Query Processing Architecture and Algorithms

Semih Salihoğlu

Nov 2\textsuperscript{nd} 2021
Announcements

➢ Assignment 4:
   ➢ Out Nov 5\textsuperscript{th} night.
   ➢ Due Nov 19\textsuperscript{th} midnight
   ➢ For the programming question, you need to wait for next week’s lecture on Tuesday but can get started in written part for sure (or even the programming part)
Outline For Today

1. DBMS Query Processing Architecture

2. Fundamental Query Processing Operators & Algorithms
   - Assumptions
   - Scan-based Operators
   - Sort-based Operators
   - Hashing-based Operators
   - Algorithms Using Indices
Recall: Overview of Compilation Steps

Text
SELECT ...
FROM ...
WHERE...

AST
Parser → Root → Binder → Normalizer
SELECT ... TABLES ... CON.
Customer Product =
cid 5

Logical Plan
Join C.cid=O.cid
Scan Tbl Order
100 < price
Scan Tbl Customer cid=3
Project oid

Physical Plan Translator
Physical Plan
Project $5
HashJoin L.$2=R.$3
Filter $2=5
IndexScan cust.mkd

Optimizer

Logical Plan Translator

Query Executor
RESULT
cid
...

SELECT ... FROM ... WHERE...
Query Processor of DBMSs

- The component that executes a physical plan:
  - A tree of operators that manipulate files and tuples to produce the output asked in a query.
  - Operators implement core:
    1. data access methods
    2. query processing algorithms

```sql
SELECT cid
FROM Customer C, Order O, Product P
WHERE C.cid = O.cid AND O.pid = P.pid
  AND P.name = BookA
```

Note: the more operators a system has, the larger set of query plans (i.e., algorithms) it can stitch together to evaluate queries
(Simplified) Physical Plan Architecture

- Tuples flow from leaves to root
- Operators produce tables
  - not necessarily in full; can be in pieces
- Several designs exist. Most popular choices
  - push vs pull (will not simulate)
  - materialized vs pipelined
  - iterator model (Volcano) vs block-at-a-time
Materialized vs Pipelined (1)

- Materialized: All ops are “blocking”, i.e. materialize all their inputs to disk or temp. memory buffers
- Simple to implement
- Earlier DBMSs adopted materialization but now obsolete

Diagram:
- Disk
- FileScan Customer
- Filter name=Alice
- Project cid
- ...
Pipelined: When possible, ops take 1 or more tuples-at-a-time, process, and pass to parent ops

More efficient (avoids temp file writing, reading)

Not always possible: e.g., ORDER BY

- to sort a table, cannot pipeline tuples. need to see all tuples before computing the order.
Iterator Model vs Block-at-a-time

- **Iterator Model** (by Goetz Graefe, 1990)
  - Pull data from children “one tuple a a time”
  - E.g.: PostgreSQL
- **Block-at-a-time**:
  - Pull data many, e.g., 1024, tuples at a time
  - Better CPU utilization b/c fewer function calls

Goetz Graefe

Main architect of first version of MS SQL Server
Outline For Today

1. DBMS Query Processing Architecture

2. Fundamental Query Processing Operators & Algorithms
   - Assumptions
   - Scan-based Operators
   - Sort-based Operators
   - Hashing-based Operators
   - Algorithms Using Indices
Costs of Query Processing Algorithms

- In algorithm analysis often “runtime”, i.e., # CPU cycles is the metric
- In DBMSs, most algs (but not all) are linear time or almost-linear time (i.e., with $O(\log(|R|))$ factors) in terms of runtime.
- Will use \textit{I/O cost} to analyze the main algorithms because DBMSs are disk-based systems.
- Disclaimer 1: Simplification to study the general behavior of algs
- Disclaimer 2: All of the algs we describe are integrated in many systems and have scenarios when one is used over the other
Given operator $o$ processing 1 or 2 tables (e.g., scan or join)

Recall: $o$ runs in memory

Select * from Customer, Order where Customer.cid = Order.cid;

Number of rows for a table $|Customer|$

Number of disk blocks for a table

$$B(Customer) = \frac{|Customer|}{\text{# of rows per block}}$$
Notation

- Relations: $R, S$
- Tuples: $r, s$
- Number of tuples: $|R|, |S|$
- Number of disk blocks: $B(R), B(S)$
- Assume row-oriented physical design (i.e., all column values of tuples are in the page/block)
- Number of memory blocks available: $M$
- Cost metric: # I/O’s
  - And sometimes # memory blocks required
Outline For Today

1. DBMS Query Processing Architecture

2. Fundamental Query Processing Operators & Algorithms
   - Assumptions
   - Scan-based Operators
   - Sort-based Operators
   - Hashing-based Operators
   - Algorithms Using Indices
Table Scan Operators

- Scan table $R$ and optionally perform a:
  - Selection over $R$
  - Projection of $R$ without duplicate elimination

- I/O's: $B(R)$
  - Optimization for selection:
    - Stop early if it is a lookup by key

- Memory requirement: 2 (blocks)
  - 1 for input, 1 for buffer output
  - Increasing memory does not improve I/O

- Not counting I/O cost (if any), of writing the result out
  - Same for any algorithm!
Nested-loop Join Operator

- Takes 2 tables as inputs and implements: $R \bowtie_p S$
- Basic/Naive version:
  for each block of $R$, and for each $r$ in the block:
    for each block of $S$, and for each $s$ in the block:
      output $rs$ if $p$ evaluates to true over $r$ and $s$
- $R$ is called the outer table; $S$ is called the inner table
- I/O’s: $B(R) + |R| \cdot B(S)$
  - Blocks of $R$ are moved into memory only once
  - Blocks of $S$ are moved into memory with $|R|$ number of times
- Memory requirement: 3
- This is a terribly slow algorithm: has quadratic runtime
- But when is it necessary?
  - When doing Cartesian product-like operations, e.g., “difficult” join conditions
  - SELECT * FROM R, P WHERE sqrt(R.A * S.B) > 5

Note: No other operation except table scan
Simulation of Basic Nested-loop Join

- 1 block = 2 tuples, 3 blocks of memory

Number of I/O:
\[ B(R) + |R| \times S(R) = 2 \text{ blocks} + 4 \times 3 \text{ blocks} = 14 \]
Block-Nested-loop join Operator

- Improvement: block-based nested-loop join

  for each block of $R$, for each block of $S$:
  
  for each $r$ in the $R$ block, for each $s$ in the $S$ block:
  
  ... 

- I/O's: $B(R) + B(R) \cdot B(S)$

- Memory requirement: same as before
Simulation of Block Nested-loop Join

- 1 block = 2 tuples, 3 blocks of memory

Number of I/O:

\[ B(R) + B(R) \times B(S) = 2 \text{ blocks} + 2 \times 3 \text{ blocks} = 8 \]
Improvement to Block Nested Loop Join

- Make use of available memory
  - Read into memory as many blocks of $R$ as possible, stream $S$ by one-block at a time & join every $S$ tuple w/ all $R$ tuples in memory

- I/O’s: $B(R) + \left\lceil \frac{B(R)}{M-2} \right\rceil \cdot B(S)$
  - Or, roughly: $B(R) \cdot B(S)/M$

- Memory requirement: $M$ (as much as possible)

- Which table would you pick as the outer? (exercise)
Simulation After Improvement

- 1 block = 2 tuples, 4 blocks of memory

- Number of I/O:
  \[ B(R) + \frac{B(R)}{(M-2)} \times S(R) = 2 \text{ blocks} + 1 \times 3 \text{ blocks} = 5 \]
Outline For Today

1. DBMS Query Processing Architecture

2. Fundamental Query Processing Operators & Algorithms
   - Assumptions
   - Scan-based Operators
   - Sort-based Operators
   - Hashing-based Operators
   - Algorithms Using Indices
At a high-level majority of DBMS operators require: (1) sorting; or (2) hashing of input tables.

If you have a DBMS that has these core operators implemented in a very performant, robust, and scalable manner, you have a solid query processing foundation.

Key point: Keep optimizing these core algorithms!
A robust DBMS has solutions to the following problem:

Consider an operator \( o \) that needs to sort a table \( R \), maybe a base table or an intermediate table (e.g., implementing ORDER BY clause)

Assume \( o \) is given \( M \) blocks of memory by system’s memory manager but \( M \ll B(R) \). (Not an infrequent scenario)

How can we sort if data does not fit into system’s memory?
Recall in-memory merge sort: Sort progressively larger runs, 2, 4, 8, ..., |R|, by merging consecutive “runs”.

- **Phase 0**: read $M$ blocks of $R$ at a time, sort them, and write out a level-0 run.

- **Phase 1**: merge $(M - 1)$ level-0 runs at a time, and write out a level-1 run.

- **Phase 2**: merge $(M - 1)$ level-1 runs at a time, and write out a level-2 run.

... 

- Final phase produces one sorted run.
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

Arrows indicate the blocks in memory.
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

Arrows indicate the blocks in memory.
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1

Arrows indicate the blocks in memory
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1
- Phase 2 (final)
Example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3
- Phase 0
- Phase 1
- Phase 2 (final)

Arrows indicate the blocks in memory
I/O Cost Analysis

- **Phase 0:** read $M$ blocks of $R$ at a time, sort them, and write out a level-0 run: $2 \times B(R)$ I/Os
  - There are $\left\lceil \frac{B(R)}{M} \right\rceil$ level-0 sorted runs

- **Phase $i$:** merge $(M - 1)$ level-$(i - 1)$ runs at a time, and write out a level-$i$ run: $2 \times B(R)$ I/Os as well

- **Total I/O:** $2B(R) \times \text{Num phases}$

- **# phases:** $1 + \left\lceil \log_{M-1} \left( \frac{B(R)}{M} \right) \right\rceil$

- **Total I/O:** $2B(R) \cdot \left( 1 + \left\lceil \log_{M-1} \left( \frac{B(R)}{M} \right) \right\rceil \right) \approx 2B(R) \log_M (B(R))$

- Observe: As $M$ increases cost decreases. E.g: when $M$ is $B(r)$ cost is $2B(R)$ as expected
Operators That Use Sorting

- Pure Sort: e.g., ORDER BY
- Set Union, Difference, Intersection, or Join or R and S (next slide):
  - When the join condition is an equality condition e.g., R.A = S.B,
    - All can be implemented by walking relations “in tandem” as in the merge step of merge sort.
- Group-By-and-Aggregate: Exercise: Think about how you can implement group-by-and-aggregate with sorting?
- DISTINCT (Related to group-by-and-aggregate)
Sort-merge Join

\[ R \bowtie_{R.A=S.B} S \]

- Sort \( R \) and \( S \) by their join attributes; then merge
  
  \( r, s = \) the first tuples in sorted \( R \) and \( S \)
  
  Repeat until one of \( R \) and \( S \) is exhausted:
  
  If \( r.A > s.B \) then \( s = \) next tuple in \( S \)
  
  else if \( r.A < s.B \) then \( r = \) next tuple in \( R \)
  
  else output all matching tuples, and
  
  \( r, s = \) next in \( R \) and \( S \)

- I/O's: Depends on how many tuples match.
  
  - Common case each \( r \) matches 1 \( s \): \( \text{sorting} + O(B(R) + B(S)) \)
  
  - If every \( r, s \) join (worst-case) \( B(R) \cdot B(S) \):
Example

\[ R: \]
\[ r_1.A = 1 \quad r_2.A = 3 \quad r_3.A = 3 \quad r_4.A = 5 \quad r_5.A = 7 \quad r_6.A = 7 \quad r_7.A = 8 \]

\[ S: \]
\[ s_1.B = 1 \quad s_2.B = 2 \quad s_3.B = 3 \quad s_4.B = 3 \quad s_5.B = 8 \]

\[ R \bowtie_{R.A=S.B} S: \]
\[ r_1s_1 \]
Example

\[ R: \]
\[ r_1.A = 1 \]
\[ r_2.A = 3 \]
\[ r_3.A = 3 \]
\[ r_4.A = 5 \]
\[ r_5.A = 7 \]
\[ r_6.A = 7 \]
\[ r_7.A = 8 \]

\[ S: \]
\[ s_1.B = 1 \]
\[ s_2.B = 2 \]
\[ s_3.B = 3 \]
\[ s_4.B = 3 \]
\[ s_5.B = 8 \]

\[ R \bowtie_{R.A=S.B} S: \]
\[ r_1s_1 \]
Example

\[
R:\quad r_1.A = 1 \\
r_2.A = 3 \\
r_3.A = 3 \\
r_4.A = 5 \\
r_5.A = 7 \\
r_6.A = 7 \\
r_7.A = 8
\]

\[
S:\quad s_1.B = 1 \\
s_2.B = 2 \\
s_3.B = 3 \\
s_4.B = 3 \\
s_5.B = 8
\]

\[
R \bowtie_{R.A=S.B} S:
\]

\[
r_1s_1
\]
Example

\[
R: \\
r_1.A = 1 \\
r_2.A = 3 \\
r_3.A = 3 \\
r_4.A = 5 \\
r_5.A = 7 \\
r_6.A = 7 \\
r_7.A = 8 \\
\]

\[
S: \\
s_1.B = 1 \\
s_2.B = 2 \\
s_3.B = 3 \\
s_4.B = 3 \\
s_5.B = 8 \\
\]

\[
R \Join_{R.A=S.B} S: \\
r_1s_1 \\
\]
Example

\[ R: \]
\[ r_1.A = 1 \]
\[ r_2.A = 3 \]
\[ r_3.A = 3 \]
\[ r_4.A = 5 \]
\[ r_5.A = 7 \]
\[ r_6.A = 7 \]
\[ r_7.A = 8 \]

\[ S: \]
\[ s_1.B = 1 \]
\[ s_2.B = 2 \]
\[ s_3.B = 3 \]
\[ s_4.B = 3 \]
\[ s_5.B = 8 \]

\[ R \Join_{R.A=S.B} S: \]
\[ r_1s_1 \]
\[ r_2s_3 \]
Example

\[ R: \]
\[ r_1.A = 1 \]
\[ r_2.A = 3 \]
\[ r_3.A = 3 \]
\[ r_4.A = 5 \]
\[ r_5.A = 7 \]
\[ r_6.A = 7 \]
\[ r_7.A = 8 \]

\[ S: \]
\[ s_1.B = 1 \]
\[ s_2.B = 2 \]
\[ s_3.B = 3 \]
\[ s_4.B = 3 \]
\[ s_5.B = 8 \]

\[ R \bowtie_{R.A=S.B} S: \]
\[ r_1s_1 \]
\[ r_2s_3 \]
\[ r_2s_4 \]
Example

\[ R : \]
\[ r_1.A = 1 \]
\[ r_2.A = 3 \]
\[ r_3.A = 3 \]
\[ r_4.A = 5 \]
\[ r_5.A = 7 \]
\[ r_6.A = 7 \]
\[ r_7.A = 8 \]

\[ S : \]
\[ s_1.B = 1 \]
\[ s_2.B = 2 \]
\[ s_3.B = 3 \]
\[ s_4.B = 3 \]
\[ s_5.B = 8 \]

\[ R \bowtie_{R.A=S.B} S : \]
\[ r_1s_1 \]
\[ r_2s_3 \]
\[ r_2s_4 \]
Example

\[ R: \]
\[
\begin{align*}
& r_1.A = 1 \\
& r_2.A = 3 \\
& r_3.A = 3 \\
& r_4.A = 5 \\
& r_5.A = 7 \\
& r_6.A = 7 \\
& r_7.A = 8 \\
\end{align*}
\]

\[ S: \]
\[
\begin{align*}
& s_1.B = 1 \\
& s_2.B = 2 \\
& s_3.B = 3 \\
& s_4.B = 3 \\
& s_5.B = 8 \\
\end{align*}
\]

\[ R \bowtie_{R.A=S.B} S: \]
\[
\begin{align*}
& r_1s_1 \\
& r_2s_3 \\
& r_2s_4 \\
& r_3s_3 \\
\end{align*}
\]
Example

\[ R: \]
\[ r_1.A = 1 \]
\[ r_2.A = 3 \]
\[ r_3.A = 3 \]
\[ r_4.A = 5 \]
\[ r_5.A = 7 \]
\[ r_6.A = 7 \]
\[ r_7.A = 8 \]

\[ S: \]
\[ s_1.B = 1 \]
\[ s_2.B = 2 \]
\[ s_3.B = 3 \]
\[ s_4.B = 3 \]
\[ s_5.B = 8 \]

\[ R \bowtie_{R.A=S.B} S: \]
\[ r_1s_1 \]
\[ r_2s_3 \]
\[ r_2s_4 \]
\[ r_3s_3 \]
\[ r_3s_4 \]
Example

\[ R: \]
\[
\begin{align*}
  r_1.A &= 1 \\
  r_2.A &= 3 \\
  r_3.A &= 3 \\
  r_4.A &= 5 \\
  r_5.A &= 7 \\
  r_6.A &= 7 \\
  r_7.A &= 8 \\
\end{align*}
\]

\[ S: \]
\[
\begin{align*}
  s_1.B &= 1 \\
  s_2.B &= 2 \\
  s_3.B &= 3 \\
  s_4.B &= 3 \\
  s_5.B &= 8 \\
\end{align*}
\]

\[ R \bowtie_{R.A=S.B} S: \]
\[
\begin{align*}
  r_1s_1 \\
  r_2s_3 \\
  r_2s_4 \\
  r_3s_3 \\
  r_3s_4 \\
\end{align*}
\]
Example

\[ R: \]
\[
\begin{align*}
    r_1.A &= 1 \\
    r_2.A &= 3 \\
    r_3.A &= 3 \\
    r_4.A &= 5 \\
    r_5.A &= 7 \\
    r_6.A &= 7 \\
    r_7.A &= 8
\end{align*}
\]

\[ S: \]
\[
\begin{align*}
    s_1.B &= 1 \\
    s_2.B &= 2 \\
    s_3.B &= 3 \\
    s_4.B &= 3 \\
    s_5.B &= 8
\end{align*}
\]

\[ R \bowtie_{R.A=S.B} S: \]
\[
\begin{align*}
    r_1s_1 \\
    r_2s_3 \\
    r_2s_4 \\
    r_3s_3 \\
    r_3s_4
\end{align*}
\]
Example

\[ R: \]
\[ r_1.A = 1 \]
\[ r_2.A = 3 \]
\[ r_3.A = 3 \]
\[ \textbf{r}_4.A = 5 \]
\[ r_5.A = 7 \]
\[ r_6.A = 7 \]
\[ r_7.A = 8 \]

\[ S: \]
\[ s_1.B = 1 \]
\[ s_2.B = 2 \]
\[ s_3.B = 3 \]
\[ s_4.B = 3 \]
\[ s_5.B = 8 \]

\[ R \bowtie_{R.A=S.B} S: \]
\[ r_1s_1 \]
\[ r_2s_3 \]
\[ r_2s_4 \]
\[ r_3s_3 \]
\[ r_3s_4 \]
Example

\[ R: \]
\[ r_1.A = 1 \]
\[ r_2.A = 3 \]
\[ r_3.A = 3 \]
\[ r_4.A = 5 \]
\[ r_5.A = 7 \]
\[ r_6.A = 7 \]
\[ r_7.A = 8 \]

\[ S: \]
\[ s_1.B = 1 \]
\[ s_2.B = 2 \]
\[ s_3.B = 3 \]
\[ s_4.B = 3 \]
\[ s_5.B = 8 \]

\[ R \bowtie_{R.A = S.B} S: \]
\[ r_1s_1 \]
\[ r_2s_3 \]
\[ r_2s_4 \]
\[ r_3s_3 \]
\[ r_3s_4 \]
Example

$R$:
- $r_1.A = 1$
- $r_2.A = 3$
- $r_3.A = 3$
- $r_4.A = 5$
- $r_5.A = 7$
- $r_6.A = 7$
- $r_7.A = 8$

$S$:
- $s_1.B = 1$
- $s_2.B = 2$
- $s_3.B = 3$
- $s_4.B = 3$
- $s_5.B = 8$

$R \bowtie_{R.A = S.B} S$: $r_1s_1$
- $r_2s_3$
- $r_2s_4$
- $r_3s_3$
- $r_3s_4$
Example

\[ R: \]
- \( r_1.A = 1 \)
- \( r_2.A = 3 \)
- \( r_3.A = 3 \)
- \( r_4.A = 5 \)
- \( r_5.A = 7 \)
- \( r_6.A = 7 \)
- \( r_7.A = 8 \)

\[ S: \]
- \( s_1.B = 1 \)
- \( s_2.B = 2 \)
- \( s_3.B = 3 \)
- \( s_4.B = 3 \)
- \( s_5.B = 8 \)

\[ R \Join_{R.A=S.B} S: \]
- \( r_1s_1 \)
- \( r_2s_3 \)
- \( r_2s_4 \)
- \( r_3s_3 \)
- \( r_3s_4 \)
- \( r_7s_5 \)
Outline For Today

1. DBMS Query Processing Architecture

2. Fundamental Query Processing Operators & Algorithms
   - Assumptions
   - Scan-based Operators
   - Sort-based Operators
   - Hashing-based Operators
   - Algorithms Using Indices
Hash Join

➢ Consider an equality join, e.g., join of \( R(X, A) \bowtie S(B, Y) \) where \( A=B \).

➢ Let \( h \) be hash function hashing \( A \) or \( B \) values to \( 1, \ldots, k \).

➢ If \( r \in R \) and \( s \in S \) “join”, i.e., \( r[A] = s[B] \) then:

\[
h(r[A]) = h(s[B])
\]

➢ Question: Why is this a useful observation?

➢ Answer: We can:

1. partition \( R \) by hashing its tuples on \( A \) into \( R_1, \ldots, R_k \)
2. partition \( S \) by hashing its tuples on \( B \) into \( S_1, \ldots, S_k \)
3. where each partitions are (i) much smaller tables (e.g., they may fit in memory) & (ii) tuples in \( R_i \) can only join with tuples in \( S_i \)

If the join will be computed externally, i.e., using disk, hash join can be an efficient algorithm.
Hash Join vs Nested Loop Join Pictorially

- \( R \bowtie_{R.A=S.B} S \)
- If \( r.A \) and \( s.B \) get hashed to different partitions, they don’t join

Nested-loop join considers all slots

Hash join considers only those along the diagonal!
(External) Hash Join Algorithm

- let $h$ be a hash function mapping $A$ or $B$ columns into $1, \ldots, k$
- Pick $k < M$ (why?)

Phase 1: partition $R$ and $S$ into $k$ partitions using $h$

Phase 2:
for $i = 1, \ldots, k$:

- read $R_i$ and $S_i$ into memory and use in-memory hash-join alg.
  (i.e., build a hash table of $R_i$ and probe $S_i$ tuples)

Q: Systems make the smaller (estimated) relation the build side? Why?
1. Quicker to build & search in hash table

Note: do not have to use any memory for probe side, e.g., can just stream $S$. 

Phase 1: Partitioning

Partition \( R \) and \( S \) according to the same hash function on their join attributes.

Each partition takes \( \sim B(R)/k \) blocks.
Phase 2: Probing

- Read in $R_i$, stream in the corresponding partition $S_i$ and join
  - Typically use in-memory hash join: build a hash table of $R_i$
  - Often use a different hash function for in-memory hash join
I/O Cost of Hash Join

- Assuming no edges cases (e.g., a very large hash partition) and hash join completes in two phases:

  I/O’s: \( 3 \cdot (B(R) + B(S)) \)
  
  Phase 1: read \( B(R) + B(S) \) into memory to partition and write partitioned \( B(R) + B(S) \) to disk
  
  Phase 2: read \( B(R) + B(S) \) into memory to perform the join
What if \( R_i \) Does Not Fit In Memory

- Read it back in and partition it again!
- Note however, in the worst-case all A or B values could be the same and one simply cannot generate useful partitions.
- Systems either fail in such cases or fall back to alternative slower solutions
Outline For Today

1. DBMS Query Processing Architecture

2. Fundamental Query Processing Operators & Algorithms
   - Assumptions
   - Scan-based Operators
   - Sort-based Operators
   - Hashing-based Operators
   - Algorithms Using Indices
Selection Using Index

- Equality predicate: $\sigma_{A=v}(R)$
  - Use an ISAM, B+-tree, or hash index on $R(A)$

- Range predicate: $\sigma_{A>v}(R)$
  - Use an ordered index (e.g., ISAM or B+-tree) on $R(A)$
  - Hash index is not applicable

- Indexes other than those on $R(A)$ may be useful
  - Example: B+-tree index on $R(A, B)$
  - How about B+-tree index on $R(B, A)$?
    - Not useful because A values will be scattered.
Index vs Table Scan (1)

- Situations where index clearly is the better choice:
  - Index-only equality queries on a unique column (e.g., primary key)
    - E.g. \( \sigma_{A=v}(R) \) where A is a unique column and has an index.
  - Index guarantees 1 I/O. Table scan can lead up to \( B(R) \) I/Os.

- Index-only range queries on clustered indices:
  - \( \sigma_{A>v}(R) \): guarantee that only the blocks that contain answers are read (aside from blocks of the index)
BUT(!): Consider \( \sigma_{A>v}(R) \) and a secondary, non-clustered index on \( R(A) \)

- Need to follow pointers to get the actual result tuples
- Say that 20% of \( R \) satisfies \( A > v \)
  - Could happen even for equality predicates
- Back-of-the-envelope calculation:
  
  I/O’s for table scan: \( B(R) \)
  
  I/O’s for index scan up to: lookup + 20% \(|R|\) (assume no cache hits)

Table scan is faster if a block contains more than 5 tuples!

\[
B(R) = \frac{|R|}{5} < 20|R| + \text{lookup}
\]

Systems should not do this to be safe. Table scan might be slow but its slowness is bounded by \( B(R) \) and not a function of \( R \).
Index Nested-loop Join

\[ R \bowtie_{R.A = S.B} S \]

- Idea: use a value of \( R.A \) to probe the index on \( S(B) \) for each block of \( R \), and for each \( r \) in the block:
  - use the index on \( S(B) \) to retrieve \( s \) with \( s.B = r.A \)
  - output \( rs \)

- I/O’s: \( B(R) + |R| \cdot (\text{index lookup}) \)
  - Let’s assume the cost of an index lookup is 2-4 I/O’s (depends on the index tree height if B+ tree)
  - Key takeaway 1: Can be faster than hash/sort-merge join if \( |R| \) is small
  - Key takeaway 2: Better pick \( R \) to be the smaller relation

- Memory requirement: \( O(1) \) (extra memory can be used to cache index, e.g. root of B+ tree).
Zig-zag Join Using Ordered Indexes

\( R \bowtie_{R.A=S.B} S \)

- Idea: use the ordering provided by the indexes on \( R(A) \) and \( S(B) \) to eliminate the sorting step of sort-merge join
- Use the larger key to probe the other index
  Possibly skipping many keys that don’t match
Zig-zag Join Using Ordered Indexes

\[ R \bowtie_{R.A=S.B} S \]

- Idea: use the ordering provided by the indexes on \( R(A) \) and \( S(B) \) to eliminate the sorting step of sort-merge join
- Use the larger key to probe the other index
  - Possibly skipping many keys that don’t match
Some systems use estimated I/O costs of core ops to pick how to translate logical to physical plans, i.e., logical-physical plan translation is a cost-based optimization:

Logical Plan

Physical Plan 1  
(cost = C1)

Project cid

HashJoin  
L.$3=R.$3

NestLoopJ  
L.$3=R.2

FileScan Order

FileScan Customer

IndexScan Product name=BookA

Physical Plan 2  
(cost = C2)

Project cid

Join  
P.pid=O.pid

Join  
C.cid=O.cid

Scan Table Order

Scan Table Customer

Scan Table Product name=BookA

Filter name=BookA

HashJoin  
L.$3=R.$3

FileScan Order

FileScan Customer

IndexScan Product name=BookA
Note on Role of (I/O) Costs Of Operators For Picking Physical Plans

- Some systems use estimated I/O costs of core ops to pick how to translate logical to physical plans, i.e., logical-physical plan translation is a cost-based optimization:
  1. Enumerate different physical plan translations
  2. Estimate the cost of each physical plan
  3. Pick the minimum cost estimated plan
- Next lecture on cost-based optimization at logical plan picking level

- Others pick physical operators in a rule-based manner, so the I/O costs are there to determine these rules.
  - Always make scans IndexScans if possible
  - All equality joins should be HashJoins except if the tables are already sorted, in which case use SortMerge Join.
  - All other types of joins should be NesteLoopJoins etc.