Lecture 14: Access Methods & Operators
Pace & Lecture content

• We are slowing down:
  • In response to feedback from some that they like this pace better!
  • Due to great questions! Makes us seriously happy!
  • The details are more fun! Makes one of us seriously happy.

• We may cut some topics listed (maybe not).
  • We have a lot of (we think) good material 😐 but...
  • We’d prefer depth and happiness to breadth.

• Please refresh lectures before (changes are minor)
  • Cannot tell you how much time we spend tweaking... it’s sad really...
Project #2 Hint

- You may want to do *Trigger activity* for project 2.
  - We’ve noticed those who do it have less trouble with project!

- Seems like we’re good here 😊 Exciting for us!

- Definitely use piazza actively: students have been giving great answers
  - Hats are back ordered! (well not really)
Today’s Lecture

1. B+ Trees

2. Nested Loop Joins
1. B+ Trees
What you will learn about in this section

1. B+ Trees: Basics

2. B+ Trees: Design & Cost

3. Clustered Indexes
B+ Trees

• Search trees
  • B does not mean binary!

• Idea in B Trees:
  • make 1 node = 1 physical page
  • Balanced, height adjusted tree (not the B either)

• Idea in B+ Trees:
  • Make leaves into a linked list (for range queries)
B+ Tree Basics

Parameter $d =$ the degree

Each non-leaf ("interior") node has $\geq d$ and $\leq 2d$ keys*

*except for root node, which can have between 1 and 2d keys
B+ Tree Basics

The $n$ keys in a node define $n+1$ ranges:

- $k < 10$
- $10 \leq k < 20$
- $20 \leq k < 30$
- $30 \leq k$
B+ Tree Basics

Non-leaf or *internal* node

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>20</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>22</th>
<th>25</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each range, in a *non-leaf* node, there is a pointer to another node with keys in that range.
B+ Tree Basics

Leaf nodes also have between $d$ and $2d$ keys, and are different in that:
B+ Tree Basics

Leaf nodes also have between \( d \) and \( 2d \) keys, and are different in that:

Their key slots contain pointers to data records.
Leaf nodes also have between $d$ and $2d$ keys, and are different in that:

Their key slots contain pointers to data records.

They contain a pointer to the next leaf node as well, for faster sequential traversal.
B+ Tree Basics

Note that the pointers at the leaf level will be to the actual data records (rows).

We might truncate these for simpler display (as before)...

Non-leaf or internal node

Leaf nodes
Some finer points of B+ Trees
Searching a B+ Tree

• For exact key values:
  • Start at the root
  • Proceed down, to the leaf

• For range queries:
  • As above
  • *Then sequential traversal*

```sql
SELECT name
FROM people
WHERE age = 25
```

```sql
SELECT name
FROM people
WHERE 20 <= age
  AND age <= 30
```
B+ Tree Exact Search Animation

30 < 80

30 in [20,60)

30 in [30,40)

To the data!

K = 30?
B+ Tree Range Search Animation

30 < 80
30 in [20,60)
30 in [30,40)
To the data!

K in [30,85]?

Not all nodes pictured
B+ Tree Design

• How large is $d$?

• Example:
  • Key size = 4 bytes
  • Pointer size = 8 bytes
  • Block size = 4096 bytes

• We want each node to fit on a single block/page
  • $2d \times 4 + (2d+1) \times 8 \leq 4096 \Rightarrow d \leq 170$

NB: Oracle allows 64K = $2^{16}$ byte blocks
$\Rightarrow d \leq 2730$
B+ Tree: High Fanout = Smaller & Lower IO

• As compared to e.g. binary search trees, B+ Trees have high fanout \((\text{between } d+1 \text{ and } 2d+1)\)

• This means that the depth of the tree is small \(\rightarrow\) getting to any element requires very few IO operations!
  • Also can often store most or all of the B+ Tree in main memory!

• A TiB = \(2^{40}\) Bytes. What is the height of a B+ Tree (with fill-factor = 1) that indexes it (with 64K pages)?
  • \((2*2730 + 1)^h = 2^{40} \rightarrow h = 4\)

The **fanout** is defined as the number of pointers to child nodes coming out of a node

*Note that fanout is dynamic*- we’ll often assume it’s constant just to come up with approximate eqns!*

The known universe contains \(~10^{80}\) particles... what is the height of a B+ Tree that indexes these?
B+ Trees in Practice

• Typical order: d=100. Typical fill-factor: 67%.
  • average fanout = 133

• Typical capacities:
  • Height 4: $133^4 = 312,900,700$ records
  • Height 3: $133^3 = 2,352,637$ records

• Top levels of tree sit in the buffer pool:
  • Level 1 = 1 page = 8 Kbytes
  • Level 2 = 133 pages = 1 Mbyte
  • Level 3 = 17,689 pages = 133 MBytes

**Fill-factor** is the percent of available slots in the B+ Tree that are filled; is usually < 1 to leave slack for (quicker) insertions

Typically, only pay for one IO!
Simple Cost Model for Search

- Let:
  - \( f = \text{fanout}, \) which is in \([d+1, 2d+1]\) (*we’ll assume it’s constant for our cost model…*)
  - \( N = \) the total number of pages we need to index
  - \( F = \) fill-factor (usually \( \approx 2/3 \))

- Our B+ Tree needs to have room to index \( N/F \) pages!
  - We have the fill factor in order to leave some open slots for faster insertions

- What height \( (h) \) does our B+ Tree need to be?
  - \( h=1 \) \( \rightarrow \) Just the root node- room to index \( f \) pages
  - \( h=2 \) \( \rightarrow \) \( f \) leaf nodes- room to index \( f^2 \) pages
  - \( h=3 \) \( \rightarrow \) \( f^2 \) leaf nodes- room to index \( f^3 \) pages
  - ...
  - \( h \rightarrow f^{h-1} \) leaf nodes- room to index \( f^h \) pages!

\[ \log_{\frac{N}{F}} \]

> We need a B+ Tree of height \( h = \left[ \log_{\frac{N}{F}} \right] ! \)
Simple Cost Model for Search

• Note that if we have $B$ available buffer pages, by the same logic:
  • We can store $L_B$ levels of the B+ Tree in memory
  • where $L_B$ is the number of levels such that the sum of all the levels’ nodes fit in the buffer:
    • $B \geq 1 + f + \cdots + f^{L_B-1} = \sum_{l=0}^{L_B-1} f^l$

• In summary: to do exact search:
  • We read in one page per level of the tree
  • However, levels that we can fit in buffer are free!
  • Finally we read in the actual record

IO Cost: \[ \left\lceil \log_f \frac{N}{F} \right\rceil - L_B + 1 \]

where $B \geq \sum_{l=0}^{L_B-1} f^l$
Simple Cost Model for Search

• To do range search, we just follow the horizontal pointers

• The IO cost is that of loading additional leaf nodes we need to access + the IO cost of loading each page of the results- we phrase this as “Cost(OUT)”

\[
\text{IO Cost: } \left\lfloor \log_f \frac{N}{F} \right\rfloor - L_B + \text{Cost(OUT)}
\]

where \( B \geq \sum_{l=0}^{L_B-1} f^l \)

**NOTE:** This has been tweaked slightly since presentation in lecture- read carefully!
Fast Insertions & Self-Balancing

• We won’t go into specifics of B+ Tree insertion algorithm, but has several attractive qualities:

  • ~ Same cost as exact search

  • **Self-balancing:** B+ Tree remains **balanced** (with respect to height) even after insert

B+ Trees also (relatively) fast for single insertions!
*However, can become bottleneck if many insertions (if fill-factor slack is used up...)*
Clustered Indexes

An index is *clustered* if the underlying data is ordered in the same way as the index’s data entries.
Clustered vs. Unclustered Index

Clustered Indexes

Unclustered Indexes
Clustered vs. Unclustered Index

• Recall that for a disk with block access, **sequential IO is much faster than random IO**

• For exact search, no difference between clustered / unclustered

• For range search over R values: difference between **1 random IO + R sequential IO**, and **R random IO**:
  • A random IO costs ~ 10ms (sequential much much faster)
  • For R = 100,000 records- **difference between ~10ms and ~17min!**
Summary [From Lecture 13 too...]

• We covered an algorithm + some optimizations for sorting larger-than-memory files efficiently
  • An IO aware algorithm!

• We create indexes over tables in order to support fast (exact and range) search and insertion over multiple search keys

• B+ Trees are one index data structure which support very fast exact and range search & insertion via high fanout
  • Clustered vs. unclustered makes a big difference for range queries too
2. 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

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

\[
\begin{align*}
R \, \bowtie \, S & \\
\text{SELECT} & \quad R.A,B,C,D \\
\text{FROM} & \quad R, S \\
\text{WHERE} & \quad R.A = S.A
\end{align*}
\]
Joins: Example

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

\[
R \bowtie S
\]

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

\[ R \bowtie S \]

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

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

\[ R \bowtie S \]

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

\[
\begin{align*}
R & \quad S \\
A & \quad A \\
B & \quad D \\
C & \\
1 & 3 & 7 \\
2 & 2 \\
2 & 3 \\
3 & 1 & 1 \\
\end{align*}
\]

\[
\begin{array}{cccc}
A & B & C & D \\
2 & 3 & 4 & 2 \\
2 & 3 & 4 & 3 \\
2 & 5 & 2 & 2 \\
2 & 5 & 2 & 3 \\
\end{array}
\]
Joins: Example

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

\[
\begin{align*}
\text{SELECT} & \quad R.A,B,C,D \\
\text{FROM} & \quad R, S \\
\text{WHERE} & \quad R.A = S.A
\end{align*}
\]
Semantically: A Subset of the Cross Product

\[ R \Join S \]

**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?**
Notes

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

• We write $\mathbf{R} \bowtie \mathbf{S}$ 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”)

However joins can merge > 2 tables, and some algorithms do support non-equality constraints!
Nested Loop Joins
Notes

• We are again considering “IO aware” algorithms: 
  * care about disk IO

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

• Note also that we omit ceilings in calculations...
  good exercise to put back in!

Recall that we read / write entire pages with disk IO
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)

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

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

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 $A$:

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

Cost:

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

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

Cost:

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

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 $A$:

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

Cost:

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

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 $P(R) \cdot P(S)$... but usually not that bad
Nested Loop Join (NLJ)

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

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

Cost:

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

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

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

Outer vs. inner selection makes a huge difference - DBMS needs to know which relation is smaller!
IO-Aware Approach
Block Nested Loop Join (BNLJ)

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:

$P(R)$

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!
Block Nested Loop Join (BNLJ)

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:

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

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 $S$

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

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

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
Block Nested Loop Join (BNLJ)

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

Again, \( \text{OUT} \) could be bigger than \( P(R) \cdot P(S) \)... but usually not that bad
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

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

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 + \(50,000*1000\) = 50 Million IOs \(\sim 140\) hours

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

A very real difference from a small change in the algorithm!
Smarter than Cross-Products
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:
    - **NLJ**: \( P(R) + T(R)P(S) + \text{OUT} \)
    - **BNLJ**: \( P(R) + \frac{P(R)}{B-1} P(S) + \text{OUT} \)

- Now we’ll see some (nearly) linear joins:
  - \( \sim O(P(R) + P(S) + \text{OUT}), \) where again \( \text{OUT} \) could be quadratic but is usually better

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:

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

where $L$ is the IO cost to access all the distinct values in the index; assuming these fit on one page, $L \sim 3$ is a good estimate.

→ We can use an index (e.g. B+ Tree) to avoid doing the full cross-product!
Lecture 15: Joins- A Cage Match
Announcements

1. There was a bug on one sub problem of the midterm
   1. Precedence order for NOT (see Piazza!)
   2. We gave everyone 5 bonus points: *this disadvantages no one!*
   3. cookies!

2. PS #3 & Project 3 out soon! The last of each! 😊
   1. We want to give you time for Project 3 (it’s more learn by doing)
   2. And we want to grade PS#3 to give feedback (exam material)
      1. I’m worried you won’t have enough time for relational algebra, which I consider important... so we built some new materials for it (next topic).
Today’s Lecture

1. Sort-Merge Join (SMJ)

2. Hash Join (HJ)

3. The Cage Match: SMJ vs. HJ
1. Sort-Merge Join (SMJ)
What you will learn about in this section

1. Sort-Merge Join
2. “Backup” & Total Cost
3. Optimizations
4. ACTIVITY: Sequential Flooding
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 subsection]*

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 \( A \)

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

1. Sort the relations $R$, $S$ on the join key (first value)
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!
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. 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 tuple we’ve already read!
Backup

• At best, no backup $\Rightarrow$ 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 have to 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 “zig-zag” (see animation)
SMJ: Total cost

• Cost of SMJ is **cost of sorting** R and S...

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

• Plus the **cost of writing out**: \(~P(R)+P(S)\) but in worst case \(T(R)\times T(S)\)

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

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

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

- If we have 100 buffer pages, \( P(R) = 1000 \) pages and \( P(S) = 500 \) pages:
  - Sort both in two passes: \( 2 \times 2 \times 1000 + 2 \times 2 \times 500 = 6,000 \text{ IOs} \)
  - Merge phase \( 1000 + 500 = 1,500 \text{ IOs} \)
  - \( = 7,500 \text{ IOs} + \text{OUT} \)

What is BNLJ?

- \( 500 + 1000 \times \left\lfloor \frac{500}{98} \right\rfloor = 6,550 \text{ IOs} + \text{OUT} \)

- But, if we have 35 buffer pages?
  - Sort Merge has same behavior (still 2 passes)
  - BNLJ? \( 15,500 \text{ IOs} + \text{OUT}! \)

SMJ is \( \sim \) linear vs. BNLJ is quadratic...
A Simple Optimization: Merges Merged!

• SMJ is composed of a **sort phase** and a **merge phase**

• During the **sort phase**, run passes of external merge sort on R and S
  • Suppose at some point, R and S have \( \leq B \) (sorted) runs in total

• We could do two merges (for each of R & S) at this point, complete the sort phase, and start the merge phase...

• OR, we could combine them: do one B-way merge and complete the join!
Un-Optimized SMJ

Sort Phase
(Ext. Merge Sort)

Sort Phase
(Ext. Merge Sort)

Merge / Join Phase

Given $B+1$ buffer pages

Unsorted input relations

Merge

Merge

Merge

Split & sort

Split & sort

Joined output file created!
Simple SMJ Optimization

Sort Phase
(Ext. Merge Sort)

Unsorted input relations

Merge / Join Phase

Given $B+1$ buffer pages

Lecture 15 > Section 1 > Backup
Simple SMJ Optimization

• Now, on this last pass, we only do \( P(R) + P(S) \) IOs to complete the join!

• If we can initially split \( R \) and \( S \) into \( B \) total runs each of length approx. \( \leq 2(B+1) \), assuming repacking lets us create initial runs of \( \sim 2(B+1) \)- then we only need \( 3(P(R) + P(S)) + OUT \) for SMJ!
  • 2 R/W per page to sort runs in memory, 1 R per page to B-way merge / join!

• How much memory for this to happen?
  • \( \frac{P(R)+P(S)}{B} \leq 2(B + 1) \Rightarrow \sim P(R) + P(S) \leq 2B^2 
  • Thus, \( \max\{P(R), P(S)\} \leq B^2 \) is an approximate sufficient condition

Given \( B+1 \) buffer pages

If the larger of \( R,S \) has \( \leq B^2 \) pages, then SMJ costs \( 3(P(R)+P(S)) + OUT \)!
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
  • If \( \max \{ P(R), P(S) \} < B^2 \) then cost is \( 3(P(R)+P(S)) + \text{OUT} \)
Activity-15.ipynb
4. Hash Join (HJ)
What you will learn about in this section

1. Hash Join

2. Memory requirements
Recall: Hashing

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

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

- We hash on an attribute $A$, so our has function is $h_B(t)$ has the form $h_B(t.A)$.
  - **Collisions** may be more frequent.
Recall: Mad Hash Collisions

Say something here to justify this slide’s existence? [TODO]
Hash Join: High-level procedure

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

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

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

   1. Use BNLJ here; or hash again $\Rightarrow$ either way, operating on small partitions so fast!

We *decompose* the problem using $h_B$, then complete the join

Note again that we are only considering equality constraints here
Hash Join: High-level procedure

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

More detail in a second...

Note our new convention: pages each have two tuples (one per row)
Hash Join: High-level procedure

2. **Matching Phase:** Take pairs of buckets whose tuples have the same values for $h_B$, and join these
Hash Join: High-level procedure

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

Don’t have to join the others! E.g. these!
Hash Join Phase 1: 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
  - The “dual” of sorting.
  - For each tuple \( t \) in input, copy to buffer page for \( h_B(t.A) \)
  - When page fills up, flush to disk.
How big are the resulting buckets?

• Given **N input pages, we partition into B buckets:**
  • → Ideally our buckets are each of size ~ \( \frac{N}{B} \) pages

• What happens if there are **hash collisions**?
  • Buckets could be > \( \frac{N}{B} \)
  • We’ll do several passes...

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

• Ideally, our buckets would be of size $\leq B - 1$ pages
  • 1 for input page, 1 for output page, $B-1$ for each bucket

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

• We can keep partitioning buckets that are $> B-1$ pages, until they are $\leq B - 1$ pages
  • Using a new hash key which will split them...

Given $B+1$ buffer pages

Recall for BNLJ:

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

We’ll call each of these a “pass” again...
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).

Given $B + 1 = 3$ buffer pages.

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 $B = 2$ buckets of size $\leq B - 1 \Rightarrow 1$ page each.
Hash Join Phase 1: Partitioning

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

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

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

Hash Join Phase 1: Partitioning

2. Then we use **hash function** $h_2$ to sort into the buckets, which each have one page in the buffer
Hash Join Phase 1: Partitioning

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
Hash Join Phase 1: Partitioning

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

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

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

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

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

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

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

Given $B + 1 = 3$ buffer pages
3. We repeat until the buffer bucket pages are full... then flush to disk
Hash Join Phase 1: Partitioning

Note that collisions can occur!

Given $B+1 = 3$ buffer pages

Collision!!!
Hash Join Phase 1: Partitioning

Finish this pass...

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

Given $B+1 = 3$ buffer pages

Finish this pass...

Disk

- R
  - (5, b)
- B0
  - (0, a)
  - (0, j)
- B1
  - (3, a)
  - (3, j)

Main Memory

Buffer

Input page

Output (bucket) pages

0

1

(0, j)

(5, a)
Hash Join Phase 1: Partitioning

Finish this pass...

Given $B+1 = 3$ buffer pages

Collision!!!

![Diagram showing partitioning and collision in hash join phase 1](image-url)
Hash Join Phase 1: Partitioning

Finish this pass...

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

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 (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:

$$h'_2(3) \neq h'_2(5)$$
Hash Join 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:

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

Given $B+1 = 3$ buffer pages

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 $S$...
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 \( \sim B - 1 \) pages, can join each such pair using BNLJ in linear time; recall (with \( P(R) = B-1 \)):

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

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

R \bowtie S on A
Hash Join Phase 2: Matching

To perform the join, we ideally just need to explore the dark blue regions

= the tuples with same values of the join key A
Hash Join Phase 2: Matching

With a join algorithm like BNLJ that doesn’t take advantage of equijoin structure, we’d have to explore this **whole grid!**
### Hash Join Phase 2: Matching

<table>
<thead>
<tr>
<th>R.A hashed values</th>
<th>S.A hashed values</th>
</tr>
</thead>
<tbody>
<tr>
<td>h(1) = 0</td>
<td>h(1) = 0</td>
</tr>
<tr>
<td>h(1) = 0</td>
<td>h(1) = 0</td>
</tr>
<tr>
<td>h(2) = 0</td>
<td>h(2) = 0</td>
</tr>
<tr>
<td>h(2) = 0</td>
<td>h(2) = 0</td>
</tr>
<tr>
<td>h(3) = 1</td>
<td>h(3) = 1</td>
</tr>
<tr>
<td>h(3) = 1</td>
<td>h(3) = 1</td>
</tr>
<tr>
<td>h(4) = 1</td>
<td>h(4) = 1</td>
</tr>
<tr>
<td>h(4) = 1</td>
<td>h(4) = 1</td>
</tr>
<tr>
<td>h(5) = 2</td>
<td>h(5) = 2</td>
</tr>
<tr>
<td>h(5) = 2</td>
<td>h(5) = 2</td>
</tr>
<tr>
<td>h(6) = 2</td>
<td>h(6) = 2</td>
</tr>
<tr>
<td>h(6) = 2</td>
<td>h(6) = 2</td>
</tr>
</tbody>
</table>

R \Join S on A

With HJ, we only explore the **blue** regions

= the tuples with same values of \( h(A) \)!

We can apply BNLJ to each of these regions
Hash Join Phase 2: Matching

R \bowtie S on A

An alternative to applying BNLJ:

We could also hash again, and keep doing passes in memory to reduce further!
How much memory do we need for HJ?

- Given B+1 buffer pages
- Suppose (reasonably) that we can partition into B buckets in 2 passes:
  - For R, we get B buckets of size \( \sim \frac{P(R)}{B} \)
  - To join these buckets in linear time, we need these buckets to fit in B-1 pages, so we have:

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

+ WLOG: Assume \( P(R) \leq P(S) \)

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

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

  • **Partitioning** requires reading + writing each page of R, S
    • \( \rightarrow 2(P(R)+P(S)) \) IOs

  • **Matching** (with BNLJ) requires reading each page of R, S
    • \( \rightarrow P(R) + P(S) \) IOs

  • **Writing out results** could be as bad as \( P(R)*P(S) \)... but probably closer to \( P(R)+P(S) \)

  HJ takes \(~3(P(R)+P(S)) + OUT\) IOs!
3. The Cage Match
Sort-Merge v. Hash Join

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

\[ \sim 3(P(R) + P(S)) + OUT \]

• “Enough” memory =

• SMJ: \( B^2 > \max\{P(R), P(S)\} \)

• HJ: \( B^2 > \min\{P(R), P(S)\} \)

Hash Join superior if relation sizes *differ greatly*. Why?
Further Comparisons of Hash and 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