Algorithms for SQL Query Operators

Query Optimization

Query Optimization

- Introduction
- Unary Operators
- External Sorting
- Projection
- Binary Operators

Equivalent Queries

```
select c.cid, c.cname, c.email, p.pid, p.pname, p.price
from customer c, sales s, product p
where c.city = ‘Vancouver’ and p.company = ‘lego’ and
s.year = 2019 and s.cid = c.cid and s.pid = p.pid
```

```
select c.cid, c.cname, c.email, p.pid, p.pname, p.price
from (select cid, cname, email from company where city = ‘Vancouver’) as c
natural inner join
(select cid, pid from sales where year = 2019) as s
natural inner join
(select pid, pname, price from products where company = ‘lego’) as p
```

```
select c.cid, c.cname, c.email, p.pid, p.pname, p.price
from       (select c1.cid, s1.pid
            from customer c1, sales s1
            where c1.city = ‘Vancouver’ and s1.year = 2019
            intersect
            select s2.cid, p1.pid
            from product p1, sales s2
            where p1.company = ‘lego’ and s2.year = 2019) as cp
natural inner join customer
natural inner join product
```

SQL is procedural

Operations are specified

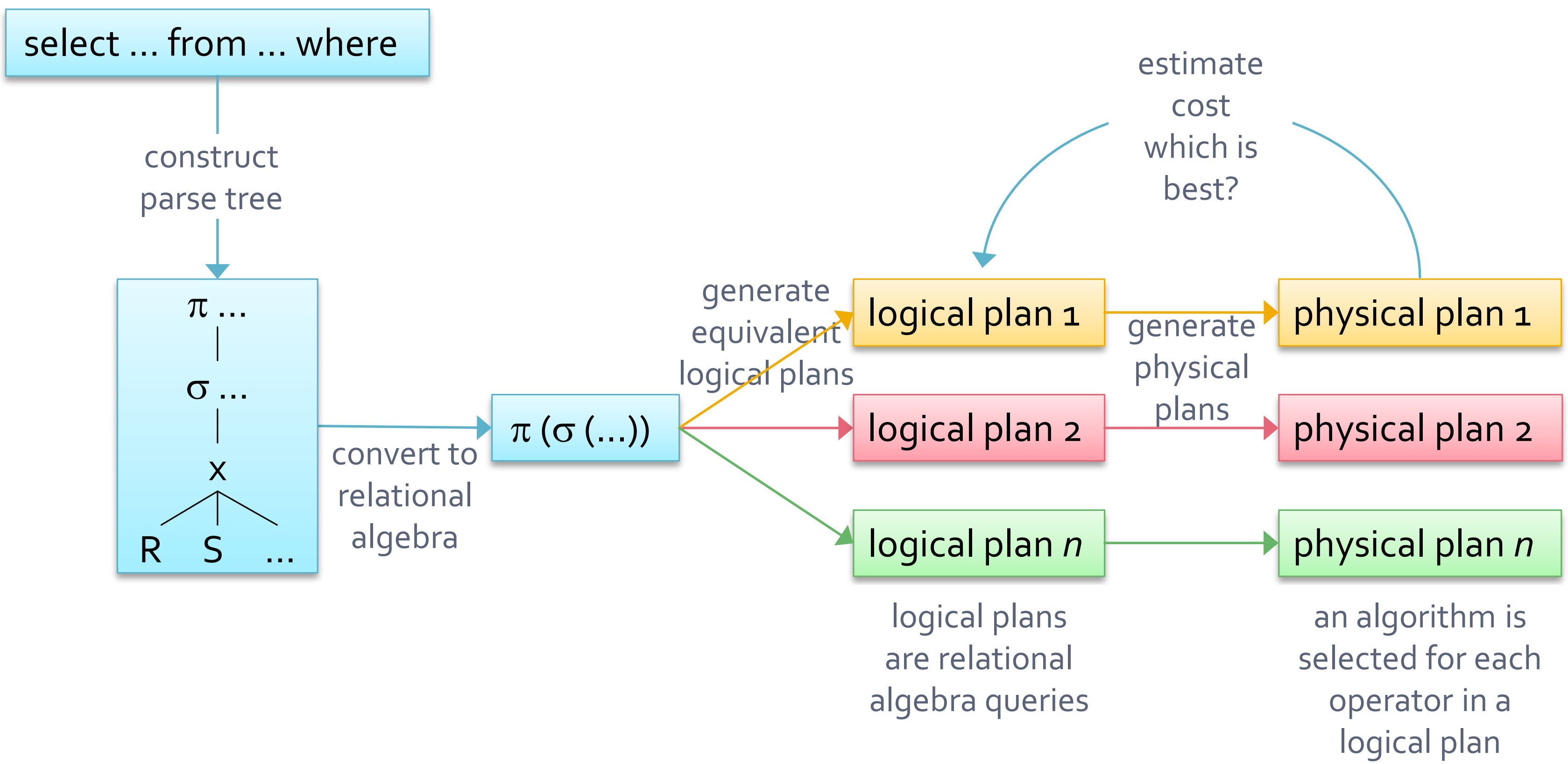
There are often equivalent queries

That are more or less efficient

Query optimization entails finding the best* query

*Actually a *good enough* query

Query Optimization



What are these algorithms?

Evaluating Queries

```
select c.cid, c.cname, c.email, p.pid, p.pname, p.price
from (select cid, cname, email from company where city = 'Vancouver') as c
natural inner join
(select cid, pid from sales where year = 2019) as s
natural inner join
(select pid, pname, price from products where company = 'lego') as p
```

$\pi_{cid,cname,email,pid,pname,price}(\pi_{cid,cname,email}(\sigma_{city = 'Vancouver'}(\texttt{Customer}))$

$\bowtie \pi_{cid,pid}(\sigma_{year = 2019}(\texttt{Sales}))$

$\bowtie \pi_{pid,pname,price}(\sigma_{company = 'lego'}(\texttt{Product})))$

Order of operations? 1 2 3 … maybe …

Size of input into next operation – intermediate relations?

Are results maintained in main memory?

What is the cost metric?

- A physical plan is made up of a sequence of steps
  - Each step corresponds to a relational algebra operation
  - Input is one or more relations
  - Output from each operation is a relation
- Some operations require low level processes
  - Scanning a table
  - Using an index to access a record

Query Model and Metrics

1.1

Terminology and Cost Metric

main memory frames

M frames available

Cost metric – number of blocks read or written from disk

why?

Cost of disk I/O dominates

For simplicity, assume data is accessed one block at a time

DB – on disk

contains

relations

B(*R*) – number of blocks of *R*

T(*R*) – number of records of *R*

bf(*R*) – records per block of *R*

Metadata – in system catalog

V(*R*,*a*) – number of distinct values for attribute *a* in *R*

Computation Model

$\pi_{\text{cid,cname,email,pid,pname,price}}(\pi_{\text{cid,cname,email}}(\sigma_{\text{city = 'Vancouver'}}(\texttt{Customer}))$

$\bowtie \pi_{\text{cid,pid}}(\sigma_{\text{year = 2019}}(\texttt{Sales}))$

$\bowtie \pi_{\text{pid,pname,price}}(\sigma_{\text{company = 'lego'}}(\texttt{Product})))$

B(Customer) = 100,000

T(Customer) = 1,000,000

V(Customer, city) = 100

Assume performed first

Cost?

- This section covers algorithms for query operations
  - There is often more than one for an operation
- Operations are considered in isolation
  - Assume that data is read from disk
    - In practice this is not always the case
  - And that the result is retained in memory – not written out

Interaction of operations discussed later

Unary Operators

1.2

Introduction

- A unary operator is an operation with a single operand
  - For SQL operators the operand is a table
    - Either a base table or the result of a previous query operation
- Unary operations
  - Selection $\sigma_{\texttt{salary > 100000}}(\texttt{Customer})$
  - Projection $\pi_{\texttt{name,salary}}(\texttt{Customer})$
    - Which may include duplicate removal
  - Sorting
  - Aggregations `AVG(salary)`

id	name	salary	...
154	bob	77000	
786	brie	120000	
001	kate	82000	
268	sue	63000	

85500

Simple Selection

```
SELECT *
FROM Customer
WHERE city = 'Vancouver'
```

don't do this …

$\sigma_{city\ =\ \text{'Vancouver'}}(\text{Customer})$

- A simple selection has a single condition
  - Complex selections are considered later
- Selections are satisfied by retrieving the matching records via an *access path*
  - Scanning the file and testing each record to determine if it matches the selection
    - Or using binary search if the file is sorted and has no index
  - which is unusual …
  - Using an index on the attribute in the condition

Cost of Simple Selections

- No index on the selection attribute
  - Linear search by scanning file, cost is *B* reads
  - If the selection attribute is a candidate key the scan can be terminated once a match has been found (cost is *B*/2)
  - If the file is sorted use binary search to find record(s)
    - $\log_2(B)$ + pages of matching records - 1

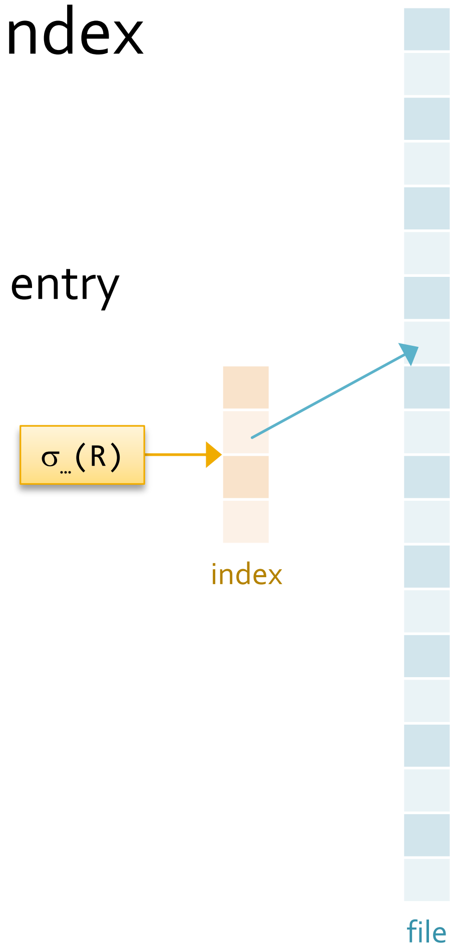
But it is unusual to have a sorted file with no primary index

- Index on the selection attribute
  - The cost is dependent on
    - The type of index – B+ tree, hash index, …
    - The height of the index
    - The number of records that match the selection
    - Whether the index is primary or secondary

Compare selections on
SIN
First name
City
Gender

Cost of Using an Index

- The cost of satisfying a selection with an index is composed of
  - Number of disk reads to use the index
    - i.e. to reach the leaf / bucket that contains the data entry
    - The number of leaves / size of the bucket
  - Number of blocks of the file with records that match the selection
    - Generally larger if the index is secondary
- Assume that indices are
  - Hash index – extensible or linear
  - B+ tree index

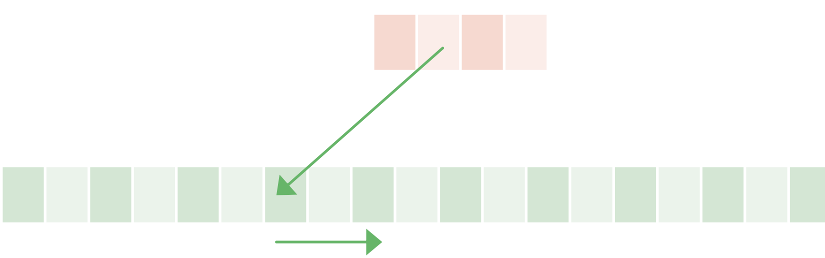


Cost to Search Index

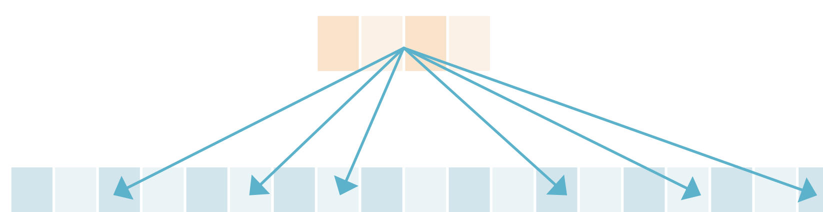
- B+ Tree
  - To find matching RIDs search tree
    - RIDs reside in leaf nodes
  - Cost: 1 disk read per level
- Additional leaf pages may have to be read
  - If index is dense or selection is inequality
    - i.e. entries are on multiple leaves
- Extensible hash index
  - Read directory
    - Probably 1 or 2 blocks
  - Read bucket
    - 1 block
- Linear hash index
  - Read bucket
    - Bucket may have overflow blocks
- Hash indexes only used for equality selections

Cost to Read Records

- Primary index
  - File is sorted by search key
    - Matching records are stored in consecutive blocks
  - Blocks read is number of records ÷ records per block
    - $1 + \lceil (\text{records} - 1) \div \text{bf}(R) \rceil$
    - Assumes worst case



- Secondary index
  - Matching records are not stored consecutively
  - Assume one disk read for each matching record
    - As records are scattered across the file
  - For large selections could be worse than a file scan



Simple Selection Cost

Access Method	Candidate Key Selection	Non Candidate Key Selection	Notes
Linear search	$B/2$	B	
Binary search	$\log_2(B)$	$\log_2(B) + x$	Must be sorted on selection attribute x = blocks of matching records
Primary B+ tree index	tree height + 1	tree height + x	x = blocks of matching records
Secondary B+ tree index	tree height + 1	tree height + w + y	w = leaf nodes of data entries - 1 y = number of matching records
Primary hash index	index height + 1	index height + w + x	w = blocks in bucket - 1 x = blocks of matching records
Secondary hash index	index height + 1	index height + w + y	w = blocks in bucket - 1 y = number of matching records

Notes: tree height usually 3 to 5; hash index "height" usually 1 or 2; root node of indexes may be resident in main memory which reduces cost by 1; value for *w* is usually 1 (particularly for a hash index); difference between *x* and *y* can be large; details on how to compute these costs follow

Complex Selections

- A complex selection is made of at least two terms connected by *and* ($\wedge$) and *or* ($\vee$)
  - The terms can reference different or the same attributes
  - Conjunctions are more selective *and* clauses
  - Disjunctions are less selective *or* clauses
- Complex selections are satisfied in much the same way as simple selections
  - If no index on any of the selection attributes scan the file
  - Use indices on selection attributes where possible
    - Use of indices is governed by the type of selection and index

Selections with no Disjunctions

- If only one index is available use the index and apply other selections in main memory
  - Either there is an index on only one of the attributes
  - Or an index with a compound key that references multiple selection attributes

    attributes in selection must form prefix of the key
  - Note the restrictions on the use of hash indices
- If multiple indexes are available

  contrast
  - Either use the most selective

    $\sigma_{\text{firstname = "Emma"} \wedge \text{lastname = "Lee"}}(\text{Patient})$
  - Or collect RIDs from leaves or buckets of indexes and take the intersection

    $\sigma_{\text{city = "Vancouver"} \wedge \text{msp} = 555123456}(\text{Patient})$

Selections with Disjunctions

- Selections with disjunctions are stated in *conjunctive normal form* (CNF) By the query optimizer
  - A collection of *conjuncts*
  - Each conjunct consists either of a single term, or multiple terms joined by *or*
    - e.g. $(A \wedge B) \vee C \vee D \equiv (A \vee C \vee D) \wedge (B \vee C \vee D)$
  - This allows each conjunct to be considered independently
- A conjunct can only be satisfied by indices if there is an index on all attributes of all of its disjunctive terms
  - If *all* the conjuncts contain at least one disjunction with no matching index a file scan is necessary

Selections with Disjunctions …

- Consider a selection of this form
  - $\sigma_{(a \vee b \vee c) \wedge (d \vee e \vee f)}(R)$
  - Where each of *a* to *f* is an equality selection on an attribute
- If each of the terms in either of the conjuncts has a matching index
  - Use the indexes to find the *rids*
  - Take the *union* of the *rids* and retrieve those records
  - For example, if there are indexes just on *a*, *b*, *c*, and *e*
    - Use the *a*, *b*, and *c* indexes and take the union of the *rids*
    - Retrieve the resulting records and apply the other criteria

if there was no index on *b* a file scan would be necessary

Projections

```
SELECT fName, lName
FROM Customer
```

$\pi_{\texttt{fName,lName}}(\texttt{Customer})$

- Only selected columns are retained
  - Reducing the size of the result relation
  - Projections can always be pipelined from other operations
    - Unless the SELECT clause includes DISTINCT
- A SELECT DISTINCT clause eliminates duplicates
  - Which requires sorting the relation
    - Or building a hash table on the relation

Processed without writing out the previous result

But what if the relation does not fit in main memory?

External Sorting

A Digression (2-1)

Sorting and Scanning

- It is sometimes necessary or useful to sort data as it is scanned (read)
  - To satisfy a query with an ORDER BY clause
  - Or because an algorithm requires sorted input
    - Such as projection or some join algorithms
- There are a number of ways in which a sort scan can be performed
  - Main memory sorting
  - B+ tree index
  - Multi-way mergesort

The cost to scan a file is $B(R)$

But only if R fits in main memory: $B(R) < M$

If $B(R) < M$ the cost to sort a file is $B(R)$

Internal vs. External Sorting

- Sorting a collection of records that fit within main memory can be performed efficiently
  - There are a number of sorting algorithms that can be performed in $n(\log_2 n)$ time
    - That is, with $n(\log_2 n)$ comparisons, e.g., Mergesort, Quicksort,
- Many DB tables are too large to fit into main memory at one time
  - So cannot simply be read into main memory and sorted
  - The focus in *external sorting* is to reduce the number of disk I/Os
    - As it is with optimization in general

Merge Sort – a Brief Reminder

- Consider the *Merge sort* algorithm
  - Input sub-arrays are repeatedly halved
  - Until they contain only one element
- Sub-arrays are then merged into sorted sub-arrays by repeated *merge* operations
  - merging two *sorted* sub-arrays can be performed in $O(n)$

```
mergesort(arr, start, end)
   if(start < end) //at least two elements
      mid = start + end / 2
      mergesort(arr, start, mid)
      mergesort(arr, mid+1, end)
      merge(arr, start, mid, mid+1, end)
```

$O(n \log_2 n)$

Naïve External Merge Sort

- Convert main memory merge sort to work on disk data
- Initial step - read 2 pages of data from file
  - Sort them and write them to disk
    - Results in *B*/2 sorted "*runs*" of size 2

Note: this does not make much sense, but is included for illustration

- Merge the first two sorted runs of size 2
  - Read the first page of the first two runs into input pages
  - Merge to a single output page, and write it out when full
  - When all records in an input page have been merged read in the second page of that run
  - Repeat for each pair of runs of size 2
  - There are now *B*/4 sorted runs of size 4
- Repeatedly merge runs until the file is sorted

Naïve External Merge Sort …

11	15	23	31	64	87	6	10	14	41	55	76	3	25	28	39	53	

disk pages contain three records

sorted runs of size 2

11	15	23
6	10	14

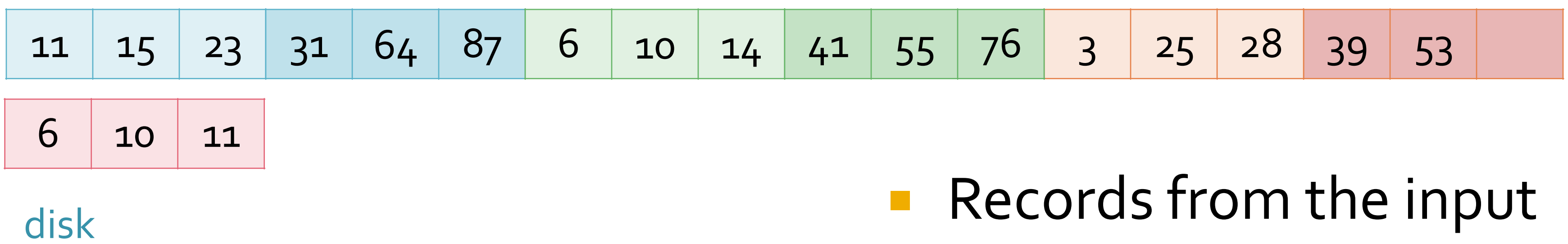
input buffers

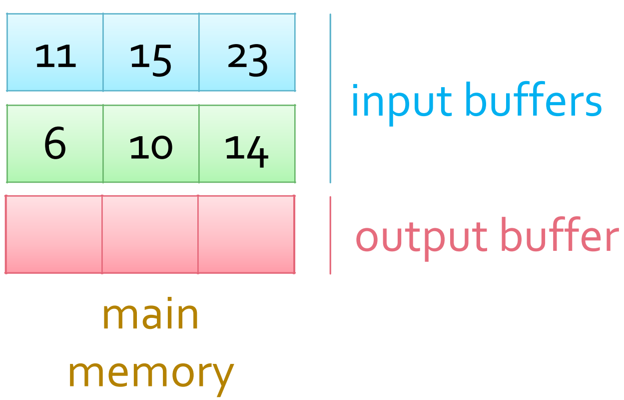
output buffer

main memory

- After the first sort pass the file consists of *B*/2 sorted runs each of two pages
- Read in the first page of each of the first two sorted runs
- Leaving a third page free as an output buffer

Naïve External Merge Sort …





- Records from the input pages are merged into the output buffer
- Once the output buffer is full it's contents are written out to disk, to form the first page of the first sorted run of length 4

Naïve External Merge Sort …

11	15	23	31	64	87	6	10	14	41	55	76	3	25	28	39	53	

6	10	11

disk

11	15	23
41	55	76
14	15	

input buffers

output buffer

main memory

uses only three main memory frames!

- At this point all of the records from one of the input pages have been processed
- The next page of that sorted run is read into the input page
- And the process continues

Cost of Naïve Merge Sort

- Assume that $B = 2^k$
  - After the first pass there are 2^{k-1} sorted runs
    - Each is two pages in size
  - After the second pass there are 2^{k-2} sorted runs, of length 4
  - After the k^{th} pass there is one sorted run of length B
- The number of passes is therefore $\lceil \log_2 B \rceil$
- In each pass all the pages of the file are read and written for a total cost of $\lceil \log_2 B \rceil * 2B$

  B pages read and *B* pages written

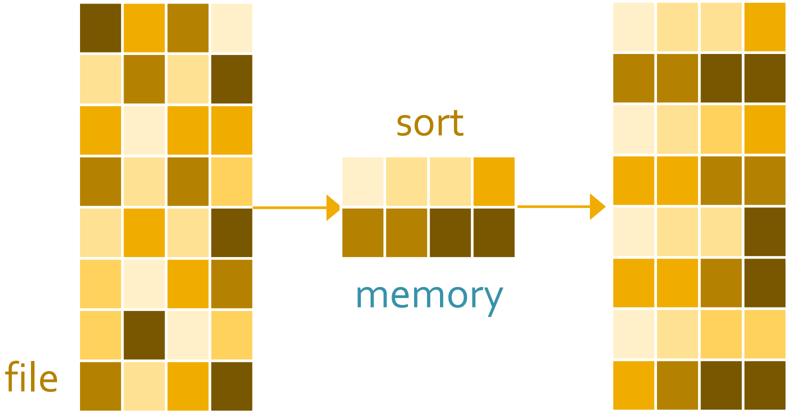
  - Note that only 3 frames of main memory are required!
  - Also note that main memory costs are ignored
- The algorithm can be improved in two ways

First Stage Improvement

- In the first stage of the naive process pairs of pages are read into main memory, sorted and written out
  - Resulting in *B*/2 runs of size 2
- To make effective use of main memory, read *M* pages, and sort them

M main memory pages available

  - After the first pass there will be *B*/*M* sorted runs, each of length *M*
  - This reduces the number of subsequent merge passes



Merge Pass Improvement

- In the merge passes perform an M-1 way merge
  - M-1 input pages, one for each of M-1 sorted runs and
  - 1 page for an output buffer
- The first items in each of the M-1 input partitions are compared to determine the smallest
- Each merge pass merges M-1 runs
  - After the first pass the runs are size $(M\text{-}1)*M$
  - This results in less merge passes, and less disk I/O

Runs were size M after first pass

Cost of External Merge Sort

- The initial pass produces B/M sorted runs of size M
- Each merge pass reduces the number of runs by a factor of M-1
  - The number of merge passes is $\lceil \log_{M-1} \lceil B/M \rceil \rceil$
- Each pass requires that the entire file is read and then written
  - Total cost is therefore $2B\,(\lceil \log_{M-1} \lceil B/M \rceil \rceil + 1)$

first pass

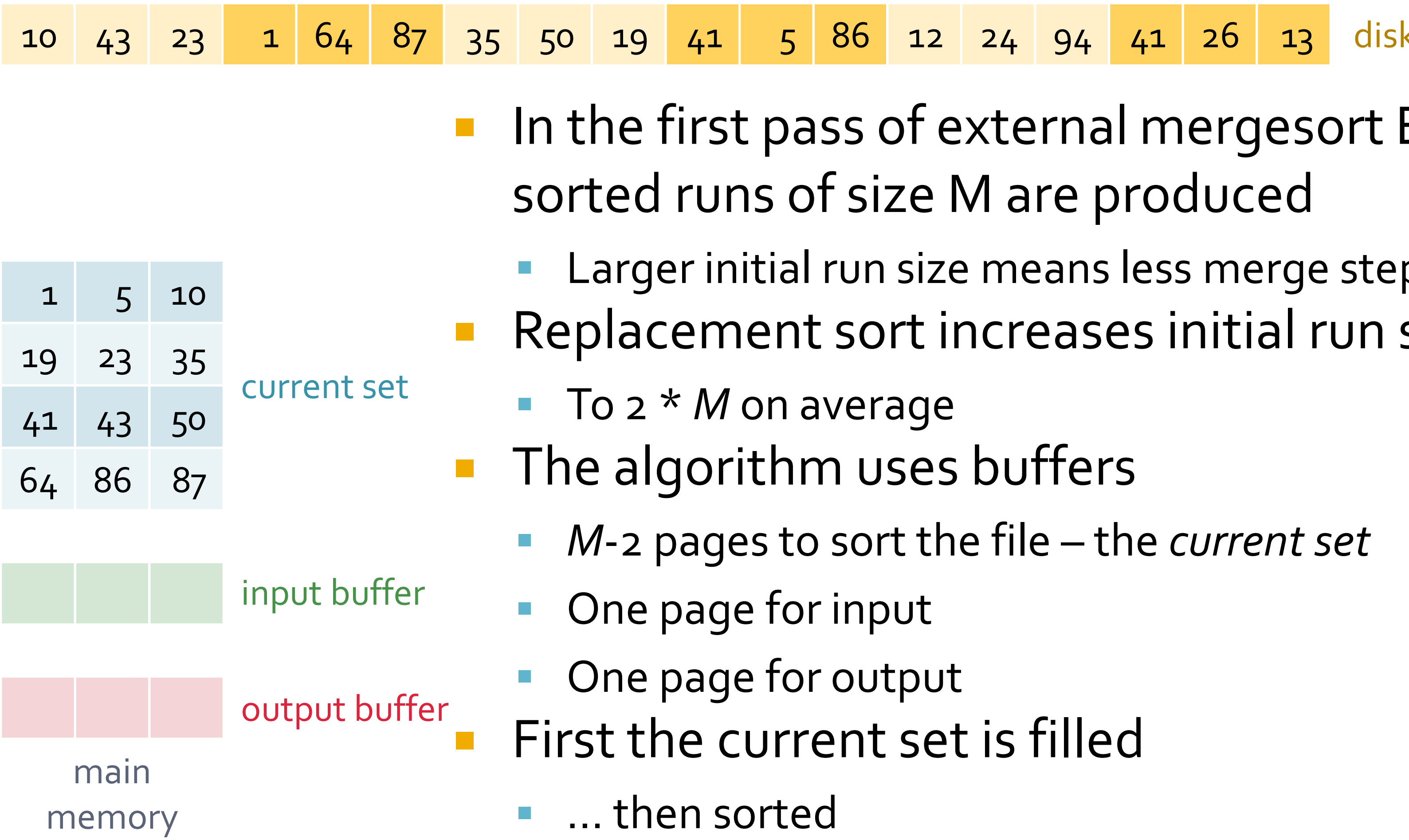
- M is typically relatively large this so this reduction over two-way merge is considerable

Number of Passes

B = 1,000,000		
M	$\lceil \log_2 B \rceil$	$\lceil \log_{b-1} \lceil B/M \rceil \rceil + 1$
3	20	20
5	20	10
200	20	3
2,000	20	2

Even a large file can usually be sorted in two passes (a cost of 4*B* I/Os to sort and write out) assuming a reasonable size for *M*

Replacement Sort



- In the first pass of external mergesort B/M sorted runs of size M are produced
  - Larger initial run size means less merge steps
- Replacement sort increases initial run size
  - To 2 * *M* on average
- The algorithm uses buffers
  - *M*-2 pages to sort the file – the *current set*
  - One page for input
  - One page for output
- First the current set is filled
  - ... then sorted

Replacement Sort

10	43	23	1	64	87	35	50	19	41	5	86	12	24	94	41	26	13

disk

- Once the current set is sorted the next page of the file is read into the input buffer
- The smallest record from the current set, and input buffer, is put in the output buffer
- The first element of the current set is now free and is replaced with the first record from the input buffer
- This process is repeated until the output buffer is full and all the values in the input buffer are in the current set

current set

12	24	94
19	23	35
41	43	50
64	86	87

input buffer

12	24	94

output buffer

1	5	10

main memory

Replacement Sort

10	43	23	1	64	87	35	50	19	41	5	86	12	24	94	41	26	13	disk

1	5	10

current set

12	24	94
19	23	35
41	43	50
64	86	87

input buffer

41	26	13

output buffer

main memory

- When the output buffer is full
  - The next page of the file is read into the input buffer
  - And the output buffer is written to disk as the first page of the first sorted run
  - As the process continues the current set is periodically re-sorted
- When a value is read that is less than the values in the output buffer the current set must be written out
  - The process starts again to generate the next run

Revisiting I/O Costs

- In practice it may be more efficient to make the input and output buffers larger than one page
  - This reduces the number of runs that can be merged at one time, so may increase the number of passes required
  - But, it allows a sequence of pages to be read or written to the buffers, decreasing the actual access time per page
- We have also ignored CPU costs
  - If double buffering is used, the CPU can process one part of a run while the next is being loaded into main memory
    - Double buffering also reduces the amount of main memory available for the sort

Note: B+ Trees and Sorting

- Primary B+ tree index
  - The index can be used to find the first page, but
  - Note that the file is already sorted!
- Secondary B+ tree index
  - Leaves point to data records that are not in sort order
    - In the worst case, each data entry could point to a different page from its adjacent entries
  - Retrieving the records in order requires reading all of the index leaf pages, plus one disk read *for each record*!
  - In practice external sort is likely to be much more efficient than using a secondary index

Secondary indices are not useful for retrieving large selections

Projections

3.1

Projection and Duplicate Removal

- Naively, projection and duplicate removal entails
  - Scan the table, remove unwanted attributes, and write it back
    - cost ≈ 2*B* disk I/Os
  - Sort the result, using all of its attributes as a compound sort key
    - cost ≈ 4*B*, possibly more if the file is very large
  - Scan the result, removing the adjacent duplicates as they are encountered; cost ≈ *B*
    - The cost to write out the result of this last stage is not included; it may be the last operation or may be pipelined into another operations
- It appears that the total cost is 7*B* disk I/Os, but this process can be much improved by combining multiple steps

SELECT DISTINCT fname, lname
FROM Customer

The cost to write the result is less than *B*

Again, less than 4*B*

And again, the relation size is now less than *B*

Sort Projection Cost

- The initial scan is performed as follows
  - Read M pages and remove unwanted attributes
  - Sort the records, and remove any duplicates
  - Write the sorted run to disk
  - Repeat for the rest of the file, for a total cost of $2B$
    - Actually less than $2B$ since the result will be smaller than B
- Perform merge passes as required on the output from the first stage
  - Remove any duplicates as they are encountered
  - If only one merge pass is required the cost is $\approx 1B$
- For a total cost of $\approx 3B$

The final result size can be estimated

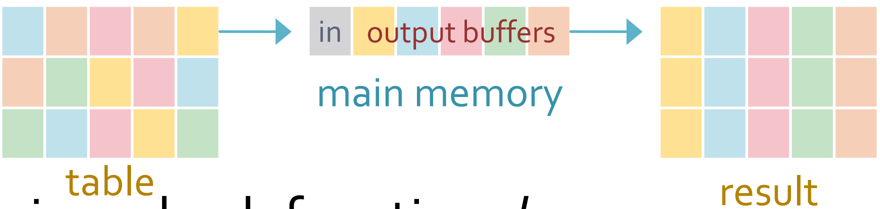
from attribute size

and other statistics

Unless more merge passes are required

Hash Projection – Partitioning

- Duplicates can also be identified by using hashing
- Duplicate removal by hashing has two stages
  - Partitioning and probing
- In the partitioning stage



  - Partition into *M*-1 partitions using a hash function, *h*
    - With an output buffer for each partition, and one input buffer
  - The file is read into main memory one page at a time, with each record being hashed to the appropriate buffer
    - Output buffers are written out when full
- Partitions contain records with different attribute values
  - Duplicates are eliminated in the next stage

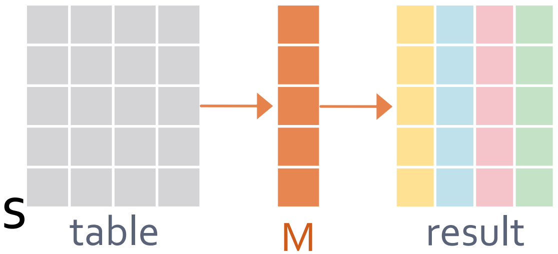
Hash Projection – Probing

- The duplicate elimination stage uses a second hash function h_2 ($h_2 \neq h$) to reduce main memory costs
  - An in-memory hash table is built using h_2
  - If two records hash to the same location they are checked to see if they are duplicates
  - Duplicates can, instead, be removed using in-memory sorting
- If each partition produced in the partitioning stage can fit in main memory the cost is
  - Partitioning stage: $2B$
  - Duplicate elimination stage: B, for a total cost of $3B$
- This is the same cost as projection using sorting

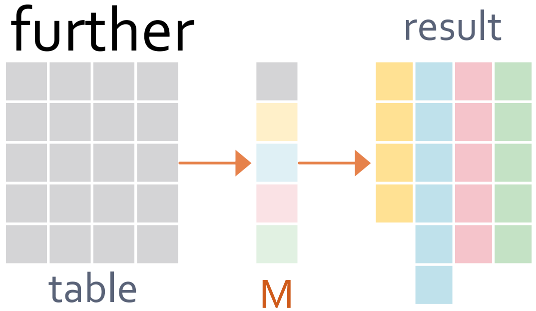
Approximate cost: actual cost is less since the result is smaller than the original file

Sort and Hash Projection Compared

- Sort and hash projection have the same cost (3*B*)
- If $M > \sqrt{(B)}$ sorting and sort projection can be performed in two passes
  - The first pass produces *B*/*M* sorted runs
  - If there are less than *M*-1 of them only one merge pass is required



- Hash projection partitions are different sizes
  - If just one partition is greater than *M*-1, further partitioning is required
    - Regardless of the overall size of the file



Aggregations

- Aggregations without groups are simple to compute
  - Scan the file and calculate the aggregate amount
  - Requires one input buffer and a variable for the result
  - Aggregations can usually be pipelined from a previous operation

```
SELECT MIN(gpa)
FROM Student
```

- Aggregations with groups require more memory
  - To keep track of the grouped data
  - They can be calculated by sorting or hashing on the group attribute(s)
  - Or by using an index with all the required attributes

```
SELECT AVG(income)
FROM Doctor
GROUP BY specialty
```

Sorting and Groups

- The table is sorted on the group attribute(s)
- The results of the sort are scanned and the aggregate operation computed
  - These two processes can be combined in a similar way to the sort based projection algorithm
- The cost is driven by the sort cost
  - $3B(R)$ if the table can be sorted in one merge pass
  - Final result is typically much smaller than the sorted table

Hashing and Groups

- In the hash based approach an in-memory hash table is build on the grouping attribute
  - Hash table entries consist of
    - ⟨grouping-value, running-information⟩

e.g. count and sum for AVG

- The table is scanned and for each record
  - Probe the hash table to find the entry for the group that the record belongs to, and
  - Update the running information for that group
- Once the table has been scanned the grouped results are computed using the hash table entries
  - If the hash table fits in main memory the cost is $B(R)$

Aggregations and Indexes

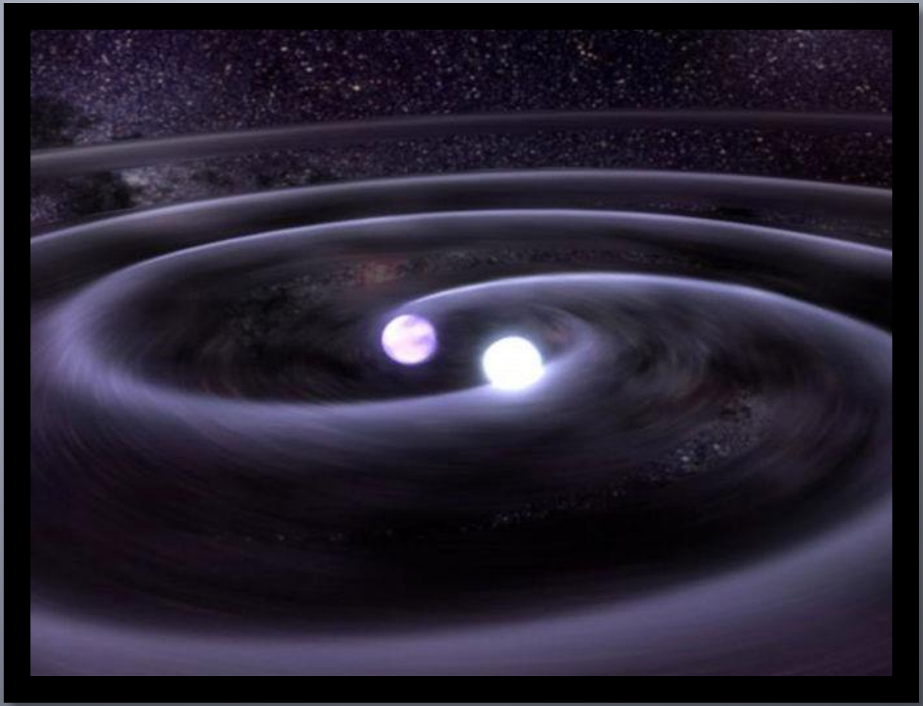
- It may be possible to satisfy an aggregate query using just the data entries of an index
  - The search key must include *all* of the attributes required for the query
    - The data entries may be sorted or hashed, and
    - No access to the records is required
  - If the GROUP BY clause is a *prefix* of a tree index, the data entries can be retrieved in the grouping order
    - The actual records may also be retrieved in this order
- This is an example of an *index-only* plan

This may seem unlikely

but an index may be created for this use

Binary Operations

3.2 / 4.1



Joins

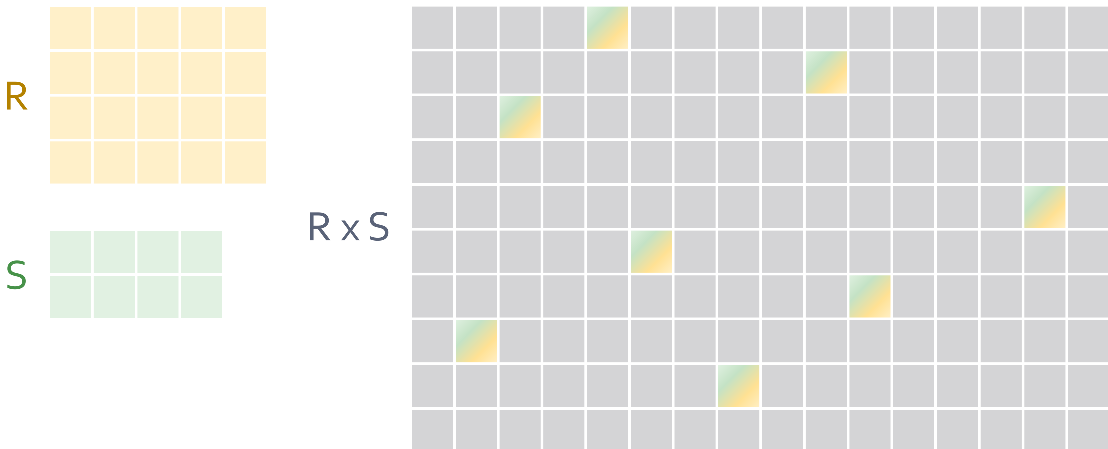
$Customer \bowtie Account$

```
SELECT *
FROM Customer NATURAL INNER JOIN Account
```

$\sigma_{C.sin = A.sin}(Customer \times Account)$

- A join is defined as a Cartesian product followed by a selection
  - Where the selection is the join condition
  - A natural join's condition is equality on all attributes in common
- Cartesian products typically result in much larger tables than joins
  - It is important to be able to efficiently implement joins

Join Algorithms



How do we find joined records without searching the entire space?

Algorithms

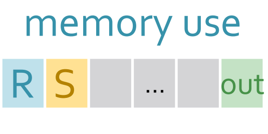
Nested loop joins
Sort-merge join
Hash join

- T(*R*) = 10,000 and T(*S*) = 4,000
  - Assume *S* has a foreign key that references *R*
    - So records in *S* relates to at most one record in *R*
- The sizes of the join and the Cartesian product are
  - Cartesian product – 40,000,000 records
  - Natural join – 4,000 (if every *s* in *S* relates to an *r* in *R*)

Simple Nested Loop Joins

- There are three nested loop join algorithms that compare each record in one relation to each record in the other
  - They differ in how often the inner relation is read
- Tuple nested loop join
  - Read one page of *R* at a time
  - For each *record* in *R*
  - Scan *S* and compare to all *S* records
  - Result has the same ordering as *R*
- Improved nested loop join
  - As tuple nested loop join but scan and compare *S* for records of *R* one page at a time

Cost = $B(R) + (T(R) * B(S))$



$R \bowtie_{R.i=S.j} S$

```
for each record r ∈ R
  for each record s ∈ S
    if r_i = s_j then
      add ⟨r,s⟩ to result
```

Cost = $B(R) + (B(R) * B(S))$

Block Nested Loop Join

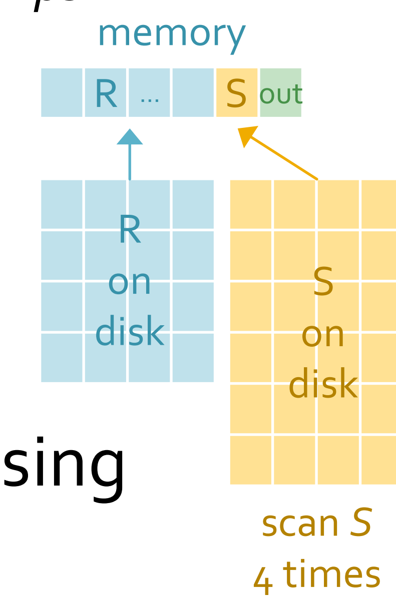
- The simple nested loop join algorithms do not make effective use of main memory
  - Both require only two input buffers and one output buffer
- The algorithm can be improved by making the input buffer for *R* as large as possible

  - Use $M - 2$ pages as an input buffer for the outer relation
  - 1 page as an input buffer for the inner relation, and
  - 1 page as an output buffer
  - If the smaller relation fits in $M - 2$ pages the cost is $B(R) + B(S)$
    - CPU costs are reduced by building an in-memory hash table on *R*, using the join attribute for the hash function

Why *Block* Nested Loop Join?

- What if the smaller relation is larger than M-2?
  - Break R, the outer relation, into *blocks* of $M - 2$ pages
    - I refer (somewhat flippantly) to these blocks as *clumps*
  - Scan S once for each *clump* of R
    - Insert concatenated records $\langle r, s \rangle$ that match the join condition into the output buffer
    - S is read $\lceil B(R)/(M\text{-}2) \rceil$ times

      M-2 is the clump size
  - The total cost is $B(R) + (\lceil B(R)/(M\text{-}2) \rceil * B(S))$



- Disk seek time can be reduced by increasing the size of the input buffer for S
  - Which may increase the number of times that S is scanned

Index Nested Loop Join

- Indexes can be used to compute a join where one relation has an index on the join attribute
  - The indexed relation is made the *inner* relation (call it *S*)
  - Scan the outer relation *B*(*R*) reads
  - While retrieving matching records of *S* using the index index cost?
- The inner relation is never scanned
  - Only records that satisfy the join condition are retrieved
    - Unlike the other nested loop joins this algorithm does not compare every record in *R* to every record in *S*
- Cost depends on the size of *R* and the type of index
  - B(R) + (T(R) * ⟨index cost⟩)

Index Nested Loop Join Cost

- The cost of index nested loop join is dependent on the type of index and the number of matching records
- The outer relation is scanned and records of S retrieved by using the index for each record of *R*
  - Search index for matching *RID*s – access leaf or bucket
    - If no matching records move on to next record of *R*
  - Retrieve matching records
    - One disk read if a single S record matches one R record
    - If multiple *S* records match to a single *R* the cost is dependent on the number of records and whether the index is primary or secondary

System catalog records data to estimate cost

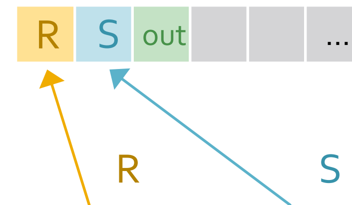
Sort-Merge Join Introduction

- Assume that both tables to be joined are sorted on the join attribute
  - The tables may be joined with one pass
    - Like merging two sorted runs

cost = $B(R) + B(S)$

- Read in pages of *R* and *S* – join on *x*
- While x_r != x_s
  - If $x_r < x_s$ move to the next *R* record else
  - Move to the *S* next record
- If $x_r == x_s$
  - Concatenate *r* and *s*, and
  - Add to output buffer
- Repeat until all records have been read

But R and S may not be sorted on the join attribute



R

id	...
11	
17	
18	
23	
34	
41	
44	
47	
49	
53	
61	
...	

S

id	...
11	
16	
20	
23	
32	
40	
41	
42	
46	
49	
52	
...	

Sort-Merge Join

- The sort-merge join* combines the join operation with the merge step of external merge sort
  - The first pass makes sorted runs of *R* and *S* of size *M*
    - *R* and *S* are processed independently
  - Merge runs of *R* and *S* as external merge sort until the combined number of sorted runs is less than *M*
    - If *M* is large or *R* and *S* are small this step may not be necessary
  - The final merge phase of the external sort algorithm is combined with the join, by comparing the runs of *R* and *S*
    - Records that do not meet the join condition are discarded
    - Records that meet the condition are concatenated and output

*aka sort-join

Memory Requirements

- Given sufficient main memory sort-merge join can be performed in two passes
  - For a cost of $3(B(R) + B(S))$

cost to write out final result not included

- Main memory must be large enough to allow an input buffer for each sorted run of *both* *R* and *S*
  - Main memory must be greater than $\sqrt{(B(R) + B(S))}$ to perform the join in two passes
    - Initial pass produces $B(R) / M + B(S) / M$ sorted runs of size M
    - If M is greater than $\sqrt{(B(R) + B(S))}$ then $(B(R) / M + B(S) / M)$ must be less than M

Memory Requirements Example

$M > \sqrt{(B(R) + B(S))}$

$M < \sqrt{(B(R) + B(S))}$

main memory

$M = 10$

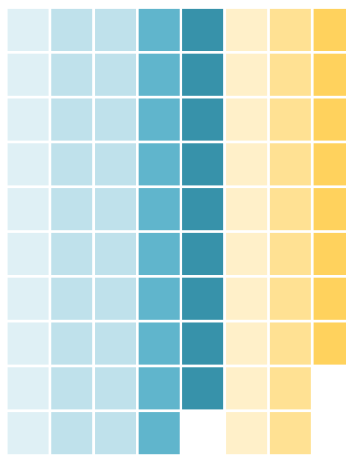
input page for each sorted run

$M = 8$

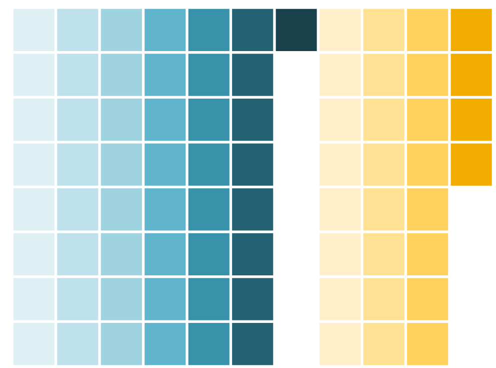
insufficient frames for page for each run

$B(R) = 49$

$B(S) = 28$



sorted runs of *R* and S after initial sort pass



sorted runs of *R* and S after initial sort pass

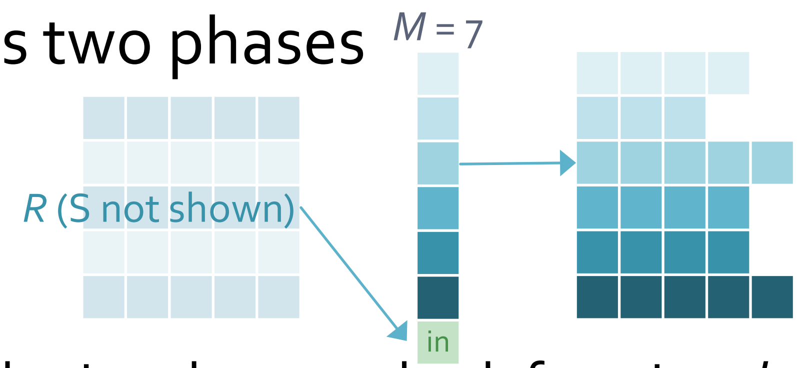
Must perform another merge pass

Zig-Zag Join

- If *both* relations have a primary tree index on the join attribute a *zig-zag join* can be performed
  - Scan the leaves of the two B+ trees in order from the left
    - i.e. from the record with the smallest value for the join attribute
  - When the search key value of one index is higher, scan the other index
  - When both indexes contain the same search key values matching records are retrieved and concatenated
  - Recall that the index is typically much smaller than the file

Cost = blocks of leaves of both indexes + blocks of matching records

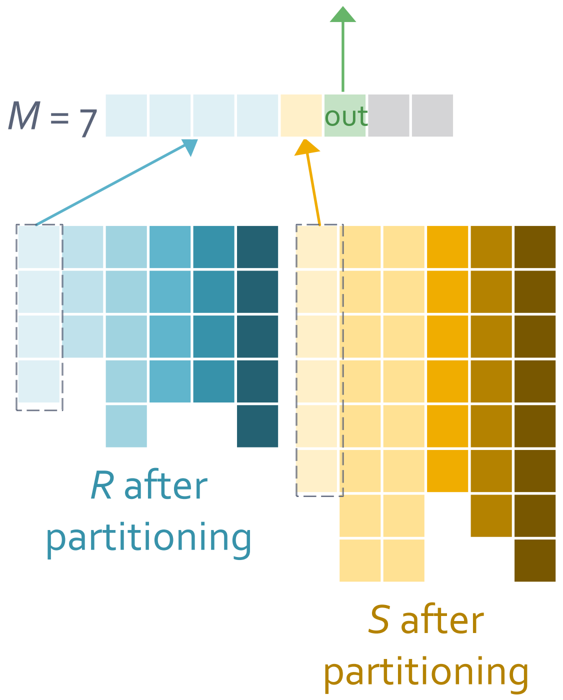
Hash Join – Partitioning



- The hash join algorithm has two phases
  - Partitioning, and
  - Probing
- Partitioning
  - Both relations are partitioned using the *same* hash function, *h*, on the join attribute
    - Records in one partition of *R* can only match records in the matching partition of *S*
  - One input buffer page and *M*- 1 output buffer pages are used to make *M* - 1 partitions for each relation
  - If the largest partitions of *both* relations do not fit in main memory, the relations must be further partitioned

Hash Join – Probing

- Probing
  - Read in one partition of *R*, where *R* is the smaller relation
  - To reduce CPU costs, build an in memory hash table using hash function h_2 ($h_2 \neq h$)
  - Read the corresponding partition of *S* into an input buffer one page at a time
  - Join matching records using the hash table
  - Repeat for each partition of *R*
- Cost
  - If each partition of one relation fits in main memory the overall cost is $3(B(R) + B(S))$



Memory Requirements

- Relations must be partitioned until the largest partition of the smallest relation (*S*) fits in main memory
- Ideally only one partitioning step is required
  - Which requires that M - 2 is > $\sqrt{(B(S))}$
    - Buffers for *S* and for the output are needed
    - Partitioning produces $B(S)$ - 1 partitions
    - Of average size $M / (B(S) - 1)$
  - If M - 2 is > $\sqrt{(B(S))}$ the cost of hash join is $3(B(R) + B(S))$
- If $M < \sqrt{(B(S))}$ then $B(S) / M$ must be larger than M, and the partitions are larger than main memory
  - Therefore the relations must be further partitioned

Assuming that partitions are the same size

Hybrid Hash Join

- Hybrid hash join can be used if *M* is large
  - Retain an entire partition of the smaller relation (*S*) during the partitioning phase
    - Eliminating the need to write out the partition, and read it back in during the probing phase
  - Matching *R* records are joined and written out to the result when *R* is partitioned
  - Hence the records of both *R* and *S* belonging to that partition are only read once
- This approach can be generalized to retain more than one partition where possible

Generalized Hybrid Hash Join

- Partition S (the smaller relation) into k partitions
  - Retain t partitions, S_1, … S_t in main memory
  - The remaining k – t partitions , S_{t+1}, … S_k are written to disk
- Partition R into k partitions
  - The first t partitions are joined to S since those t partitions of S are still in main memory
  - The remaining $k - t$ partitions are written to disk
- Join the remaining $k - t$ partitions as normal
- Cost is $B(R) + B(S) + 2 * ((k - t)/k) * (B(R) + B(S))$
  - $= (3 - 2 * t / k)(B(R) + B(S)) \approx (3 - 2 * M / B(R)) (B(R) + B(S))$

The cost improvement is incremental

Choosing Values for *k* and *t*

- There must be 1 main memory buffer for each partition
  - So $k \leq M$
    - Hybrid hash join is only used where M >> B(*S*), such that $(B(S) / k) < M$
- The ratio t / k, , should be as large as possible
  - And $t / k * B(S) + k - t \leq M$
    - The retained partitions must fit in main memory with sufficient buffers for the other $(k – t)$ partitions
  - One approach: retain one partition and make as few partitions as possible $t = 1$ and k as small as possible

k = the number of partitions

t = the number of partitions to be retained in main memory

t/k = fraction of S kept in main memory

$t = 1$, k small

Hybrid Hash Join Example

- Statistics
  - B(R) = 100,000
  - B(S) = 1,000
  - M = 200, note that √(B(S)) = 100
- Choose values for k and t
  - k is the number of partitions and t is the number to be retained in main memory
    - Select t = 1

k = 6 each partition is 167 blocks, 1 is retained leaving 33 blocks for 1 input buffer and 5 output buffers for the other partitions

  - k should be as small as possible while still allowing
    - one partition to be retained in main memory
    - one output page for each if the other (k-t) partitions
    - one input page

Hybrid Hash Join Example

partition S – read in all of S and write out $(k\text{-}t) / k = 5/6$ of S and retain one partition

use	partition 1 of S (s_1)					s_2	s_3	s_4	s_5	s_6	in		
frames			…									…	
#	0	1	…	165	166	167	168	169	170	171	172		

partition R – read in all of R, write out $(k\text{-}t) / k = 5/6$ of R and join partition 1 of R and S

use	s_1					r_2	r_3	r_4	r_5	r_6	in	result	
frames			…									…	…
#	0	1	…	165	166	167	168	169	170	171	172	173	

read in second partition of S and scan and join second partition of R

use	s_2					in (r_2)	result						
frames			…					…					
#	0	1	…	165	166	167	168						

repeat for the remaining four partitions of R and S

Hybrid Hash Join Example – Cost

- Statistics
  - B(*R*) = 100,000
  - B(*S*) = 1,000
  - k = 6, t = 1
- Cost
  - Read all of *S* – cost = B(*S*) = 1,000
  - Write out 5/6 of *S* – cost = B(*S*) * 5/6 = 833
  - Read all of *R* – cost = B(*R*) = 100,000
  - Write out 5/6 of *R* – cost = B(*S*) * 5/6 = 83,333
  - Read remaining partitions of *R* – cost = 833
  - Scan and probe matching partitions of *S* – cost = 83,333
  - Total cost = *B*(*R*) + *B*(*S*) + 2 * (5/6) * (*B*(*R*) + *B*(*S*)) = 269,333

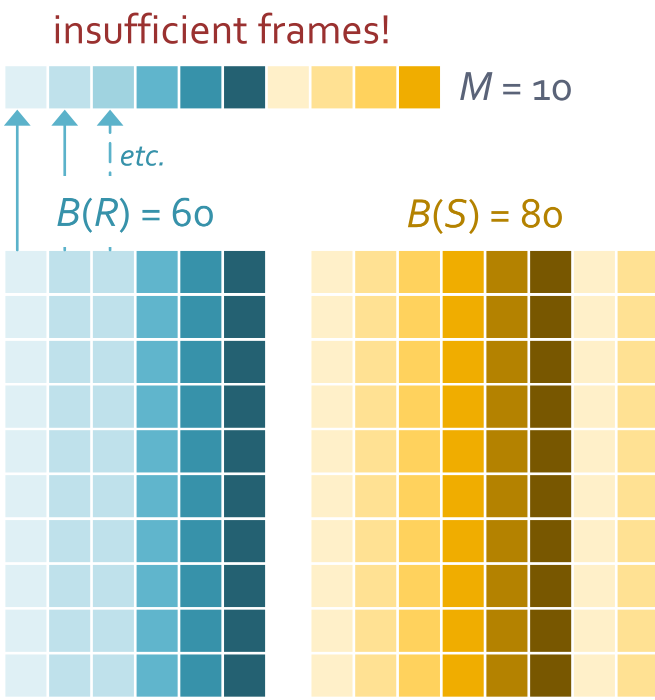
Hybrid Hash Join and Block Nested Loop Join

- If the smaller relation fits in main memory the costs are identical
  - The smaller relation is read once $B(R) + B(S)$
  - The larger relation is scanned once to join the records
- Otherwise hybrid hash join is more efficient
  - Block nested loop reads *R* once
    - But *S* once for each *clump* of *R*
  - Hybrid hash join reads one partition of *R* and *S* once
    - Reads the other partitions twice and writes them once
  - And the records of both *R* and *S* belonging to a particular partition are only read once, after the partitioning phase

Hash Join and Sort-Merge Join

Sort-join in 2 passes: $M > \sqrt{(B(R) + B(S))}$

Hash join in 2 passes: $M > \sqrt{(B(smaller))}$

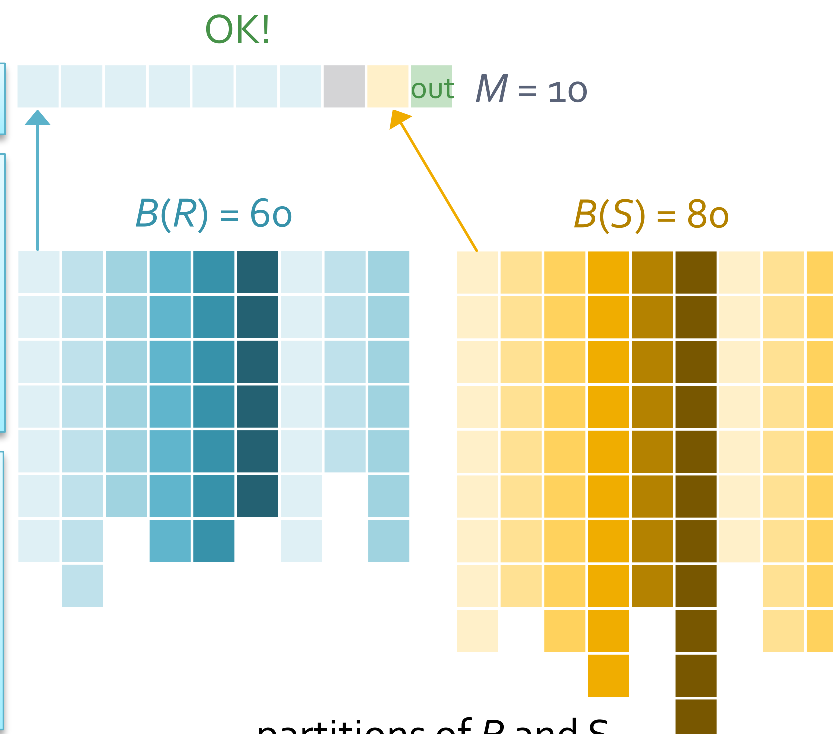


sorted runs of *R* and S after initial sort pass

But

sort-join not sensitive to data skew

sort-join results sorted on join attribute



partitions of *R* and S

Join Method Ordering

- Simple nested loop join (read *S* for each *record*)
  - Retains the original order of *R*
- Index nested loop join
  - Retains the original order of *R*
- Sort-Merge join
  - Ordered by the join attribute
- Zig-zag join
  - Ordered by the join attribute
- All other join methods
  - No order

But an awful algorithm …

Order might make an upstream operation is more efficient

Such as a join with a third table on the same join attribute

General Join Conditions

- The join process is more complex if the join condition is not simple equality on one attribute
- For equalities over several attributes
  - Sort-merge and hash join must sort (or hash) over all of the attributes in the selection
  - An index that matches one of the equalities may be used for the index nested loop join
- For inequalities ($\leq$, $\geq$, etc.)
  - Hash indexes cannot be used for index nested loop joins
  - Sort-merge and hash joins are not possible
  - Other join algorithms are unaffected

Set Operations

```
SELECT fName, lName
FROM Patient
INTERSECT
SELECT fName, lName
FROM Doctor
```

$\pi_{\texttt{fName,lName}}(\texttt{Patient}) \cap \pi_{\texttt{fName,lName}}(\texttt{Doctor})$

Note that set operations, unlike other operations remove duplicates by default

- Intersection $R \cap S$
  - A join where the condition is equality on all attributes

$\pi_{\texttt{fName,lName}}(\texttt{Patient}) \bowtie \pi_{\texttt{fName,lName}}(\texttt{Doctor})$

- Cartesian product $R \times S$
  - A special case of join where there is no join condition
  - All records are joined to each other

More Set Operations

- Union using sorting
  - Sort *R* and *S* using all fields
  - Scan and merge the results while removing duplicates
- Union using hashing
  - Partition *R* and *S* using a hash function *h*
  - For each partition of smaller relation (*S*)
    - Build an in-memory hash table (using h_2)
    - Scan the corresponding partition of *R*, and for each record probe the hash table if it is not in the table, add it
- Set difference
  - Similar to union except that for *R* – *S*, if records are not in the hash table for *S* add it to the result

The result is separate from the *S* hash table

Summary

Memory Requirements

One-Pass and Simple Algorithms

Operation	Algorithm	*M* Requirement	Disk I/O
σ, π	scan	1	B
δ, γ^*	scan	B	B
$\cup$, $\cap$, $-$, $\times$	scan	$\min(B(R), B(S))$	$B(R) + B(S)$
$\bowtie$	nested loop	$\min(B(R), B(S))$	$B(R) + B(S)$
$\bowtie$	nested loop	$M \geq 2$	$B(R) + B(R) * B(S)/M$

$^*\delta$ = duplicate removal, γ = grouping

cost is greater if *M* requirement is not met

Sort-Based Algorithms

Operation	*M* Requirement	Disk I/O	Notes
δ, γ	$\sqrt{B}$	$3B$	
$\cup$, $\cap$, $-$	$\sqrt{(B(R) + B(S))}$	$3(B(R) + B(S))$	
$\bowtie$	$\sqrt{(B(R) + B(S))}$	$3(B(R) + B(S))$	sort-merge join

cost is greater if *M* requirement is not met

Hash-Based Algorithms

Operation	M Requirement	Disk I/O	Notes
δ, γ	$\sqrt{B}$	$3B$	
$\cup$, $\cap$, $-$	$\sqrt{B(S)}$	$3(B(R) + B(S))$	$B(S)$ is smaller relation
$\bowtie$	$\sqrt{B(S)}$	$3(B(R) + B(S))$	
$\bowtie$	$> \sqrt{B(S)}$	$(3 - 2 * t / k)(B(R) + B(S))$ note	hybrid hash-join

Assume $B(S) \leq B(R)$, and $B(S) \geq M$

cost is greater if M requirement is not met

note – reduction dependent on relative sizes of M and R

Appendix

Selections with no Disjunctions

- Hash indexes can be used if there is an equality condition for *every* attribute in the search key
  - e.g. a single hash index on {*city*, *street*, *number*}
    - $\sigma_{\texttt{city="London"} \wedge \texttt{street="Baker"} \wedge \texttt{number=221}}$`(Detective)` can be used
    - $\sigma_{\texttt{city="Los Angeles"} \wedge \texttt{street="Cahuenga"}}$`(Detective)` cannot
- Tree indexes can be used if there is a selection on each of the first *n* attributes of the search key
  - e.g. B+ index on {*city*, *street*, *number*}
    - $\sigma_{\texttt{city="London"} \wedge \texttt{street="Baker"} \wedge \texttt{number=221}}$`(Detective)` can be used
    - $\sigma_{\texttt{city="Los Angeles"} \wedge \texttt{street="Cahuenga"}}$`(Detective)` can be used

Selections with no Disjunctions…

- If an index matches a *subset* of the conjuncts
  - Use the index to return a result that contains some unwanted records
  - Scan the result for matches to the other conjuncts
    - $\sigma_{\texttt{city="London"}\wedge\texttt{street="Baker"}\wedge\texttt{number=221}\wedge\texttt{fName="Sherlock"}}$ `(Detective)`
    - Use the address index and scan result for *Sherlock*s
- If more than one index matches a conjunct
  - Either use the most selective index, then scan the result, discarding records that fail to match to the other criteria
  - Or use all indexes and retrieve the *rids*
    - Then take the intersection of the *rids* and retrieve those records

Selections with no Disjunctions…

- Consider the relation and selection shown below
  - Detective = {*id*, *fName*, *lName*, *age*, *city*, *street*, *number*, *author*}
  - $\sigma_{\texttt{city="New York"} \wedge \texttt{author="Spillane"} \wedge \texttt{lName="Hammer"}}(\texttt{Detective})$
- With indexes
  - Secondary hash index, {*city*, *street*, *number*} cannot be used
  - Secondary B+ tree index, {*lName*, *fName*} can be used
  - Secondary hash index, {*author*} can be used
- There are two strategies:
  - Use the most selective of the two matching indexes, and search the results for the remaining criteria
  - Use both indexes, take the intersection of the *rid*

What if the B+ tree index is primary?

Selections with Disjunctions ...

- Consider the selections shown below
  - $\sigma_{(\texttt{author="King"} \vee \texttt{age>35}) \wedge (\texttt{lName="Tam"} \vee \texttt{id=11})}(\texttt{Detective})$
  - $\sigma_{(\texttt{author="King"}) \wedge (\texttt{lName="Tam"} \vee \texttt{id=11})}(\texttt{Detective})$
- Indexes on the relation
  - Secondary B+ tree index, {*lName*, *fName*}
  - Secondary hash index, {*author*}
- Compare the two selections
  - In the first selection each conjunct contains a disjunction without an index (*age*, *id*) so a file scan is required
  - In the second selection the index on author can be used, and records that don't meet the other criteria removed

Hybrid Hash Join and Block Nested Loop Join

- If the smaller relation fits in main memory the costs are identical
  - The smaller relation is read once

    $B(R) + B(S)$
  - The larger relation is scanned once to join the records
- Otherwise hybrid hash join is more efficient

  $B(R) + 5 * B(S) = 6n$
  - Block nested loop reads *R* once
    - But *S* once for each *clump* of *R*
  - Hybrid hash join reads one partition of
    - Reads the other partitions twice and
  - And the records of both *R* and *S* belonging to a particular partition are only read once, after the partitioning phase

$(B(R) + B(S)) / 5 +$
$(B(R) + B(S)) * 12 / 5$
$= 2n / 5 + 24n / 5$
$= 26n / 5$
$= 5.2n$