Query Execution

- Here only:
  - how to implement operators
  - what are the costs of implementations
  - how to implement queries
    - Data flow between operators

- Next part:
  - How to choose good plan

Execution Plan

- A tree (DAG) of physical operators that implement a query
- May use indices
- May create temporary relations
- May create indices on the fly
- May use auxiliary operations such as sorting

How to estimate costs

- If everything fits into memory
  - Standard computational complexity
- If not
  - Assume fixed memory available for buffering pages
  - Count I/O operations
  - Real systems combine this with CPU estimations

Estimating IOs:

- Count # of disk blocks that must be read (or written) to execute query plan
To estimate costs, we may have additional parameters:

- $B(R)$ = # of blocks containing $R$ tuples
- $f(R)$ = max # of tuples of $R$ per block
- $M$ = # memory blocks available

Clustered index

Index that allows tuples to be read in an order that corresponds to physical order

Operators Overview

- (External) Sorting
- Joins (Nested Loop, Merge, Hash, ...)
- Aggregation (Sorting, Hash)
- Selection, Projection (Index, Scan)
- Union, Set Difference
- Intersection
- Duplicate Elimination

Operator Profiles

- Algorithm
- In-memory complexity: e.g., $O(n^2)$
- Memory requirements
  - Runtime based on available memory
  - #I/O if operation needs to go to disk
- Disk space needed
- Prerequisites
  - Conditions under which the operator can be applied

Execution Strategies

- Compiled
  - Translate into C/C++/Assembler code
  - Compile, link, and execute code
- Interpreted
  - Generic operator implementations
  - Generic executor
    - Interprets query plan
Virtual Machine Approach

- Implement virtual machine of low-level DBMS operations
- Compile query into machine-code for that machine

Iterator Model

- Need to be able to combine operators in different ways
  - E.g., join inputs may be scans, or outputs of other joins, ...
  - -> define generic interface for operators
  - be able to arbitrarily compose complex plans from a small set of operators

Iterator Model - Interface

- **Open**
  - Prepare operator to read inputs
- **Close**
  - Close operator and clean up
- **Next**
  - Return next result tuple

Query Execution – Iterator Model

[Diagram showing the iterator model with open, close, and next operations]
Parallelism
- Iterator Model
  - **Pull-based** query execution
- Potential types of parallelism
  - Inter-query (every multiuser system)
  - Intra-operator
  - Inter-operator

Intra-Operator Parallelism
- Execute portions of an operator in parallel
  - Merge-Sort
    - Assign a processor to each merge phase
  - Scan
    - Partition tables
    - Each process scans one partition

Inter-Operator Parallelism
- Each process executes one or more operators
- **Pipelining**
  - **Push-based** query execution
  - Chain operators to directly produce results
  - Pipeline-breakers
    - Operators that need to consume the whole input (or large parts) before producing outputs

Pipelining Communication
- **Queues**
  - Operators push their results to queues
  - Operators read their inputs from queues
- **Direct call**
  - Operator calls its parent in the tree with results
  - Within one process

Pipeline-breakers
- **Sorting**
  - All operators that apply sorting
- **Aggregation**
- **Set Difference**
- **Some implementations of**
  - Join
  - Union

Key
- Append to queue
- Dequeue
- Direct Call

Pipelines
Operators Overview

• (External) Sorting
• Joins (Nested Loop, Merge, Hash, ...)
• Aggregation (Sorting, Hash)
• Selection, Projection (Index, Scan)
• Union, Set Difference
• Intersection
• Duplicate Elimination

Sorting

• Why do we want/need to sort
  – Query requires sorting (ORDER BY)
  – Operators require sorted input
    • Merge-join
    • Aggregation by sorting
    • Duplicate removal using sorting

In-memory sorting

• Algorithms from data structures 101
  – Quick sort
  – Merge sort
  – Heap sort
  – Intro sort
  – ...

External sorting

• Problem:
  – Sort $N$ pages of data with $M$ pages of memory

• Solutions?

First Idea

• Split data into runs of size $M$
• Sort each run in memory and write back to disk
  – $\lceil N/M \rceil$ sorted runs of size $M$
• Now what?

Merging Runs

• Need to create bigger sorted runs out of sorted smaller runs
  – Divide and Conquer
  – Merge Sort?
• How to merge two runs that are bigger than $M$?
Merging Runs using 3 pages

- Merging sorted runs $R_1$ and $R_2$
- Need 3 pages
  - One page to buffer pages from $R_1$
  - One page to buffer pages from $R_2$
  - One page to buffer the result
- Whenever this buffer is full, write it to disk

2-Way External Mergesort

- Repeat process until we have one sorted run
- Each iteration (pass) reads and writes the whole table once: $2B(R)$ I/Os
- Each pass doubles the run size
  - $1 + \lceil \log_2 (B(R) / M) \rceil$ runs
  - $2B(R) * (1 + \lceil \log_2 (B(R) / M) \rceil)$ I/Os

N-Way External Mergesort

- How to utilize $M$ buffer during merging?
- Each pass merges $M-1$ runs at once
  - One memory page as buffer for each run
- $\#I/Os$
  - $1 + \lceil \log_{M-1} (B(R) / M) \rceil$ runs
  - $2B(R) * (1 + \lceil \log_{M-1} (B(R) / M) \rceil)$ I/Os
How many passes do we need?

<table>
<thead>
<tr>
<th>N</th>
<th>M=17</th>
<th>M=129</th>
<th>M=257</th>
<th>M=513</th>
<th>M=1025</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1,000</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>10,000</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>100,000</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1,000,000</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>10,000,000</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>100,000,000</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>8</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

To put into perspective

• Scenario
  – Page size 4KB
  – 1TB of data (250,000,000)
  – 10MB of buffer for sorting (250)

• Passes
  – 4 passes

Merge

• In practice would want larger I/O buffer for each run
• Trade-off between number of runs and efficiency of I/O

Improving in-memory merging

• Merging \( M \) runs
  – To choose next element to output
  – Have to compare \( M \) elements
  ---> complexity linear in \( M \): \( O(M) \)
• How to improve that?
  – Use priority queue to store current element from each run
  ---> \( O(\log_2(M)) \)

Priority Queue

• Queue for accessing elements in some given order
  – \texttt{pop-smallest} = return and remove smallest element in set
  – \texttt{Insert(e)} = insert element into queue

Min-Heap

• Implementation of priority queue
  – Store elements in a binary tree
  – All levels are full (except leaf level)
  – Heap property
    • Parent is smaller than child
• Example: \( \{1, 4, 7, 10\} \)
Min-Heap Insertion

- **insert(e)**
  1. Add element at next free leaf node
  2. If node smaller than parent then
  3. Repeat until 2) cannot be applied anymore

Min-Heap Dequeue

- **pop-smallest**
  1. Return Root and use right-most leaf as new root
  2. If node smaller than child then
  3. Repeat until 2) cannot be applied anymore

Insert

- Insert 3
  1. Insert at first free position
  2. Restore heap property

Dequeue

- Dequeue

Min/Max-Heap Complexity

- Heap is a complete tree
  - Height is $O(\log(n))$
- Insertion
  - Maximal height of the tree switches
    - $O(\log(n))$
- Dequeue
  - Maximal height of the tree switches
    - $O(\log(n))$
Min-Heap Implementation

- Full tree
  - Use array to implement tree
- Compute positions
  - Parent(n) = ⌊(n-1)/2⌋
  - Children(n) = 2n + 1, 2n + 2

Using a heap to generate runs

- Read inputs into heap
  - Until available memory is full
- Replace elements
  - Remove smallest element from heap
    - If larger then last element written of current run then write to current run
    - Else create a new run
  - Add new element from input to heap

Merging with Priority Queue

- Inputs:
  1. 8
  2. 9
  3. 7
  4. 10
  5. 12
  6. 13
  7. 6
  8. 11

Using a heap to generate runs

- Inputs:
  1. 5
  2. 7
  3. 2
  4. 3
  5. 12
  6. 15
  7. 1

Using a heap to generate runs
Using a heap to generate runs

- Increases the run-length
  - On average by a factor of 2 (see Knuth)

Use clustered B+-tree

- Keys in the B+-tree \( I \) are in sort order
  - If B+-tree is clustered traversing the leaf nodes is sequential I/O!
  - \( K = \#\text{keys/leaf node} \)

- Approach
  - Traverse from root to first leaf: \( \text{HT}(I) \)
  - Follow sibling pointers: \( |R| / K \)
  - Read data blocks: \( B(R) \)

I/O Operations

- \( \text{HT}(I) + |R| / K + B(R) \) I/Os
- Less than \( 2B(R) = 1 \) pass external mergesort
- \( ->\)Better than external merge-sort!

Unclustered B+-tree?

- Each entry in a leaf node may point to different page of relation \( R \)
  - For each leaf page we may read up to \( K \) pages from relation \( R \)
  - Random I/O
- In worst-case we have
  - \( K \cdot B(R) \)
  - \( K = 500 \)
  - \( 500 \cdot B(R) = 250 \) merge passes

Sorting Comparison

<table>
<thead>
<tr>
<th>Property</th>
<th>Ext. Mergesort</th>
<th>( B_s ) (clustered)</th>
<th>( B_s ) (unclustered)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime</td>
<td>( O(N \log_{M}(N)) )</td>
<td>( O(N) )</td>
<td>( O(N) )</td>
</tr>
<tr>
<td>#I/O (random)</td>
<td>( 2B(R) + \left(1 + \log_{M}(B(R) / M) \right) B(R) )</td>
<td>( \text{HT} +</td>
<td>R</td>
</tr>
<tr>
<td>Memory</td>
<td>( M )</td>
<td>( 1 ) (better HT + X)</td>
<td>( 1 ) (better HT + X)</td>
</tr>
<tr>
<td>Disk Space</td>
<td>( B(R) )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Variants</td>
<td>Merge with heap</td>
<td>1) Merge with heap</td>
<td>1) Merge with heap</td>
</tr>
<tr>
<td></td>
<td>Run generation with heap</td>
<td>2) Run generation with heap</td>
<td>2) Run generation with heap</td>
</tr>
<tr>
<td></td>
<td>Larger Buffer</td>
<td>3) Larger Buffer</td>
<td>3) Larger Buffer</td>
</tr>
</tbody>
</table>

Operators Overview

- (External) Sorting
- Joins (Nested Loop, Merge, Hash, ...)
- Aggregation (Sorting, Hash)
- Selection, Projection (Index, Scan)
- Union, Set Difference
- Intersection
- Duplicate Elimination
Scan

- Implements access to a table
  - Combined with selection
  - Probably projection too
- Variants
  - Sequential
    - Scan through all tuples of relation
  - Index
    - Use index to find tuples that match selection

Operators Overview

- (External) Sorting
- Joins (Nested Loop, Merge, Hash, ...)
- Aggregation (Sorting, Hash)
- Selection, Projection (Index, Scan)
- Union, Set Difference
- Intersection
- Duplicate Elimination

Options

- Transformations: \( R_1 \bowtie_C R_2, \ R_2 \bowtie_C R_1 \)
- Joint algorithms:
  - Nested loop
  - Merge join
  - Join with index
  - Hash join
- Outer join algorithms

Nested Loop Join (conceptually)

for each \( r \in R_1 \) do
  for each \( s \in R_2 \) do
    if \((r,s) \vdash C\) then output \((r,s)\)

Applicable to:
- Any join condition \( C \)
- Cross-product

Merge Join (conceptually)

1. if \( R_1 \) and \( R_2 \) not sorted, sort them
2. \( i \leftarrow 1; j \leftarrow 1; \)
   While \((i \leq T(R_1)) \land (j \leq T(R_2))\) do
     if \( R_1\{i\}.C = R_2\{j\}.C \) then outputTuples
     else if \( R_1\{i\}.C > R_2\{j\}.C \) then \( j \leftarrow j+1 \)
     else if \( R_1\{i\}.C < R_2\{j\}.C \) then \( i \leftarrow i+1 \)

Applicable to:
- \( C \) is conjunction of equalities or \(</\):
  - \( A_1 = B_1 \) AND ... AND \( A_n = B_n \)

Procedure Output-Tuples

While \( R_1\{i\}.C = R_2\{j\}.C \land (i \leq T(R_1)) \) do
  \( jj \leftarrow j \);
  while \( R_1\{i\}.C = R_2\{jj\}.C \land (jj \leq T(R_2)) \) do
    output pair \( R_1\{i\}, R_2\{jj\}; \)
    \( jj \leftarrow jj+1 \)
  \( i \leftarrow i+1 \)
Example

<table>
<thead>
<tr>
<th>i</th>
<th>R₁(i).C</th>
<th>R₂(j).C</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td>30</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
<td>30</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>52</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

Index nested loop (Conceptually)

For each \( r \in R₁ \) do

\[ X \leftarrow \text{index} (R₂, C, r.C) \]

for each \( s \in X \) do

output \((r,s)\) pair

Assume \( R₂.C \) index

Note: \( X \leftarrow \text{index(rel, attr, value)} \)

then \( X = \text{set of rel tuples with attr = value} \)

Hash join (conceptual)

Hash function \( h \), range \( 0 \rightarrow k \)

Buckets for \( R₁ \): \( G₀, G₁, \ldots, Gₖ \)

Buckets for \( R₂ \): \( H₀, H₁, \ldots, Hₖ \)

Applicable to:

- \( C \) is conjunction of equalities
  \( A₁ = B₁ \) AND ... AND \( Aₙ = Bₙ \)

Simple example hash: even/odd

<table>
<thead>
<tr>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>

Even: \( 2, 4, 5, 8, 14 \)

Odd: \( 3, 5, 9, 11 \)

Factors that affect performance

1. Tuples of relation stored physically together?
2. Relations sorted by join attribute?
3. Indexes exist?
Example 1(a)   NL Join $R_1 \bowtie R_2$

- Relations not contiguous
- Recall $T(R_1) = 10,000\quad T(R_2) = 5,000$
  $S(R_1) = S(R_2) = 1/10$ block
  MEM = 101 blocks

Can we do better?

Use our memory
1. Read 100 blocks of $R_1$
2. Read all of $R_2$ (using 1 block) + join
3. Repeat until done

Cost: for each $R_1$ chunk:
  Read chunk: 100 IOs
  Read $R_2$: 500 IOs

\[
\text{Total} = \frac{1,000 \times 600}{100} = 6,000 \text{ IOs}
\]
• Can we do better?

• Reverse join order: $R_2 \bowtie R_1$

Total = $500 \times (100 + 1,000) = 100$

$5 \times 1,100 = 5,500$ IOs

Cost of Block Nested Loop

• Reverse join order: $R_1 \bowtie R_2$

Total = $\frac{B(R_1)}{M-1} \times (\min(B(R_1), M-1) + B(R_2))$

Block-Nested Loop Join (conceptual)

for each $M-1$ blocks of $R_1$ do

read $M-1$ blocks of $R_1$ into buffer

for each block of $R_2$ do

read next block of $R_2$

for each tuple $r$ in $R_1$ block

for each tuple $s$ in $R_2$ block

if $(r,s) \models C$ then output $(r,s)$

Note

• How much memory for buffering inner and for outer chunks?

  – 1 for inner would minimize I/O

  – But, larger buffer better for I/O
Example 1(b) Merge Join

- Both $R_1$, $R_2$ ordered by $C$; relations contiguous

Memory

```
R_1  
R_2  
```

Total cost: Read $R_1$ cost + read $R_2$ cost

$= 1000 + 500 = 1,500$ IOs
Example 1(c)  Merge Join

- $R_1, R_2$ not ordered, but contiguous

--> Need to sort $R_1, R_2$ first
One way to sort: Merge Sort

(i) For each 100 blk chunk of R:
- Read chunk
- Sort in memory
- Write to disk

(ii) Read all chunks + merge + write out

Cost: Sort
Each tuple is read, written, read, written
so...
Sort cost $R_1$: $4 \times 1,000 = 4,000$
Sort cost $R_2$: $4 \times 500 = 2,000$

Example 1(d) Merge Join (continued)
$R_1, R_2$ contiguous, but unordered
Total cost = sort cost + join cost
$= 6,000 + 1,500 = 7,500$ IOs

Example 1(c) Merge Join (continued)
$R_1, R_2$ contiguous, but unordered
Total cost = sort cost + join cost
$= 6,000 + 1,500 = 7,500$ IOs
But: Iteration cost $= 5,500$
so merge joint does not pay off!

But say $R_1 = 10,000$ blocks contiguous
$R_2 = 5,000$ blocks not ordered
Iterate: $\frac{5000 \times (100+10,000)}{100} = 50 \times 10,100$
$= 505,000$ IOs
Merge join: $5(10,000+5,000) = 75,000$ IOs
Merge Join (with sort) WINS!
How much memory do we need for merge sort?

E.g: Say I have 10 memory blocks

\[ 10 \rightarrow \]

\( R_1 \)

100 chunks \( \Rightarrow \) to merge, need 100 blocks!

In general:

Say \( k \) blocks in memory

\( x \) blocks for relation sort

\# chunks = \( \frac{x}{k} \)  

size of chunk = \( k \)

# chunks < buffers available for merge

In our example

\( R_1 \) is 1000 blocks, \( k \geq 31.62 \)

\( R_2 \) is 500 blocks, \( k \geq 22.36 \)

Need at least 32 buffers

Again: in practice we would not want to use only one buffer per run!

Can we improve on merge join?

Hint: do we really need the fully sorted files?

\[ \begin{array}{c}
R_1 \\
\text{Join?} \\
R_2
\end{array} \]
Cost of improved merge join:
\[ C = \text{Read } R_1 + \text{write } R_1 \text{ into runs} + \text{read } R_2 + \text{write } R_2 \text{ into runs} + \text{join} \]
\[ = 2,000 + 1,000 + 1,500 = 4,500 \]
--> Memory requirement?

Example 1(d)  Index Join
- Assume \( R_1.C \) index exists; 2 levels
- Assume \( R_2 \) contiguous, unordered
- Assume \( R_1.C \) index fits in memory

Cost: Reads: 500 IOs
for each \( R_2 \) tuple:
- probe index - free
- if match, read \( R_1 \) tuple: 1 IO

What is expected # of matching tuples?
(a) say \( R_1.C \) is key, \( R_2.C \) is foreign key
then expect = 1
(b) say \( V(R_1,C) = 5000, \ T(R_1) = 10,000 \)
with uniform assumption
expect = \( 10,000/5,000 = 2 \)
(c) Say \( \text{DOM}(R_1,C) = 1,000,000 \)
\[ T(R_1) = 10,000 \]
with alternate assumption
Expect = \( \frac{10,000}{1,000,000} = \frac{1}{100} \)

Total cost with index join
(a) Total cost = 500+5000(1)1 = 5,500
(b) Total cost = 500+5000(2)1 = 10,500
(c) Total cost = 500+5000(1/100)1=550
What if index does not fit in memory?

Example: say \( R_1.C \) index is 201 blocks

- Keep root + 99 leaf nodes in memory
- Expected cost of each probe is
  \[ E = \frac{0 \times 99 + 1 \times 101}{200} \approx 0.5 \]

\[ \frac{200}{200} \]

Total cost (including probes)

\[ = 500 + 5000 \] [Probe + get records]
\[ = 500 + 5000 \] [0.5 + 2] uniform assumption
\[ = 500 + 12,500 = 13,000 \] (case b)

For case (c):
\[ = 500 + 5000 \times 0.5 \times 1 + \frac{1}{100} \times 1 \]
\[ = 500 + 2500 + 50 = 3050 \] IOs

Total cost (including probes)

\[ = 500 + 5000 \] [Probe + get records]
\[ = 500 + 5000 \] [0.5 + 2] uniform assumption
\[ = 500 + 12,500 = 13,000 \] (case b)

So far

\[
\begin{align*}
\text{Nested Loop} & : 5500 \\
\text{Merge Join} & : 1500 \\
\text{Sort-Merge Join} & : 7500 \rightarrow 4500 \\
\text{R}_1.C \text{ Index} & : 5500 \rightarrow 3050 \rightarrow 550 \\
\end{align*}
\]

Example 1(e) Partition Hash Join

- \( R_1, R_2 \) contiguous (un-ordered)
  - Use 100 buckets
  - Read \( R_1 \), hash, + write buckets

\[ R_1 \rightarrow R_1 \:	ext{memory} \]

- Same for \( R_2 \)
  - Read one \( R_1 \) bucket; build memory hash table
    - using different hash function \( h' \)
  - Read corresponding \( R_2 \) bucket + hash probe

\[ R_1 \rightarrow R_2 \]

\[ \Rightarrow \text{Then repeat for all buckets} \]
Cost:

“Bucketize:”   Read R₁ + write
                 Read R₂ + write
Join:           Read R₁, R₂

Total cost = 3 x [1000+500] = 4500

Cost:

“Bucketize:”   Read R₁ + write
                 Read R₂ + write
Join:           Read R₁, R₂

Total cost = 3 x [1000+500] = 4500

Note: this is an approximation since buckets will vary in size and we have to round up to blocks

Minimum memory requirements:

Size of R₁ bucket = \( \frac{x}{k} \)

\( k \) = number of memory buffers
\( x \) = number of R₁ blocks

So... \( \frac{x}{k} < k \)

\( k > \sqrt{x} \) need: \( k+1 \) total memory buffers

Can we use Hash-join when buckets do not fit into memory?:

• Treat buckets as relations and apply Hash-join recursively

Duality Hashing-Sorting

• Both partition inputs
• Until input fits into memory
• Logarithmic number of phases in memory size
Trick: keep some buckets in memory
E.g., \( k' = 33 \)  
\( R_1 \) buckets = 31 blocks  
keep 2 in memory

Next: Bucketize \( R_2 \)
- \( R_2 \) buckets = \( 500/33 = 16 \) blocks  
- Two of the \( R_2 \) buckets joined immediately with \( G_0, G_1 \)

Finally: Join remaining buckets
- for each bucket pair:
  - read one of the buckets into memory  
  - join with second bucket

Cost
- Bucketize \( R_1 \) = \( 1000 + 31 \times 31 = 1961 \)
- To bucketize \( R_2 \), only write 31 buckets:  
  so, cost = \( 500 + 31 \times 16 = 996 \)
- To compare join (2 buckets already done)  
  read \( 31 \times 31 + 31 \times 16 = 1457 \)

Total cost = \( 1961 + 996 + 1457 = 4414 \)

How many buckets in memory?
- See textbook for answer...
Another hash join trick:
• Only write into buckets
  \(<val, ptr>\) pairs
• When we get a match in join phase, must fetch tuples

To illustrate cost computation, assume:
– 100 \(<val, ptr>\) pairs/block
– expected number of result tuples is 100

To illustrate cost computation, assume:
– 100 \(<val, ptr>\) pairs/block
– expected number of result tuples is 100

• Build hash table for \(R_2\) in memory
  5000 tuples \(\rightarrow\) 5000/100 = 50 blocks
• Read \(R_1\) and match
• Read \(\sim 100 \ R_2\) tuples

• Build hash table for \(R_2\) in memory
  5000 tuples \(\rightarrow\) 5000/100 = 50 blocks
• Read \(R_2\) and match
• Read \(\sim 100 \ R_2\) tuples

Total cost =
\[
\begin{align*}
\text{Read } R_1 : & \quad 500 \\
\text{Read } R_2 : & \quad 1000 \\
\text{Get tuples:} & \quad 100 \\
& \quad 1600
\end{align*}
\]

So far:
<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterate</td>
<td>5500</td>
</tr>
<tr>
<td>Merge join</td>
<td>1500</td>
</tr>
<tr>
<td>Sort+merge join</td>
<td>7500</td>
</tr>
<tr>
<td>(R_1.C) index</td>
<td>5500 (\rightarrow) 550</td>
</tr>
<tr>
<td>(R_2.C) index</td>
<td></td>
</tr>
<tr>
<td>Build (R_1.C) index</td>
<td></td>
</tr>
<tr>
<td>Build (R_2.C) index</td>
<td></td>
</tr>
<tr>
<td>Hash join</td>
<td>4500+</td>
</tr>
<tr>
<td>with trick, (R_1) first</td>
<td>4414</td>
</tr>
<tr>
<td>with trick, (R_2) first</td>
<td></td>
</tr>
<tr>
<td>Hash join, pointers</td>
<td>1600</td>
</tr>
</tbody>
</table>

Yet another hash join trick:
• Combine the ideas of
  – block nested-loop with hash join
• Use memory to build hash-table for one chunk of relation
• Find join partners in \(O(1)\) instead of \(O(M)\)
• Trade-off
  – Space-overhead of hash-table
  – Time savings from look-up
Summary

- Nested Loop ok for “small” relations (relative to memory size)
  - Need for complex join condition
- For equi-join, where relations not sorted and no indexes exist, hash join usually best

- Sort + merge join good for non-equi-join (e.g., R₁.C > R₂.C)
- If relations already sorted, use merge join
- If index exists, it could be useful (depends on expected result size)

Join Comparison

\[ N_i = \text{number of tuples in } R_i \]
\[ B(R_i) = \text{number of blocks of } R_i \]
\[ \#P = \text{number of partition steps for hash join} \]
\[ P_{ij} = \text{average number of join partners} \]

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>#I/O</th>
<th>Memory</th>
<th>Disk Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested Loop</td>
<td>( \frac{B(R_i)}{(M-1)} \times \text{sort} + B(R_i) )</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Index Nested Loop</td>
<td>( B(R_i) + N_i + P_{ij} \times \text{sort} + 2 )</td>
<td>B(Index) + 2</td>
<td>0</td>
</tr>
<tr>
<td>Merge (sorted)</td>
<td>( B(R_i) + B(R_j) )</td>
<td>Max tuples = 0</td>
<td></td>
</tr>
<tr>
<td>Merge (unsorted)</td>
<td>( B(R_i) + B(R_j) \times \text{sort} - 1 )</td>
<td>B(R_i) + B(R_j)</td>
<td></td>
</tr>
<tr>
<td>Hash</td>
<td>( (2#P + 1) \times (B(R_i) + B(R_j)) )</td>
<td>root(max(B(R_i), B(R_j)), #P + 1)</td>
<td>~B(R_i) + B(R_j)</td>
</tr>
</tbody>
</table>

Why do we need nested loop?

- Remember not all join implementations work for all types of join conditions

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Type of Condition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested Loop</td>
<td>any</td>
<td>a LIKE &quot;%hello%&quot;</td>
</tr>
<tr>
<td>Index Nested Loop</td>
<td>Supported by index:</td>
<td>a = b</td>
</tr>
<tr>
<td></td>
<td>Equi-join (hash)</td>
<td>a &lt; b</td>
</tr>
<tr>
<td></td>
<td>Equi or range (B-tree)</td>
<td></td>
</tr>
<tr>
<td>Merge</td>
<td>Equalities and ranges</td>
<td>a &lt; b, a = b AND c = d</td>
</tr>
<tr>
<td>Hash</td>
<td>Equi-join</td>
<td>a = b</td>
</tr>
</tbody>
</table>

Outer Joins

- How to implement (left) outer joins?
- Nested Loop and Merge
  - Use a flag that is set to true if we find a match for an outer tuple
  - If flag is false fill with NULL
- Hash
  - If no matching tuple fill with NULL

Merge Left Outer Join

\[ R \bowtie_{B=C} S \]

Output: \((a,1,X)\)
Operators Overview

- (External) Sorting
- Joins (Nested Loop, Merge, Hash, …)
- Aggregation (Sorting, Hash)
- Selection, Projection (Index, Scan)
- Union, Set Difference
- Intersection
- Duplicate Elimination

Aggregation Example

```
SELECT sum(a), b
FROM R
GROUP BY b
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>sum(a)</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Aggregation

- Have to compute aggregation functions
  - for each group of tuples from input
- Groups
  - Determined by equality of group-by attributes

Aggregation Function Interface

- `init()`
  - Initialize state
- `update(tuple)`
  - Update state with information from tuple
- `close()`
  - Return result and clean-up
Implementation SUM(A)

- **init()**
  - `sum := 0`
- **update(tuple)**
  - `sum += tuple.A`
- **close()**
  - `return sum`

Aggregation Implementations

- **Sorting**
  - Sort input on group-by attributes
  - On group boundaries output tuple
- **Hashing**
  - Store current aggregated values for each group in hash table
  - Update with newly arriving tuples
  - Output result after processing all inputs

Aggregation Example

```sql
SELECT sum(a), b
FROM R
GROUP BY b
```

Aggregation Example

```sql
SELECT sum(a), b
FROM R
GROUP BY b
```

```sql
update(3, 1)
```

```sql
update(3, 1)
```

Grouping by sorting

- Similar to Merge join
- Sort R on group-by attribute
- Scan through sorted input
  - If group-by values change
    - Output using close() and call init()
  - Otherwise
    - Call update()
Aggregation Example

```
SELECT sum(a), b
FROM R
GROUP BY b
```

```
a  b
3  1
4  2
1  2
```

Grouping by Hashing

- Create in-memory hash-table
- For each input tuple probe hash table with group by values
  - If no entry exists then call `init()`, `update()`, and add entry
  - Otherwise call `update()` for entry
- Loop through all entries in hash-table and output calling `close()`

```
Group by changed!
```

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
```

```
Grouping by Hashing
```

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
```

```
Grouping by Hashing
```

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
```
### Aggregation Example

**SELECT** `sum(a), b`  
**FROM** `R`  
**GROUP BY** `b`

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

![Image](image.png)

### Aggregation Summary

- **Hashing**
  - No sorting -> no extra I/O
  - Hash table has to fit into memory
  - No outputs before all inputs have been processed
- **Sorting**
  - No memory required
  - Output one group at a time

### Operators Overview

- (External) Sorting  
- Joins (Nested Loop, Merge, Hash, ...)
- Aggregation (Sorting, Hash)
- Selection, Projection (Index, Scan)
- Union, Set Difference
- Intersection
- Duplicate Elimination

### Duplicate Elimination

- Equivalent to group-by on all attributes
- Can use aggregation implementations

**Optimization**

- Hash
  - Directly output tuple and use hash table only to avoid outputting duplicates

### Set Operations

- Can be modeled as join
  - with different output requirements
- As aggregation/group by on all columns
  - with different output requirements
Union

- Bag union
  - Append the two inputs
  - E.g., using three buffers
- Set union
  - Apply duplicate removal to result

Intersection

- Set version
  - Equivalent to join + project + duplicate removal
  - 3-state aggregate function (found left, found right, found both)
- Bag version
  - Join + project + min(i,j)
  - Aggregate min(count(i),count(j))

Set Difference

- Using join methods
  - Find matching tuples
  - If no match found, then output
- Using aggregation
  - count(i) – count(j) (bag)
  - true(i) AND false(j) (set)

Summary

- Operator implementations
  - Joins!
  - Other operators
- Cost estimations
  - I/O
  - memory
- Query processing architectures

Next

- Query Optimization Physical
- -> How to efficiently choose an efficient plan