Example: External Sorting Using Sort-Merge (N >= M)

M = 3
N = 12

b_r / 2 \lfloor \log_{b_r}(b_r / M) \rfloor + 1) blocks

seeks:
2 \lfloor b_r / M \rfloor + \lfloor b_r / b_r \rfloor (2 \lfloor \log_{b_r}(b_r / M) \rfloor - 1)

\text{Initial relation}
\begin{align*}
g & 24 \\
a & 24 \\
d & 31 \\
c & 33 \\
b & 14 \\
e & 16 \\
r & 16 \\
d & 21 \\
m & 3 \\
p & 2 \\
d & 7 \\
a & 14 \\
\end{align*}

\text{Create runs}
\begin{align*}
a & 19 \\
d & 31 \\
g & 24 \\
b & 14 \\
c & 33 \\
e & 16 \\
p & 2 \\
\end{align*}

\text{Merge pass 1}
\begin{align*}
a & 19 \\
b & 14 \\
c & 33 \\
d & 31 \\
e & 16 \\
g & 24 \\
p & 2 \\
\end{align*}

\text{Merge pass 2}
\begin{align*}
a & 14 \\
b & 19 \\
c & 33 \\
d & 31 \\
e & 16 \\
g & 24 \\
p & 2 \\
\end{align*}

\text{Sorted output}
\begin{align*}
a & 14 \\
b & 19 \\
c & 33 \\
d & 31 \\
e & 16 \\
g & 24 \\
p & 2 \\
\end{align*}
Merge-Join (Sort-merge join)

- Pre-condition:
  - equi-/natural joins
  - The relations must be sorted by the join attribute
  - If not sorted, can sort first, and then use this
- Called “sort-merge join” sometimes

```
SELECT *
FROM r, s
WHERE r.a1 = s.a1
```

Step:
1. Compare the tuples at pr and ps
2. Move pointers down the list
   - Depending on the join condition
3. Repeat

Cost:
- If the relations sorted, then just
  - \( b_r + b_s \) block transfers, some seeks depending on memory size
- What if not sorted?
  - Then sort the relations first
  - In many cases, still very good performance
  - Typically comparable to hash join

Observation:
- The final join result will also be sorted on \( a1 \)
- This might make further operations easier to do
  - E.g. duplicate elimination
Project Code

```python
if self.jointype == self.INNER_JOIN:
    ptr_l = 0
    ptr_r = 0
    while ptr_l < len(left_input) and ptr_r < len(right_input):
        l_attr = left_input[ptr_l].getAttribute(self.left_attribute)
        ptr_l += 1
        while ptr_l < len(left_input):
            if left_input[ptr_l].getAttribute(self.left_attribute) == l_attr:
                set_L.append(left_input[ptr_l])
                ptr_l += 1
            else:
                break
        while ptr_r < len(right_input) and right_input[ptr_r].getAttribute(self.right_attribute) == l_attr:
            if right_input[ptr_r].getAttribute(self.right_attribute) == l_attr:
                for l in set_L:
                    output = list(l.t)
                    output.extend(list(right_input[ptr_r].t))
                    yield Tuple(None, output)
                ptr_r += 1
    else:
        while ptr_l < len(left_input):
            set_L.append(left_input[ptr_l])
            ptr_l += 1
else:
    if self.jointype == self.FULL_OUTER_JOIN:
```

Project Code: $B^+$-trees

```
# Update the parent pointer for the block pointed to be that ptr
otherBlock.keysAndPointers[0].getBlock().parent = Pointer(otherBlock.blockNumber)
```
Project Code: B$^+$-trees

```python
# Update the parent pointer for the block pointed to be that ptr
otherBlock.keysAndPointers[0].getBlock().parent = Pointer(otherBlock.blockNumber)
```

So far...

- **Block Nested-loops join**
  - Can always be applied irrespective of the join condition
- **Index Nested-loops join**
  - Only applies if an appropriate index exists
  - Very useful when we have selections that return small number of tuples
    - `select balance from customer, accounts where customer.name = "j. s." and customer.SSN = accounts.SSN`
- **Merge joins**
  - Join algorithm of choice when the relations are large
  - Sorted results commonly desired at the output
    - To answer group by queries, for duplicate elimination, because of ASC/DSC
Hash Join

- **Case 1: Smaller relation \( (S) \) fits in memory**
- Nested-loops join:
  
  \[
  \text{for each tuple } r \text{ in } R \\
  \quad \text{for each tuple } s \text{ in } S \\
  \quad \text{check if } r.a = s.a
  \]

- Cost: \( b_r + b_s \) transfers, 2 seeks
- The inner loop is not exactly cheap (high CPU cost)

- Hash join:
  
  \[
  \text{read } S \text{ in memory and build a hash index on it} \\
  \quad \text{for each tuple } r \text{ in } R \\
  \quad \text{use the hash index on } S \text{ to find tuples such that } S.a = r.a
  \]

- Cost: \( b_r + b_s \) transfers, 2 seeks (unchanged)
- Why good?
  - CPU cost is much better (even though we don't care about it too much)
  - Much better than nested-loops join when \( S \) doesn't fit in memory (next)
Hash Join

- **Case 2: Smaller relation (S) doesn’t fit in memory**
- **Basic idea:**
  - partition tuples of each relation into sets that have same value on join attributes
  - must be equi-/natural join
- **Phase 1:**
  - Read $R$ block by block and partition it using a hash function: $h_1(a)$
    - Create one partition for each possible value of $h_1(a)$ ($n_h$ partitions)
  - Write the partitions to disk
    - $R$ gets partitioned into $R_1$, $R_2$, ..., $R_k$
  - Similarly, read and partition $S$, and write partitions $S_1$, $S_2$, ..., $S_k$ to disk
  - Only requirements:
    - Room for a single input block and one output block for each hash value
    - Each $S$ partition fits in memory

---

Hash Join

- **Case 2: Smaller relation (S) doesn’t fit in memory**
- **Two “phases”**
- **Phase 2:**
  - Read $S_i$ into memory, and build a hash index on it ($S_i$ fits in memory)
    - Using a different hash function from the partition hash: $h_2(a)$
  - Read $R_i$ block by block, and use the hash index to find matches.
  - Repeat for all $i$. 

---
Hash Join

- **Case 2**: Smaller relation \( (S) \) doesn’t fit in memory
- Two “phases”:
  - Phase 1:
    - Partition the relations using one hash function, \( h_1(a) \)
  - Phase 2:
    - Read \( S_i \) into memory, and build a hash index on it (\( S_i \) fits in memory)
    - Read \( R_i \) block by block, and use the hash index to find matches.
- **Cost**?
  - \( 3(b_r + b_s) \) block transfers
    - \( R \) or \( S \) might have partially full block to be read and written (ignored)
  - \( + 2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil) \) seeks (seek count unclear)
    - Where \( b_b \) is the size of each input buffer (p 560)
  - Much better than Nested-loops join under the same conditions
Hash Join: Issues

- How to guarantee that each partition of S fits in memory?
  - Say S = 10,000 blocks, Memory = M = 100 blocks
  - Use a hash function that hashes to 100 different values?
    - Eg. \( h_1(a) = a \mod 100 \)
    - Problem: Impossible to guarantee uniform split
      - Some partitions will be larger than 100 blocks, some will be smaller
    - Use a hash function that hashes to \( 100f \) different values
      - \( f \) is called fudge factor, typically around 1.2
      - So we may consider \( h_1(a) = a \mod 120 \).
      - This is okay IF \( a \) is uniformly distributed
  - Why can’t we just set \( h_n \) to 200?
    - need to have a per-value output block in mem during build phase

Hash Join: Issues

- Memory required?
  - Say S = 10000 blocks, Memory = M = 100 blocks
  - So 120 different partitions
  - During phase 1:
    - Need 1 block for storing \( R \)
    - Need 120 blocks for storing each partition of \( R \)
  - So must have at least 121 blocks of memory
  - We only have 100 blocks
  - Typically need \( \sqrt{|S| \cdot f} \) blocks of memory
    - So if S is 10000 blocks, and \( f = 1.2 \), need 110 blocks of memory
    - Need:
      - \( M > n_h + 1 \)
      - each partition of S to fit in M-1 (why not R?)
      - space for hash build on \( h2() \) (usually ignored)
  - Example:
    - \( h_n = 109 \), average size = 10,000/109 = 91.7
Hash Join: If $S_i$ Too Large

- **Avoidance**
  - Fudge factor

- **Resolution**
  - partition w/ a third hash $h3()$
  - also partition $R_i$
  - go through each sub-partition

  - this approach could be used for every partition

---

Merge-Join (Sort-merge join)

- **Pre-condition:**
  - equi-/natural joins
  - The relations must be sorted by the join attribute
  - If not sorted, can sort first, and then use this

- Called “sort-merge join” sometimes

\[
\text{select } * \\
\text{from } r, s \\
\text{where } r.a1 = s.a1
\]

Step:
1. Compare the tuples at $pr$ and $ps$
2. Move pointers down the list
   - Depending on the join condition
3. Repeat
**Merge-Join (Sort-merge join)**

- **Cost:**
  - If the relations sorted, then just
    - \( b_r + b_s \) block transfers, some seeks depending on memory size
  - What if not sorted?
    - Then sort the relations first
    - In many cases, still very good performance
    - Typically comparable to hash join

- **Observation:**
  - The final join result will also be sorted on \( a_1 \)
  - This might make further operations easier to do
    - E.g. duplicate elimination

---

**Joins: Summary**

- **Block Nested-loops join**
  - Can always be applied irrespective of the join condition

- **Index Nested-loops join**
  - Only applies if an appropriate index exists

- **Hash joins – only for equi-joins**
  - Join algorithm of choice when the relations are large

- **Sort-merge join**
  - Very commonly used – especially since relations are typically sorted
  - Sorted results commonly desired at the output
    - To answer group by queries, for duplicate elimination, because of ASC/DSC
Query Processing

- Overview
- Selection operation
- Join operators
- Other operators
- Putting it all together…
- Sorting

Group By and Aggregation

```
select a, count(b)
from R
group by a;
```

- Hash-based algorithm:
  - Create a hash table on a, and keep the \textit{count(b) so far}
  - Read R tuples one by one
  - For a new R tuple, “r”
    - Check if r.a exists in the hash table
    - If yes, increment the count
    - If not, insert a new value
Group By and Aggregation

select a, count(b) from R group by a;

- **Sort-based algorithm:**
  - Sort $R$ on $a$
  - Now all tuples in a single group are contiguous
  - Read tuples of $R$ (sorted) one by one and compute the aggregates

Group By and Aggregation

select a, AGGR(b) from R group by a;

- **sum(), count(), min(), max():** only need to maintain one value per group
  - Called “distributive”
- **average():** need to maintain the “sum” and “count” per group
  - Called “algebraic”
- **stddev():** algebraic, but need to maintain some more state
- **median():** can do efficiently with sort, but need two passes (called “holistic”)
  - First to find the number of tuples in each group, and then to find the median tuple in each group
- **count(distinct b):** must do duplicate elimination before the count
Duplicate Elimination

\[ \text{select distinct } a \\
\text{from } R ; \]

- Best done using sorting – Can also be done using hashing
- Steps:
  - Sort the relation \( R \)
  - Read tuples of \( R \) in sorted order
  - \( \text{prev} = \text{null} ; \)
  - for each tuple \( r \) in \( R \) (sorted)
    - if \( r \neq \text{prev} \) then
      - Output \( r \)
      - \( \text{prev} = r \)
    - else
      - Skip \( r \)

Set operations

\[ (\text{select } * \text{ from } R) \text{ union } (\text{select } * \text{ from } S) ; \]
\[ (\text{select } * \text{ from } R) \text{ intersect } (\text{select } * \text{ from } S) ; \]
\[ (\text{select } * \text{ from } R) \text{ union all } (\text{select } * \text{ from } S) ; \]
\[ (\text{select } * \text{ from } R) \text{ intersect all } (\text{select } * \text{ from } S) ; \]

- Remember the rules about duplicates
- “union all”: just append the tuples of \( R \) and \( S \)
- “union”: append the tuples of \( R \) and \( S \), and do duplicate elimination
- “intersection”: similar to joins
  - Find tuples of \( R \) and \( S \) that are identical on all attributes
  - Can use hash-based or sort-based algorithm
Query Processing

- Overview
- Selection operation
- Join operators
- Other operators
- Putting it all together…
- Sorting

Evaluation of Expressions

select customer-name
from account a, customer c
where a.SSN = c.SSN and
a.balance < 2500

- Two options:
  - Materialization
  - Pipelining
Evaluation of Expressions

- **Materialization**
  - Evaluate each expression separately
  - Store its result on disk in temporary relations
  - Read it for next operation

- **Pipelining**
  - Evaluate multiple operators simultaneously
    - Do not go to disk
  - Usually faster, but requires more memory
  - Also not always possible..
    - E.g. Sort-Merge Join
  - Harder to reason about