SQL and Query Execution for Aggregation

Example Instances

<table>
<thead>
<tr>
<th>Reserves</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td></td>
<td>95</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sailors</th>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>22</td>
<td>Dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>Lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td></td>
<td>95</td>
<td>Bob</td>
<td>3</td>
<td>63.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Boats</th>
<th>bid</th>
<th>bname</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>101</td>
<td>Interlace</td>
<td>blue</td>
</tr>
<tr>
<td></td>
<td>102</td>
<td>Interlace</td>
<td>red</td>
</tr>
<tr>
<td></td>
<td>103</td>
<td>Clipper</td>
<td>green</td>
</tr>
<tr>
<td></td>
<td>104</td>
<td>Marine</td>
<td>red</td>
</tr>
</tbody>
</table>

Queries With GROUP BY

To generate values for a column based on groups of rows, use aggregate functions in SELECT statements with the GROUP BY clause

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
GROUP BY grouping-list
```

- The target-list contains
  - (i) list of column names &
  - (ii) terms with aggregate operations (e.g., \( \text{MIN} (S.\text{age}) \)).
- column name list (i) can contain only attributes from the grouping-list.

Group By Examples

For each rating, find the average age of the sailors

```
SELECT S.rating, AVG (S.age)
FROM Sailors S
GROUP BY S.rating
```

For each rating find the age of the youngest sailor with age \( \geq 18 \)

```
SELECT S.rating, MIN (S.age)
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
```

Conceptual Evaluation

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
GROUP BY grouping-list
```

- The cross-product of relation-list is computed, tuples that fail qualification are discarded, ‘unnecessary’ fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in grouping-list.
- One answer tuple is generated per qualifying group.
- If DISTINCT is specified: drop duplicate answer tuples.
Find the number of reservations for each red boat.

SELECT B.bid, COUNT(*) AS numres
FROM Boats B, Reserves R
WHERE R.bid=B.bid
    AND B.color='red'
GROUP BY B.bid

- Grouping over a join of two relations.

**Queries With GROUP BY and HAVING**

- Use the HAVING clause with the GROUP BY clause to restrict which group-rows are returned in the result set.

**Conceptual Evaluation**

- Form groups as before.
- The group-qualification is then applied to eliminate some groups.
- Expressions in group-qualification must have a single value per group!
- That is, attributes in group-qualification must be arguments of an aggregate op or must also appear in the grouping-list. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.

Find the age of the youngest sailor with age ≥ 18, for each rating with at least 2 such sailors.

SELECT S.rating, MIN(S.age)
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT(*) > 1

**Sort GROUP BY: Naïve Solution**

- The Sort Iterator ensures that all tuples are output in sequence.
- The Aggregate Iterator keeps running info ("transition values") on agg functions in the SELECT list, per group
  - E.g., for COUNT, it keeps count-so-far
  - For SUM, it keeps sum-so-far
  - For AVERAGE it keeps sum-so-far and count-so-far
- As soon as the Aggregate Iterator sees a tuple from a new group:
  - It produces an output for the old group based on the agg function
  - E.g. for AVERAGE it returns (sum-so-far/count-so-far)
  - It resets its running info.
  - It updates the running info with the new tuple's info
An Alternative to Sorting: Hashing!

- Idea:
  - Many of the things we use sort for don’t exploit the order of the sorted data
  - E.g.: forming groups in GROUP BY
  - E.g.: removing duplicates in DISTINCT
  - Often good enough to match all tuples with equal field-values
  - Hashing does this!
    - And may be cheaper than sorting! (Hmmm…!)
  - But how to do it for data sets bigger than memory??

General Idea

- Two phases:
  - Partition: use a hash function \( h_p \) to split tuples into partitions on disk.
    - We know that all matches live in the same partition.
    - Partitions are “spilled” to disk via output buffers
  - ReHash: for each partition on disk, read it into memory and build a main-memory hash table based on a hash function \( h_r \)
    - Then go through each bucket of this hash table to bring together matching tuples

Two Phases

- Partition:
  - B main memory buffers
  - Disk

- ReHash:
  - Disk
  - B main memory buffers

Analysis

- How big of a table can we hash in two passes?
  - B-1 "spill partitions" in Phase 1
  - Each should be no more than B blocks big
  - Answer: \( B(B-1) \).
    - Said differently: We can hash a table of size \( N \) blocks in about \( \sqrt{N} \) space
    - Much like sorting!
- Have a bigger table? Recursive partitioning!
  - In the ReHash phase, if a partition \( b \) is bigger than \( B \), then recurse:
    - pretend that \( b \) is a table we need to hash, run the Partitioning phase on \( b \), and then the ReHash phase on each of its (sub)partitions

Hash GROUP BY: Naïve Solution (similar to the Sort GROUPBY)

- The Hash iterator permutes its input so that all tuples are output in groups
- The Aggregate iterator keeps running info ("transition values") on agg functions in the SELECT list, per group
  - E.g., for COUNT, it keeps count-so-far
  - For SUM, it keeps sum-so-far
  - For AVERAGE it keeps sum-so-far and count-so-far
- When the Aggregate iterator sees a tuple from a new group:
  1. It produces an output for the old group based on the agg function
     - E.g. for AVERAGE it returns \( \text{(sum-so-far/count-so-far)} \)
  2. It resets its running info.
  3. It updates the running info with the new tuple’s info

We Can Do Better!

- Combine the summarization into the hashing process
  - During the ReHash phase, don’t store tuples, store pairs of the form: \( <\text{GroupVals}, \text{TransVals}> \)
  - When we want to insert a new tuple into the hash table
    - If we find a matching GroupVals, just update the TransVals appropriately
    - Else insert a new \( <\text{GroupVals}, \text{TransVals}> \) pair
- What’s the benefit?
  - Q: How many pairs will we have to handle?
    - A: Number of distinct values of GroupVals columns
      - Not the number of tuples!
      - Also probably “narrower” than the tuples
    - Can we play the same trick during sorting?
**Even Better: Hybrid Hashing**

- What if the set of \(<\text{GroupVals}, \text{TransVals}\>) pairs fits in memory
  - It would be a waste to spill it to disk and read it all back!
  - Recall this could be true even if there are tons of tuples!
- Idea: keep a smaller 1st partition in memory during phase 1!
  - Output its stuff at the end of Phase 1.
- Q: how do we choose the number \(k\)?

**A Hash Function for Hybrid Hashing**

- Assume we like the hash-partition function \(h_p\)
- Define \(h_h\) operationally as follows:
  - \(h_h(x) = 1\) if in-memory hashtable is not yet full
  - \(h_h(x) = 1\) if \(x\) is already in the hashtable
  - \(h_h(x) = h_p(x)\) otherwise
- This ensures that:
  - Bucket 1 fits in \(k\) pages of memory
  - If the entire set of distinct hashtable entries is smaller than \(k\), we do no spilling!