Algorithms for query evaluation

Join
How to parse SQL query

SELECT a,b
FROM X,Y,Z
WHERE X.c=Y.c AND Z.d > 12

1. What relations are involved: FROM clause
2. Selection condition on rows: WHERE clause
3. Projection on columns: SELECT clause
Cartesian product (cross-product)

If there is no WHERE clause for 2 relations, it is probably a bug, as it will produce a Cartesian product (cross-product) – a huge relation of size T(R)*T(S)
Join: reminder

• **Natural join** (⋈) - a Cartesian product with equality condition on common attributes

  Example:
  
  • If $R$ has schema $R(A, B, C, D)$, and if $S$ has schema $S(E, B, D)$
  
  • Common attributes: $B$ and $D$
  
  • Then:

  $$ R \bowtie S = \pi_{A, B, C, D, E} [\sigma_{R.B = S.B \land R.D = S.D} (R \times S)] $$

  • In SQL:

    ```
    ```

    ```
    SELECT * FROM R NATURAL JOIN S
    ```
Join: Example

$R \bowtie S$

**Example:** Returns all pairs of tuples $r \in R, s \in S$ such that $r.A = s.A$

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

**SQL:**

```
SELECT R.A, B, C, D
FROM R, S
WHERE R.A = S.A
```
Join: Example

Example: Returns all pairs of tuples $r \in R, s \in S$ such that $r.A = s.A$
Join: Example

\[ R \bowtie S \]

**Example:** Returns all pairs of tuples \( r \in R, s \in S \) such that \( r.A = s.A \)

**SQL:**

```
SELECT R.A, B, C, D
FROM R, S
WHERE R.A = S.A
```
Join: Example

R \bowtie S

```sql
SELECT R.A, B, C, D
FROM R, S
WHERE R.A = S.A
```

Example: Returns all pairs of tuples $r \in R, s \in S$ such that $r.A = s.A$
Join: Example

\[ R \bowtie S \]

**Example:** Returns all pairs of tuples \( r \in R, s \in S \) such that \( r.A = s.A \)

**SQL Query:**

```
SELECT R.A,B,C,D
FROM R, S
WHERE R.A = S.A
```
Semantically: A Subset of the Cross Product

\[ \mathbf{R} \times \mathbf{S} \]

**Example:** Returns all pairs of tuples \( r \in R, s \in S \) such that \( r.A = s.A \)

Cross Product

Filter by conditions (\( r.A = s.A \))

Can we actually implement a join this way?

| \( R \) | \( S \ |
|---|---|---|---|---|---|---|---|
| A | B | C | A | D |
| 1 | 0 | 1 | 3 | 7 |
| 2 | 3 | 4 | 2 | 2 |
| 2 | 5 | 2 | 2 | 3 |
| 3 | 1 | 1 | 3 | 1 | 1 | 7 |

**SQL:**

```sql
SELECT R.A, B, C, D
FROM R, S
WHERE R.A = S.A
```
Join algorithm I: Nested Loop
Setup

• We write $R \bowtie S$ to mean *join R and S by returning all tuple pairs where all shared attributes are equal*

• We write $R \bowtie S$ on $A$ to mean *join R and S by returning all tuple pairs where attribute(s) A are equal*

• For simplicity, we’ll consider joins on **two tables** and with **equality constraints** ("equijoins")

• Given a relation $R$, let:
  • $T(R)$ = # of tuples in $R$
  • $B(R)$ = # of blocks (pages) in $R$

Recall that we read / write entire pages with disk IO

However joins *can* merge > 2 tables, and some algorithms do support non-equality constraints!
Nested Loop Join (NLJ)

Compute $R \bowtie S$ on $A$:

for $r$ in $R$:
    for $s$ in $S$:
        if $r[A] == s[A]$:
            OUT ($r,s$)
Nested Loop Join (NLJ)

Compute $R \bowtie S$ on $A$:

```python
for r in R:
    for s in S:
        if r[A] == s[A]:
            OUT (r,s)
```

Cost:

$B(R)$

1. Loop over the tuples in $R$

Note that our IO cost is based on the number of pages loaded, not the number of tuples!
Nested Loop Join (NLJ)

Compute $R \bowtie S$ on $A$:

for $r$ in $R$:
  for $s$ in $S$:
    if $r[A] == s[A]$:
      OUT ($r, s$)

Cost:

$$B(R) + T(R) \times B(S)$$

1. Loop over the tuples in $R$
2. For every tuple in $R$, loop over all the tuples in $S$

Have to read all of $S$ from disk for every tuple in $R$!
Nested Loop Join (NLJ)

Compute $R \bowtie S$ on $A$:

for $r$ in $R$:
    for $s$ in $S$:
        if $r[A] == s[A]$:
            OUT $(r,s)$

Note that NLJ can handle things other than equality constraints... just change the if statement!

Cost:

$B(R) + T(R) \times B(S)$

1. Loop over the tuples in $R$
2. For every tuple in $R$, loop over all the tuples in $S$
3. Check against join conditions
Nested Loop Join (NLJ)

Compute $R \bowtie S$ on $A$:

```
for r in R:
    for s in S:
        if r[A] == s[A]:
            OUT (r,s)
```

What would the result be if our join condition is trivial (if TRUE)?

Cost:

$B(R) + T(R) \times B(S)$

1. Loop over the tuples in $R$
2. For every tuple in $R$, loop over all the tuples in $S$
3. Check against join conditions
4. Output combined tuple if match
Nested Loop Join (NLJ)

Compute $R \bowtie S$ on $A$:

```
for r in R:
    for s in S:
        if r[A] == s[A]:
            OUT (r,s)
```

Cost:

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

What if $R$ ("outer") and $S$ ("inner") switched?

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

Outer vs. inner selection makes a huge difference - DBMS needs to know which relation is smaller!
Join algorithm IA:
Block Nested Loop
IO-aware modification
Block Nested Loop Join (BNLJ)

Compute $R \bowtie S$ on $A$:

for each chunk $c_R$ of $R$ of size $M-1$:
  load $c_R$ pages of $R$ into mem

  for each $p_s$ page of $S$:
    for each tuple $s$ in $p_s$:
      for each tuple $r$ in $c_R$
        if $r[A] == s[A]$: 
          OUT $(r,s)$

Cost:

1. Load in $M-1$ pages of $R$ at a time (leaving 1 page free for $S$)

Note: There could be some speedup here due to the fact that we’re reading multiple pages sequentially however we’ll ignore this here!
Block Nested Loop Join (BNLJ)

Compute $R \bowtie S$ on $A$:

for each chunk $c_R$ of $R$ of size M-1:
  load $c_R$ pages of $R$ into mem

  for each $p_s$ page of $S$:
    for each tuple $s$ in $p_s$:
      for each tuple $r$ in $c_R$
        if $r[A] == s[A]$:
          OUT (r,s)

Cost:

Given $M$ pages of memory

\[
B(R) + \frac{B(R)}{M - 1} B(S)
\]

1. Load in M-1 pages of R at a time (leaving 1 page free for S)

2. For each (M-1)-page segment of R, load each page of S

Note: Faster to iterate over the smaller relation first!
Block Nested Loop Join (BNLJ)

Compute $R \bowtie S$ on $A$:

for each chunk $c_R$ of $R$ of size $M-1$:
  load $c_R$ pages of $R$ into mem
  for each $p_s$ page of $S$:
    for each tuple $r$ in $c_R$
      if $r[A] == s[A]$
        OUT ($r,s$)

Given $M$ pages of memory

Cost:

$$B(R) + \frac{B(R)}{M-1} B(S)$$

1. Load in M-1 pages of $R$ at a time (leaving 1 page free for $S$)

2. For each (M-1)-page segment of $R$, load each page of $S$

3. Check against the join conditions with all in-mem tuples

BNLJ can also handle non-equality constraints
BNLJ vs. NLJ: Benefits of IO Aware

• In BNLJ, by loading larger chunks of R, we minimize the number of full disk reads of S
  - We only read all of S from disk for every (M-1)-page segment of R!
  - Still the full cross-product, but more done in memory

\[
B(R) + T(R) \times B(S) \\
B(R) + \frac{B(R)}{M-1} B(S)
\]

BNLJ is faster by roughly \(\frac{(M-1)T(R)}{B(R)}\)!
BNLJ vs. NLJ: Benefits of IO Aware

• Example:
  • B(R) = 500 pages
  • B(S) = 1000 pages
  • T (R) = 50,000 tuples
  • T (S) = 100,000 tuples
  • We have 11 pages of memory (M = 11)

• **NLJ**: Cost = 500 + \(50,000 \times 1000\) = 50 Million IOs \(\approx 140\) hours

• **BNLJ**: Cost = \(500 + \frac{500 \times 1000}{10}\) = 50 Thousand IOs \(\approx 0.14\) hours

A very real difference from a small change in the algorithm!
Can we do better than Cross-Product?
Smarter than cross-products: from quadratic to nearly linear

• All joins that compute the **full cross-product** have some **quadratic** term
  • For example we saw:

\[
\text{NLJ: } B(R) + T(R)B(S)
\]

\[
\text{BNLJ: } B(R) + \frac{B(R)}{M - 1} B(S)
\]

• Now we’ll see some (nearly) linear joins:
  • \(\sim O(B(R) + B(S))\)

We get this gain by *taking advantage of data structures and algorithms* – for simplicity considering equality constraints ("equijoin") only!
Join algorithms II:
Index Nested Loop
Index Nested Loop Join (INLJ)

Compute \( R \bowtie S \) on \( A \):

Given index \( I \) on \( S.A \):

\[
\text{for } r \text{ in } R: \\
\quad s_L = \text{index } I(r[A]) \\
\text{for } s \text{ in } s_L: \\
\quad \text{OUT } r,s
\]

→ We can use an index (e.g. B+ Tree) to **avoid doing the full cross-product**!

Cost:

\[
P(R) + T(R) \times (\text{TH}_i + SC(S,A))
\]

where \( \text{TH}_i \) is the height of a B-tree and \( SC(S,A) \) is the IO cost to collect all values equal to \( r[A] \) in the index of \( S.A \); assuming these fit on one page, \(~3\) is a good estimate.

\[
P(R) + 3 \times T(R)
\]
We want to compute $R(X,Y) \bowtie S(Y,Z)$ on $Y$.

Suppose there is an index on $S[Y]$. 

Cost:

- $B(R)$ to read entire $R$ once
- Each tuple of $R$ joins with $SC(S,Y) = T(S)/V(S,Y)$ tuples of $S$, on average.
- If $S$ has a non-clustered index on $Y$:
  - I/O cost is $B(R) + T(R) \times (TH_i + T(S)/V(S,Y))$
- If $S$ has a clustered index on $Y$:
  - I/O cost is $B(R) + T(R) \times (TH_i + B(S)/V(S,Y))$

**Algorithm:**
for each tuple $r$ of $R$, lookup all tuples in $S$ with key-value $r[Y]$ and output their join with $r$. 
INLJ: cost example

- \( T(R) = 10,000 \), \( B(R) = 1000 \)
- \( T(S) = 5000 \), \( B(S) = 500 \), \( V(S,Y) = 100 \)
- \( M = 11 \)

INLJ:
- To compute \( R(X,Y) \bowtie S(Y,Z) \) using a clustered index on \( S[Y] \):
  \[ 1000 + 10,000 \times 3 \times \left( \frac{500}{100} \right) = 153,000 \text{ I/O’s} \]
- Even if the top level of B-tree is buffered:
  \[ 1000 + 10,000 \times \left( \frac{500}{100} \right) = 51,000 \text{ I/O’s} \]

BNLJ:
- \( 1000 + 100 \times 500 = \textbf{51,000 I/Os} \)

\[ \text{Use of index is not beneficial if selection cardinality is high (50 in this example)} \]
Join using sorted indexes

• We want to compute \( R(X,Y) \bowtie S(Y,Z) \) on Y

• If both R and S have sorted (B-tree) index on \( Y \), do a **zigzag-join**:
  
  • We scan the leaves of both B-trees in order. In the best case, we use just \( B(R) + B(S) \) disk I/O’s to read the blocks of R and S through their indexes (if there are no matching values).
Zigzag Join - example

Leaves of B-tree index on R[Y]

Leaves of B-tree index on S[Y]

• Start with the 1 and 2. Since 1<2 skip 1 in R’s index.
Zigzag Join - example

Leaves of B-tree index on R[Y]

Leaves of B-tree index on S[Y]

- Start with the 1 and 2. Since 1<2 skip 1 in R’s index.
- Since 2<3 skip the 2’s in S’s index.
Zigzag Join - example

Leaves of B-tree index on R[Y]

Leaves of B-tree index on S[Y]

• Start with the 1 and 2. Since 1<2 skip 1 in R’s index.
• Since 2<3 skip the 2’s in S’s index.
Zigzag Join - example

Leaves of B-tree index on R[Y]
Leaves of B-tree index on S[Y]

• Start with the 1 and 2. Since 1<2 skip 1 in R’s index.
• Since 2<3 skip the 2’s in S’s index.
• Since 3<4 skip 3 in R.
Zigzag Join - example

Leaves of B-tree index on R[Y]

Leaves of B-tree index on S[Y]

- Start with the 1 and 2. Since 1<2 skip 1 in R’s index.
- Since 2<3 skip the 2’s in S’s index.
- Since 3<4 skip 3 in R.
- Join 4’s (retrieve records).
Zigzag Join - example

Leaves of B-tree index on R[Y]

Leaves of B-tree index on S[Y]

• Start with the 1 and 2. Since 1<2 skip 1 in R’s index.
• Since 2<3 skip the 2’s in S’s index.
• Since 3<4 skip 3 in R.
• Join 4’s (retrieve records).
• …
Zigzag Join

- We jump back and forth between the indexes finding Y-values that they share in common.
- Tuples from R with Y-value that don’t appear in S need never be retrieved, and similarly tuples of S whose Y-value doesn’t appear in R need never be retrieved.
- The worst-case cost (clustered indexes, R < S):
  \[ B(R) + B(S) + \frac{B(R) \times B(S)}{V(S, a)} \]
Join algorithm III:
Sort-Merge Join (SMJ)
Sort Merge Join (SMJ): Basic Procedure

To compute $R \bowtie S$ on $A$:

1. Sort $R, S$ on $A$ using **external merge sort**

2. **Scan** sorted files and “merge”

3. [May need to “backup” - see next]

Note that if $R, S$ are already sorted on $A$, SMJ will be awesome!
SMJ Example: $R \bowtie S$ on $A$

with 3-page buffer

- For simplicity: Let each page be *one tuple*, and let the first value be of column $A$
SMJ Example: $R \bowtie S$ on $A$
with 3-page buffer

1. Sort the relations $R$, $S$ on the join key (first value)

```
R
(5,b) (3,j) (0,a)

S
(0,j) (3,g) (7,f)
```
SMJ Example: $R \bowtie S$ on $A$
with 3-page buffer

2. Scan and “merge” on join key!

We show the current file pointer, which is the value currently in buffer

Disk

Output

(0,a) (3,j) (5,b)
(0,j) (3,g) (7,f)

Main Memory

Buffer
SMJ Example: $R \bowtie S$ on $A$

with 3-page buffer

2. Scan and “merge” on join key!
SMJ Example: $R \bowtie S$ on $A$ with 3-page buffer

2. Scan and “merge” on join key!

```

<table>
<thead>
<tr>
<th></th>
<th>Main Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>$\langle 3, j \rangle$</td>
</tr>
<tr>
<td>S</td>
<td>$\langle 0, j \rangle$</td>
</tr>
<tr>
<td>Output</td>
<td>$\langle 0, a, j \rangle$</td>
</tr>
</tbody>
</table>
```

Disk

$R$ with 3-page buffer

$S$ with 3-page buffer

Output with 3-page buffer

$R \bowtie S$ on $A$
SMJ Example: $R \bowtie S$ on $A$
with 3-page buffer

2. Done!
What happens if join keys have many duplicates?
Multiple tuples with same join key: “backup”

1. Start with sorted relations, and begin scan / merge...
1. Start with sorted relations, and begin scan / merge...

Multiple tuples with same join key: “backup”
Multiple tuples with same join key: “backup”

1. Start with sorted relations, and begin scan / merge...
Multiple tuples with same join key: “backup”

1. Start with sorted relations, and begin scan / merge...

Have to “backup” in the scan of S and read tuples we’ve already read!
SMJ: cost of a final scan

- At best, no backup $\rightarrow$ final scan takes $B(R) + B(S)$ reads
  - For ex.: if no duplicate values in join attribute

- At worst (e.g. full backup each time), scan could take $B(R) \times B(S)$ reads!
  - For ex.: if all duplicate values in join attribute, i.e. all tuples in R and S have the same value for the join attribute
  - Roughly: For each page of R, we’ll have to back up and read each page of S...
  - Not a very realistic scenario
SMJ: Total cost

- Cost of SMJ is **cost of sorting** R and S and **writing** temporary sorted files: $4B(R) + 4B(S)$

- Plus the **cost of scanning**: $\sim B(R) + B(S)$
  - Because of *backup*: in worst case $B(R) \times B(S)$; but this would be very unlikely

$$5B(R) + 5B(S)$$
SMJ cost: example

• We have 101 buffer pages,
  • \( B(R) = 1000 \) and \( B(S) = 500 \) pages:
    • **SMJ**:
      • Sort both in two passes: \( 4 \times 1000 + 4 \times 500 = \textbf{6,000 IOs} \)
      • Merge-join phase \( 1000 + 500 = 1,500 \) IOs
      • \( = 7,500 \) IOs
    • What with **BNLJ**?
      • \( 500 + 1000 \left\lfloor \frac{500}{100} \right\rfloor = \textbf{5,500 IOs} \)
    • But, if we have 26 buffer pages?
      • **SMJ** has same behavior (still 2 passes): \( \textbf{= 7,500 IOs} \)
      • **BNLJ?** \textbf{25,500 IOs}!

SMJ is ~ linear vs. BNLJ is quadratic...
A simple optimization for SMJ: join during sort

- SMJ is composed of a 2PMMS sort and a join of sorted tuples

- During the 2PMMS, if R and S have \( \leq (M - 1) \) (sorted) runs in total:
  - We could do two separate merges (for each of R & S) at this point, complete the sort phase, and start the join phase...
  - OR, we could combine them: do one \((M - 1)\)-way merge simultaneously for R and S and complete the join!
Un-Optimized SMJ

Sort Phase
(Ext. Merge Sort)

- Split & sort
- Merge

Merge / Join Phase

Join output file created!

Unsorted input relations

Given $M$ buffer pages
Simple SMJ Optimization

Partition sort Phase (2PMMS)

Given $M$ buffer pages

Unsorted input relations

Split & sort

Split & sort

(M-1)-Way Merge / Join

<= (M-1) total runs

Merge / Join Phase

Joined output file created!
Optimized SMJ: memory requirements

- If we can initially split R and S into total $M-1$ runs, each run of length $\leq M$, then we only need $3(B(R) + B(S))$ for SMJ!
  - 2 Read/Write per page to sort runs in memory, 1 Read per page to $(M-1)$-way merge / join!
- How much memory for this to happen?
  - $\frac{B(R)+B(S)}{M-1} \leq M \Rightarrow B(R) + B(S) \leq M^2$
- Thus, $M \geq \sqrt{B(R) + B(S)}$ is an approximate sufficient condition for this algorithm

If the sum of R,S has $\leq M^2$ pages, then SMJ costs $3(B(R)+B(S))$!
Takeaway points from SMJ

If input already sorted on join key, skip the sorts.

- SMJ is basically linear.
- Nasty but unlikely case: too many duplicate join keys.

SMJ needs to sort both relations

- If max $B(R) + B(S) \leq M^2$ then cost is $3(B(R) + B(S))$
Join algorithm IV: Hash Join (HJ)
Recall: Hashing

- **Magic of hashing:**
  - A hash function $h_M$ maps into $[0,M-1]$
  - And maps nearly uniformly

- A hash **collision** is when $x \neq y$ but $h_M(x) = h_M(y)$
  - Note however that it will **never** occur that $x = y$ but $h_M(x) \neq h_M(y)$

- We hash on an attribute $A$, so our hash function $h_M(t)$ has the form $h_M(t.A)$.
  - **Collisions** may be more frequent, as we have much more tuples than the buckets
Hash Join: High-level

To compute $R \bowtie S$ on $A$:

1. **Partition Phase**: Using one (shared) hash function $h_M$, partition $R$ and $S$ into $M-1$ buckets.

2. **Matching Phase**: Take pairs of buckets whose tuples have the same values for $h$, and join these.

We *decompose* the problem using $h_M$, then complete the join.
HJ: high-level

1. **Partition Phase:** Using one (shared) hash function $h_M$, partition $R$ and $S$ into $M-1$ buckets.

Note our new convention: pages each have two tuples (one per row).
HJ: high-level

2. **Matching Phase**: Take pairs of buckets whose tuples have the same values for $h_M$, and join these.
HJ: high-level

2. Matching Phase: Take pairs of buckets whose tuples have the same values for $h_M$, and join these.
Hash Join phase 1: partitioning

**Goal:** For each relation, partition relation into **buckets** such that if $h_M(t.A) = h_M(t'.A)$ they are in the same bucket

Given $M$ buffer pages, we partition into $M-1$ buckets:

- We use $M-1$ buffer pages for output (one for each bucket), and 1 for input
  - The “dual” of merge-sorting.
  - For each tuple $t$ in input, copy to a buffer page $h_M(t.A)$
  - When buffer fills up, flush to disk.
How big are the resulting buckets?

• Given B blocks of R, we partition into M-1 buckets:
  • → Ideally our buckets are each of equal size $\sim B/M$ pages

• What happens if there are many hash collisions?
  • Some buckets could be $> B/M$

• What happens if there are multiple duplicate join keys?
  • Nothing we can do here... could have some skew in size of the buckets
How big at most do we want the resulting buckets?

• Ideally, our buckets would be of size $\leq M - 1$ pages

• Recall: If we want to join a bucket $R_i$ from R and one from S, we can do BNLJ in linear time if for one of them (say $R_i$), $B(R_i) \leq M - 1$!
  • And more generally, being able to fit bucket in memory is advantageous

Recall for BNLJ:

$$B(R) + \frac{B(R)B(S)}{M - 1} = 1$$

Given $M$ buffer pages
We partition into \( M-1 = 2 \) buckets using hash function \( h_2 \) so that we can have one buffer page for each partition (and one for input).

For simplicity, we’ll look at partitioning one of the two relations— we just do the same for the other relation!

Recall: our goal will be to get \( M - 1 = 2 \) buckets of size \( \leq M - 1 \Rightarrow 2 \) pages each.
Hash Join Phase 1: Example

1. We read pages from R into the “input” page of the buffer...

Given $M = 3$ buffer pages
Hash Join Phase 1: Example

2. Then we use hash function $h_2$ to find the output bucket, which each has one page in the buffer
Hash Join Phase 1: Example

2. Then we use **hash function** $h_2$ to find the output bucket, which each has one page in the buffer.

![Diagram showing disk and main memory with hash function application](image_url)
Hash Join Phase 1: Example

3. We repeat until the buffer bucket pages are full...

Given $M = 3$ buffer pages
Hash Join Phase 1: Example

3. We repeat until the buffer bucket pages are full... then flush to disk

Given $M = 3$ buffer pages
Hash Join Phase 1: Example

Note that collisions can occur!

Given $M = 3$ buffer pages

Collision!!!

$\text{hash}(5) = \text{hash}(3) = 1$
Hash Join Phase 1: Example

Finished phase I for R

Given $M = 3$ buffer pages
Hash Join Phase 1: complete

Given $M = 3$ buffer pages

We wanted buckets of size $M-1 = 2$... Some of them could be larger due to:

1. Duplicate join keys
2. Hash collisions
Now that we have partitioned R and S...
Hash Join Phase 2: Matching

- Now, we just join pairs of buckets from R and S that have the same hash value to complete the join!
Hash Join Phase 2: Matching

• Note that since $x = y \rightarrow h(x) = h(y)$, we only need to consider pairs of buckets (one from R, one from S) that have the same hash function value

• If our buckets are $\sim M - 1$ pages each, can join each such pair using BNLJ \textit{in linear time}; recall (with $B(R) = M - 1$):

BNLJ Cost: $B(R) + \frac{B(R)B(S)}{M-1} = B(R) + \frac{(M-1)B(S)}{M-1} = B(R) + B(S)$

Joining the pairs of buckets is linear! (As long as smaller bucket $\leq M-1$ pages)
Hash Join Phase 2: Matching

If condition is an equality – we explore only matching buckets – diagonal
Hash Join Phase 2: Matching

If it is not an equijoin, we explore this whole grid!
Hash Join: memory requirements

• Given M buffer pages

• Suppose (reasonably) that we can partition into M buckets in 1 pass:
  • For R, we get M buckets of size \( \sim B(R)/M \)
  • To join these buckets in linear time, we need each bucket of R to fit in M-1 pages, so we have:

\[
M - 1 \geq \frac{B(R)}{M} \Rightarrow \sim M^2 \geq B(R)
\]

Quadratic relationship between smaller relation’s size & memory!
Hash Join: cost

- *Given enough buffer pages as on previous slide...*

  - **Partitioning** requires reading + writing each page of R, S
    - → $2(\text{B}(R) + \text{B}(S))$ IOs

  - **Matching** (with BNLJ) requires reading each page of R, S
    - → $\text{B}(R) + \text{B}(S)$ IOs

  **HJ takes $\sim 3(B(R)+B(S))$ !**
Sort-Merge vs. Hash Join

• *Given enough memory*, both SMJ and HJ have performance:

  \[ \sim 3(B(R)+B(S)) \]

• “Enough” memory =

  • SMJ: \( M^2 > B(R) + B(S) \)
  
  • HJ: \( M^2 > \min\{B(R), B(S)\} \)

Hash Join superior if relation sizes *differ greatly*. Why?
Further Comparison of Hash vs. Sort Joins

• Hash Joins are highly parallelizable.

• Sort-Merge less sensitive to data skew and result is sorted
Summary

• Saw IO-aware join algorithms
  • Massive difference

• Memory sizes key in hash versus sort join
  • Hash Join = Little dog (depends on smaller relation)

• Skew is also a major factor
Impact of Buffering

• If several operations are executing concurrently, estimating the number of available buffer pages is guesswork.

• Repeated access patterns interact with buffer replacement policy.
  • e.g., Inner relation is scanned repeatedly in Simple Nested Loop Join. With enough buffer pages to hold inner, replacement policy does not matter. Otherwise, MRU is best, LRU is worst (sequential flooding).

• Does replacement policy matter for Block Nested Loops?
• What about Index Nested Loops? Sort-Merge Join?