Optimization!
Optimizing Queries and workloads

Workload = <Query, Frequency of query>
Example:

Basic SFW queries

Workload description

SELECT pname
FROM    Product
WHERE year = ? AND category =?

SELECT pname
FROM    Product
WHERE year = ? AND Category =?
AND manufacturer = ?

Lower cost (query and update cost)

1. How to execute? Sort, Hash first …?
2. Maintain indexes for Year? Category? Manufacturer?
3. For query, check multiple indexes?
4. What’s cost of maintaining index?
5. Use multiple machines? ...

Intuition

Manufacturers likely most Selective.

Many more manufacturers than Categories. Maintain index, if this query happens a lot.
1. For SFW, Joins queries
   b. Pre-build an index? B+ tree, Hash?

2. What statistics can I keep to optimize?
   a. E.g. Selectivity of columns, values

Build Query Plans

Cost in I/O, resources?
To query, maintain?

Analyze Plans
1. Nested Loop Joins
What you will learn about in this section

1. RECAP: Joins
2. Nested Loop Join (NLJ)
3. Block Nested Loop Join (BNLJ)
4. Index Nested Loop Join (INLJ)
RECAP: Joins
Joins: Example

\[ R \Join S \]

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

\[
\begin{array}{|c|c|c|}
\hline
A & B & C \\
\hline
1 & 0 & 1 \\
2 & 3 & 4 \\
2 & 5 & 2 \\
3 & 1 & 1 \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
A & D \\
\hline
3 & 7 \\
2 & 2 \\
2 & 3 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
A & B & C & D \\
\hline
2 & 3 & 4 & 2 \\
\hline
\end{array}
\]
Joins: Example

\[ R \bowtie S \]

\[
\begin{align*}
R & \quad S \\
\begin{array}{ccc}
1 & 0 & 1 \\
2 & 3 & 4 \\
2 & 5 & 2 \\
3 & 1 & 1 \\
\end{array} & \quad \begin{array}{cc}
3 & 7 \\
2 & 2 \\
2 & 3 \\
\end{array}
\end{align*}
\]

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

**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 \( r.A = s.A \)

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

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
## Joins: Example

\[ R \Join S \]

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

### SQL Query

\[
\text{SELECT } R.A, B, C, D \\
\text{FROM } R, S \\
\text{WHERE } R.A = S.A
\]
Joins: 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
```
Semantically: A Subset of the Cross Product

```
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 \( r.A = s.A \)

Can we actually implement a join in this way?
Nested Loop Joins
We consider “IO aware” algorithms: care about disk IO

Given a relation R, let:
• $T(R) = \# \text{ of tuples in } R$
• $P(R) = \# \text{ of pages in } R$

We’ll see lots of formulae from now
⇒ Hint: Focus on how it works. Much easier to derive from 1st principles (vs recalling formula soup)
Nested Loop Join (NLJ)

Compute $R \bowtie S$ on $A$:
for $r$ in $R$:
    for $s$ in $S$:
        if $r[A] == s[A]$:
            yield $(r,s)$
Nested Loop Join (NLJ)

Cost:

\[ P(R) \]

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 $r[A] == s[A]$: yield $(r,s)$

Have to read all of $S$ from disk for every tuple in $R$!

Cost:
$P(R) + T(R)P(S)$

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

Cost:
\[ P(R) + T(R) * P(S) \]

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**

Note that NLJ can handle things other than equality constraints... just check in the *if* statement!
Nested Loop Join (NLJ)

Compute $R \bowtie S$ on $A$:
for r in R:
  for s in S:
    if r[A] == s[A]:
      yield (r,s)

Cost:
$P(R) + T(R) \times P(S) + \text{OUT}$

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. Write out (to page, then when page full, to disk)

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

OUT could be bigger than $P(R) \times P(S)$... but usually not that bad
Nested Loop Join (NLJ)

Compute $R \bowtie S$ on $A$:
for $r$ in $R$:
  for $s$ in $S$:
    if $r[A] == s[A]$:
      yield $(r,s)$

Cost:
$P(R) + T(R) \times P(S) + OUT$

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

$P(S) + T(S) \times P(R) + OUT$

Outer vs. inner selection makes a huge difference—DBMS needs to know which relation is smaller!
IO-Aware Approach
Quick Intuition

For 1 Billion people
Goal: Compute Census JOIN TaxInfo
Data stored in RowStores, 1000 tuples/page (million pages)

Census (SSN, Address, ...)
TaxInfo (SSN, TaxPaid, ...)

Census table (row store)
- 132-567-789
- 432-567-789
- 134-562-184
- 613-416-452

TaxInfo table (row store)
- 134-562-184
- 613-416-452
- 432-567-789

BNLJ
For each pair of pages in Census and TaxInfo...
Example

For 1 Billion people

Goal: Compute Census JOIN TaxInfo

Data stored in RowStores, 1000 tuples/page (million pages)

Census (SSN, Address, ...)  TaxInfo (SSN, TaxPaid, ...)

Better?
For each pair of pages in Census and TaxInfo...
Block Nested Loop Join (BNLJ)

Given $B+1$ pages of memory (For $B \ll P(R), P(S)$)

Cost:

$P(R)$

1. Load in $B-1$ pages of $R$ at a time (leaving 1 page each free for $S$ & output)

Note: There could be some speedup here due to the fact that we're reading in multiple pages sequentially however we'll ignore this here!

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

for each $B-1$ pages $pr$ of $R$:

for page $ps$ of $S$:

for each tuple $r$ in $pr$:

for each tuple $s$ in $ps$:

if $r[A] == s[A]$:

yield $(r,s)$
Block Nested Loop Join (BNLJ)

Given B+1 pages of memory

Cost:
\[ P(R) + \frac{P(R)}{B - 1} P(S) \]

1. Load in B-1 pages of R at a time (leaving 1 page each free for S & output)
2. For each (B-1)-page segment of R, load each page of S

Compute \( R \bowtie S \) on \( A \):
for each B-1 pages pr of R:
  for each page ps of S:
    for each tuple r in pr:
      for each tuple s in ps:
        if \( r[A] = s[A] \):
          yield (r,s)

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

Given \( B+1 \) pages of memory

Cost:

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

1. Load in \( B-1 \) pages of R at a time (leaving 1 page each free for S & output)
2. For each \( (B-1) \)-page segment of R, load each page of S
3. Check against the join conditions

BNLJ can also handle non-equality constraints

Compute \( R \bowtie S \) on \( A \):
for each \( B-1 \) pages pr of R:
  for page ps of S:
    for each tuple r in pr:
      for each tuple s in ps:
        if \( r[A] == s[A] \):
          yield \( (r,s) \)
Given $B+1$ pages of memory

Cost:

\[ P(R) + \frac{P(R)}{B-1} P(S) + \text{OUT} \]

1. Load in $B-1$ pages of $R$ at a time (leaving 1 page each free for $S$ & output)
2. For each $(B-1)$-page segment of $R$, load each page of $S$
3. Check against the join conditions
4. Write out
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 \((B-1)\)-page segment of R!
- Still the full cross-product, but more done only in memory

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

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

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

Example:
- R: 500 pages
- S: 1000 pages
- 100 tuples / page
- We have 12 pages of memory ($B = 11$)

- 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!
Quick Recap

Block Nested Loop Joins

For 1 Billion people

Goal: Compute Census JOIN TaxInfo

Data stored in RowStores, 1000 tuples/page (million pages)

Given: B + 1 buffer space
Idea: Use B-1 pages for Census, 1 page each for TaxInfo and output

Steps: Repeat till done
Read B-1 pages from Census into Buffer
Read 1 page from TaxInfo
Partial Join into 1 output page
Block Nested Loop Join (BNLJ)

Compute $R \bowtie S$ on $A$:
for each B-1 pages $pr$ of $R$:
  for each page $ps$ of $S$:
    for each tuple $r$ in $pr$:
      for each tuple $s$ in $ps$:
        if $r[A] == s[A]$:
          yield $(r,s)$

Cost:
Given $B+1$ pages of memory

$P(R) + \frac{P(R)}{B-1} P(S) + \text{OUT}$

How many R pages to read?
$P(R)$

How many times is each S page read?
$P(R)/(B-1)$
Example NLJ vs. BNLJ: Steel Cage Match

Example: \( P(R) = 1000, P(S) = 500, \)
100 tuples/page \( \Rightarrow \) \( T(R) = 1000 \times 100, T(S) = 500 \times 100 \]

\[
\begin{array}{|c|c|}
\hline
 & B + 1 = 100 & B + 1 = 20 \\
 & (i.e., B = 99) & (i.e., Buffer B = 19) \\
\hline
\text{NLJ} & (1000 + 1000 \times 100 \times 500 + \text{OUT}) & (1000 + 1000 \times 100 \times 500 + \text{OUT}) \\
& \Rightarrow \text{IO} = \sim 5,001,000 + \text{OUT} & \Rightarrow \text{IO} = \sim 5,001,000 + \text{OUT} \\
\text{BNLJ} & (500 + 1000 \times 500/(99-1)) & (500 + 1000 \times 500/(19-1)) \\
& \Rightarrow \text{IO} = \sim 5.6K + \text{OUT} & \Rightarrow \text{IO} = 28.2 \text{K IOs} + \text{OUT} \\
\hline
\end{array}
\]

But it’s all about the memory.
Smarter than Cross-Products
Smarter than Cross-Products: From Quadratic to Nearly Linear

All joins computing the *full cross-product* have a quadratic term

- For example we saw:

\[
P(R) + T(R)P(S) + \text{OUT}
\]

Now we’ll see some (nearly) linear joins:

- \(\sim O(P(R) + P(S) + \text{OUT})\)

We get this gain by *taking advantage of structure*—moving to equality constraints (“equijoin”) only!
Index Nested Loop Join (INLJ)

Compute $R \bowtie S$ on $A$:
Given index idx on $S.A$:
for $r$ in $R$:
  s in idx($r[A]$):
  yield $r,s$

Cost:
$P(R) + T(R)L + OUT$
Where $L$ is the IO cost to access each distinct values in index
Recall: $L$ is usually small (e.g., 3-5)

→ We can use an index (e.g. B+ Tree) to avoid full cross-product!
Optimizing Joins
(the good stuff, multi table joins)

Message: It’s all about the IO and memory!
0. Intuition for smarter joins

1. Sort-Merge Join

2. HashPartition Joins
Example

Census (SSN, Address, ...)
TaxInfo (SSN, TaxPaid, ...)

For 1 Billion people
Goal: Compute Census JOIN TaxInfo
Data stored in RowStores, 1000 tuples/page (million pages)

BNLJ
For each pair of pages in Census and TaxInfo...
For 1 Billion people

Goal: Compute Census JOIN TaxInfo

Data stored in RowStores, 1000 tuples/page (million pages)

Example
Pre-process data before JOINing

- Sort(Census), Sort(TaxInfo) on SSN
- Merge sorted pages

-- Sort(Census), Sort(TaxInfo) on SSN
-- Merge sorted pages

-- Hash(Census), Hash(TaxInfo) on SSN
-- Merge partitioned pages
Speedy Joins: With Sorting and Hashing

- **Given enough memory**, SortMergeJoin and HashJoins cost
  \[ \sim 3(P(R)+P(S)) + OUT \]

- Hash Joins are highly parallelizable

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

⇒ **Big takeaway**: IO-aware join algorithms
  - Massive difference vs brute-force
  - Nearly linear vs quadratic (or worse)
0. Intuition for smarter joins

1. Sort-Merge Join

2. HashPartition Joins
Sort-Merge Join (SMJ)
1. Sort-Merge Join

2. “Backup” & Total Cost

3. Optimizations
Sort Merge Join (SMJ)

**Goal:** Execute \( R \bowtie S \) on \( A \)

**Key Idea:**
We can sort \( R \) and \( S \) [with external sort]
Then just merge-scan over them!

**IO Cost:**
- *Sort phase:* \( \text{Sort}(R) + \text{Sort}(S) \approx 2 (P(R) + P(S)) \)
- *Merge/join phase:* \( \approx P(R) + P(S) + \text{OUT} \)
SMJ Example: $R \bowtie S$ with 3 page buffer

For simplicity: Let each page be *one tuple*, and let the first value be join key.

We show the file HEAD, which is the next value to be read!
SMJ Example: $R \bowtie S$ with 3 page buffer
SMJ Example: $R \bowtie S$ with 3 page buffer

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

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

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

2. Done!
What happens with duplicate join keys?
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…
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 page we’ve already read!
Backup

- At best, no backup → scan takes $P(R) + P(S)$ reads
  - For ex: if no duplicate values in join attribute

- At worst (e.g. full backup each time), scan could take $P(R) \times P(S)$ 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 back up and read each page of S...

- Often not that bad however, plus we can:
  - Leave more data in buffer (for larger buffers)
  - Can try other algorithms
SMJ: Total cost

- Cost of SMJ is **cost of sorting** R and S…

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

- Plus the **cost of writing out**: OUT

\[
\sim \text{Sort}(P(R)) + \text{Sort}(P(S)) + P(R) + P(S) + \text{OUT}
\]

Recall: \(\text{Sort}(N) \approx 2N \left(\left\lfloor \log_B \frac{N}{2(B+1)} \right\rfloor + 1 \right)\)

*Note: this is using repacking, where we estimate that we can create initial runs of length \(\sim 2(B+1)\)*
Un-Optimized SMJ

Sort Phase (Ext. Merge Sort)

Merge / Join Phase

Given $B+1$ buffer pages

Unsorted input relations

Joined output file created!
Simple SMJ Optimization

Given $B+1$ buffer pages

Unsorted input relations

Sort Phase (Ext. Merge Sort)

$\leq B$ total runs

Merge / Join Phase

$B$-Way Merge / Join

Joined output file created!
Takeaway points from SMJ

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

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

SMJ needs to sort **both** relations
Example SMJ Number of passes

Consider $P(R) = 1000$, $P(S) = 500$

Num passes for $R$ (for $B+1 = 100$) =
$$K = \lceil \log_{99} \frac{1000}{2 \times 99} \rceil + 1) = 2$$

Num passes for $R$ (for $B+1 = 20$)
$$K = \lceil \log_{19} \frac{1000}{2 \times 19} \rceil + 1) = 3$$

(Repeat for $S$, and you get $k = 2$ and $3$)

Reminder: More Buffer? Fewer passes for Sorting

Recall: $Sort(N) \approx 2N \left( \left\lfloor \log_2 \frac{N}{2(B+1)} \right\rfloor + 1 \right)$

Note: this is using repacking, where we estimate that we can create initial runs of length $\sim 2(B+1)$
Example SMJ vs. BNLJ: Steel Cage Match

Consider $P(R) = 1000$, $P(S) = 500$

<table>
<thead>
<tr>
<th></th>
<th>Buffer = 100</th>
<th>Buffer = 20</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(i.e., $B+1=100$)</td>
<td>(i.e., $B+1=20$)</td>
</tr>
<tr>
<td>SMJ</td>
<td>$\sim \text{Sort}(P(R)) + \text{Sort}(P(S)) + P(R) + P(S) + \text{OUT}$</td>
<td>$\sim \text{Sort}(P(R)) + \text{Sort}(P(S)) + P(R) + P(S) + \text{OUT}$</td>
</tr>
<tr>
<td></td>
<td>$2^* (k^* 1000 + k^* 500) = 6000$</td>
<td>$2^* (k^* 1000 + k^* 500) = 9000$</td>
</tr>
<tr>
<td></td>
<td>Merge: 1000 + 500 = 1500 IOs)</td>
<td>Merge: 1000 + 500 = 1500 IOs)</td>
</tr>
<tr>
<td></td>
<td>$\Rightarrow \text{IO} = 7500 \text{ IOs} + \text{OUT}$</td>
<td>$\Rightarrow \text{IO} = 10,500 \text{ IOs} + \text{OUT}$</td>
</tr>
<tr>
<td>BNLJ</td>
<td>$500 + 1000*500/(99-1))$</td>
<td>$(500 + 1000*500/(19-1))$</td>
</tr>
<tr>
<td></td>
<td>$\Rightarrow \text{IO} = \sim 5.6 \text{K} + \text{OUT}$</td>
<td>$\Rightarrow \text{IO} = 28.2 \text{K} \text{ IOs} + \text{OUT}$</td>
</tr>
</tbody>
</table>

SMJ is $\sim$ linear vs. BNLJ is quadratic…
But it’s all about the memory.
0. Intuition for smarter joins

1. Sort-Merge Join

2. HashPartition Joins
Pre-process data before JOINing

- Sort(Census), Sort(TaxInfo) on SSN
- Merge sorted pages

- Hash(Census), Hash(TaxInfo) on SSN
- Merge partitioned pages
Hash Join (HJ) or Hash Partition Join (HPJ)
Hash Join

- **Goal:** Execute $R \bowtie S$ on $A$
- **Key Idea:** We can partition $R$ and $S$ into buckets by hashing the join attribute-then just join the pairs of (small) matching buckets!
Hash Partition Join: High-level

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

1. Hash Partition: Split $R$, $S$ into $B$ buckets, using $h_B$ on $A$

2. Per-Partition Join: JOIN tuples in same partition (i.e, same hash value)

Note again that we are only considering equality constraints here.

We *decompose* the problem using $h_B$, then complete the join.
1. Hash Partition: Split R, S into B buckets, using $h_B$ on A
HPJ: High-level procedure

2. Per-Partition Join: JOIN tuples in same partitions
HPJ: High-level procedure

2. Per-Partition Join: JOIN tuples in same partition

Don’t have to join the others! E.g. (S_1 and R_2)!
HPJ Phase 1: Hash Partitioning

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

Given $B+1$ buffer pages, we partition into $B$ buckets:
- We use $B$ buffer pages for output (one for each bucket), and 1 for input
  - For each tuple $t$ in input, copy to buffer page for $h_B(t.A)$
  - When page fills up, flush to disk.
HPJ Phase 1: Partitioning

We partition into \( B = 2 \) 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!
1. We read pages from R into the “input” page of the buffer…

Given $B+1 = 3$ buffer pages
2. Then we use **hash function** $h_2$ to sort into the buckets, which each have one page in the buffer.

Given $B+1 = 3$ buffer pages.

$$h_2(0) = 0$$
HPJ Phase 1: Partitioning

2. Then we use hash function $h_2$ to sort into the buckets, which each have one page in the buffer.
3. We repeat until the buffer bucket pages are full…

Given \( B+1 = 3 \) buffer pages.
3. We repeat until the buffer bucket pages are full…

Given $B+1 = 3$ buffer pages
HPJ Phase 1: Partitioning

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

Given $B+1 = 3$ buffer pages
HPJ Phase 1: Partitioning

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

Given $B+1 = 3$ buffer pages
HPJ Phase 1: Partitioning

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

Given $B+1 = 3$ buffer pages
HPJ Phase 1: Partitioning

Note that collisions can occur!

Given $B+1 = 3$ buffer pages

Collision!!!

$$h_2(5) = h_2(3) = 1$$
HPJ Phase 1: Partitioning

Finish this pass…

Given $B+1 = 3$ buffer pages

Disk

Main Memory

<table>
<thead>
<tr>
<th>Buffer</th>
<th>Input page</th>
<th>Output (bucket) pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, b)</td>
<td>0</td>
<td>(5, a)</td>
</tr>
<tr>
<td>(0, a)</td>
<td>1</td>
<td>(0, j)</td>
</tr>
<tr>
<td>(0, j)</td>
<td></td>
<td>(0, j)</td>
</tr>
<tr>
<td>(3, a)</td>
<td></td>
<td>(5, a)</td>
</tr>
<tr>
<td>(3, j)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$h_2(0) = 0$
HPJ Phase 1: Partitioning

Finish this pass...

Given $B+1 = 3$ buffer pages
HPJ Phase 1: Partitioning

Finish this pass…

Given $B+1 = 3$ buffer pages

Main Memory

$$h_2(5) = h_2(3) = 1$$

Collision!!!
HPJ Phase 1: Partitioning

Finish this pass…

Given $B+1 = 3$ buffer pages

Disk

Main Memory

Buffer

Input page 0 1

Output (bucket) pages
Given $B+1 = 3$ buffer pages

We wanted buckets of size $B-1 = 1$... however we got larger ones due to:

1. Duplicate join keys
2. Hash collisions
Given $B + 1 = 3$ buffer pages

To take care of larger buckets caused by hash collisions, we can just do another pass!

What hash function should we use?

Do another pass with a different hash function, $h'_2$, ideally such that:

$$h'_2(3) \neq h'_2(5)$$
HPJ Phase 1: Partitioning

Given $B+1 = 3$ buffer pages.

To take care of larger buckets caused by (2) hash collisions, we can just do another pass!

What hash function should we use?

Do another pass with a different hash function, $h'_2$, ideally such that:

$h'_2(3) \neq h'_2(5)$
HPJ Phase 1: Partitioning

What about duplicate join keys? Unfortunately this is a problem… but usually not a huge one.

We call this unevenness in the bucket size **skew**.
Now that we have partitioned R and S...
HPJ Phase 2: Partition Join

Now, we just join pairs of buckets from R and S that have the same hash value to complete the join!
HPJ Summary

Given enough buffer pages...

- **Hash Partition** requires reading + writing each page of R,S
  - $\rightarrow 2(P(R)+P(S))$ IOs

- **Partition Join** (with BNLJ) requires reading each page of R,S
  - $\rightarrow P(R) + P(S)$ IOs

HJ takes $\sim 3(P(R)+P(S)) + OUT$ IOs!
SMJ vs HPJ Joins Summary

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

  $\sim 3(P(R) + P(S)) + OUT$

- Hash Joins are highly parallelizable

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

⇒ **Big takeaway**: IO-aware join algorithms
  - Massive difference vs brute-force
  - Nearly linear vs quadratic (or worse)