Recap column stores & paper

**Join Processing:**

S = \{S\} tuples, ISI pages
R = \{R\} tuples, IRI pages
IRI < ISI
M pages of memory

**Types of joins**

**Nested Loops**

```
for s in S
    for r in R
        if pred(s,r) output s join r
```

Does it matter which is inner, outer? (yes)
\{S\} * \{R\} compares in either case

suppose ISI = 4, IRI = 2, M=3, LRU

S inner = 2 + 4 + 4 = 10 pages read
R inner = 4 + 2 = 6 pages read

w/out cache: ISI + \{S\}*IRI  i/o s

**Index Nested Loops (Index on R)**

```
for s in S
    find matches in R
```

ISI + \{S\} i/o
**Block Nested Loops**

\[
\begin{align*}
B &= \text{block size} < M \\
\text{while (not at end of S)} \\
&\quad S' = \text{read } B \text{ records from } S \\
&\quad \text{for } r \text{ in } R \\
&\quad \quad \text{for } s \text{ in } S' \\
&\quad \quad \quad \text{if } \text{pred}(s, r) \text{ output } s \text{ join } r
\end{align*}
\]

**(Sort)Merge (IRI + ISI < M -- in memory)**

1< 2<

Sort R, Sort S

1 2 3

merge

5 7

7 7

IRI + ISI i/os

7 9

9

Show example dealing with duplicates

**Hash (simple, IRI < M -- in memory)**

Build hash on R, probe with S

IRI + ISI i/os

**Blocked hash (IRI > M)**

Repeat until read all of R:

Read in M pages of R, probe with S

IRI + ISI * ceil (IRI / M) i/os

**Pipelined Hash**

As tuples of R arrive, add to hash on R, probe into hash on S

As tuples of S arrive, add to hash on S, probe into hash on R

Study break

**Shapiro: (Gossip about paper)**

What's this paper about?

(Join algorithms for two relations when size of either relation exceeds available RAM)

Equality joins only

What's the big takeaway?

(hash join outperforms sort-merge join)

Always?

(at least, if you have to sort the relations)
Why is this only for "large" memories?
(Requires memory equal to \sqrt{|S|}, where S is the larger relation)
\[ M \geq \sqrt{|S|} \geq \sqrt{|R|} \]

How do these external algorithms work?

2 phases
- Phase 1: partition the relation into (sorted/hashed) runs
- Phase 2: join the partitions

Sort-merge:
\[ \text{phase 1} \]
repeat until done:
- read a run of S
- sort
- write out
repeat until done:
- read a run of R
- sort
- write out

\[ \text{phase 2} \]

begin reading from each run of R and S (requires one block of memory for each run)
join R and S as they appear

Example:
R = 1, 4, 3, 6, 9, 14, 1, 7, 11
S = 2, 3, 7, 12, 9, 8, 4, 15, 6

R1 = 1, 4, 3 R2 = 6, 9, 14 R3 = 1, 7, 11, etc

run size = 3

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1&lt;</td>
<td>6&lt;</td>
<td>1&lt;</td>
<td>2&lt;</td>
<td>8&lt;</td>
<td>4&lt;</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>7</td>
<td>3</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>11</td>
<td>7</td>
<td>12</td>
<td>15</td>
</tr>
<tr>
<td>1</td>
<td>6&lt;</td>
<td>1</td>
<td>2&lt;</td>
<td>8&lt;</td>
<td>4&lt;</td>
</tr>
<tr>
<td>3&lt;</td>
<td>9</td>
<td>7&lt;</td>
<td>3</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>11</td>
<td>7</td>
<td>12</td>
<td>15</td>
</tr>
<tr>
<td>1</td>
<td>6&lt;</td>
<td>1</td>
<td>2</td>
<td>8&lt;</td>
<td>4&lt;</td>
</tr>
<tr>
<td>3&lt;</td>
<td>9</td>
<td>7&lt;</td>
<td>3&lt;</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>11</td>
<td>7</td>
<td>12</td>
<td>15</td>
</tr>
</tbody>
</table>
How do I pick the run size?  
(Make it as big as possible, to minimize the number of runs. Runs can be at most \( M \geq \sqrt{|S|} \) to sort in memory.)

Claim: Can't be more than \( M \) runs. Why?

Suppose we set the length of a run to \( \sqrt{|I|} \)

Then, there will be \( |I|/\sqrt{|I|} = \sqrt{|I|} \) runs, which is good b/c \( M > \sqrt{|I|} \), and need one one page of memory for each run to do merge concurrently

Paper confusingly claims if you have \( M=\sqrt{|I|} \) memory, runs will be size \( 2 \times \sqrt{|I|} \)

(Where does 2 come from. Using "selection replacement tree" -- idea is that you store values in a heap, read in a new value whenever you output an old value. Average run length will be \( 2 \times \) size of memory). See Knuth.
\[ M = 3 \]
\[
\begin{array}{c}
23 \\
23 45 \\
23 45 64 \\
\end{array} \\
\]

\[ \text{Run} = 23 \]
\[
\begin{array}{c}
12 \\
12 45 \\
12 45 64 \\
\end{array} \\
\]

\[ \text{Run} = 23, 45 \]
\[
\begin{array}{c}
12 \\
12 19 \\
12 19 64 \\
\end{array} \\
\]

\[ \text{Run} = 23, 45, 64 \]
\[
\begin{array}{c}
12 \\
12 19 \\
12 19 82 \\
\end{array} \\
\]

\[ \text{Run} = 23, 45, 64, 82 \]
\[
\begin{array}{c}
12 \\
12 19 \\
12 19 97 \\
\end{array} \\
\]

\[ \text{Run1} = 23, 45, 64, 82, 97 \]
\[
\begin{array}{c}
12 \\
12 19 \\
12 19 44 \\
\end{array} \\
\]

\[ \text{Run2} = 12, 19, 44 \]

#I/OS:

- Read R+S once (seq)
- Write R+S once (seq)
- Read R+S once (random -- can do better if we read long runs instead of 1 page at a time, but that requires memory! Might be better to hierarchically merge sequential
Simple hash:

\[ i = 0; \]
\[ \text{pass size = } v \text{ (e.g., } v = 1) \quad \text{// if } P \text{ partitions, hash into } [1...n], \text{ e.g., } h(x) = x \mod P \]
for partition \( i \) (on hash values in range range \( [v^i , v^{(i+1)}) \) )

- scan \( S \), hash, if in partition, insert into hash table
- o.w., write back out

- scan \( R \), hash, if in partition, lookup in hash table, output matches
- o.w., write back out

repeat with reduced \( R \) and \( S \), in round \( i+1 \) ; example:

\( R = 1, 4, 3, 6, 9, 14, 1, 7, 11 \)
\( S = 2, 3, 7, 12, 9, 8, 4, 15, 6 \)
\( h(x) = x \mod 3, \text{ pass size = } 1 \)

Pass 1: \( h(x) \) in range \( [0..1) \)
\( R \) hash table: 3 6 9
remainder: 1 4 14 1 7 11
\( S \) probe with : 3 12 9 15 6 --> 3 6 9 join
remainder: 2 7 8 4

Pass 2: \( h(x) \) in range \( [1..2) \)
\( R \) hash table: 1 4 1 7
remainder: 14 11
\( S \) probe with : 7 4 --> 7 4 join
remainder: 2 8

Pass 2: \( h(x) \) in range \( [1..2) \)
\( R \) hash table: 14 11
\( S \) probe with : 2 8 --> no join

Somewhat complex to analyze:

Read \( R, S \) (seq)

Amount we write depends on number of passes. In pass 1, we write:

\((p-1)/p)R, (p-1)/p)S \text{ (seq)}

We then read all this data back in (seq), and in pass 2, we write:

\((p-2)/p)R, (p-2)/p)S \text{ (seq)}
And so on...

So for 2 passes, we get:

Read $|R+S|$, Write $(1/2)(|R|+|S|)$, Read $(1/2)(|R|+|S|)$ and are done.
Total IO is $2(|R|+|S|)$

For 3 passes, total IO is $3(|R|+|S|)$
For $n$ passes, total IO is $n(|R|+|S|)$

Is this better than blocked hash?
(Depeps on relative size of $|R|$ and $|S|$ -- if $|S|$ is much smaller than $R$, blocked has will be better since it doesn't rewrite $R$)

**Grace hash:**

choose $P$ partitions, with one page per partition
hash $r$ into partitions, flushing pages as they fill
hash $s$ into partitions, flushing pages as they fill
for each partition $p$
  build a hash table $T$ on $r$ tuples in $p$
  lookup each $s$ in $T$ outputting matches

example:

$R = 1, 4, 3, 6, 9, 14, 1, 7, 11$
$S = 2, 3, 7, 12, 9, 8, 4, 15, 6$
$h(x) = x \text{ mod } 3$

$R_0 = 3 \ 6 \ 9$
$R_1 = 1 \ 4 \ 1 \ 7$
$R_2 = 14 \ 11$

$S_0 = 3 \ 12 \ 9 \ 15 \ 6$
$S_1 = 7 \ 4$
$S_2 = 2 \ 8$

Now, join $R_0$ with $S_0$, $R_1$ with $S_1$, $R_2$ with $S_2$

*Because we are using the same hash function for $R$ and $S$ we can guarantee that the only tuples that will join with partition $R_i$ are those in $S_i$*

How do I pick the partition size?

(Assume uniform distribution of tuples to partitions, make each partition equal to $M$ pages (minus a couple for active pages of $S$ being read.)
\( \sqrt{|R|} < M \)

Partition size \( = M > \sqrt{|R|} \)

\#parts \( P = \frac{|R|}{M} \leq \frac{|R|}{\sqrt{|R|}} = \sqrt{|R|} \)

\( h(v) \rightarrow [1,k] \)

Each covers \( k/P \) hash values

Need \( \sqrt{|R|} \) pages of memory b/c we need at least one page per partition as we write out (note that simple hash doesn't have this requirement)

I/O:

Read \( R+S \) (seq)

Write \( R+S \) (semi-random)

Read \( R+S \) (seq)

Also \( 3(|R|+|S|) \) I/OS

What's hard about this?

Possible that some partitions will overflow -- e.g., if many duplicate values

What do they say we should do?

(Leave some more slop (assign fewer values to each partition) by assuming that each record takes a few more bytes to store in hash table.)

Split partitions that overflow

When does grace outperform simple?

(When there are many partitions, since we avoid the cost of re-reading tuples from disk in building partitions)

When does simple outperform grace?

(When there are few partitions, since grace re-reads hash tables from disk)

So what does Hybrid do?

\( M = \sqrt{|R|} + E \)

Make first partition of size \( E \), do it on the fly.

Do remaining partitions as in grace.
Why does grace/hybrid outperform sort-merge?

CPU Costs!
I/O costs are comparable
690 / 1000 seconds in sort merge are due to the costs of sorting
17.4 in the case of CPU for grace/hybrid!

Will this still be true today?
(Yes)