Intro to Database Systems (15-445/645)

11 Join Algorithms
 ADMINISTRIVIA

Homework 3 due Sunday
→ You may not turn in Homework 3 late

Midterm exam Wednesday, March 1st
→ Practice exam coming tomorrow
→ Exam accommodations? Schedule with EOS

Project 2 is available
→ First checkpoint due Friday, March 3rd (15% of P2 grade)
→ Overall due Wednesday, March 22nd (85% of P2 grade)
LAST TIME

Finished concurrent B+Trees

Sorting
→ Top-k heap sort
→ External merge sort

Aggregations
→ External hashing
The operators are arranged in a tree.

Data flows from the leaves of the tree up towards the root.
→ We will discuss the granularity of the data movement next week.

The output of the root node is the result of the query.
WHY DO WE NEED TO JOIN?

We normalize tables in a relational database to avoid unnecessary repetition of information.

We then use the **join operator** to reconstruct the original tuples without any information loss.
JOIN ALGORITHMS

We will focus on performing binary joins (two tables) using **inner equijoin** algorithms.
→ These algorithms can be tweaked to support other joins.
→ Multi-way joins exist primarily in research literature.

In general, we want the smaller table to always be the left table ("outer table") in the query plan.
→ The optimizer will (try to) figure this out when generating the physical plan.
JOIN OPERATORS

Decision #1: Output
→ What data does the join operator emit to its parent operator in the query plan tree?

Decision #2: Cost Analysis Criteria
→ How do we determine whether one join algorithm is better than another?

```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```
For tuple \( r \in R \) and tuple \( s \in S \) that match on join attributes, concatenate \( r \) and \( s \) together into a new tuple.

Output contents can vary:
→ Depends on processing model
→ Depends on storage model
→ Depends on data requirements in query
Early Materialization:
→ Copy the values for the attributes in outer and inner tuples into a new output tuple.

Subsequent operators in the query plan never need to go back to the base tables to get more data.

```sql
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```
**OPERATOR OUTPUT: RECORD IDS**

**Late Materialization:**
→ Only copy the joins keys along with the Record IDs of the matching tuples.

Ideal(?) for column stores because the DBMS does not copy data that is not needed for the query.

```sql
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```
COST ANALYSIS CRITERIA

Assume:
→ \( M \) pages in table \( R \), \( m \) tuples in \( R \)
→ \( N \) pages in table \( S \), \( n \) tuples in \( S \)

Cost Metric: # of I/Os to compute join

We ignore overall output costs because it depends on the data and is the same for all algorithms.

\[
\text{SELECT R.id, S.cdate}
\text{FROM R JOIN S}
\text{ON R.id = S.id}
\text{WHERE S.value > 100}
\]
**JOIN VS CROSS-PRODUCT**

$R \bowtie S$ is the most common operation and thus must be carefully optimized.

$R \times S$ followed by a selection is inefficient because the cross-product is large.

There are many algorithms for reducing join cost, but no algorithm works well in all scenarios.
JOIN ALGORITHMS

Nested Loop Join
→ Naïve
→ Block
→ Index

Sort-Merge Join

Hash Join
→ Simple
→ GRACE (Externally Partitioned)
→ Hybrid
NAÏVE NESTED LOOP JOIN

foreach tuple \( r \in R \):
  foreach tuple \( s \in S \):
    if \( r \) and \( s \) match then emit

\[ R(id, name) \]

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>600</td>
<td>MethodMan</td>
</tr>
<tr>
<td>200</td>
<td>GZA</td>
</tr>
<tr>
<td>100</td>
<td>Andy</td>
</tr>
<tr>
<td>300</td>
<td>ODB</td>
</tr>
<tr>
<td>500</td>
<td>RZA</td>
</tr>
<tr>
<td>700</td>
<td>Ghostface</td>
</tr>
<tr>
<td>400</td>
<td>Raekwon</td>
</tr>
</tbody>
</table>

\[ S(id, value, cdate) \]

<table>
<thead>
<tr>
<th>id</th>
<th>value</th>
<th>cdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>2222</td>
<td>2/23/23</td>
</tr>
<tr>
<td>500</td>
<td>7777</td>
<td>2/23/23</td>
</tr>
<tr>
<td>400</td>
<td>6666</td>
<td>2/23/23</td>
</tr>
<tr>
<td>100</td>
<td>9999</td>
<td>2/23/23</td>
</tr>
<tr>
<td>200</td>
<td>8888</td>
<td>2/23/23</td>
</tr>
</tbody>
</table>
NAÏVE NESTED LOOP JOIN

Why is this algorithm bad?
→ For every tuple in \( R \), it scans \( S \) once

Cost: \( M + (m \cdot N) \)

\( R(id, name) \)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>600</td>
<td>MethodMan</td>
</tr>
<tr>
<td>200</td>
<td>GZA</td>
</tr>
<tr>
<td>100</td>
<td>Andy</td>
</tr>
<tr>
<td>300</td>
<td>ODB</td>
</tr>
<tr>
<td>500</td>
<td>RZA</td>
</tr>
<tr>
<td>700</td>
<td>Ghostface</td>
</tr>
<tr>
<td>400</td>
<td>Raekwon</td>
</tr>
</tbody>
</table>

\( S(id, value, cdate) \)

<table>
<thead>
<tr>
<th>id</th>
<th>value</th>
<th>cdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>2222</td>
<td>2/23/23</td>
</tr>
<tr>
<td>500</td>
<td>7777</td>
<td>2/23/23</td>
</tr>
<tr>
<td>400</td>
<td>6666</td>
<td>2/23/23</td>
</tr>
<tr>
<td>100</td>
<td>9999</td>
<td>2/23/23</td>
</tr>
<tr>
<td>200</td>
<td>8888</td>
<td>2/23/23</td>
</tr>
</tbody>
</table>

\( M \) pages
\( m \) tuples

\( N \) pages
\( n \) tuples
NAÏVE NESTED LOOP JOIN

Example database:
→ Table R: \( M = 1000, \ m = 100,000 \)
→ Table S: \( N = 500, \ n = 40,000 \)

Cost Analysis:
→ \( M + (m \cdot N) = 1000 + (100000 \cdot 500) = 50,001,000 \) IOs
→ At 0.1 ms/IO, Total time ≈ 1.3 hours

What if smaller table (S) is used as the outer table?
→ \( N + (n \cdot M) = 500 + (40000 \cdot 1000) = 40,000,500 \) IOs
→ At 0.1 ms/IO, Total time ≈ 1.1 hours

4 KB pages \( \rightarrow \) 6 MB
**BLOCK NESTED LOOP JOIN**

```plaintext
foreach block \( B_R \in R: \\
foreach block \( B_S \in S: \\
foreach tuple \( r \in B_R: \\
foreach tuple \( s \in B_S: \\
\text{if } r \text{ and } s \text{ match then emit}
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>600</td>
<td>MethodMan</td>
</tr>
<tr>
<td>200</td>
<td>GZA</td>
</tr>
<tr>
<td>100</td>
<td>Andy</td>
</tr>
<tr>
<td>300</td>
<td>ODB</td>
</tr>
<tr>
<td>500</td>
<td>RZA</td>
</tr>
<tr>
<td>700</td>
<td>Ghostface</td>
</tr>
<tr>
<td>400</td>
<td>Raekwon</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>value</th>
<th>cdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>2222</td>
<td>2/23/23</td>
</tr>
<tr>
<td>500</td>
<td>7777</td>
<td>2/23/23</td>
</tr>
<tr>
<td>400</td>
<td>6666</td>
<td>2/23/23</td>
</tr>
<tr>
<td>100</td>
<td>9999</td>
<td>2/23/23</td>
</tr>
<tr>
<td>200</td>
<td>8888</td>
<td>2/23/23</td>
</tr>
</tbody>
</table>

\( M \) pages \( m \) tuples

\( N \) pages \( n \) tuples
This algorithm performs fewer disk accesses.
→ For every block in \( R \), it scans \( S \) once.

Cost: \( M + ((\# \text{ blocks in } R) \cdot N) \)

**R(id, name)**

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>600</td>
<td>MethodMan</td>
</tr>
<tr>
<td>200</td>
<td>GZA</td>
</tr>
<tr>
<td>100</td>
<td>Andy</td>
</tr>
<tr>
<td>300</td>
<td>ODB</td>
</tr>
<tr>
<td>500</td>
<td>RZA</td>
</tr>
<tr>
<td>700</td>
<td>Ghostface</td>
</tr>
<tr>
<td>400</td>
<td>Raekwon</td>
</tr>
</tbody>
</table>

\( M \) pages \( m \) tuples

**S(id, value, cdate)**

<table>
<thead>
<tr>
<th>id</th>
<th>value</th>
<th>cdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>2222</td>
<td>2/23/23</td>
</tr>
<tr>
<td>500</td>
<td>7777</td>
<td>2/23/23</td>
</tr>
<tr>
<td>400</td>
<td>6666</td>
<td>2/23/23</td>
</tr>
<tr>
<td>100</td>
<td>9999</td>
<td>2/23/23</td>
</tr>
<tr>
<td>200</td>
<td>8888</td>
<td>2/23/23</td>
</tr>
</tbody>
</table>

\( N \) pages \( n \) tuples
The smaller table should be the outer table. We determine size based on the number of pages, not the number of tuples.

\[ M \text{ pages} \quad m \text{ tuples} \]

\[ N \text{ pages} \quad n \text{ tuples} \]
If we have \( B \) buffers available:
→ Use \( B - 2 \) buffers for each block of the outer table.
→ Use one buffer for the inner table, one buffer for output.

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>value</th>
<th>cdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>GZA</td>
<td>2222</td>
<td>2/23/23</td>
</tr>
<tr>
<td>500</td>
<td>RZA</td>
<td>7777</td>
<td>2/23/23</td>
</tr>
<tr>
<td>400</td>
<td>Raekwon</td>
<td>6666</td>
<td>2/23/23</td>
</tr>
<tr>
<td>200</td>
<td>RZA</td>
<td>8888</td>
<td>2/23/23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>value</th>
<th>cdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>GZA</td>
<td>8888</td>
<td>2/23/23</td>
</tr>
<tr>
<td>100</td>
<td>GZA</td>
<td>9999</td>
<td>2/23/23</td>
</tr>
<tr>
<td>500</td>
<td>RZA</td>
<td>7777</td>
<td>2/23/23</td>
</tr>
<tr>
<td>400</td>
<td>RZA</td>
<td>6666</td>
<td>2/23/23</td>
</tr>
<tr>
<td>100</td>
<td>GZA</td>
<td>2222</td>
<td>2/23/23</td>
</tr>
</tbody>
</table>

\( M \) pages \( m \) tuples \n\( N \) pages \( n \) tuples
**BLOCK NESTED LOOP JOIN**

If we have \( B \) buffers available:

→ Use \( B - 2 \) buffers for each block of the outer table.
→ Use one buffer for the inner table, one buffer for output.

```
foreach \( B - 2 \) pages \( p_R \in R \):
  foreach page \( p_S \in S \):
    foreach tuple \( r \in B - 2 \) pages:
      foreach tuple \( s \in p_S \):
        if \( r \) and \( s \) match then emit
```

<table>
<thead>
<tr>
<th>( R(id, name) )</th>
<th>( S(id, value, cdate) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>name</td>
</tr>
<tr>
<td>600</td>
<td>MethodMan</td>
</tr>
<tr>
<td>200</td>
<td>GZA</td>
</tr>
<tr>
<td>100</td>
<td>Andy</td>
</tr>
<tr>
<td>300</td>
<td>ODB</td>
</tr>
<tr>
<td>500</td>
<td>RZA</td>
</tr>
<tr>
<td>700</td>
<td>Ghostface</td>
</tr>
<tr>
<td>400</td>
<td>Raekwon</td>
</tr>
</tbody>
</table>

\( M \) pages \( m \) tuples

\( N \) pages \( n \) tuples
BLOCK NESTED LOOP JOIN

This algorithm uses $B-2$ buffers for scanning $R$.

Cost: $M + \left( \left\lceil \frac{M}{B-2} \right\rceil \cdot N \right)$

If the outer relation fits in memory ($M < B-2$):
$\rightarrow$ Cost: $M + N = 1000 + 500 = 1500$ I/Os
$\rightarrow$ At 0.1ms per I/O, Total time $\approx 0.15$ seconds

If we have $B=102$ buffer pages:
$\rightarrow$ Cost: $M + \left( \left\lceil \frac{M}{B-2} \right\rceil \cdot N \right) = 1000 + 10 \times 500 = 6000$ I/Os
$\rightarrow$ Or can switch inner/outer relations, giving us cost:
$\quad 500 + 5 \times 1000 = 5500$ I/Os
NESTED LOOP JOIN

Why is the basic nested loop join so bad?
→ For each tuple in the outer table, we must do a sequential scan to check for a match in the inner table.

We can avoid sequential scans by using an index to find inner table matches.
→ Use an existing index for the join.
INDEX NESTED LOOP JOIN

Assume the cost of each index probe is some constant \( C \) per tuple.

Cost: \( M + (m \cdot C) \)

### Example

#### R(id, name)
- **id**
  - 600
  - 200
  - 100
  - 300
  - 500
  - 700
  - 400
- **name**
  - MethodMan
  - GZA
  - Andy
  - ODB
  - RZA
  - Ghostface
  - Raekwon

#### S(id, value, cdate)
- **id**
  - 100
  - 500
  - 400
  - 100
  - 200
- **value**
  - 2222
  - 7777
  - 6666
  - 9999
  - 8888
- **cdate**
  - 2/23/23
  - 2/23/23
  - 2/23/23
  - 2/23/23
  - 2/23/23
NESTED LOOP JOIN SUMMARY

**Key Takeaways**

→ Pick the smaller table as the outer table.
→ Buffer as much of the outer table in memory as possible.
→ Loop over the inner table (or use an index).

**Algorithms**

→ Naïve
→ Block
→ Index
SORT-MERGE JOIN

Phase #1: Sort
→ Sort both tables on the join key(s).
→ You can use any appropriate sort algorithm
→ These phases are distinct from the sort/merge phases of an external merge sort, from the previous class

Phase #2: Merge
→ Step through the two sorted tables with cursors and emit matching tuples.
→ May need to backtrack depending on the join type.
SORT-MERGE JOIN

sort \( R, S \) on join keys

cursor_\( R \) ← \( R_{\text{sorted}} \), cursor_\( S \) ← \( S_{\text{sorted}} \)

while cursor_\( R \) and cursor_\( S \):
    if cursor_\( R \) > cursor_\( S \):
        increment cursor_\( S \)
    if cursor_\( R \) < cursor_\( S \):
        increment cursor_\( R \) (and possibly backtrack cursor_\( S \))
    elif cursor_\( R \) and cursor_\( S \) match:
        emit
        increment cursor_\( S \)
**SORT-MERGE JOIN**

**R(id,name)**

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Andy</td>
</tr>
<tr>
<td>200</td>
<td>GZA</td>
</tr>
<tr>
<td>200</td>
<td>GZA</td>
</tr>
<tr>
<td>300</td>
<td>ODB</td>
</tr>
<tr>
<td>400</td>
<td>Raekwon</td>
</tr>
<tr>
<td>500</td>
<td>RZA</td>
</tr>
<tr>
<td>600</td>
<td>MethodMan</td>
</tr>
<tr>
<td>700</td>
<td>Ghostface</td>
</tr>
</tbody>
</table>

**S(id,value,cdate)**

<table>
<thead>
<tr>
<th>id</th>
<th>value</th>
<th>cdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>2222</td>
<td>2/23/23</td>
</tr>
<tr>
<td>100</td>
<td>9999</td>
<td>2/23/23</td>
</tr>
<tr>
<td>200</td>
<td>8888</td>
<td>2/23/23</td>
</tr>
<tr>
<td>400</td>
<td>6666</td>
<td>2/23/23</td>
</tr>
<tr>
<td>500</td>
<td>7777</td>
<td>2/23/23</td>
</tr>
</tbody>
</table>

**Output Buffer**

```sql
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```

<table>
<thead>
<tr>
<th>R.id</th>
<th>R.name</th>
<th>S.id</th>
<th>S.value</th>
<th>S.cdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Andy</td>
<td>100</td>
<td>2222</td>
<td>2/23/23</td>
</tr>
<tr>
<td>100</td>
<td>Andy</td>
<td>100</td>
<td>9999</td>
<td>2/23/23</td>
</tr>
<tr>
<td>200</td>
<td>GZA</td>
<td>200</td>
<td>8888</td>
<td>2/23/23</td>
</tr>
<tr>
<td>200</td>
<td>GZA</td>
<td>200</td>
<td>8888</td>
<td>2/23/23</td>
</tr>
<tr>
<td>400</td>
<td>Raekwon</td>
<td>200</td>
<td>6666</td>
<td>2/23/23</td>
</tr>
<tr>
<td>500</td>
<td>RZA</td>
<td>500</td>
<td>7777</td>
<td>2/23/23</td>
</tr>
</tbody>
</table>
SORT-MERGE JOIN

Sort Cost (R): \(2M \cdot (1 + \lceil \log_{B-1} \left\lfloor M / B \right\rfloor \rceil)\)

Sort Cost (S): \(2N \cdot (1 + \lceil \log_{B-1} \left\lfloor N / B \right\rfloor \rceil)\)

Merge Cost: \((M + N)\)

Total Cost: Sort + Merge
SORT-MERGE JOIN

Example database:
→ **Table R**: \( M = 1000, \ m = 100,000 \)
→ **Table S**: \( N = 500, \ n = 40,000 \)

With \( B=100 \) buffer pages, both \( R \) and \( S \) can be sorted in two passes:
→ Sort Cost (\( R \)) = \( 2000 \cdot (1 + \lceil \log_{99} \frac{1000}{100} \rceil) = 4000 \) I/Os
→ Sort Cost (\( S \)) = \( 1000 \cdot (1 + \lceil \log_{99} \frac{500}{100} \rceil) = 2000 \) I/Os
→ Merge Cost = \( 1000 + 500 = 1500 \) I/Os
→ Total Cost = \( 4000 + 2000 + 1500 = 7500 \) I/Os
→ At 0.1 ms/IO, Total time ≈ 0.75 seconds
SORT-MERGE JOIN

The worst case for the merging phase is when the join attribute of all the tuples in both relations contains the same value.

Cost: \((M \cdot N) + (\text{sort cost})\)
WHEN IS SORT-MERGE JOIN USEFUL?

One or both tables are already sorted on join key. Output must be sorted on join key.

The input relations may be sorted either by an explicit sort operator, or by scanning the relation using an index on the join key.
HASH JOIN

If tuple \( r \in R \) and a tuple \( s \in S \) satisfy the join condition, then they have the same value for the join attributes.

If that value is hashed to some partition \( i \), the \( R \) tuple must be in \( r_i \) and the \( S \) tuple in \( s_i \).

Therefore, \( R \) tuples in \( r_i \) need only to be compared with \( S \) tuples in \( s_i \).
SIMPLE HASH JOIN ALGORITHM

Phase #1: Build
→ Scan the outer relation and populate a hash table using the hash function $h_1$ on the join attributes.

Phase #2: Probe
→ Scan the inner relation and use $h_1$ on each tuple to jump to a location in the hash table and find a matching tuple.
**SIMPLE HASH JOIN ALGORITHM**

**build** hash table $HT_R$ for $R$

**foreach** tuple $s \in S$

**output**, if $h_1(s) \in HT_R$
HASH TABLE CONTENTS

**Key:** The attribute(s) that the query is joining on
→ The hash table needs to store the key to verify that we have a correct match, in case of hash collisions.

**Value:** It varies
→ Depends on what the next query operators will do with the output from the join
→ Early vs. Late Materialization
OPTIMIZATION: PROBE FILTER

Create a probe filter (such as a Bloom Filter) during the build phase if the key is likely to not exist in the inner relation
→ Check the filter before probing the hash table
→ Fast because the filter fits in CPU cache
BLOOM FILTERS

Uses a bitmap to probabilistically answer set membership queries
→ False negatives will never occur
→ False positives can sometimes occur

**Insert(x):**
→ Use $k$ hash functions to set bits in the filter to 1

**Lookup(x):**
→ Check whether the bits are 1 for each hash function

See the [Bloom Filter Calculator](#) if you build one
BLOOM FILTERS

Bloom Filter

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Insert('RZA')

Insert('GZA')

Lookup(RZA) → TRUE

Lookup('Raekwon') → FALSE

Lookup('ODB') → TRUE

hash₁('RZA') = 3333 % 8 = 5
hash₂('RZA') = 7777 % 8 = 1

hash₁('GZA') = 5555 % 8 = 3
hash₂('GZA') = 7777 % 8 = 1

hash₁('ODB') = 6699 % 8 = 3
hash₂('ODB') = 9966 % 8 = 1

hash₁('Raekwon') = 3333 % 8 = 5
hash₂('Raekwon') = 8899 % 8 = 5
What happens if we do not have enough memory to fit the entire hash table?

We do not want to let the buffer pool manager swap out the hash table pages at random.
PARTITIONED HASH JOIN

Hash join when tables do not fit in memory.

→ **Partition Phase:** Hash both tables on the join attribute into partitions.

→ **Probe Phase:** Compares tuples in corresponding partitions for each table.

Sometimes called **GRACE Hash Join.**

→ Named after the GRACE database machine from Japan in the 1980s.
Hash join when tables do not fit in memory.

→ **Build Phase:** Hash both tables on the join attribute into partitions.

→ **Probe Phase:** Compares tuples in corresponding partitions for each table.

Sometimes called **GRACE Hash Join**.

→ Named after the GRACE database machine from Japan in the 1980s.
PARTITIONED HASH JOIN PARTITION PHASE

Hash $R$ into $k$ buckets.
Hash $S$ into $k$ buckets with same hash function.
Write buckets to disk when they get full.
PARTITIONED HASH JOIN PROBE PHASE

Read corresponding partitions into memory one pair at a time, hash join their contents.
PARTITIONED HASH JOIN EDGE CASES

If a partition does not fit in memory, recursively partition it with a different hash function
→ Repeat as needed
→ Eventually hash join the corresponding (sub-)partitions

If a single join key has so many matching records that they don’t fit in memory, use a block nested loop join for that key
RECURSIVE PARTITIONING

\[ R(id,\text{name}) \]

\[ h_1 \]

\[ h_2 \]

\[ \vdots \]

\[ k-1 \]

\[ S(id,\text{value},\text{cdate}) \]
How big a table can be joined without recursive partitioning?
→ Up to $B-1$ partitions
→ Each could be about as big as $B-2$ pages

Answer: About $(B-1) \cdot (B-2)$ pages
→ If the partitions are approximately equal size, a table of $N$ pages needs about $\sqrt{N}$ buffers
→ In practice, use a "fudge factor" $f > 1$: $\sqrt{f \cdot N}$
→ Only partitions of the outer table need to fit in memory
COST OF PARTITIONED HASH JOIN

If we don’t need recursive partitioning:
→ Cost: $3(M + N)$

Partition phase:
→ Read+write both tables
→ $2(M+N)$ I/Os

Probe phase:
→ Read both tables (in total, one partition at a time)
→ $M+N$ I/Os
PARTITIONED HASH JOIN

Example database:
→ $M = 1000, \ m = 100,000$
→ $N = 500, \ n = 40,000$

Cost Analysis:
→ $3 \cdot (M + N) = 3 \cdot (1000 + 500) = 4,500\ IOs$
→ At 0.1 ms/IO, Total time ≈ 0.45 seconds
OPTIMIZATION: HYBRID HASH JOIN

Use some buckets for a simple in-memory hash join, have some buckets spill to disk.

\[
\begin{align*}
R & (id, name) \\
0 & \quad 1 & \quad 2 \\
S & (id, value, cdate)
\end{align*}
\]
The inner table can be any size.
→ Only outer table (or its partitions) need to fit in memory

If we know the size of the outer table, then we can use a static hash table.
→ Less computational overhead

If we do not know the size, then we must use a dynamic hash table or allow for overflow pages.
## JOIN ALGORITHMS: SUMMARY

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>IO Cost</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve Nested Loop Join</td>
<td>$M + (m \cdot N)$</td>
<td>1.3 hours</td>
</tr>
<tr>
<td>Block Nested Loop Join</td>
<td>$M + \left( \left\lceil \frac{M}{B-2} \right\rceil \cdot N \right)$</td>
<td>0.55 seconds</td>
</tr>
<tr>
<td>Index Nested Loop Join</td>
<td>$M + (m \cdot C)$</td>
<td>Variable</td>
</tr>
<tr>
<td>Sort-Merge Join</td>
<td>$M + N + \text{(sort cost)}$</td>
<td>0.75 seconds</td>
</tr>
<tr>
<td>Hash Join</td>
<td>$3 \cdot (M + N)$</td>
<td>0.45 seconds</td>
</tr>
</tbody>
</table>
CONCLUSION

Hashing is almost always better than sorting for operator execution.

Caveats:
→ Sorting is better on non-uniform data.
→ Sorting is better when result needs to be sorted.

Good DBMSs use either (or both).
NEXT CLASS

Composing operators together to execute queries.