- Pipelined plans can be used to avoid writing temporary files or delays to wait until some other operator has completely finished

### Materialized evaluation



**Pipelined Evaluation** enables each operator to pass the result directly to the next operator. Each partial result may be passed, other operators can start computation as early as possible and execute in parallel.

### Pipelined evaluation



Demand-driven or volcano style pipelining (open-next-close) was examined in the exercises and previously when talking about iterators.

## Iterator Model

- To simplify the code to coordinate plan execution, all operators implement a **uniform iterator interface**
  - hides **internal implementation details** of each operators
  - assumes **pipelined evaluation** of the query plan (see next slide)

### Iterator interface

- |                |  |
|----------------|--|
| <b>open()</b>  | <ul style="list-style-type: none"> <li>initializes iterator by allocating input and output buffers</li> <li>used to pass parameters that modify operator behavior, e.g., a selection condition</li> </ul>  |
| <b>next()</b>  | <ul style="list-style-type: none"> <li>calls <b>next</b> on each input operator</li> <li>executes operator-specific code to process input tuples</li> <li>places results in output buffer</li> <li>updates iterator state to keep track of how much input has been consumed</li> </ul> |
| <b>close()</b> | <ul style="list-style-type: none"> <li>deallocates state information</li> </ul>  |

## Iterator Model

Iterator	open()	next()	close()	State
print	open input	call <b>next()</b> on input, format item on screen	close input	
scan	open file	read next item	close file	open file descriptor
select	open input	call <b>next()</b> on input, until an item qualified	close input	
hash join without overflow resolution	allocate hash directory, open left build input, build hash table calling <b>next()</b> on build input, close build input, open right probe input	call <b>next()</b> in <i>probe</i> input until a match is found	close <i>probe</i> input, deallocate hash directory	hash directory
merge join without duplicates	open both inputs	get <b>next()</b> item from input with smaller key until a match is found	close both inputs	
sort	open input, build all initial run files calling <b>next()</b> on input, close input, merge run files until only one step is left	determine next output item, read new item from the correct run file	destroy remaining run files	merge heap, open file descriptors for run files

Note that there is also data-driven pipelining:

- Producer and consumer are connected by a queue
- operators execute asynchronously, in parallel
- operators are suspended by blocking queue calls
- higher resource requirements; systolic arrays

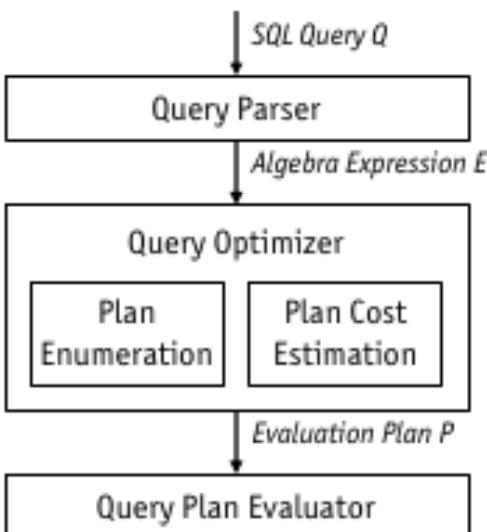
## Chapter 4

# Query Optimization and Database Tuning

The following steps are performed to answer a query Q:

1. Query Parser: Parse Q and derive a relational algebra expression E
2. **Rewrite optimization (logical level):** From E generate set of logical plans L transforming and simplifying E
3. **Cost-based Optimization (physical level):** Generate a set of physical plans P by annotating the plans in L with access paths and operator algorithms
4. **Plan cost estimator:** Estimate the costs of each plan and chose the best one
5. Query Plan Evaluator: Execute the plan and return the result to the UI

**Search space**  $\equiv$  logical level  $\cup$  physical level



## 4.1 Relational Algebra Equivalences & Rewriting

### 4.1.1 Relational Algebra Equivalences

1. Cascading selections

$$\sigma_{c_1 \wedge \dots \wedge c_n}(R) \equiv \sigma_{c_1}(\dots \sigma_{c_n}(R) \dots)$$

2. commutativity of selections

$$\sigma_{c_q}(\sigma_{c_p}(R)) \equiv \sigma_{c_p}(\sigma_{c_q}(R))$$

3. Cascading Projections

$$\pi_{c_1 \wedge \dots \wedge c_n}(R) \equiv \pi_{c_1}(\dots \pi_{c_n}(R) \dots)$$

4. **Folding selections:** with  $a_i \subseteq a_{i+1}$  only the last projection is needed (cutting off step by step vs. doing all cutoffs once)

$$\sigma_{a_1}(R) \equiv \sigma_{a_1}(\dots \sigma_{a_n}(R) \dots)$$

5. **Cross-Product and all Joins are associative** with r involves only T and S, p only involves R and S

$$(R \bowtie_p S) \bowtie_{q \wedge r} T \equiv R \bowtie_{p \wedge q} (S \bowtie_r T)$$

6. **Cross-Product and Natural Join are commutative**

$$R \times S \equiv S \times R$$

$$R \bowtie S \equiv S \bowtie R$$

7. **Selection and cross product form a join**

$$\sigma_p(R \times S) \equiv R \bowtie_p S$$

8. **Selections and joins can be combined**

$$\sigma_q(R \bowtie_p S) \equiv R \bowtie_{p \wedge q} S$$

9. **Selections commutes with Joins and cross-product** let q only involves R

$$\sigma_q(R \bowtie_p S) \equiv \sigma_q(R) \bowtie_p S$$

10. **Selections and projections distribute over joins and cross products** let p only involve R, q only involve S

$$\sigma_{p \wedge q}(R \bowtie_r S) \equiv \sigma_p(R) \bowtie_r \sigma_q(S)$$

$$\pi_a(R \bowtie_r S) \equiv \pi_{a_1}(R) \bowtie_r \pi_{a_2}(S)$$

11. **Selections and Projections commute** if the projections keeps all attributes involved in the selection predicates

$$\pi_a(\sigma_p(R)) \equiv \sigma_p(\pi_a(R))$$

12. **commutativity of Union and Intersection**

$$R \cup S \equiv S \cup R$$

$$R \cap S \equiv S \cap R$$

13. **Union and Intersection are associative**

$$(R \cup S) \cup T \equiv R \cup (S \cup T)$$

$$(R \cap S) \cap T \equiv R \cap (S \cap T)$$

14. **Projection distributes over Union**

$$\pi_a(R \cup S) \equiv \pi_a(R) \cup \pi_a(S)$$

15. **Selection, Union, Intersection and Difference are distributive**

$$\sigma_p(R \cup S) \equiv \sigma_p(R) \cup \sigma_p(S)$$

$$\sigma_p(R \cap S) \equiv \sigma_p(R) \cap \sigma_p(S)$$

$$\sigma_p(R \setminus S) \equiv \sigma_p(R) \setminus \sigma_p(S)$$

16. **Selection, Intersection and Difference are commutative**

$$\sigma_p(R \cap S) \equiv \sigma_p(R) \cap S$$

$$\sigma_p(R \setminus S) \equiv \sigma_p(R) \setminus S$$

### 4.1.2 Relational Algebra Rewriting

1. Break apart conjunctive selections (Rule 1)

2. Move selections down the query tree (Rules 2, 9, 15)

3. Replace selection-cross product pairs with joins (Rule 8)

4. Break list of projections apart & move them down, create new projections where possible (Rules 3, 10, 14)

5. Perform joins with the smallest expected result first

## 4.2 Plan Enumeration

### 4.2.1 Single-Relation Queries

### 4.2.2 Multiple-Relation Queries

### 4.2.3 Dynamic Programming

## 4.3 Cardinality Estimation 30 Slides

### 4.3.1 System Catalog

### 4.3.2 Histograms

## 4.4 Nested Subqueries: 30 Slides