TUTORIAL ON QUERY OPTIMIZATION
DB Logical Architecture

queries

Query Execution engine

Access Plan Executor
Parser
Operator Evaluator
Optimizer

Concurrency control

Transaction Manager
Lock Manager

Access Methods
Buffer Manager
Disk Manager

Recovery Manager
Relational Operators

Selection

Projection

Join

Union

Set-difference
Measures of Query Cost

- **Cost** is generally measured as **total elapsed time for answering query**
  - Many **factors** contribute to time cost: disk accesses, CPU, or even network communication

- Typically **disk access is the predominant cost**, and is also relatively easy to estimate
  - Measured by taking into account
    - Number of blocks read * average-block-read-cost
    - Number of blocks written * average-block-write-cost
  - Cost to write a block is greater than cost to read a block
    - Data is read back after being written to ensure that the write was successful
Measures of Query Cost

- For simplicity we just use number of block transfers from disk as the cost measure
  - We ignore the difference in cost between sequential and random I/O for simplicity
  - We also ignore CPU costs for simplicity
  - We do not include cost to writing output to disk in our cost formulae

- Costs depends on the size of the buffer in main memory
  - Having more memory reduces need for disk access
  - Amount of real memory available to buffer depends on other concurrent OS processes, and hard to determine ahead of actual execution
  - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available

- Real systems take CPU cost into account, differentiate between sequential and random I/O, and take buffer size into account
Nested-Loop Join

- **Read in outer relation R block by block**
  - Then, for each tuples in R, we scan the entire inner relation S (scan means read in S block by block)

- **n_R**: no. of record for R

- **b_R**: no. of block for R

- **Worst Cost**: \( b_R + n_R \times b_S \)

- **Best Cost**: \( b_R + b_S \) (if smaller relation can fit in memory)

- Use small relation as outer relation

- **Buffer**: 3 pages (1 for R, 1 for S, 1 for output)
Nested Loops Join
Exercise

- Relations: R1(A,B,C) and R2(C,D,E)
- R1 has 20,000 tuples
- R2 has 45,000 tuples
- 25 tuples of R1 fit on one block (blocking factor)
- 30 tuples of R2 fit on one block
- R1 JOIN R2
- R1 need 800 blocks (20000/25)
- R2 need 1500 blocks (45000/30)
- Assume M pages in memory
- If \( M > 800 \), cost = \( b_R + b_S = 1500 + 800 = 2300 \) I/Os

- Consider only \( M \leq 800 \),
- Using R1 as outer relation
  - Cost: \( 20000*1500 + 800 = 30000800 \) I/Os
- If R2 as outer relation
  - Cost: \( 45000*800 + 1500 = 36001500 \) I/Os
Block Nested Loop Join

- **Cost:** \( b_R + b_R \cdot b_S \)

- If \( M \) buffer pages available
  - **Cost:** \( b_R + \left\lfloor \frac{b_R}{M-2} \right\rfloor \cdot b_S \)
  - \( M \) buffer pages (1 for inner S, 1 for output and all remaining \( M-2 \) pages to hold “block” of outer R

- If \( R1 \) is outer
  - **Cost:** \( \left\lfloor \frac{800}{M-2} \right\rfloor \cdot 1500 + 800 \) I/Os

- If \( R2 \) is outer
  - **Cost:** \( \left\lfloor \frac{1500}{M-2} \right\rfloor \cdot 800 + 1500 \) I/Os
Index Nested-Loop Join

Table R

Table S

key

DB Server

Table R \Join S
Index Nested-Loop Join

- Primary B+tree index on the join attribute of R2:
  \[ b_{R1} + n_{R1} \cdot (x_{R2} + 1) \]
  where:
  - \( n_{R1} \cdot n_{R2} \) is the number of \( R_1 \) (\( R_2 \)) tuples
  - \( x_{R2} \) is the height of the B+-tree index on the join attribute
  - \( n_{R1} \cdot (x_{R2} + 1) \) is the cost of using B+-tree index to find matching tuple in \( R_2 \)

- Secondary B+tree index on the join attribute of R2:
  \[ b_{R1} + n_{R2} \cdot (x_{R2} + 1) \]
  where \( n_{R2} \cdot (x_{R2} + 1) \) is the cost of using B+-tree index to find matching tuple in \( R_2 \)
Index Nested-Loop Join

- Hash index on the join attribute of R2:
  - $b_{R1} + n_{R1} \times H$
  - Where $H$ is the average number of page accesses necessary to retrieve a tuple from R2 with a given key

- We use:
  - $H = 1.2$ for a primary hash index and
  - $H = 2.2$ for a secondary hash index
External Sorting

- File has $b_R$ pages

- Buffer $M$: number of main memory page buffers

- No. of runs in the first pass $R = b_R / M$

- No. of passes to sort file completely
  \[
  P = \lceil \log_{M-1} \left( \frac{b_R}{M} \right) \rceil + 1
  \]
  \[
  P = \lceil \log_{M-1} R \rceil + 1
  \]

- Total cost for sorting
  \[
  = b_R \times \left( 2 \times \lceil \log_{M-1} R \rceil + 1 \right)
  \]
  \[
  = b_R \times 2 \times \lceil \log_{M-1} R \rceil + b_R
  \]
Merge Join

- Assuming R1 and R2 are not initially sorted on the join key

- Cost = Sorting + $b_R + b_s$

- Sorting = $1500 \times (2 \times \left\lfloor \log_{M-1} (1500/M) \right\rfloor + 1) + 800 \times (2 \times \left\lfloor \log_{M-1} (800/M) \right\rfloor + 1)$

- Assuming all tuples with same values for join attributes fit in memory (each block needs to be read only once)
Merge Join

- Assuming that there is a secondary B+tree on Rx

- Cost = $C_{R1} + C_{R2}$

- where $C_{Rx} = (n_{Rx} \times ps)/(0.69 \times bs) + b_{Rx}$ for the R which has the index on the join attribute
  - $ps$: the size of the tuple reference (tuple identifier, rid)
  - $bs$: the size of the block

- i.e.: the leaf nodes of the index tree (assumed to be 69% full) have to be scanned for pointers to the tuples of the relation and the blocks containing the tuples itself must be read at least once
Hash join

- Hash both relations on the join attribute using the same hash function

- Since R1 is smaller, we use it as the build relation and R2 as probe relation

- Assume no overflow occurs

- If \( M \geq 800/M \), no need for recursive partitioning, cost = \( 3(1500 + 800) = 6900 \) disk access = \( 3(b_R + b_s) \)

- Else, cost = \( 2(1500 + 800) \lceil \log_{M-1} (800) - 1 \rceil + 1500 + 800 \) disk access
  = \( 2(b_R + b_s) \lceil \log_{M-1} (b_s) - 1 \rceil + b_R + b_s \)
Why Optimize?

- Given a query of size $n$ and a database of size $m$, how big can the output of applying the query to the database be?

- Example: $R(A)$ with 2 rows. One row has value 0. One row has value 1.
  - How many rows are in $R \times R$?
  - How many in $R \times R \times R$?

  $\Rightarrow$ Size of output as a function of input: $O(\ ? )$

- Usually, queries are small
  - Therefore, it is usually assumed that queries are of a fixed size
  - Use term data complexity when we analyze time, assuming that query is constant

- What is the size of the output in this case?
Optimizer Architecture

- Rewriter
- Algebraic Space
- Planner
- Cost Model
- Method-Structure Space
- Size-Distribution Estimator
Optimizer Architecture

- **Rewriter**: Finds equivalent queries that, perhaps can be computed more efficiently; all such queries are passed on to the Planner
  - Examples of Equivalent queries: Join orderings
- **Planner**: Examines all possible execution plans and chooses the cheapest one, i.e., fastest one
  - Uses other modules to find best plan
- **Algebraic Space**: Determines which types of queries will be examined
  - Example: Try to avoid Cartesian Products
- **Method-Structure Space**: Determines what types of indexes are available and what types of algorithms for algebraic operations can be used
  - Example: Which types of join algorithms can be used
- **Cost Model**: Estimates the cost of execution plans
  - Uses Size-Distribution Estimator for this
- **Size-Distribution Estimator**: Estimates size of tables, intermediate results, frequency distribution of attributes and size of indexes
Algebraic Space

- We consider queries that consist of **select**, **project** and **join** (Cartesian product is a special case of join).
- Such queries can be represented by a tree.
- Example: 
  
  ```
  emp(name, age, sal, dno)
  dept(dno, dname, floor, mgr, ano)
  act(ano, type, balance, bno)
  bank(bno, bname, address)
  ```

  select name, floor
  from emp, dept
  where emp.dno=dept.dno and sal > 100K
3 Trees

\[ \pi_{\text{name, floor}} \]

\[ \sigma_{\text{sal}>100K} \]

\[ \pi_{\text{name, floor}} \]

\[ \sigma_{\text{sal}>100K} \]

\[ \pi_{\text{dno, name}} \]

\[ \pi_{\text{dno, floor}} \]

T1

T2

T3
Restriction 1 of Algebraic Space

- Algebraic space may contain many equivalent queries

- Important to restrict space

- Restriction (heuristic) 1: Only allow queries for which selection and projection:
  - are processed as early as possible
  - are processed on the fly
Performing Selection and Projection "On the Fly"

- Selection and projection are performed as part of other actions.
- Projection and selection that appear one after another are performed one immediately after another.
  - Projection and Selection do not require writing to the disk.
- Selection is performed while reading relations for the first time.
- Projection is performed while computing answers from previous action.
Processing Selection/Projection as Early as Possible

- The three trees differ in the way that selection and projection are performed

- In T3, there is "maximal pushing of selection and projection"
  - Rewriter finds such expressions
Restriction 2 of Algebraic Space

- Since the order of selection and projection is determined, we can write trees only with joins
- **Restriction (heuristic) 2**: Cross products are never formed, unless the query asks for them
- Example:

```sql
select name, floor, balance
from emp, dept, acnt
where emp.dno=dept.dno and dept.ano = acnt.ano
```
3 Trees

Which trees have cross products?

T1

EMP

ACNT

EMP

DEPT

T2

EMP

ACNT

DEPT

T3

EMP

ACNT

DEPT

EMP

ACNT
Restriction 3 of Algebraic Space

- The left relation is called the outer relation in a join and the right relation is the inner relation (as in terminology of nested loops algorithms).

- **Restriction (heuristic) 3**: The inner operand of each join is a database relation, not an intermediate result (left-deep plans).

- Example:

```sql
select name, floor, balance
from emp, dept, acnt, bank
where emp.dno=dept.dno and dept.ano=acnt.ano
and acnt.bno = bank.bno
```
Which trees follow restriction 3?
Pipelining Joins

- Consider computing: \((\text{Emp} \bowtie \text{Dept}) \bowtie \text{Acnt}\). In principle, we should:
  - compute \(\text{Emp} \bowtie \text{Dept}\), write the result to the disk
  - then read it from the disk to join it with \text{Acnt}

- When using block and index nested loops join, we can avoid the step of writing to the disk

- We allow plans that:
  - Perform selection and projection early and on the fly
  - Do not create cross products
  - Use database relations as inner relations (also called left–deep trees)
Pipelining Joins - Example

1. Read block from Emp
2. Find matching Dept tuples using index
3. Find matching Acnt tuples using index
4. Write final output

Emp blocks → Dept blocks → Acnt blocks → Output blocks → Buffer
Planner

- Dynamic programming algorithm to find best plan for performing join of $N$ relations

- Intuition:
  - Find all ways to access a single relation
    - Estimate costs and choose best access plan(s)
  - For each pair of relations, consider all ways to compute joins using all access plans from previous step
    - Choose best plan(s)...
  - For each i-1 relations joined, find best option to extend to i relations being joined...
  - Given all plans to compute join of n relations, output the best
Reminder: Dynamic Programming

To find an optimal plan for joining $R_1$, $R_2$, $R_3$, $R_4$, choose the best among:

- Optimal plan for joining $R_2$, $R_3$, $R_4$ + for reading $R_1$ + optimal join of $R_1$ with result of previous joins
- Optimal plan for joining $R_1$, $R_3$, $R_4$ + for reading $R_2$ + optimal join of $R_2$ with result of previous joins
- Optimal plan for joining $R_1$, $R_2$, $R_4$ + for reading $R_3$ + optimal join of $R_3$ with result of previous joins
- Optimal plan for joining $R_1$, $R_2$, $R_3$ + for reading $R_4$ + optimal join of $R_4$ with result of previous joins
Not Good Enough: Interesting Orders

- Example, suppose we are computing \((R(A,B) \bowtie S(B,C)) \bowtie T(B,D))\)
  - Maybe merge-sort join of \(R\) and \(S\) is not the most efficient, but the result is sorted on \(B\)
  - If \(T\) is sorted on \(B\), the performing a sort-merge join of \(R\) and \(S\), and then of the result with \(T\), maybe the cheapest total plan

- For some joins, such as sort-merge join, the cost is cheaper if relations are ordered
  - Therefore, it is of interest to create plans where attributes that participate in a join are ordered on attributes in joins later on

- For each interesting order, save the best plan
  - We save plans for non interesting order if it better than all interesting order costs
We want to compute the query:

```
select name, mgr
from emp, dept
where emp.dno=dept.dno and sal>30K and floor = 2
```

- Available Indexes: B+tree index on emp.sal, B+tree index on emp.dno, hashing index on dept.floor
- Join Methods: Block nested loops, index nested loops and sort-merge
- In the example, all cost estimations are fictional
### Step 1 – Accessing Single Relations

<table>
<thead>
<tr>
<th>Relation</th>
<th>Interesting Order</th>
<th>Plan</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>emp</td>
<td>emp.dno</td>
<td>Access through B+tree on emp.dno</td>
<td>700</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Access through B+tree on emp.sal</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sequential scan</td>
<td>600</td>
</tr>
<tr>
<td>dept</td>
<td></td>
<td>Access through hashing on dept.floor</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sequential scan</td>
<td>200</td>
</tr>
</tbody>
</table>

Which do we save for the next step?
## Step 2 – Joining 2 Relations

<table>
<thead>
<tr>
<th>Join Method</th>
<th>Outer/Inner</th>
<th>Plan</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>nested loops</td>
<td>empt/dept</td>
<td>● For each emp tuple obtained through B+Tree on emp.sal, scan dept through hashing index on dept.floor to find tuples matching on dno&lt;br&gt;● For each emp tuple obtained through B+Tree on emp.dno and satisfying selection, scan dept through hashing index on dept.floor to find tuples matching on dno</td>
<td>1800</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3000</td>
</tr>
</tbody>
</table>
### Step 2 – Joining 2 Relations

<table>
<thead>
<tr>
<th>Join Method</th>
<th>Outer/Inner</th>
<th>Plan</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>nested loops</td>
<td>dept/emp</td>
<td>- For each dept tuple obtained through hashing index on dept.floor, scan emp through B+Tree on emp.sal to find tuples matching on dno</td>
<td>2500</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- For each dept tuple obtained through hashing index on dept.floor, scan emp through B+Tree on emp.dno to find tuples satisfying the selection on emp.sal</td>
<td>1500</td>
</tr>
</tbody>
</table>
### Step 2 – Joining 2 Relations

<table>
<thead>
<tr>
<th>Join Method</th>
<th>Outer/Inner</th>
<th>Plan</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort merge</td>
<td></td>
<td>● Sort the emp tuples resulting from accessing the B+Tree on emp.sal into L1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Sort the dept tuples resulting from accessing the hashing index on dept.floor into L2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Merge L1 and L2</td>
<td>2300</td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Sort the dept tuples resulting from accessing the hashing index on dept.floor into L2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>● Merge L2 and the emp tuples resulting from accessing the B+Tree on emp.dno and satisfying the selection on emp.sal</td>
<td>2000</td>
</tr>
</tbody>
</table>
Picking a Query Plan

- Suppose we want to find the natural join of: Reserves, Sailors, Boats
- The 2 options that appear the best are (ignoring the order within a single join):
  (Sailors $\bowtie$ Reserves) $\bowtie$ Boats
  Sailors $\bowtie$ ((Reserves $\bowtie$ Boats)
- We would like intermediate results to be as small as possible

--> Generating and comparing plans

Generate Pruning
Estimate Cost
Select
Pick Min

Cost

Plans
Analyzing Result Sizes

- In order to answer the question in the previous slide, we must be able to estimate the size of (Sailors→←Reserves) and (Reserves→←Boats)

- The DBMS stores statistics about the relations and indexes
  - **Cardinality**: Num of tuples $NTuples(R)$ in each relation $R$
  - **Size**: Num of pages $NPages(R)$ in each relation $R$
  - **Index Cardinality**: Num of distinct key values $NKeys(I)$ for each index $I$
  - **Index Size**: Num of pages $INPages(I)$ in each index $I$
  - **Index Height**: Num of non-leaf levels $IHeight(I)$ in each B+ Tree index $I$
  - **Index Range**: The minimum $ILow(I)$ and maximum value $IHigh(I)$ for each index $I$

- They are updated periodically (*not* every time the underlying relations are modified)
Estimating Result Sizes

- Consider

```sql
SELECT attribute-list
FROM relation-list
WHERE term₁ and ... and termₙ
```

- The maximum number of tuples is the product of the cardinalities of the relations in the FROM clause

- The WHERE clause is associating a reduction factor with each term
  - `column = value`: $1/N\text{Keys}(I)$ if there is an index I on `column`. This assumes a uniform distribution; otherwise, System R assumes $1/10$
  - `column₁ = column₂`: $1/\max(N\text{Keys}(I₁),N\text{Keys}(I₂))$ if there is an index I₁ on `column₁` and I₂ on `column₂`. If only one column has an index, we use it to estimate the value; otherwise, use $1/10$
  - `column > value`: $(\text{High}(I)-\text{value})/(\text{High}(I)-\text{Low}(I))$ if there is an index I on `column`

- Estimated result size is: maximum size times product of reduction factors
Example

SELECT *
FROM Reserves R, Sailors S
WHERE R.sid = S.sid and S.rating > 3
    and R.agent = 'Joe'

- Cardinality(R) = 1,000 * 100 = 100,000
- Cardinality(S) = 500 * 80 = 40,000
- NKeys(Index on S.sid) = 40,000
- NKeys(Index on R.agent) = 100
- High(Index on Rating) = 10, Low = 0
- Maximum cardinality: 100,000 * 40,000
- Reduction factor of R.sid = S.sid: 1/40,000
- Reduction factor of S.rating > 3: (10–3)/(10-0) = 7/10
- Reduction factor of R.agent = ‘Joe’: 1/100
- Total Estimated size: (Maximum cardinality) * (Reduction factor of R.sid) * 
  (Reduction factor of S.rating) * (Reduction factor of R.agent = S.sid) = 100,000 * 
  40,000 * (1/40,000) * (7/10) * (1/100) = 700
Consider the join of the four relations named R, S, T, U:

<table>
<thead>
<tr>
<th>R(a,b), 800 total tuples</th>
<th>S(b,c), 10,000 total tuples</th>
<th>T(c,d), 4,000 total tuples</th>
<th>U(a,d), 2,500 total tuples</th>
</tr>
</thead>
<tbody>
<tr>
<td>V(R,a) = 500</td>
<td></td>
<td></td>
<td>V(U,a) = 10</td>
</tr>
<tr>
<td>V(R,b) = 100</td>
<td>V(S,b) = 2,000</td>
<td></td>
<td>V(U,b) = 50</td>
</tr>
<tr>
<td></td>
<td>V(S,c) = 1,000</td>
<td>V(T,c) = 1,000</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>V(T,d) = 1,000</td>
<td>V(U,d) = 500</td>
</tr>
</tbody>
</table>
Notes

- $V(R,a)$: # of **distinct** values for attribute

- Cost $\{R,S\} = \frac{\text{size of } R \times \text{size of } S}{\max(V(R,\_), V(S,\_))}$, where $\_$ is the join attribute

- Cost $\{R,S,U\} = \frac{\text{size of } R \times \text{size of } S \times \text{size of } U}{\text{2 greater nums from } (V(R,\_), V(S,\_), V(U,\_))}$, where $\_$ is the join attribute
Second Example of Join Order Selection

- For the singleton sets, the costs and best plans are given in the table below:

<table>
<thead>
<tr>
<th></th>
<th>{R}</th>
<th>{S}</th>
<th>{T}</th>
<th>{U}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>800</td>
<td>10.000</td>
<td>4.000</td>
<td>2.500</td>
</tr>
<tr>
<td>Cost</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Best plan</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
</tr>
</tbody>
</table>

- As the costs for all relations are the same, the dynamic programming algorithm will consider them all. The greedy algorithm however, must choose one, or also consider them all. Let’s assume it takes the one with least cost, and if they are more present of these, take the one with smallest length. So, the plan called “R” is chosen.
Second Example of Join Order Selection

- Now, we consider the pairs of relations
  - Again, the cost is 0 for each, as we do not have intermediate results

<table>
<thead>
<tr>
<th></th>
<th>{R,S}</th>
<th>{R,T}</th>
<th>{R,U}</th>
<th>{S,T}</th>
<th>{S,U}</th>
<th>{T,U}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>4.000</td>
<td>3.200</td>
<td>4.000</td>
<td>40.000</td>
<td>12.500</td>
<td>10.000</td>
</tr>
<tr>
<td>Cost</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Best plan</td>
<td>RxS</td>
<td>RxT</td>
<td>RxU</td>
<td>SxT</td>
<td>SxU</td>
<td>TxU</td>
</tr>
</tbody>
</table>

- The dynamic programming algorithm again keeps them all for the next run, as the costs are 0. The greedy algorithm had chosen R in the previous run, so it must choose a plan based on this choice. Since there are two best plans here, it has to make a choice of them. Let’s assume it makes the wrong choice, and takes “RxU”.
Now, we consider the join of three out of these four relations:

As you can see, the best plan is clearly "(RxS)xU", with the least cost and size. However, the greedy algorithm will not consider this plan, as it chose "RxU" as the best plan in previous run. Instead, it is forced to take "(RxU)xT" as best plan now.
Finally, we consider the join of all relations. We come to these four final results (for dynamic programming):

<table>
<thead>
<tr>
<th>Plan</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$((R \times S) \times T) \times U$</td>
<td>20.000</td>
</tr>
<tr>
<td>$((R \times S) \times U) \times T$</td>
<td>9.000</td>
</tr>
<tr>
<td>$((R \times U) \times T) \times S$</td>
<td>20.000</td>
</tr>
<tr>
<td>$((T \times U) \times S) \times R$</td>
<td>60.000</td>
</tr>
</tbody>
</table>

The dynamic programming algorithm finds the optimal solution $((R \times S) \times U) \times T$, while the greedy algorithm, based on results of the previous run, comes to the more expensive solution $((R \times U) \times T) \times S$. 
Selecting Algorithms for Plan Operators

- For each logical plan operator, select algorithms based on I/O cost estimation

- For selection operator, consider
  - Index-scan algorithms that use single attribute indexes, multiple indexes, or multidimensional indexes
  - Table-scan algorithm using no index

- For join operator, consider
  - All types of join algorithms if enough statistics is available
  - If statistics is in sufficient, follow some simple ideas
    - Try one-pass algorithm or nested-loops
    - Use sort-join if one or both arguments are already sorted
    - If index is available, use index-join
    - If sort and index are not available and multi-pass join is necessary, use a hash join
Pipelining Example

- Relations:
  - $R(W,X), b_R = 5000$
  - $S(X,Y), b_S = 10000$
  - $U(Y,Z), b_U = 10000$

- Buffer: $M = 101$ blocks

- Both joins are hash join

- Size $k$ is estimated, and used to choose join algorithms
Case 1: $k \leq 49$

- Can pipeline result of 1st join into 2nd join

- Two-pass hash join for $R \bowtie S$:
  - Both $R$ and $S$ are hashed into 100 partitions, where each $R$ partition has 50 blocks
  - Join corresponding $R$ & $S$ partitions using 50 buffer blocks for $R$ partition, 1 block for $S$ partition, and store the result in 49 blocks as a hash table

- One-pass hash join for the 2nd join:
  - Use 1 buffer block for $U$ (no need to partition $U$), join with the intermediate result that is already in buffer

- Cost = $3(5000+10000) + 10000 = 55000$
Case 2: 49 < k ≤ 5000

- Overlap the 1st join with the hash partitioning of the 2nd join

- Two-pass hash join for the 1st join:
  - Partition R & S into 100 partitions, so that each R partition contains 50 blocks
  - Join corresponding R & S partitions (using 51 buffer blocks)
  - During the join, hash the result into 50 partitions (using the remaining 50 buffer blocks) & write the partitions to disk

- Two-pass hash join for the 2nd join:
  - Partition U into 50 partitions
  - Join corresponding partitions of intermediate result & U, using intermediate result partition as build relation (use 1 to 100 buffer blocks)

- Cost = 3(10000+5000) + k + 2(10000) + (k+10000) = 75000 + 2k
Case 3: k > 5000

- Cannot use pipelining

- Two-pass hash join for the 1st join:
  - Partition R & S into 51 partitions, so that each R partition has <100 blocks
  - Join corresponding R & S partitions, write results to disk

- Two-pass hash join for the 2nd join:
  - Partition intermediate result & U into more than 50 partitions
  - Join corresponding partitions of U & intermediate result, using the smaller partition as the build relation

- Cost = 3(5000+10000) + k + 3(10000+k) = 75000 + 4k
Pipelining vs. Materialization

- **Pipelining**: Apply next operator to the output of one stage, as the output is generated.

- **Materialization**: Create a temporary relation as the output of a stage, pass to next stage
Pipelining vs. Materialization

**Advantages** of 64 bit processors

- More main memory possible
  - And so, more pipelining operations possible without having to write intermediate results to disk
- Complex in-memory processing does not require intermediate results being temporarily written to disk
  - Saves costly disk I/O’s and increases scalability

**Disadvantages** of 64 bit processors

- Application must be fully supporting 64 bit to make full use of the speed advantages
- Upgrading to a 32 bit system with (more) parallel processors (using shared memory perhaps) might be cheaper to implement

**DBMS’s implementing 64 bit are** e.g. Oracle 10g
Ordering of Physical Operations

- Pre-order traversal
- Post-order traversal
Notation for Physical Query Plans

- Non-standard among DBMSs
- Typical physical plan operators include
  - For leaf nodes: `TableScan(R)`, `SortScan(R, AttrList)`, `IndexScan(R, A)`, `IndexScan(R, Aθc)`
  - For selection nodes: combination of `TableScan(R)`, `Filter(Cond)`, `SortScan(R, AttrList)`

Two-pass hash join
101 buffers

TableScan(U)

Two-pass hash join
101 buffers

TableScan(R)

TableScan(S)
Points to Remember

- **Step 1: Choose a logical plan**
  - Involves choosing a query tree, which indicates the order in which algebraic operations are applied
  - **Heuristic**: Pushed trees are good, but sometimes “nearly fully pushed” trees are better due to indexing (as we saw in the example)
  - **So**: Take the initial “master plan” tree and produce a fully pushed tree plus several nearly fully pushed trees

- **Step 2: Reduce search space**
  - Deal with associativity of binary operators (join, union, …)
  - Choose a particular shape of a tree (left-deep trees)
    - Equals the number of ways to parenthesize N-way join – grows very rapidly
  - Choose a particular permutation of the leaves
    - E.g., 4! permutations of the leaves A, B, C, D

- **Step 3: Use a heuristic search to further reduce complexity**
  - The choice of left-deep trees still leaves open too many options
  - A heuristic algorithm is used to get a ‘good’ plan