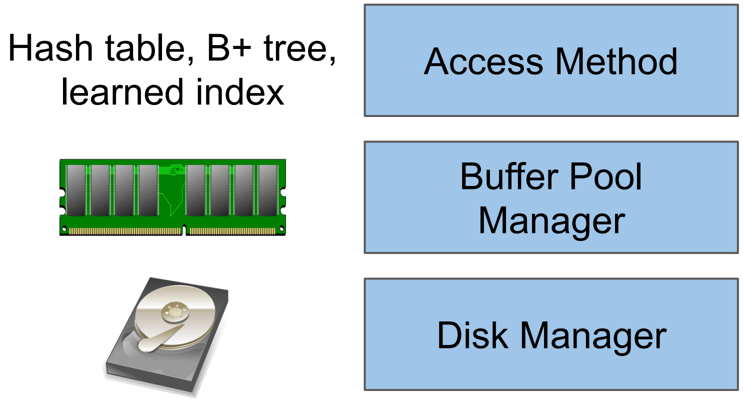
Databases and Big Data

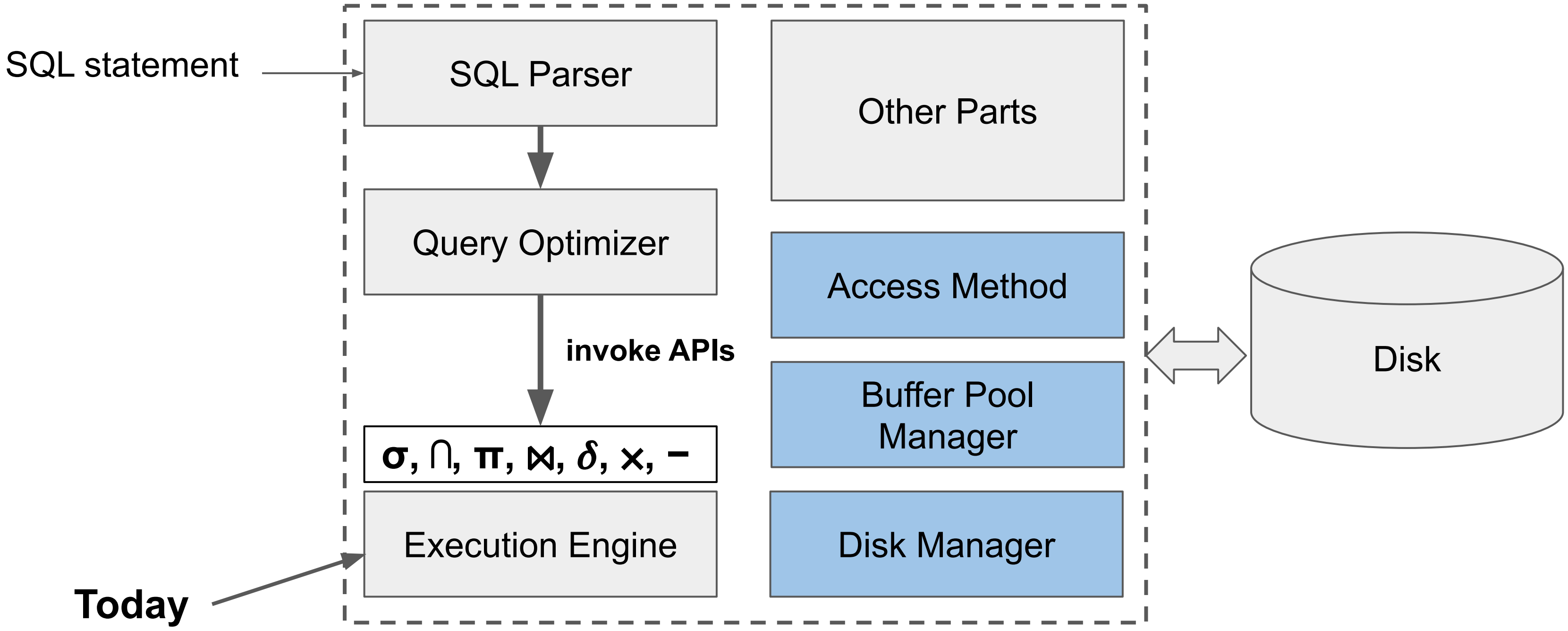
Sort and Join

Recap

- Database stores data in files
- Disk Manager: decides page layout on disk
- Buffer Manager moves pages in and out of memory
- Access methods: build indices for efficient access



So Far



SQL statement
SQL Parser
Query Optimizer
invoke APIs
σ, ∩, π, ⋈, δ, ×, −
Execution Engine
Today
Other Parts
Access Method
Buffer Pool Manager
Disk Manager
Disk

Today

- How does DBMS execute
  - Select
  - Aggregate, Distinct
  - Join
  - etc.
- Warning: no one-size-fit-all algorithms
  - Design problem: choose an appropriate one!

Selectivity: within this relation, how many tuples will satisfy this condition
Naive approach (worst):
have a heap file, when you execute the query, read page by page to see if the tuples can satisfy the condition
Cost in terms of I/O: #pages storing R

Select

- $\sigma_C(R)$:
  - Select tuples satisfying C from R

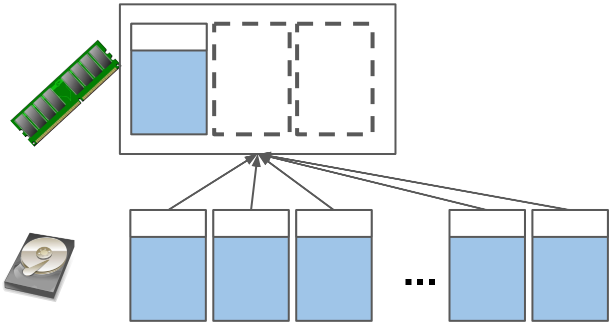
Notation

$\lvert R \rvert$:	# tuples in R
B(R):	# pages storing R
M:	buffer size (#pages in buffer pool)
α(C,R):	selectivity: # tuples satisfying C / $\lvert R \rvert$

- Approach 1: scan heap file
  - Read pages one by one
  - Check for condition C

Cost: **B(R)**

Almost always the worst way to go.



Select

Goes through the tree and find the leaf and scan through the whole of heap file
(don't need to scan every heap file, just the heap files that your index points to (from the correct leaf))

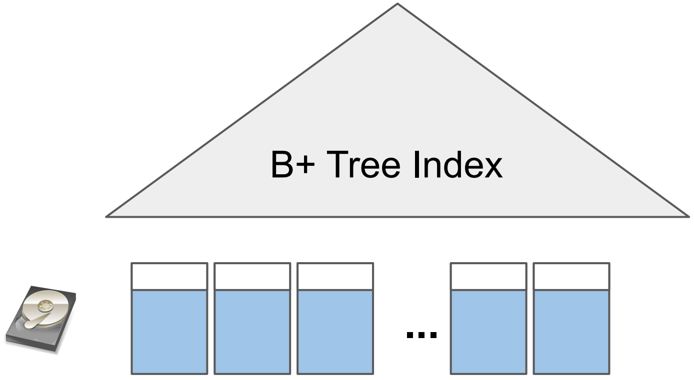
- Approach 2: scan index file
  - When C is =, <, >
  - Find the correct leaf, then scan

Notation

$\|R\|$:	# tuples in R
B(R):	# pages storing R
M:	buffer size (#pages in buffer pool)
α(C,R):	selectivity: # tuples satisfying C / $\|R\|$

Linear scan: **B(R)**

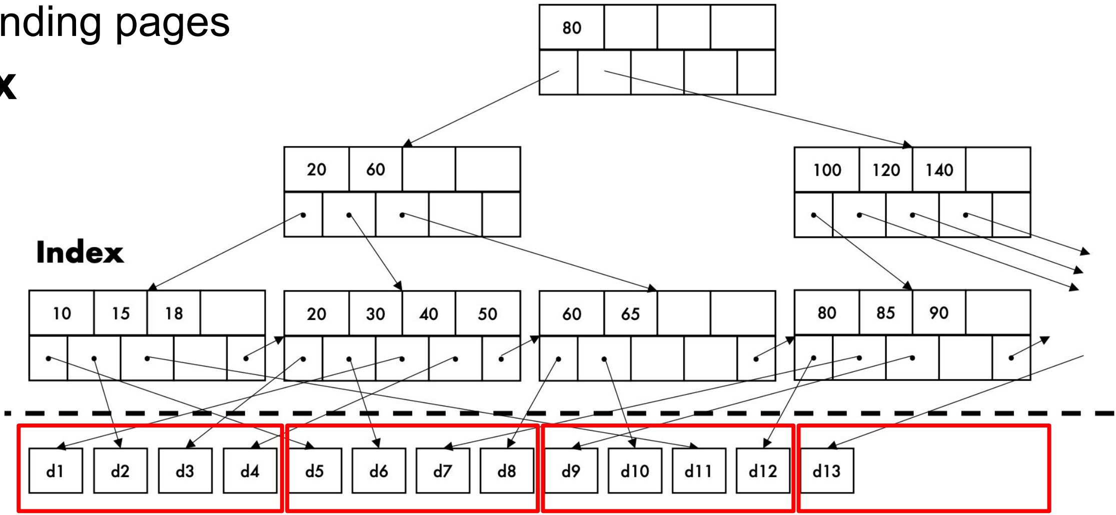
Almost always the worst way to go.



Select

- Approach 2: scan index file
  - When C is =, <, >
  - Find the correct leaf, then scan
    - Need to fetch corresponding pages
  - Using **unclustered index**

C: *key >= 40 and key <=85*



Select

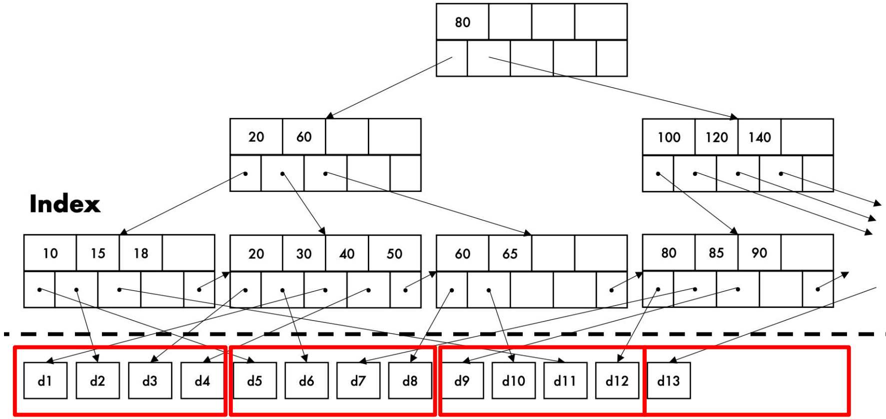
log (B(R)) *= cost of traversing the tree (log(height))*
alpha . |R| *= #tuples satisfying condition*
worst case: for each tuple you need to fetch a new page

- Approach 2: scan index file
  - When C is =, <, >
  - Find the correct leaf, then scan
    - Need to fetch corresponding pages
  - Using **unclustered index**

Notation

|R|:	# tuples in R
B(R):	# pages storing R
M:	buffer size (#pages in buffer pool)
α(C,R):	selectivity: # tuples satisfying C / |R|

Cost: ***log(B(R)) + α(C,R).|R|***



for clustered, you know that 50 will be next to 40.

cost is alpha . B(R)

*= # tuples satisfy C / # tuples in R * #pages store R*

≈ #pages that satisfy the condition

you assume tuples will be near each other, so you don't have to read a new page for each tuple

Select

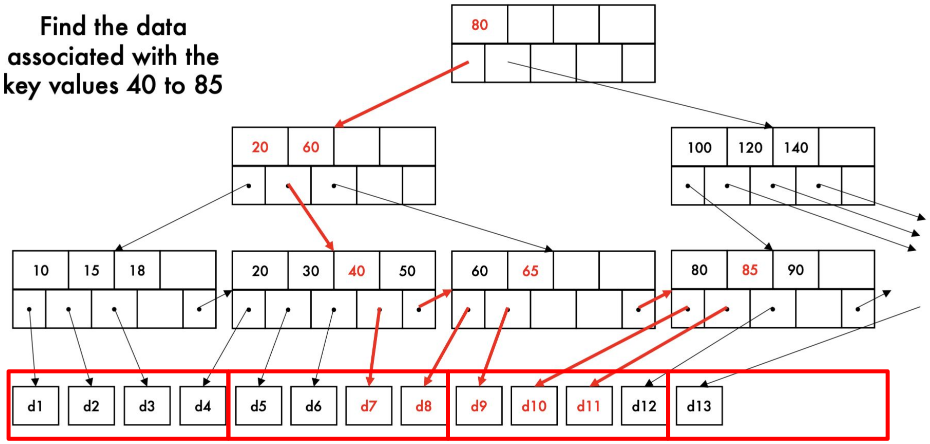
- Approach 2: scan index file
  - When C is =, <, >
  - Find the correct leaf, then scan
    - Need to fetch corresponding pages
  - Using **clustered index**

Notation

|R|:	# tuples in R
B(R):	# pages storing R
M:	buffer size (#pages in buffer pool)
α(C,R):	selectivity: # tuples satisfying C / |R|

Cost: $log(B(R)) + \alpha(C,R).B(R)$

Find the data associated with the key values 40 to 85

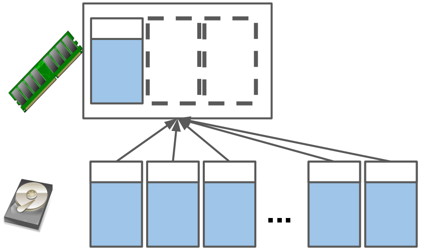


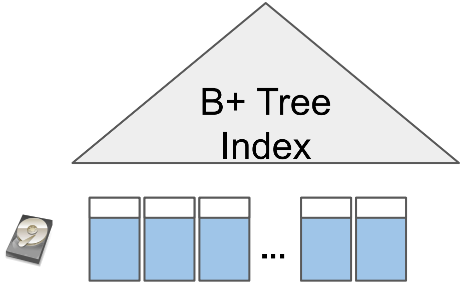
Select

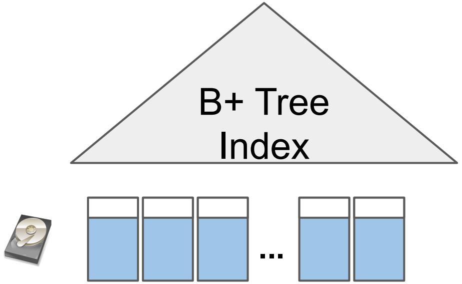
Trade-offs: *selectivity* is the key

DBMS will choose for you

good for small data
tree -> got cost of building the tree







B(R)

$log(B(R)) + \alpha(C,R).|R|$

(unclustered)

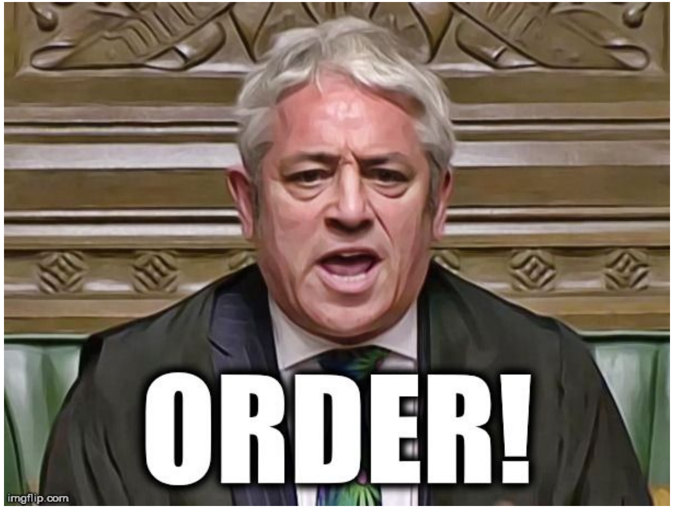
$log(B(R)) + \alpha(C,R).B(R)$

(clustered)

Bad, but not always the worst

Sort

- Order is important
  - For B+ Tree bulk loading
  - For duplicate elimination (DISTINCT)
  - For aggregations (GROUP BY)
  - Because user wants it (ORDER BY)



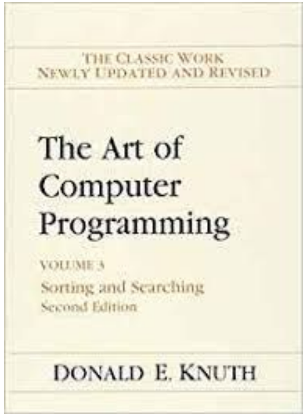
(a,3)	(c,5)	(a,2)	(d,6)	(e,9)	(c,9)	(f,7)

Order by first attribute

(a,3)	(a,2)	(c,5)	(c,9)	(d,6)	(e,9)	(f,7)

Sort

- Sorting algorithms
  - Stable
  - In-place
  - Quick, bubble, merge, bucket, radix, etc.
- In this class: external sort
  - Data **doesn't fit in DRAM**



No, thanks!

External Sort

- How to sort data on disk
  - Must be **I/O efficient**
- Merge sort:
  - Divide in equal parts, sort each part
  - Then merge

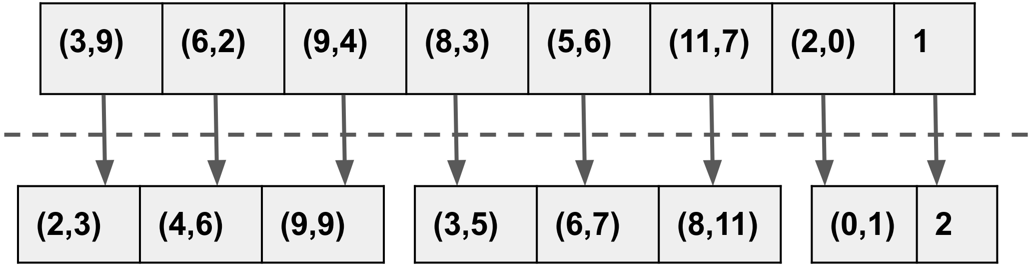
Example

N = 8 pages to sort, all on disk

M = 3 pages buffer

(3,9)	**(6,2)**	**(9,4)**	**(8,3)**	**(5,6)**	**(11,7)**	**(2,0)**	**1**

External Sort



Example

N = 8 pages to sort, all on disk

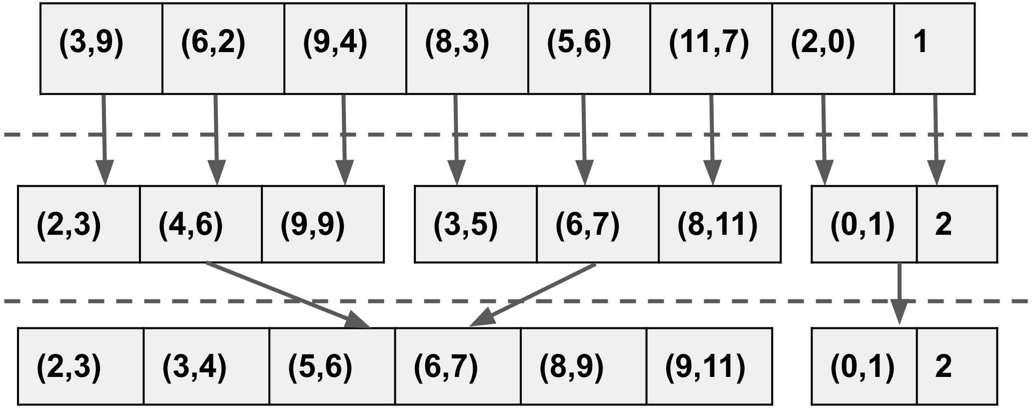
B = 3 pages buffer

each page have at most 2 values

Pass 0:

1. **Read B pages from disk,**
2. **Sort them in memory**
3. **Write to disk**

External Sort



Example

N = 8 pages to sort, all on disk

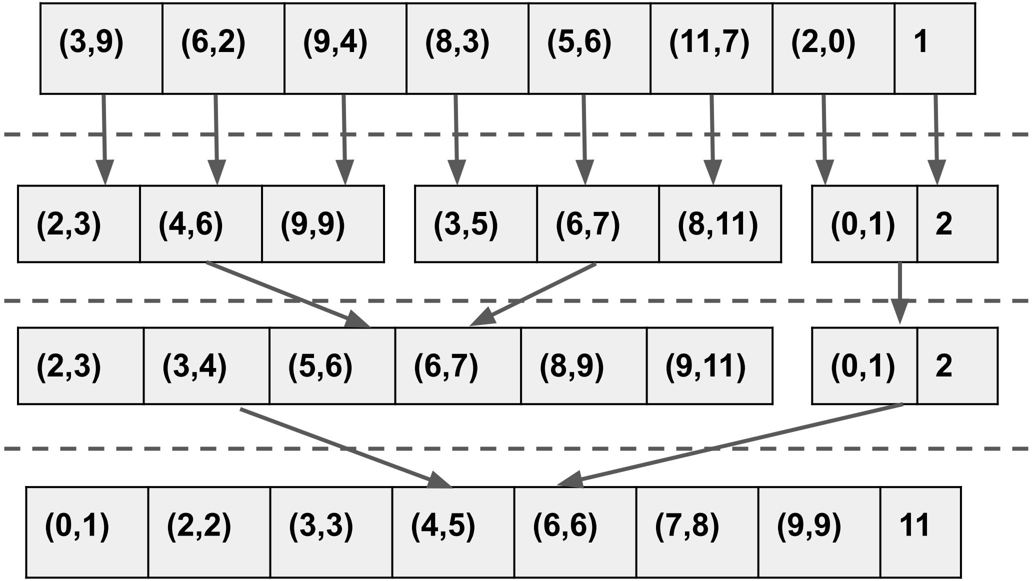
B = 3 pages buffer

2-Way Merge
so read 2 pages at a time to merge them

Pass 1:

1. **Read 2 pages from disk,**
2. **Sort them in memory**
3. **Write to disk**

External Sort



Example

N = 8 pages to sort, all on disk

B = 3 pages buffer

Pass 2:

1. Read 2 pages from disk,

2. Sort them in memory

3. Write to disk

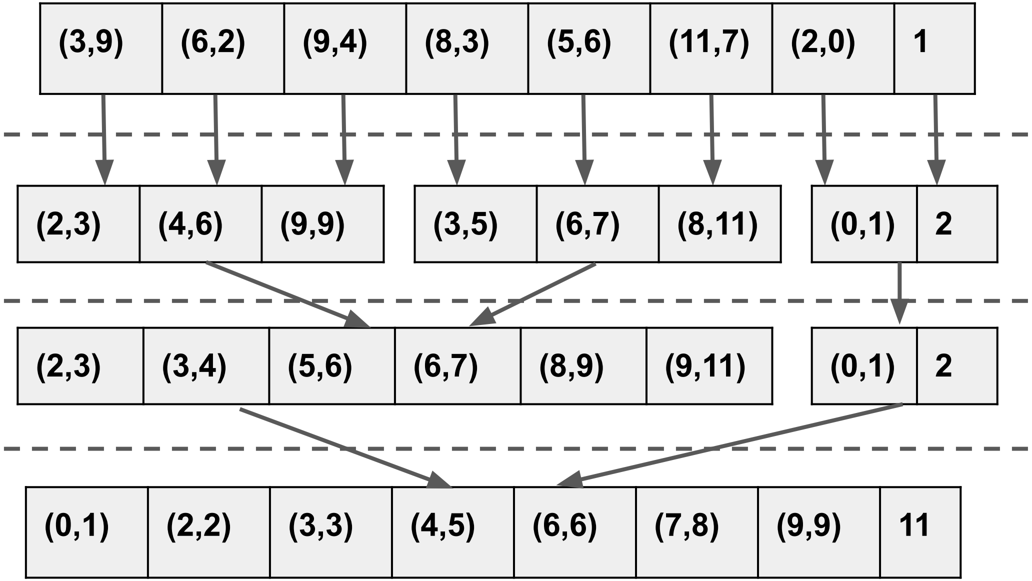
Ceiling *to account for all the smaller than B scenarios*
(output up to 3 pages/ 6 pages...)
output *= what you write to disk*

External Sort

Example

N = 8 pages to sort, all on disk

B = 3 pages buffer



Pass 0: output **3-page** files
Pass 1: output **6-page** files
Pass 2: output **12-page** files
…
Pass $\lceil \log_2 \lceil N/B \rceil \rceil$: output **N-page** files

# passes:	$1 + \lceil \log_2 \lceil N/B \rceil \rceil$
One pass:	2.N

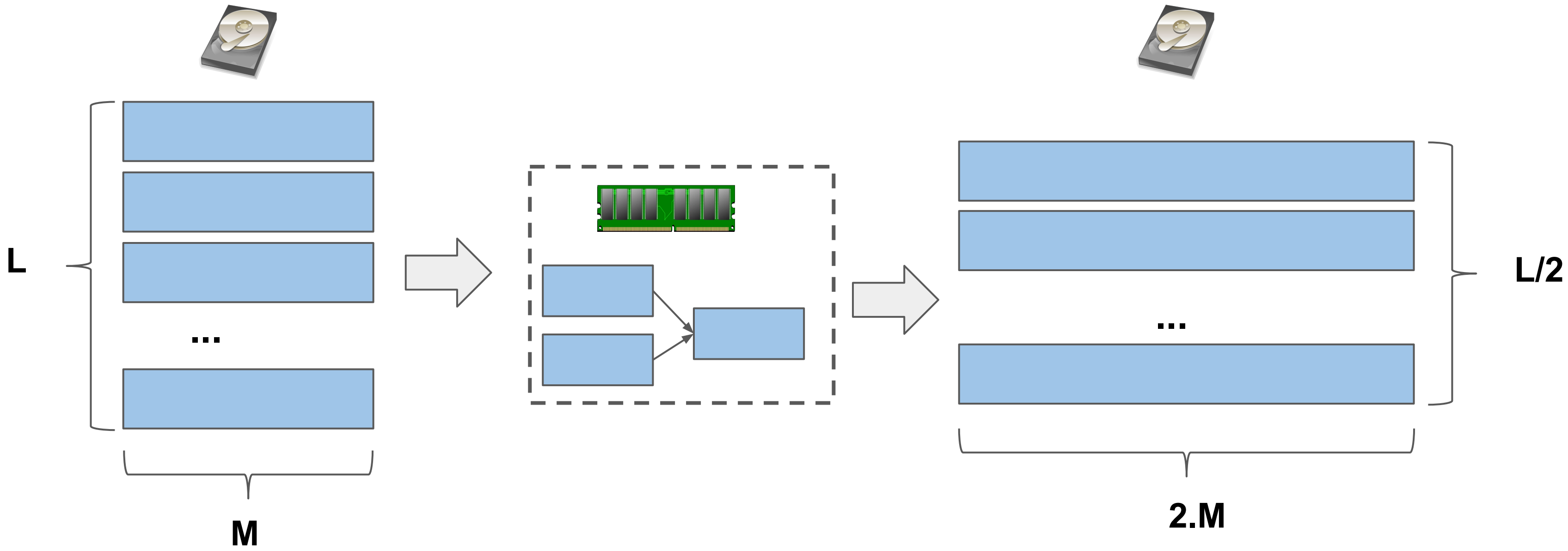
Total cost: $\mathbf{2.N.(1+ \lceil \log_2 \lceil N/B \rceil \rceil)}$

N/B = how many pages you get when you do the first pass
2.N = first read in N and write out N

External Sort

B = 3
2-way merge
need to store 2 pages to merge and output 1 page

- How does Merge(.) work?

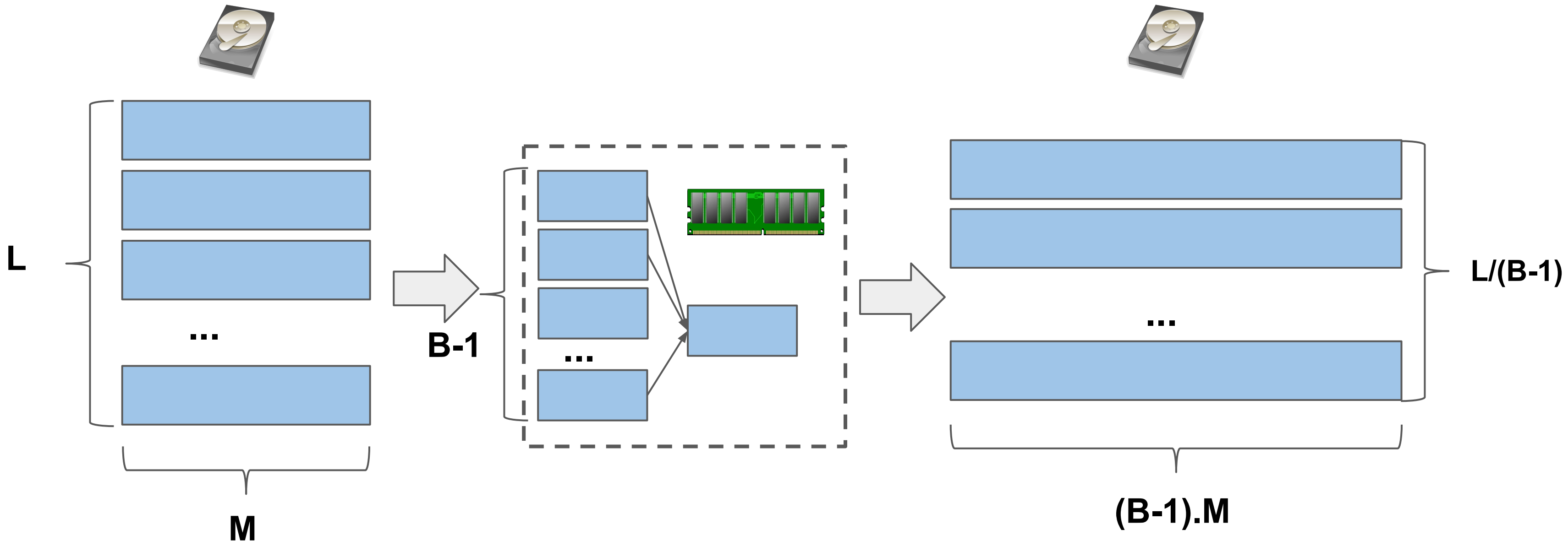


External Sort

- What if B > 3

Multi Way Merge
Use priority queues

B-1 way merge



External Sort

- B > 3

Pass 0: output **B-page** files
Pass 1: output **B(B-1)-page** files
Pass 2: output **$B(B-1)^2$-page** files
…
Pass $\lceil \log_{B-1} \lceil (N/B) \rceil \rceil$: output **N-page** files

passes: $1 + \lceil \log_{B-1} \lceil (N/B) \rceil \rceil$ passes
One pass: 2.N

Total cost: **$2.N.(1+ \lceil \log_{B-1} \lceil (N/B) \rceil \rceil)$**

Memory > sqrt (#pages needed)

It takes 2 passes.

N/sqrt(N) = sqrt(N)
log (sqrt(N)) = 0.5 logN

What happen when

$$B \geq \sqrt{N}$$

External Sort

- N=108, B=5
    - Pass 0: 22 files of 5 pages each*
    - Pass 1: 6 files of 20 pages each*
    - Pass 2: a 80-page, and a 28-page file
    - Pass 3: sorted file

20 pages because we are using a 4-way merge

$$1 + \lceil \log_{B-1} \lceil (N/B) \rceil \rceil = 1 + \lceil \log_{4}(22) \rceil = 4$$

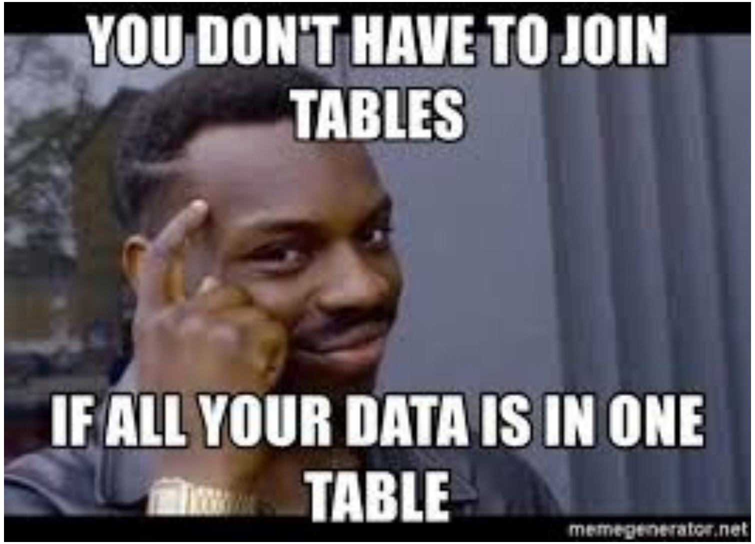
External Sort

Top Results

	Daytona	Indy
Gray	2016, 44.8 TB/min **Tencent Sort** 100 TB in 134 Seconds 512 nodes x (2 OpenPOWER 10-core POWER8 2.926 GHz, 512 GB memory, 4x Huawei ES3600P V3 1.2TB NVMe SSD, 100Gb Mellanox ConnectX4-EN) Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao Tencent Corporation Mark R. Nutter, Jeremy D. Schaub	2016, 60.7 TB/min **Tencent Sort** 100 TB in 98.8 Seconds 512 nodes x (2 OpenPOWER 10-core POWER8 2.926 GHz, 512 GB memory, 4x Huawei ES3600P V3 1.2TB NVMe SSD, 100Gb Mellanox ConnectX4-EN) Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao Tencent Corporation Mark R. Nutter, Jeremy D. Schaub
Cloud	2016, $1.44 / TB **NADSort** 100 TB for $144 394 Alibaba Cloud ECS ecs.n1.large nodes x (Haswell E5-2680 v3, 8 GB memory, 40GB Ultra Cloud Disk, 4x 135GB SSD Cloud Disk) Qian Wang, Rong Gu, Yihua Huang Nanjing University Reynold Xin Databricks Inc. Wei Wu, Jun Song, Junluan Xia Alibaba Group Inc.	2016, $1.44 / TB **NADSort** 100 TB for $144 394 Alibaba Cloud ECS ecs.n1.large nodes x (Haswell E5-2680 v3, 8 GB memory, 40GB Ultra Cloud Disk, 4x 135GB SSD Cloud Disk) Qian Wang, Rong Gu, Yihua Huang Nanjing University Reynold Xin Databricks Inc. Wei Wu, Jun Song, Junluan Xia Alibaba Group Inc.
Minute	2016, 37 TB **Tencent Sort** 512 nodes x (2 OpenPOWER 10-core POWER8 2.926 GHz, 512 GB memory, 4x Huawei ES3600P V3 1.2TB NVMe SSD, 100Gb Mellanox ConnectX4-EN) Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao Tencent Corporation Mark R. Nutter, Jeremy D. Schaub	2016, 55 TB **Tencent Sort** 512 nodes x (2 OpenPOWER 10-core POWER8 2.926 GHz, 512 GB memory, 4x Huawei ES3600P V3 1.2TB NVMe SSD, 100Gb Mellanox ConnectX4-EN) Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao Tencent Corporation Mark R. Nutter, Jeremy D. Schaub
Joule 10^{10} recs	2013, 168,242 Joules **NTOSort** 59,444 records sorted / joule Intel i7-3770K, 16GB RAM, Nsort, Windows 8, 16 Samsung 840 Pro 256GB SSDs, 1 Samsung 840 Pro 128GB SSD Andreas Ebert Microsoft	2013, 168,242 Joules **NTOSort** 59,444 records sorted / joule Intel i7-3770K, 16GB RAM, Nsort, Windows 8, 16 Samsung 840 Pro 256GB SSDs, 1 Samsung 840 Pro 128GB SSD Andreas Ebert Microsoft

Join

- Most common operation
  - Need to be heavily optimized
- Why Join
  - We normalize tables to avoid redundancy
  - Need to join them to get original tuples
- Naive way:
  - Cross product, followed by selection
  - NOT PRACTICAL!



Nested Loop Join

- Simple loop

```
for r in R:
  for s in S:
    if condition(r,s) output
```

R $\bowtie_{id}$ S

Cost: **B(R) + |R|.B(S)**

S $\bowtie_{id}$ R ← Better!

Cost: **B(S) + |S|.B(R)**

R

id	..
1	
2	
3	
4	
5	
6	
7	
8	
9	

S

id	..
1	
3	
5	
7	
9	
11	

Notation

|R|, |S|: # tuples in R, S

B(R), B(S): # pages for R,S

M: buffer size

R $\bowtie_{id}$ S = (1,1)
(3,3)
(7,7)
(9,9)

2nd term usually more exp
usually | x | is order of mag >> B(x)
(i.e. # tuples in X >> # pages for X

read blindly, dont store in buffer :(

Nested Loop Join

R ⋈$_{id}$ S

Cost: **B(R) + |R|.B(S)**

S ⋈$_{id}$ R

Cost: **B(S) + |S|.B(R)**

For R join S:
+ B(R) *cause you need to scan through R (read the entire R eventually)*
+ |R| . B(S) *cause you have to scan through S for |R| times.*

Example

R = 100,000 S = 40,000
B(R) = 1,000 B(S) = 500

~ 83 minutes

R ⋈$_{id}$ S:

B(R) + |R|.B(S) = 50,001,000 (I/Os)

20%

S ⋈$_{id}$ R:

B(S) + |S|.B(R) = 40,000,500 (I/Os)

~ 66 minutes

Block Nested Loop Join

- Use available buffers
  - M-2 for outer, 1 for inner relation, 1 for output

```
for blockR in R:
  for blockS in S:
    for r in blockR:
      for s in blockS:
        if condition(r,s) output
```

$R \bowtie_{id} S$

Cost: **B(R) + B(R).B(S)/(M-2)**

$S \bowtie_{id} R$

Cost: **B(S) + B(R).B(S)/(M-2)**

R

id	..
1	
2	
3	
4	
5	
6	
7	
8	
9	

S

id	..
1	
3	
5	
7	
9	
11	

Notation
|R|, |S|: # tuples in R, S
B(R), B(S): # pages for R,S
M: buffer size

$R \bowtie_{id} S$ = (1,1) (3,3) (7,7) (9,9)

Use same join algo but join block by block
For R join S

M-2 -> for R, 1 for S, 1 for output

- Read block in R, then read all blocks of S the last 2 loops no I/O, purely in memory
- Read B(S)/(M-2) at a time
(M-2) -> join (M-2) blocks at a time

best case: *(M-2) = B(S)*
cost = B(S) + B(R)

Hash Join:

For every value u wanna join, you store in hash table

Assumption: hash table can fit in memory

(Similar to: 2 arrays, wanna find the common elements. Build HT on smaller array, scan through larger array to see if element is in HT anot)

Block Nested Loop Join

$R \bowtie_{id} S$

Cost: **B(R) + B(R).B(S)/(M-2)**

$S \bowtie_{id} R$

Cost: **B(S) + B(R).B(S)/(M-2)**

Example

R = 100,000 S = 40,000

B(R) = 1,000 B(S) = 500

$R \bowtie_{id} S$ (Nested Loop)

B(R) + |R|.B(S) = 50,001,000 (I/Os) **~ 83 minutes**

$R \bowtie_{id} S$ (Block Nested Loop)

B(R) + B(R).B(S) = 501,000 (I/Os) **~ 50 second**

$S \bowtie_{id} R$ (Block Nested Loop), M=B(S)+2

B(S) + B(R) = 1,500 (I/Os) **~ 0.1 second**

Sort-Merge Join

- Sort-Merge:
  - First, sort both relation
  - Then scan both relations, like a merge

Good if relations are sorted already

Sort R: $2.B(R).(1+ \lceil \log_M \lceil B(R)/M \rceil \rceil)$

Sort S: $2.B(S).(1+ \lceil \log_M \lceil B(S)/M \rceil \rceil)$

Scan: $B(R) + B(S)$

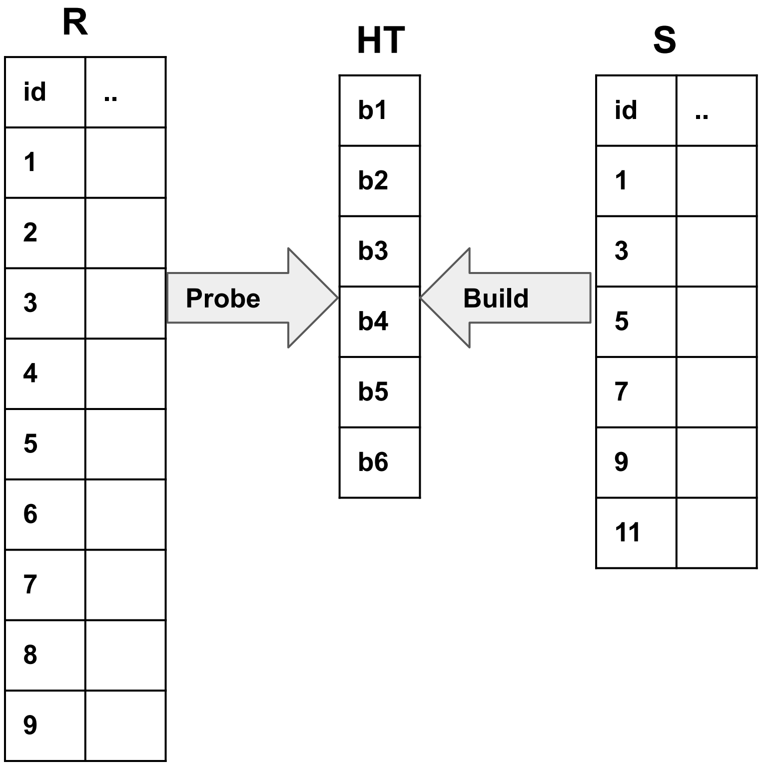
Total: $B(R) + B(S) + 2.B(R).(1+ \lceil \log_M \lceil B(R)/M \rceil \rceil) + 2.B(S).(1+ \lceil \log_M \lceil B(S)/M \rceil \rceil)$

Hash Join

- Basic Hash Join:
    - Build hash table, HT, on S
    - Assume: B(HT) <= M
    - Scan through R to check with the hash table

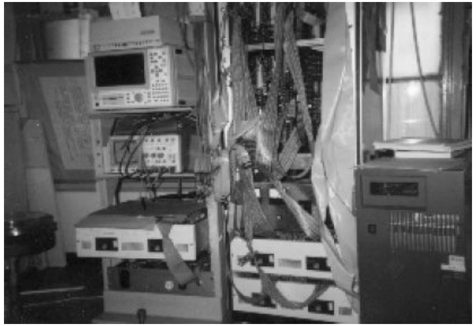
```
build hash table HT on S
for r in R:
    ff r in HT output
```

Cost: B(R) + B(S)



Hash Join

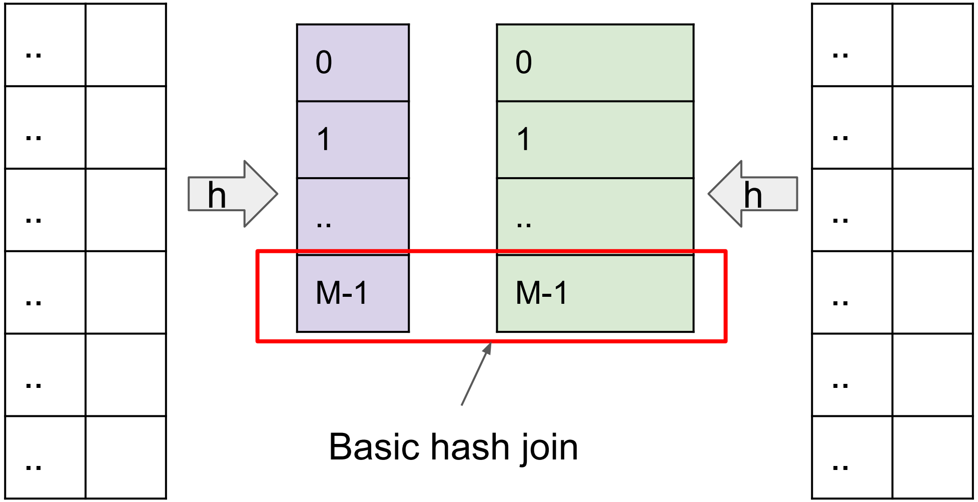
- Grace Hash Join
  - When HT doesn't fit in M buffer pages
  - Idea:
    - Partition R into M buckets, using *h*
    - Partition S into M buckets, using *h*
    - We can assume that each bucket fit into M pages (there're tricks to make sure this!)
    - Join buckets at the same positions

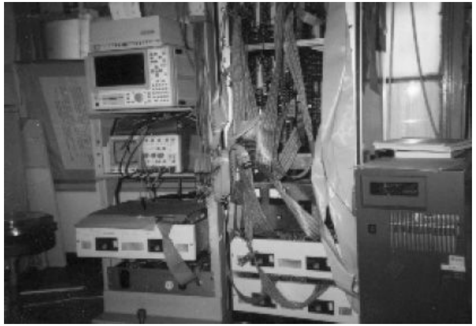


GRACE Parallel Relational Database Machine (1981-)

Hash Join

- Grace Hash Join





GRACE Parallel Relational Database Machine (1981-)

Cost: 3.(B(R) + B(S))

Join Recap

Example
R = 100,000 S = 40,000
B(R) = 1,000 B(S) = 500
M = 100

Algorithm	**Cost**	**Time**
Nested Loop	B(S) + |S|.B(R)	40,000,500 I/Os = **4000s**
Block Nested Loop	B(S) + B(R).B(S)/(M-2)	6,102 I/Os = **0.61s**
Sort Merge	B(S) + B(R) + sort(R) + sort(S)	5,849 I/Os = **0.58s**
Grace Hash	3(B(R) + B(S))	4,500 I/Os = **0.45s**

4 orders of magnitudes

DBMS will choose for you

Summary

- Many different ways to execute operators
  - Select: scan vs. index
  - Sort: external sort
  - Join: nested loop, block nested loop, index, Grace hash
  - And many more
- No clear winner
  - DBMS has to choose