Optimization: Cost Estimation & Joins

R&G Chapter 15

(slides adapted from content by J.Gehrke, J.Shanmugasundaram, and/or C.Koch)

Project 2 Will Be Posted Tonight

Build …

A Static Hash Index

A ISAM Index

Extra Credit…

Dynamic Version of Any of the Above

Cost Estimation

- The IO cost for each relational operator can be estimated from the size of its input.
- Pipelined operators can be merged together with respect to their IO cost (i.e., selection, projection, union all).
- The Reduction Factor of an operation states what % of the data makes it through the operation.

Data Access IO Costs

	Full Scan	Range Scan	Lookup
Raw File	N	N	N

For a raw file, we need to scan the entire file to get any sort of results.
For a sorted file, we can do a range scan by finding the first entry with a binary sort (log steps), and then scanning until we hit the upper end of the range (# pages in the range)
For any kind of hash index, lookups are fast, but scans can be *more* expensive, since pages are typically not utilized fully.
For tree indices, we can do range scans and point lookups by touching fewer pages than the same on a sorted file, since each index page has |T| pointers.
Note that scans of B+ trees and extendible hashes are always slower than their counterparts, as these systems use pointers between pages, and one page of data must be fully loaded before we can identify the next page to load.

Note that since static and linear hashes, as well as ISAM trees can have overflow pages, the cost of doing a lookup/scan may require touching these pages as well. Hence the numbers provided are approximations.

Data Access IO Costs

	Full Scan	Range Scan	Lookup
Raw File	N	N	N
Sorted File			

For a raw file, we need to scan the entire file to get any sort of results.
For a sorted file, we can do a range scan by finding the first entry with a binary sort (log steps), and then scanning until we hit the upper end of the range (# pages in the range)
For any kind of hash index, lookups are fast, but scans can be *more* expensive, since pages are typically not utilized fully.
For tree indices, we can do range scans and point lookups by touching fewer pages than the same on a sorted file, since each index page has |T| pointers.
Note that scans of B+ trees and extendible hashes are always slower than their counterparts, as these systems use pointers between pages, and one page of data must be fully loaded before we can identify the next page to load.

Note that since static and linear hashes, as well as ISAM trees can have overflow pages, the cost of doing a lookup/scan may require touching these pages as well. Hence the numbers provided are approximations.

Data Access IO Costs

	Full Scan	Range Scan	Lookup
Raw File	N	N	N
Sorted File	N	$\log_2(N)+\|R\|$	$\log_2(N)$

For a raw file, we need to scan the entire file to get any sort of results.
For a sorted file, we can do a range scan by finding the first entry with a binary sort (log steps), and then scanning until we hit the upper end of the range (# pages in the range)
For any kind of hash index, lookups are fast, but scans can be *more* expensive, since pages are typically not utilized fully.
For tree indices, we can do range scans and point lookups by touching fewer pages than the same on a sorted file, since each index page has |T| pointers.
Note that scans of B+ trees and extendible hashes are always slower than their counterparts, as these systems use pointers between pages, and one page of data must be fully loaded before we can identify the next page to load.

Note that since static and linear hashes, as well as ISAM trees can have overflow pages, the cost of doing a lookup/scan may require touching these pages as well. Hence the numbers provided are approximations.

Data Access IO Costs

	Full Scan	Range Scan	Lookup
Raw File	N	N	N
Sorted File	N	$\log_2(N)+\|R\|$	$\log_2(N)$
Static Hash Index			

For a raw file, we need to scan the entire file to get any sort of results.
For a sorted file, we can do a range scan by finding the first entry with a binary sort (log steps), and then scanning until we hit the upper end of the range (# pages in the range)
For any kind of hash index, lookups are fast, but scans can be *more* expensive, since pages are typically not utilized fully.
For tree indices, we can do range scans and point lookups by touching fewer pages than the same on a sorted file, since each index page has |T| pointers.
Note that scans of B+ trees and extendible hashes are always slower than their counterparts, as these systems use pointers between pages, and one page of data must be fully loaded before we can identify the next page to load.

Note that since static and linear hashes, as well as ISAM trees can have overflow pages, the cost of doing a lookup/scan may require touching these pages as well. Hence the numbers provided are approximations.

Data Access IO Costs

	Full Scan	Range Scan	Lookup
Raw File	N	N	N
Sorted File	N	$\log_2(N)+\|R\|$	$\log_2(N)$
Static Hash Index	>N	>N	~1

For a raw file, we need to scan the entire file to get any sort of results.
For a sorted file, we can do a range scan by finding the first entry with a binary sort (log steps), and then scanning until we hit the upper end of the range (# pages in the range)
For any kind of hash index, lookups are fast, but scans can be *more* expensive, since pages are typically not utilized fully.
For tree indices, we can do range scans and point lookups by touching fewer pages than the same on a sorted file, since each index page has |T| pointers.
Note that scans of B+ trees and extendible hashes are always slower than their counterparts, as these systems use pointers between pages, and one page of data must be fully loaded before we can identify the next page to load.

Note that since static and linear hashes, as well as ISAM trees can have overflow pages, the cost of doing a lookup/scan may require touching these pages as well. Hence the numbers provided are approximations.

Data Access IO Costs

	Full Scan	Range Scan	Lookup
Raw File	N	N	N
Sorted File	N	$\log_2(N)+\|R\|$	$\log_2(N)$
Static Hash Index	>N	>N	~1
Extendible Hash Index			

For a raw file, we need to scan the entire file to get any sort of results.
For a sorted file, we can do a range scan by finding the first entry with a binary sort (log steps), and then scanning until we hit the upper end of the range (# pages in the range)
For any kind of hash index, lookups are fast, but scans can be *more* expensive, since pages are typically not utilized fully.
For tree indices, we can do range scans and point lookups by touching fewer pages than the same on a sorted file, since each index page has |T| pointers.
Note that scans of B+ trees and extendible hashes are always slower than their counterparts, as these systems use pointers between pages, and one page of data must be fully loaded before we can identify the next page to load.

Note that since static and linear hashes, as well as ISAM trees can have overflow pages, the cost of doing a lookup/scan may require touching these pages as well. Hence the numbers provided are approximations.

Data Access IO Costs

	Full Scan	Range Scan	Lookup
Raw File	N	N	N
Sorted File	N	$\log_2(N)+\|R\|$	$\log_2(N)$
Static Hash Index	>N	>N	~1
Extendible Hash Index	>N+|D| (random)	>N+|D| (random)	2

For a raw file, we need to scan the entire file to get any sort of results.
For a sorted file, we can do a range scan by finding the first entry with a binary sort (log steps), and then scanning until we hit the upper end of the range (# pages in the range)
For any kind of hash index, lookups are fast, but scans can be *more* expensive, since pages are typically not utilized fully.
For tree indices, we can do range scans and point lookups by touching fewer pages than the same on a sorted file, since each index page has |T| pointers.
Note that scans of B+ trees and extendible hashes are always slower than their counterparts, as these systems use pointers between pages, and one page of data must be fully loaded before we can identify the next page to load.

Note that since static and linear hashes, as well as ISAM trees can have overflow pages, the cost of doing a lookup/scan may require touching these pages as well. Hence the numbers provided are approximations.

Data Access IO Costs

	Full Scan	Range Scan	Lookup
Raw File	N	N	N
Sorted File	N	$\log_2(N)+\|R\|$	$\log_2(N)$
Static Hash Index	>N	>N	~1
Extendible Hash Index	>N+|D| (random)	>N+|D| (random)	2
Linear Hash Index			

For a raw file, we need to scan the entire file to get any sort of results.
For a sorted file, we can do a range scan by finding the first entry with a binary sort (log steps), and then scanning until we hit the upper end of the range (# pages in the range)
For any kind of hash index, lookups are fast, but scans can be *more* expensive, since pages are typically not utilized fully.
For tree indices, we can do range scans and point lookups by touching fewer pages than the same on a sorted file, since each index page has |T| pointers.
Note that scans of B+ trees and extendible hashes are always slower than their counterparts, as these systems use pointers between pages, and one page of data must be fully loaded before we can identify the next page to load.

Note that since static and linear hashes, as well as ISAM trees can have overflow pages, the cost of doing a lookup/scan may require touching these pages as well. Hence the numbers provided are approximations.

Data Access IO Costs

	Full Scan	Range Scan	Lookup
Raw File	N	N	N
Sorted File	N	$\log_2(N)+\|R\|$	$\log_2(N)$
Static Hash Index	>N	>N	~1
Extendible Hash Index	>N+|D| (random)	>N+|D| (random)	2
Linear Hash Index	>N	>N	~1

For a raw file, we need to scan the entire file to get any sort of results.
For a sorted file, we can do a range scan by finding the first entry with a binary sort (log steps), and then scanning until we hit the upper end of the range (# pages in the range)
For any kind of hash index, lookups are fast, but scans can be *more* expensive, since pages are typically not utilized fully.
For tree indices, we can do range scans and point lookups by touching fewer pages than the same on a sorted file, since each index page has |T| pointers.
Note that scans of B+ trees and extendible hashes are always slower than their counterparts, as these systems use pointers between pages, and one page of data must be fully loaded before we can identify the next page to load.

Note that since static and linear hashes, as well as ISAM trees can have overflow pages, the cost of doing a lookup/scan may require touching these pages as well. Hence the numbers provided are approximations.

Data Access IO Costs

	Full Scan	Range Scan	Lookup
Raw File	N	N	N
Sorted File	N	$\log_2(N)+\|R\|$	$\log_2(N)$
Static Hash Index	>N	>N	~1
Extendible Hash Index	>N+|D| (random)	>N+|D| (random)	2
Linear Hash Index	>N	>N	~1
ISAM Tree Index			

For a raw file, we need to scan the entire file to get any sort of results.
For a sorted file, we can do a range scan by finding the first entry with a binary sort (log steps), and then scanning until we hit the upper end of the range (# pages in the range)
For any kind of hash index, lookups are fast, but scans can be *more* expensive, since pages are typically not utilized fully.
For tree indices, we can do range scans and point lookups by touching fewer pages than the same on a sorted file, since each index page has |T| pointers.
Note that scans of B+ trees and extendible hashes are always slower than their counterparts, as these systems use pointers between pages, and one page of data must be fully loaded before we can identify the next page to load.

Note that since static and linear hashes, as well as ISAM trees can have overflow pages, the cost of doing a lookup/scan may require touching these pages as well. Hence the numbers provided are approximations.

Data Access IO Costs

	Full Scan	Range Scan	Lookup
Raw File	N	N	N
Sorted File	N	$\log_2(N)+\|R\|$	$\log_2(N)$
Static Hash Index	>N	>N	~1
Extendible Hash Index	>N+|D| (random)	>N+|D| (random)	2
Linear Hash Index	>N	>N	~1
ISAM Tree Index	~N	$\sim\log_{\|T\|}(N)+\|R\|$	$\sim\log_{\|T\|}(N)$

For a raw file, we need to scan the entire file to get any sort of results.
For a sorted file, we can do a range scan by finding the first entry with a binary sort (log steps), and then scanning until we hit the upper end of the range (# pages in the range)
For any kind of hash index, lookups are fast, but scans can be *more* expensive, since pages are typically not utilized fully.
For tree indices, we can do range scans and point lookups by touching fewer pages than the same on a sorted file, since each index page has |T| pointers.
Note that scans of B+ trees and extendible hashes are always slower than their counterparts, as these systems use pointers between pages, and one page of data must be fully loaded before we can identify the next page to load.

Note that since static and linear hashes, as well as ISAM trees can have overflow pages, the cost of doing a lookup/scan may require touching these pages as well. Hence the numbers provided are approximations.

Data Access IO Costs

	Full Scan	Range Scan	Lookup
Raw File	N	N	N
Sorted File	N	$\log_2(N)+\|R\|$	$\log_2(N)$
Static Hash Index	>N	>N	~1
Extendible Hash Index	>N+|D| (random)	>N+|D| (random)	2
Linear Hash Index	>N	>N	~1
ISAM Tree Index	~N	$\sim\log_{\|T\|}(N)+\|R\|$	$\sim\log_{\|T\|}(N)$
B+ Tree Index			

For a raw file, we need to scan the entire file to get any sort of results.
For a sorted file, we can do a range scan by finding the first entry with a binary sort (log steps), and then scanning until we hit the upper end of the range (# pages in the range)
For any kind of hash index, lookups are fast, but scans can be *more* expensive, since pages are typically not utilized fully.
For tree indices, we can do range scans and point lookups by touching fewer pages than the same on a sorted file, since each index page has |T| pointers.
Note that scans of B+ trees and extendible hashes are always slower than their counterparts, as these systems use pointers between pages, and one page of data must be fully loaded before we can identify the next page to load.

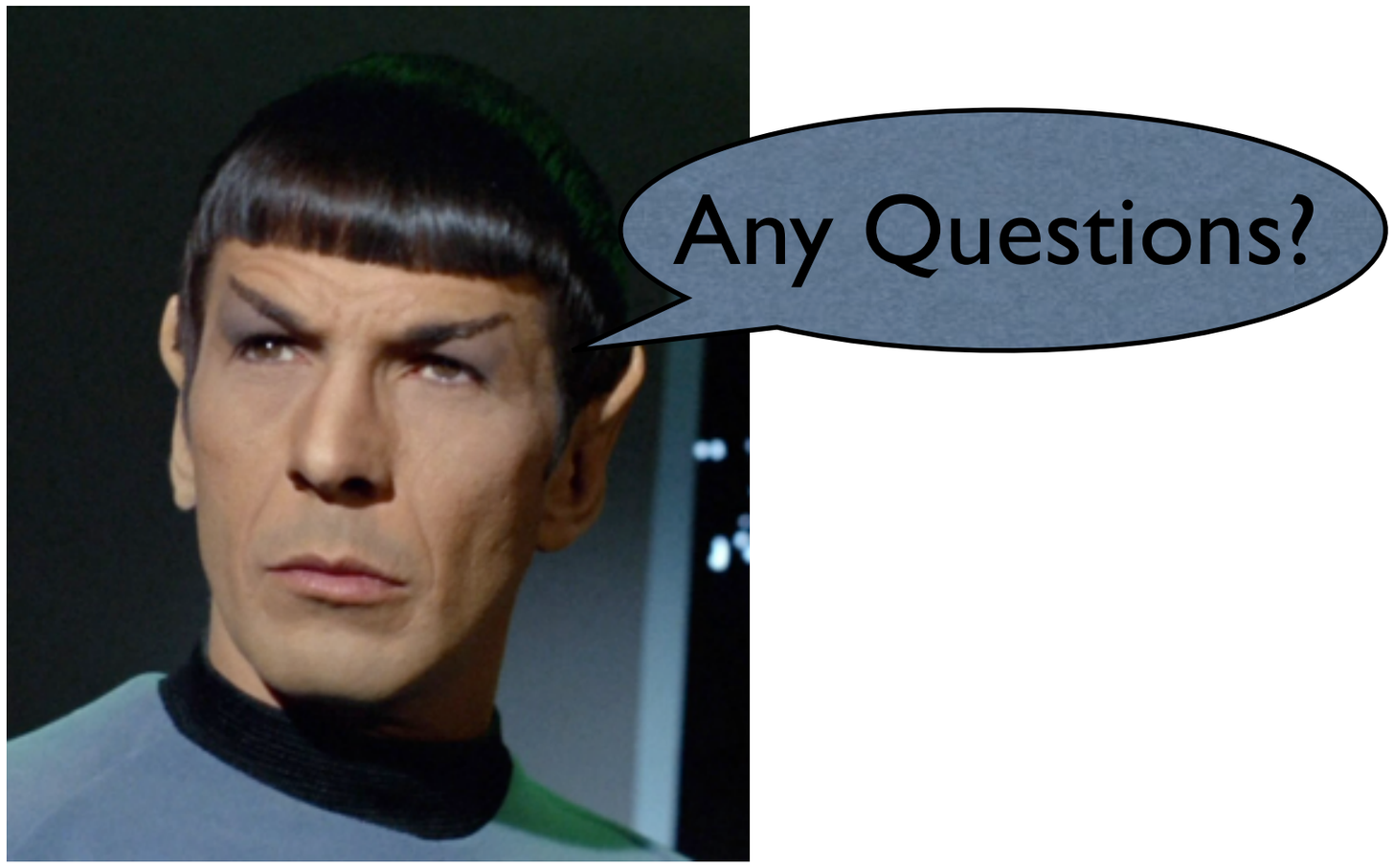
Note that since static and linear hashes, as well as ISAM trees can have overflow pages, the cost of doing a lookup/scan may require touching these pages as well. Hence the numbers provided are approximations.

Data Access IO Costs

	Full Scan	Range Scan	Lookup
Raw File	N	N	N
Sorted File	N	$\log_2(N)+\|R\|$	$\log_2(N)$
Static Hash Index	>N	>N	~1
Extendible Hash Index	>N+|D| (random)	>N+|D| (random)	2
Linear Hash Index	>N	>N	~1
ISAM Tree Index	~N	$\sim\log_{\|T\|}(N)+\|R\|$	$\sim\log_{\|T\|}(N)$
B+ Tree Index	N (random)	$\log_{\|T\|}(N)+\|R\|$ (random)	$\log_{\|T\|}(N)$

For a raw file, we need to scan the entire file to get any sort of results.
For a sorted file, we can do a range scan by finding the first entry with a binary sort (log steps), and then scanning until we hit the upper end of the range (# pages in the range)
For any kind of hash index, lookups are fast, but scans can be *more* expensive, since pages are typically not utilized fully.
For tree indices, we can do range scans and point lookups by touching fewer pages than the same on a sorted file, since each index page has |T| pointers.
Note that scans of B+ trees and extendible hashes are always slower than their counterparts, as these systems use pointers between pages, and one page of data must be fully loaded before we can identify the next page to load.

Note that since static and linear hashes, as well as ISAM trees can have overflow pages, the cost of doing a lookup/scan may require touching these pages as well. Hence the numbers provided are approximations.



Cost Estimation

- Thus far, we've considered only operators that have a reduction factor, or that have IO costs.
- A join operator has both!
  - What is the size of a join output?
  - What is the IO cost of a join?

Join Size

How do we estimate the size of a join output?

A join is a cross product followed by a selection. Although the join produces an increased output size, we consider its output relative to the worst case scenario (where the join is effectively a cross product). The reduction factor of the join is the reduction factor of the join predicate.

Join Size

How do we estimate the size of a join output?

What is a join?

A join is a cross product followed by a selection. Although the join produces an increased output size, we consider its output relative to the worst case scenario (where the join is effectively a cross product). The reduction factor of the join is the reduction factor of the join predicate.

Join Size

How do we estimate the size of a join output?

What is a join?

What is the output size of a cross product?

A join is a cross product followed by a selection. Although the join produces an increased output size, we consider its output relative to the worst case scenario (where the join is effectively a cross product). The reduction factor of the join is the reduction factor of the join predicate.

Join Size

How do we estimate the size of a join output?

What is a join?

What is the output size of a cross product?

What is the output size of a join?

A join is a cross product followed by a selection. Although the join produces an increased output size, we consider its output relative to the worst case scenario (where the join is effectively a cross product). The reduction factor of the join is the reduction factor of the join predicate.

Join IO Cost

What is the IO cost of a join?

It depends!
(on what?)

What Join algorithm is being used?
What properties of the data can we assume?

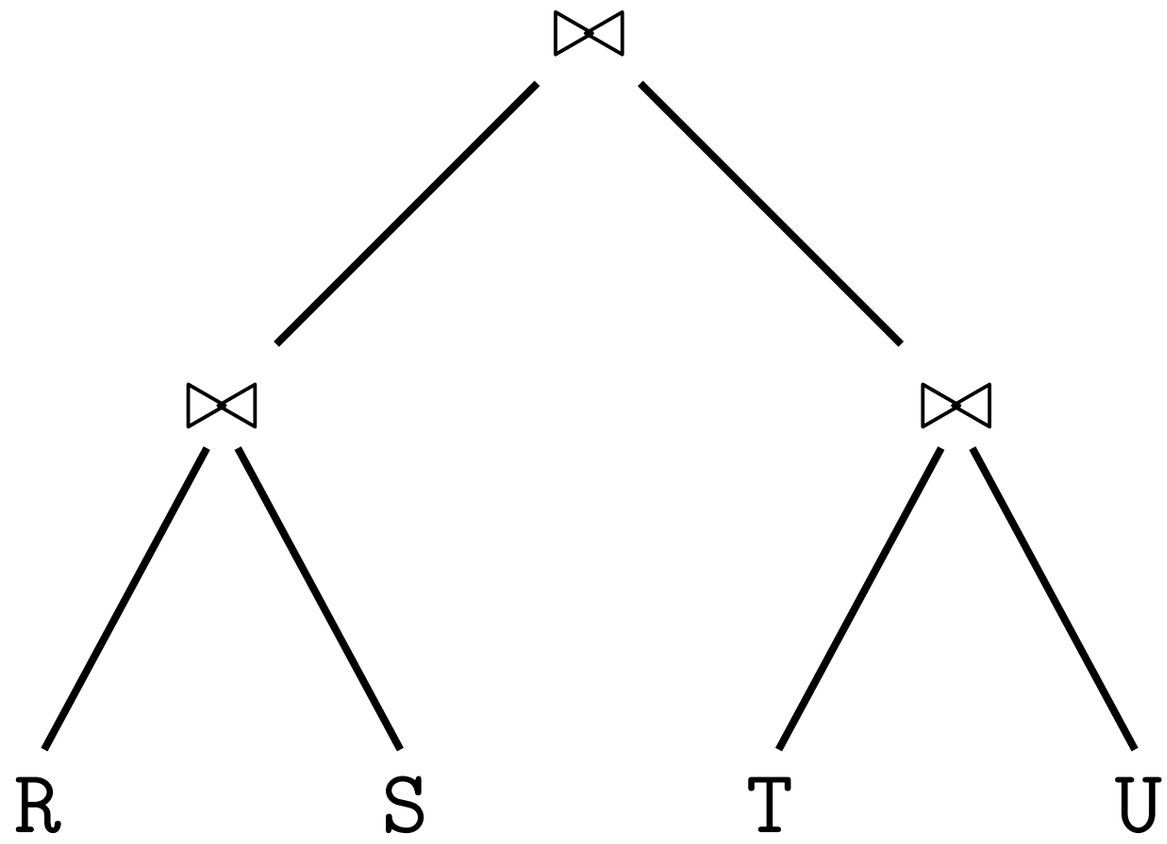
Time for a (slight, but related) detour!

Estimating Join Costs

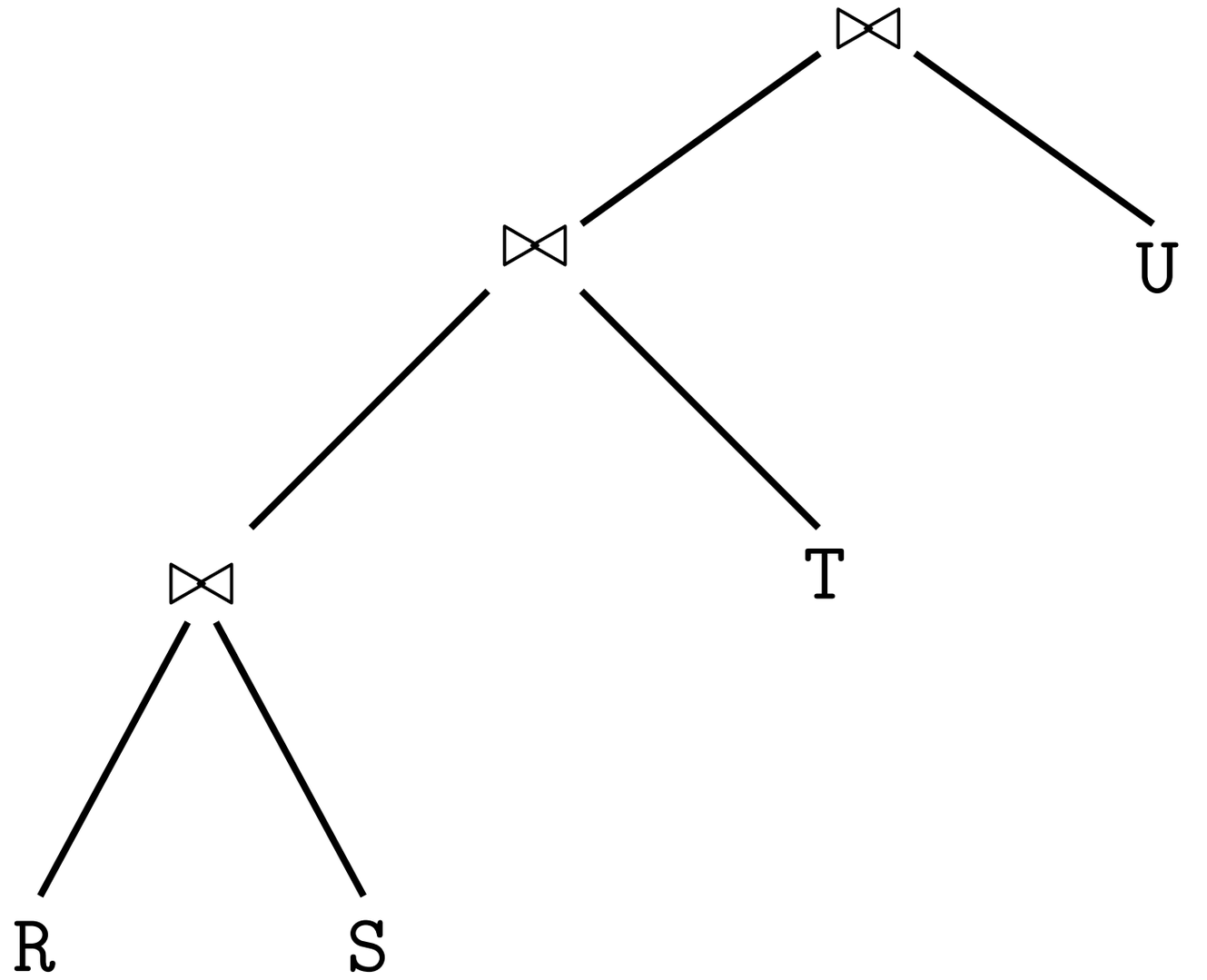
How many query plans are there?

$$R \bowtie S \bowtie T \bowtie U$$

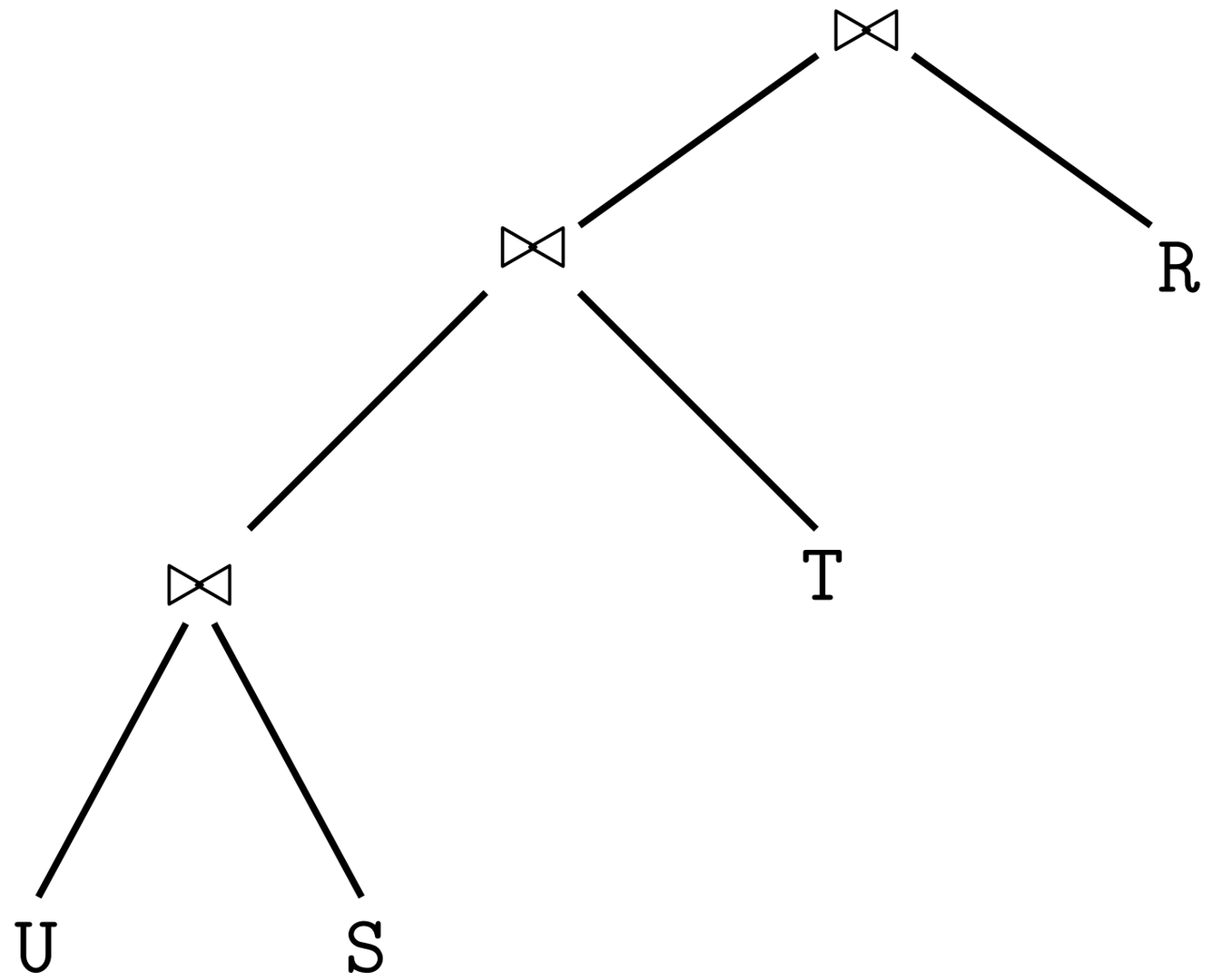
Estimating Join Costs



Estimating Join Costs



Estimating Join Costs

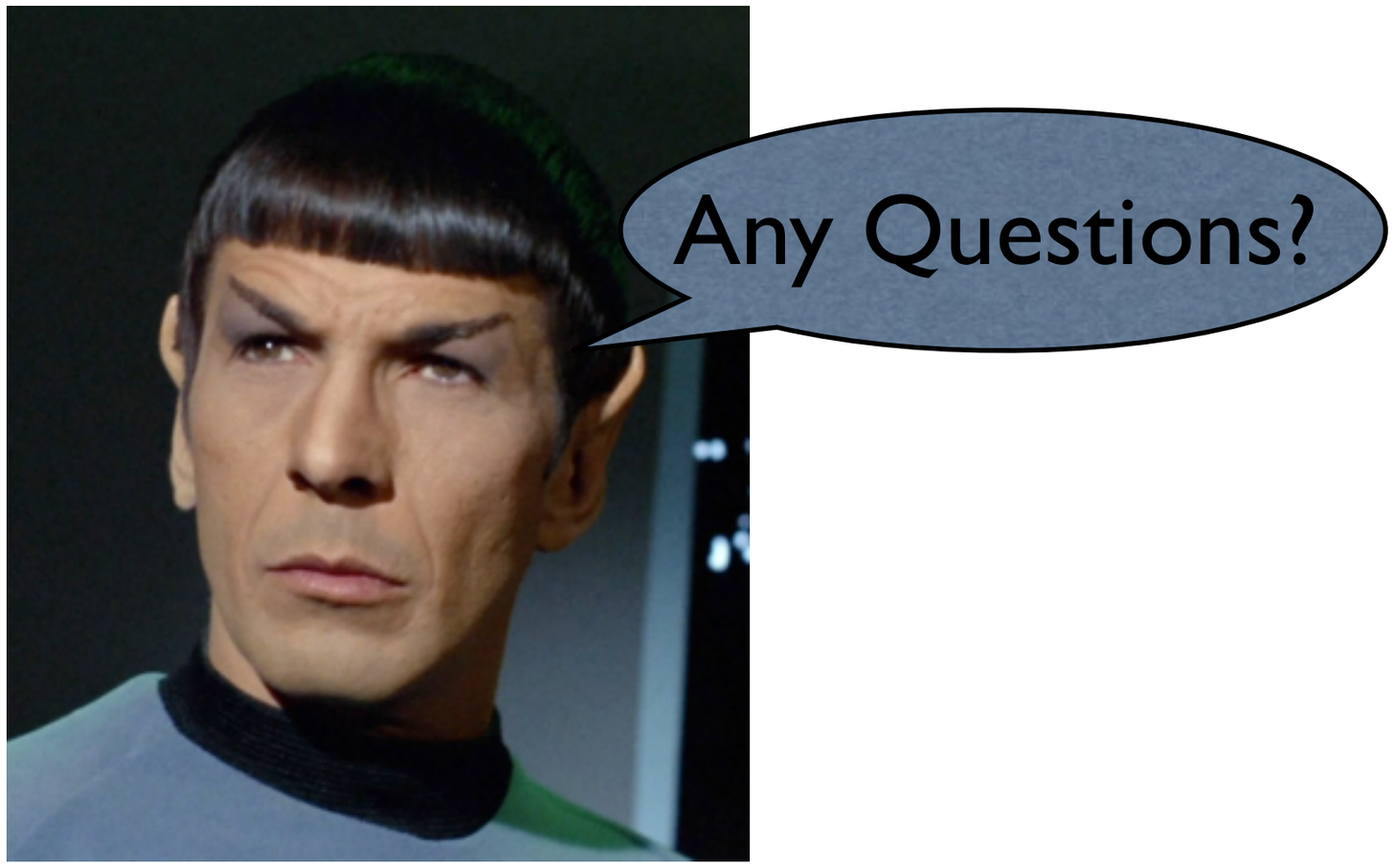


N! ways to join -- Expensive

Estimating Join Costs

There are (N-1)! (factorial) different ways (plans) to evaluate this join.

Computing costs for all of these plans is expensive!



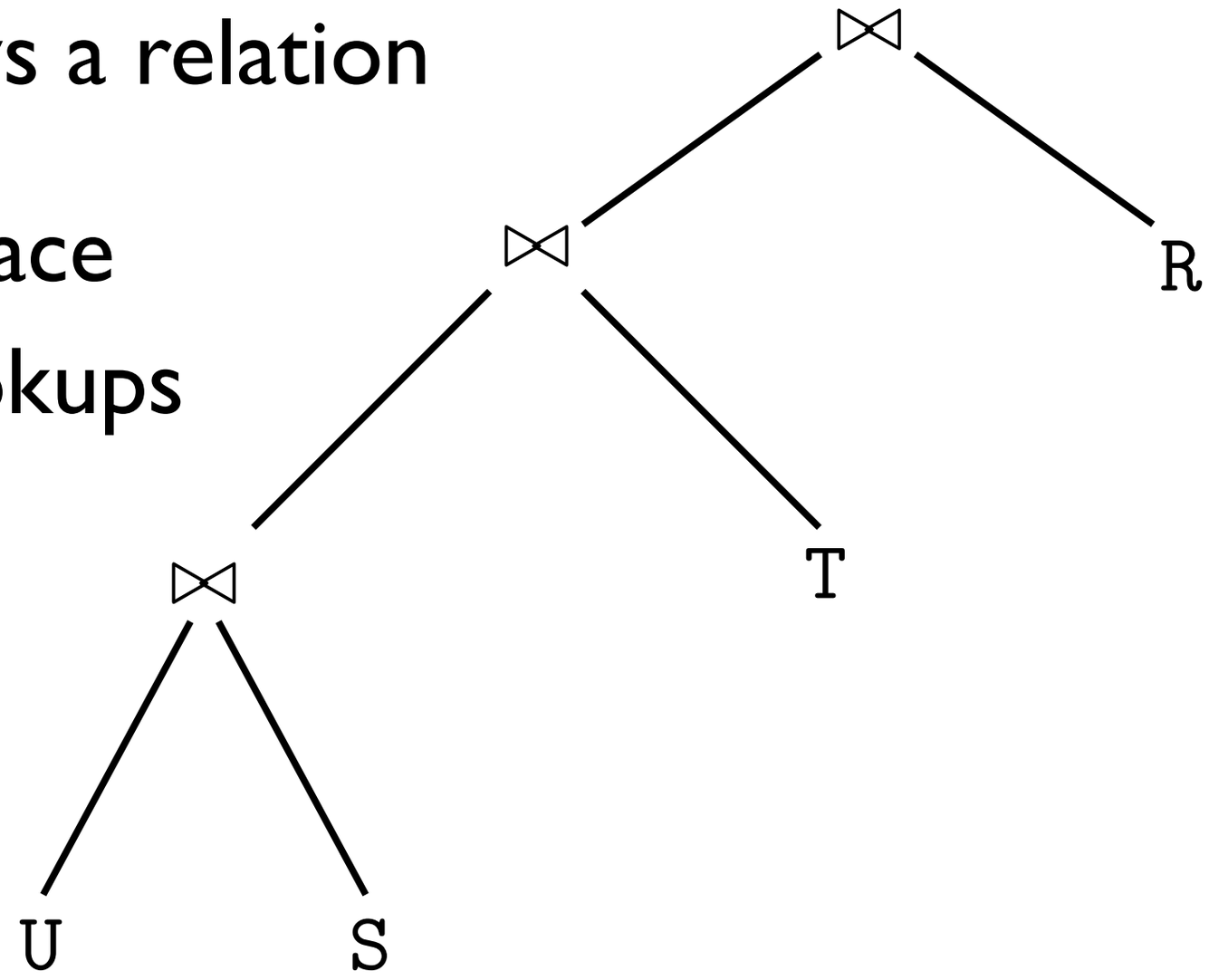
Left-Deep Plans

RHS Join Input is always a relation

1) Shrinks join search space

2) Allows index scans/lookups

Technique Pioneered by
the System R Optimizer



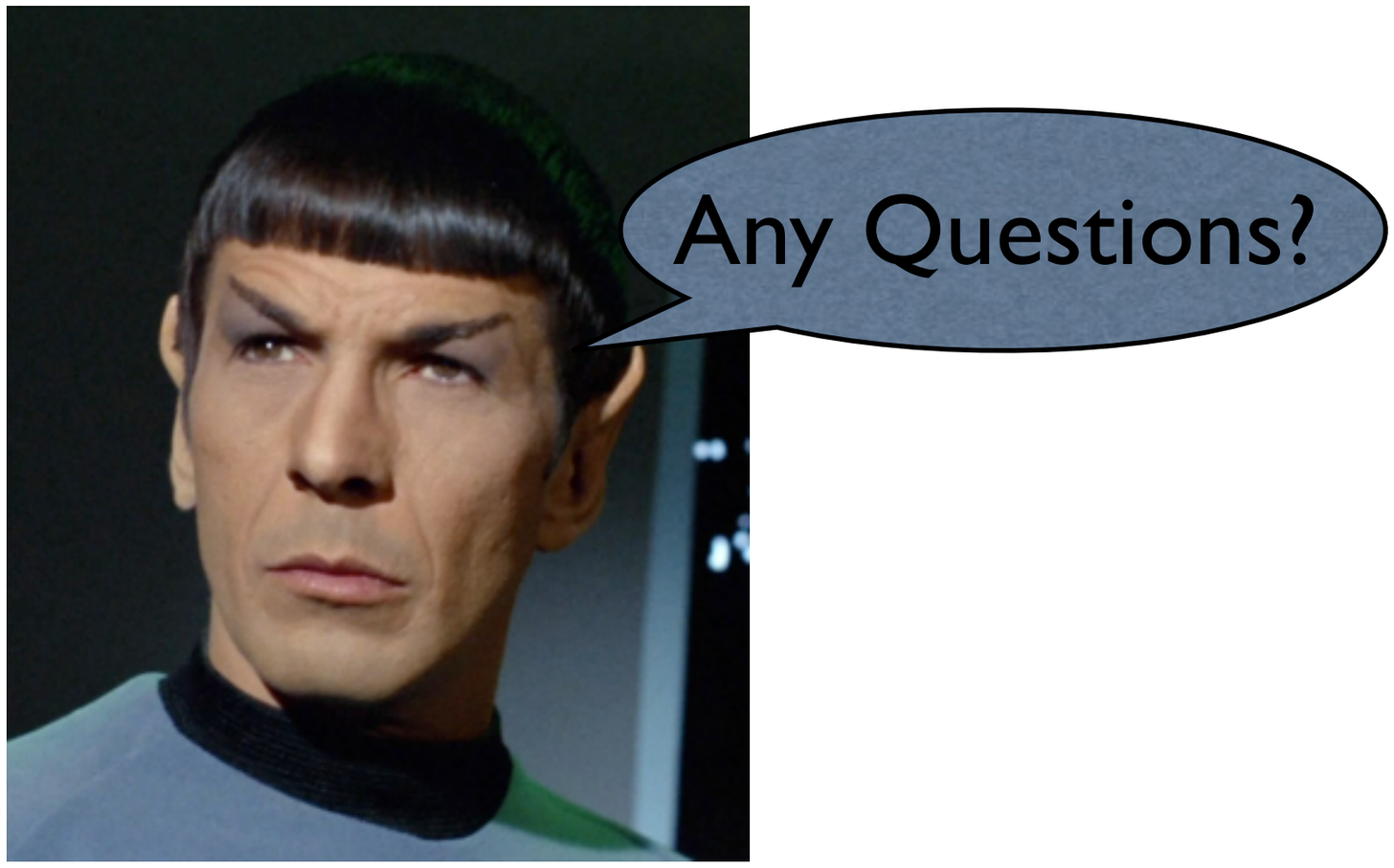
Left Deep Plan

Can Be Pipelined

(costs merged with prior step)

Direct From Disk

(can exploit indexes for INL or hybrid hash join)



Join Algorithm Comparison

	Can Support Pipelining?	But?
Hybrid Hash		
Index Nested Loop		
Sort/Merge Join		
(Block) Nested Loop		
Hash Join		

Join Algorithm Comparison

	Can Support Pipelining?	But?
Hybrid Hash	**Yes**	
Index Nested Loop		
Sort/Merge Join		
(Block) Nested Loop		
Hash Join		

Join Algorithm Comparison

	Can Support Pipelining?	But?
Hybrid Hash	**Yes**	RHS Hash Table needs to fit in memory
Index Nested Loop		
Sort/Merge Join		
(Block) Nested Loop		
Hash Join		

Join Algorithm Comparison

	Can Support Pipelining?	But?
Hybrid Hash	**Yes**	RHS Hash Table needs to fit in memory
Index Nested Loop	**Yes**	
Sort/Merge Join		
(Block) Nested Loop		
Hash Join		

Join Algorithm Comparison

	Can Support Pipelining?	But?
Hybrid Hash	**Yes**	RHS Hash Table needs to fit in memory
Index Nested Loop	**Yes**	RHS Table needs an index on the join key
Sort/Merge Join		
(Block) Nested Loop		
Hash Join		

Join Algorithm Comparison

	Can Support Pipelining?	But?
Hybrid Hash	**Yes**	RHS Hash Table needs to fit in memory
Index Nested Loop	**Yes**	RHS Table needs an index on the join key
Sort/Merge Join	**Yes**	
(Block) Nested Loop		
Hash Join		

Join Algorithm Comparison

	Can Support Pipelining?	But?
Hybrid Hash	**Yes**	RHS Hash Table needs to fit in memory
Index Nested Loop	**Yes**	RHS Table needs an index on the join key
Sort/Merge Join	**Yes**	LHS and RHS must both be sorted on the join key
(Block) Nested Loop		
Hash Join		

Join Algorithm Comparison

	Can Support Pipelining?	But?
Hybrid Hash	**Yes**	RHS Hash Table needs to fit in memory
Index Nested Loop	**Yes**	RHS Table needs an index on the join key
Sort/Merge Join	**Yes**	LHS and RHS must both be sorted on the join key
(Block) Nested Loop	**Yes**	
Hash Join		

Join Algorithm Comparison

	Can Support Pipelining?	But?
Hybrid Hash	**Yes**	RHS Hash Table needs to fit in memory
Index Nested Loop	**Yes**	RHS Table needs an index on the join key
Sort/Merge Join	**Yes**	LHS and RHS must both be sorted on the join key
(Block) Nested Loop	**Yes**	RHS Table needs to fit in memory
Hash Join		

Join Algorithm Comparison

	Can Support Pipelining?	But?
Hybrid Hash	**Yes**	RHS Hash Table needs to fit in memory
Index Nested Loop	**Yes**	RHS Table needs an index on the join key
Sort/Merge Join	**Yes**	LHS and RHS must both be sorted on the join key
(Block) Nested Loop	**Yes**	RHS Table needs to fit in memory
Hash Join	No	

Join Algorithm Comparison

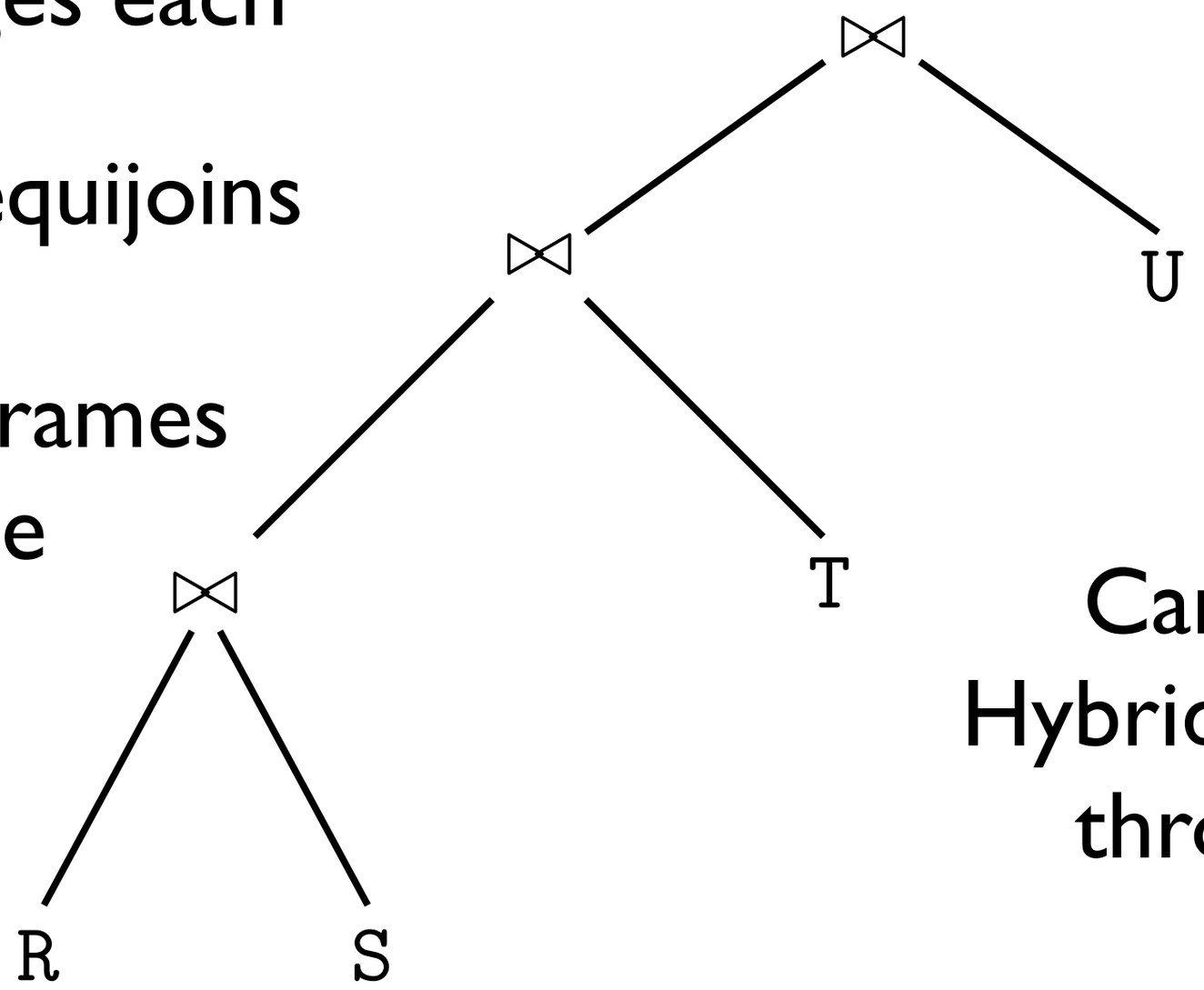
	Can Support Pipelining?	But?
Hybrid Hash	**Yes**	RHS Hash Table needs to fit in memory
Index Nested Loop	**Yes**	RHS Table needs an index on the join key
Sort/Merge Join	**Yes**	LHS and RHS must both be sorted on the join key
(Block) Nested Loop	**Yes**	RHS Table needs to fit in memory
Hash Join	No	No buts. Hash Join always materializes

Pipelining Limitations

R: 1000 pages
S, T, U: 40 pages each

All joins are equijoins

100 buffer frames available



Can we use Hybrid Hash joins throughout?

No, we can't use hybrid hash joins, as that would require 120 pages of free space in memory. If we have an index on S, T, or U, we can do an index-nested-loop join on those relations.

Option 2 would be to materialize the output of RxS or RxSxT. How do we pick between these two options? Based on cost. If RxS has a low reduction factor, we can use that. if xT has a reduction factor low enough to counteract the additional cross product, we can (and should) use that.

Join Algorithm IO Costs

$R \bowtie S$	IO Cost
Hybrid Hash	
Index Nested Loop	
Sort/Merge Join	
Nested Loop	
Block Nested Loop	
Hash Join	

Join Algorithm IO Costs

$R \bowtie S$	IO Cost
Hybrid Hash	[#pages of S] (if fits in mem)
Index Nested Loop	
Sort/Merge Join	
Nested Loop	
Block Nested Loop	
Hash Join	

Join Algorithm IO Costs

$R \bowtie S$	IO Cost
Hybrid Hash	[#pages of S] (if fits in mem)
Index Nested Loop	|R| * [cost of one scan/lookup on S]
Sort/Merge Join	
Nested Loop	
Block Nested Loop	
Hash Join	

Join Algorithm IO Costs

$R \bowtie S$	IO Cost
Hybrid Hash	[#pages of S] (if fits in mem)
Index Nested Loop	|R| * [cost of one scan/lookup on S]
Sort/Merge Join	[#pages of S] (+sorting costs)
Nested Loop	
Block Nested Loop	
Hash Join	

Join Algorithm IO Costs

$R \bowtie S$	IO Cost
Hybrid Hash	[#pages of S] (if fits in mem)
Index Nested Loop	|R| * [cost of one scan/lookup on S]
Sort/Merge Join	[#pages of S] (+sorting costs)
Nested Loop	[#pages of S] (if fits in mem)
Block Nested Loop	
Hash Join	

Join Algorithm IO Costs

$R \bowtie S$	IO Cost
Hybrid Hash	[#pages of S] (if fits in mem)
Index Nested Loop	|R| * [cost of one scan/lookup on S]
Sort/Merge Join	[#pages of S] (+sorting costs)
Nested Loop	[#pages of S] (if fits in mem)
Block Nested Loop	[#pages of R] + [#of block pairs] * ([#pages per block of R]+[#pages per block of S])
Hash Join	

Join Algorithm IO Costs

$R \bowtie S$	IO Cost
Hybrid Hash	[#pages of S] (if fits in mem)
Index Nested Loop	|R| * [cost of one scan/lookup on S]
Sort/Merge Join	[#pages of S] (+sorting costs)
Nested Loop	[#pages of S] (if fits in mem)
Block Nested Loop	[#pages of R] + [#of block pairs] * ([#pages per block of R]+[#pages per block of S])
Hash Join	2*([#pages of R]+[#pages of S]) + [#pages of S]

Summary

- 2 dimensions to search along for plans (or more)
  - What is the best access path? (π/σ equivs)
  - What is the best join order? (⋈ equivs)
- Consider the **cost** of each allowable plan.
- Understanding how each operator's output size relates to its input size makes it possible to accurately estimate the cost of a plan.
- Simplify the join order problem by exploring the cost of left-deep plans only.