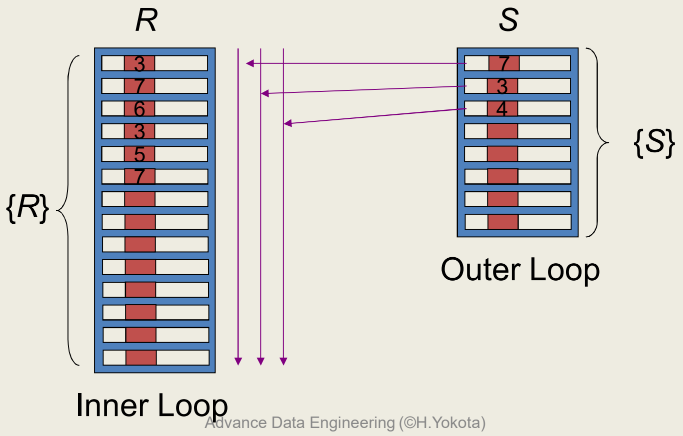
Join Operation

- Join is very time consuming operation
- Here we consider eq-join
- Eq-join is one of the most frequently used relational operation
  - Especially for the third normal form
- There are three algorithms
  - Nested Loop Join
  - Sort Merge Join
  - Hash Join
    - Hash Join cannot be used for θ -Join

Nested Loop Join

- The most naïve algorithm
- Check all tuples in one relation (R) for each tuple in another relation (S)



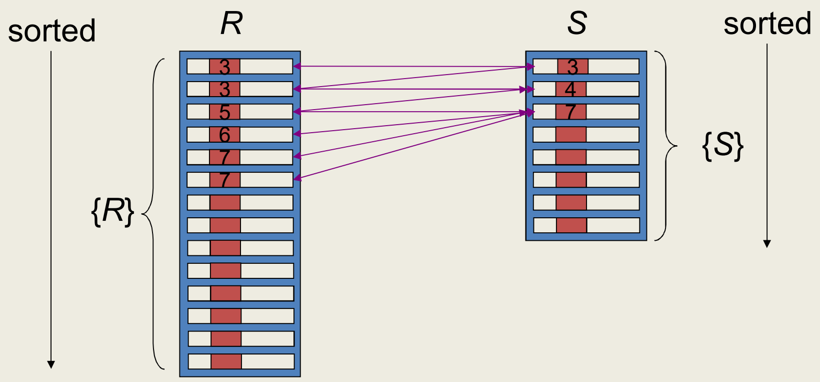
Pseudo Code for Nested Join

```
for (i = 1; i ≤ {S}; i++) {            ← Outer Loop
  f := true;
  get i-th tuple t_S and attribute value v_S in t_S;
  for (j = 1: j ≤ {R}; j++) {          ← Inner Loop
    get j-th tuple t_R and attribute value v_R in t_R;
    if v_R == v_S then output (t_R, t_S);
  }
}
```

- Comparisons : $\{R\}$ x $\{S\}$
- Disk I/Os : $|R|$ x $|S|$

2020/7/16 Advance Data Engineering (©H.Yokota) 152

Sort Merge Join



2020/7/16 Advance Data Engineering (©H.Yokota) 153

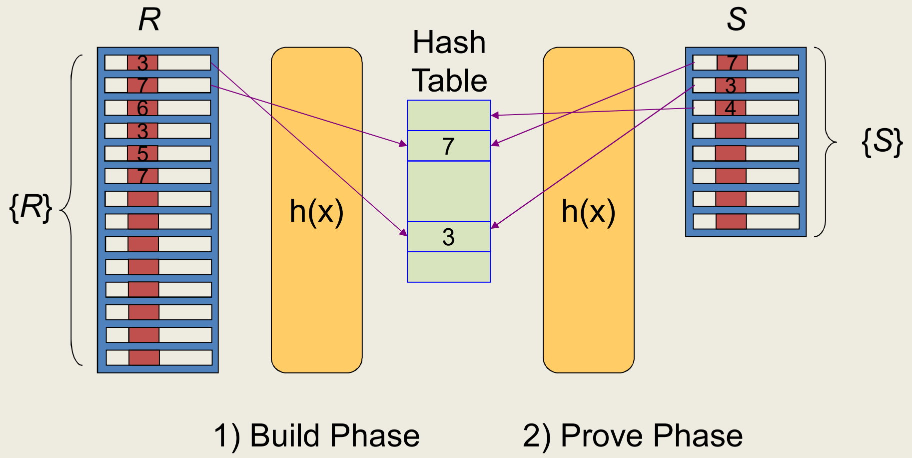
Pseudo Code for Sort Merge Join

i := 1;
j := 1;
while (i ≤ {S} and j ≤ {R}) {
  get i-th tuple t_S and attribute value v_S in t_S;
  get j-th tuple t_R and attribute value v_R in t_R;
  if v_S == v_R then {
    output (t_R, t_S);
    if j ≤ {R} then j := j +1 else i := i + 1;
  }
  else if (v_S < v_R) then i := i + 1;
  else j := j +1
  }
}

Costs for Sort Merge Join

- Comparisons (worst) : {R} + {S} -1
- Costs for sorting are ignored
- Applying 2-way merge sort in advance
  - Comparisons : {R}log{R} + {S}log{S} +1
- Disk I/Os:
  - 2-way merge sort: $2|R|(\log_2|R|+1)$
  - Sort merge join: |R| + |S|
  - Disk I/O : $|R|(2\log|R| + 3) + |S|(2\log|S| +3)$

Hash Join



Question (7-1)

1. Write a pseudo code for memory based hash join with handling hash collision.

Costs for Hash Join

- Assuming all tuples of both *R* and *S* can be place on main memory
  - Consider disk base hash join later
- We have to consider the following costs
  - Applying Hash Function: {*R*} + {*S*}
  - Building the Hash Table: {*R*}
  - Comparisons: {*S*}

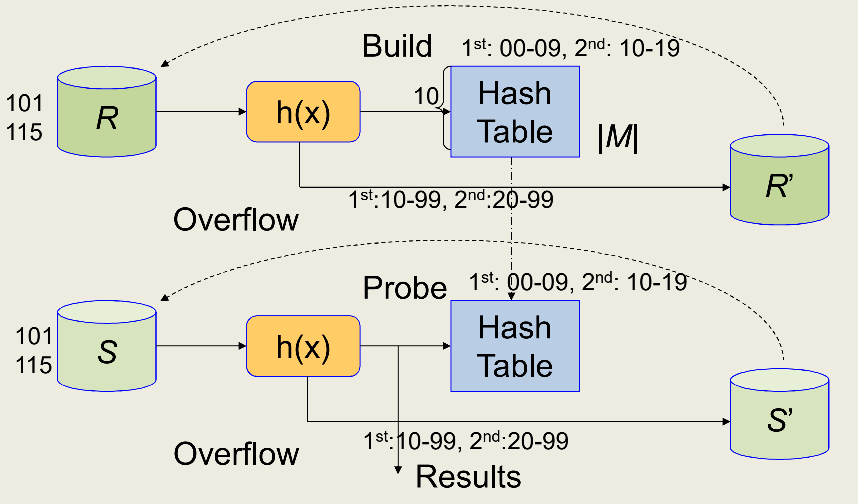
Hash Join with Disk Accesses

- Three Algorithms
  - Simple Hash Join
  - GRACE Hash Join
  - Hybrid Hash Join
- Assumptions for cost estimation
  - Memory size for hash table: $|M|$
  - The distribution of attribute value is flat
  - $|R| > |S|$

Simple Hash Join

- Restore tuples that cannot be placed in main memory hash table
  - Repeat until no overflow
  - Both *R* and *S* (because of the identical hash table)
- The number of repeat (Expectation):
  - $L = |R|/|M|$ (≥ 1)

Illustration of Simple Hash Join



Example of Overflow Control

- Suppose an attribute storing integer value
- Cardinality of R is 100
- The hash table |M| can keep 10 tuples of R
- Apply hash function h(X)=mod(X,100)
  - It generates two-digit numbers
  - Lower digit is used to choose an entry of the hash table
  - Higher digit is used to decide overflow tuples
- The first iteration: tuples of R, having higher digit of hash value is 0, are used to build the hash table, and others become overflow
- The second iteration: tuples having higher digit ≠ 1 becomes overflow
- And so on
- Iteration count becomes 10 or more

Cost for Simple Hash Join (1)

- In the first loop
  - Applying Hash Function: $\{R\} + \{S\}$
  - Comparison: $\{S\} / L$
  - Disk I/O for read: $|R| + |S|$
  - Disk I/O for write: $|R| - |M| + |S| - |S| / L$
  - or $(|R| + |S|) - (|R| + |S|) / L$
  - from $|M| = |R| / L$

Cost for Simple Hash Join (2)

- In the i-th loop (i = 2, …, L)
  - Applying Hash Function:
    $\{R\} - (i - 1) \times \{R\} / L + \{S\} - (i - 1) \times \{S\} / L$
  - Comparison: $\{S\} / L$
  - Disk I/O for read:
    $(|R| + |S|) - (i - 1) \times (|R| + |S|) / L$
  - Disk I/O for write:
    $(|R| + |S|) - i \times (|R| + |S|) / L$

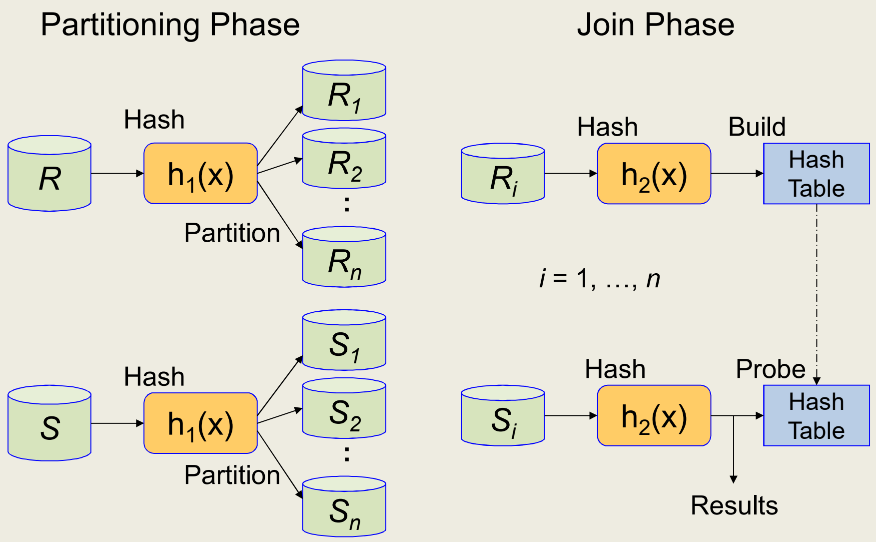
2020/7/16 Advance Data Engineering (©H.Yokota) 164

Cost for Simple Hash Join (3)

- Summation from the first to L-th (last) loop
  - Applying Hash Function:
    $L \times (\{R\} + \{S\}) – (L \times (L - 1)/2 \times (\{R\}/L + \{S\}/L))$
    $= (L + 1) / 2 \times (\{R\} + \{S\})$
    $= (|R| + |M|) \times (\{R\} + \{S\}) / 2|M|$
  - Comparison: $(\{S\} / L) \times L = \{S\}$
  - Disk I/O :
    $2L \text{ x } (|R|+|S|) - (L \text{ x } (L+1)) / 2$
    $+ L \text{ x } (L-1) / 2) \text{ x } (|R|+|S|) / L$
    $= L \times (|R| + |S|)$
    $= (|R| \times (|R| + |S|)) / |M|$

2020/7/16 Advance Data Engineering (©H.Yokota) 165

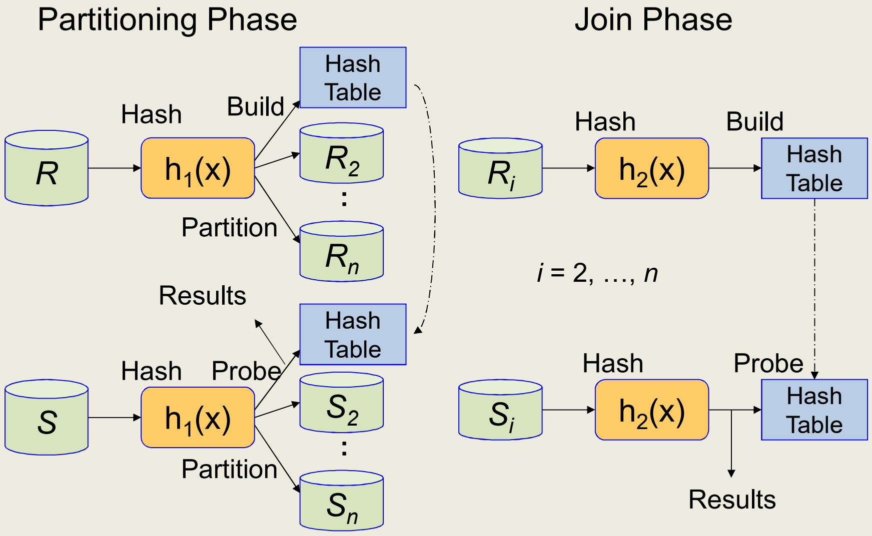
Illustration of GRACE Hash Join



Cost for GRACE Hash Join

- Disk I/O : 3 (|R| + |S|)
  - Read whole |R| and |S| and write them in the partition phase, and read them again in the join phase
- Comparison : $\Sigma S_i = (\{S\} / L) \times L = \{S\}$
  - Assume Hash Join in Join Phase
- Applying Hash Function:
  - Partitioning Phase: $\{R\} + \{S\}$
  - Join Phase: $\Sigma(\{R_i\} + \{S_i\}) = \{R\} + \{S\}$
  - Whole: $2 (\{R\} + \{S\})$

Illustration of Hybrid Hash Join



Cost for Hybrid Hash Join

- Disk I/O :
  - $3\,(|R| + |S|) - 2\,(|R| + |S|)\,/\,L$
    $= (3 - 2|M|\,/\,|R|\,) \times (|R| + |S|)$
- Comparison and Applying Hash Function
  - The same as GRACE Hash Join

Comparison of Hash Join Algorithm

	Disk I/O	Applying Hash	Comparison
Simple	$\frac{\|R\|}{\|M\|}(\|R\|+\|S\|)$	$\frac{\|R\|+\|M\|}{2\|M\|}(\{R\}+\{S\})$	$\{S\}$
GRACE	$3\ (\|R\|+\|S\|)$	$2\ (\{R\}+\{S\})$	$\{S\}$
Hybrid	$(3-2\frac{\|M\|}{\|R\|})(\|R\|+\|S\|)$	$2\ (\{R\}+\{S\})$	$\{S\}$

Question (7-2)

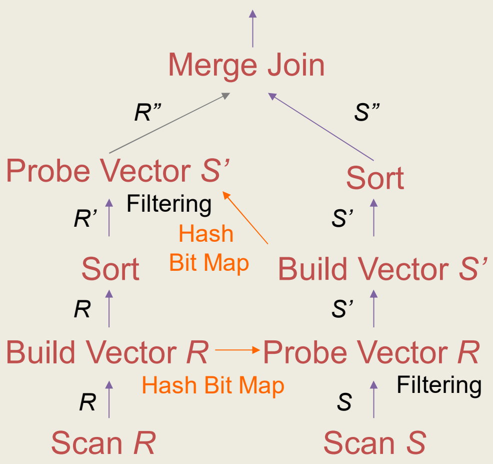
- Assume that the page numbers of relations R and S are 300,000 and 30,000 pages respectively, and the hash table is built by the relation S.
  1. Estimate the disk I/O counts for Simple, GRACE, and Hybrid hash join respectively, under the condition that 100 pages can be used for the hash table.
  2. Estimate the page numbers for the hash table to make the Simple hash join superior to GRACE hash join.
  3. Estimate the page numbers for the hash table to make the Simple hash join superior to Hybrid hash join.

Optimization of Join (1)

- Star Query (Oracle 7)
  - Derive Cartesian Products among Dimension Tables
    - to avoid handling the huge Fact Table
  - Tradeoff between the cost for Cartesian Product and that for handling the Fact Table
  - The dimension tables can be filtered by conditions in advance
    - e.g. A4 Laptop, Aug., etc

Optimization of Join (2)

- Bit Vector Filtering
  - Bloom Filtering
  - Make each entry of the Hash Table 1 bit
    - Only Existence (Allow collisions)
  - Bit Vector can be place in main memory
  - This filtering can apply both sort merge and hash join

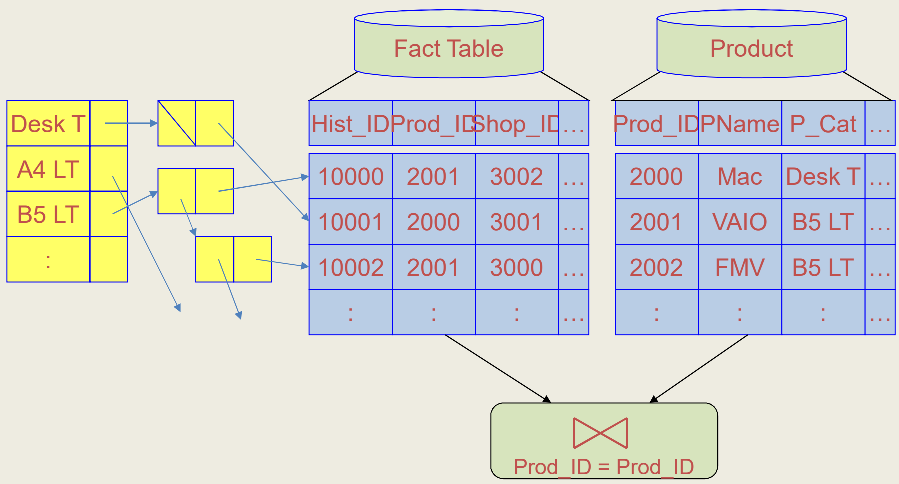


Join Index (1)

- Assuming join attributes for relations *R* and *S* are *R.A* and *S.B*, respectively, an index from the other attribute of *R.C* of the relation *R* is a Join Index.
  - Structure: Inverted File or B^+-tree
  - Collisions for one entry: Inverted List or in Inverted File / B-tree

2020/7/16 Advance Data Engineering (©H.Yokota) 174

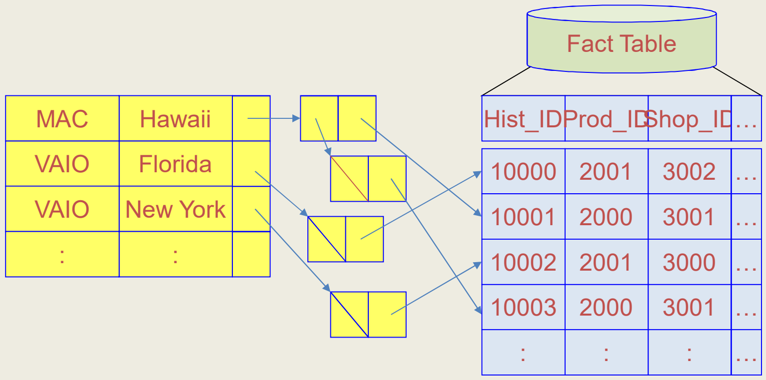
Join Index (2)



2020/7/16 Advance Data Engineering (©H.Yokota) 175

Multidimensional Join Index

- Join Index for combination of multiple attributes



2020/7/16 Advance Data Engineering (©H.Yokota) 176

Bitmap Index

- Toggle 0 or 1 by existence of a value of the attribute
- When the variety of values is small, it provides good space efficiency
  - Bitmap can be place in main memory
  - Bitmap can be calculated as an array
    - AND/OR operation can be used for filtering

```
for (i = 0; i < len(B1);  i++)
  B3[i] = B1[i] & B2[i];
```

- On the other hand, large variety of attribute values make the space efficiency worse.
  - combination with value index by segmentation is proposed

2020/7/16 Advance Data Engineering (©H.Yokota) 177

An Example of Bitmap Index

Condition: Prod_ID=2001 and Shop_ID=3001

Fact Table

Hist_ID	Prod_ID	Shop_ID	…
10000	2001	3002	…
10001	2000	3001	…
10002	2001	3000	…
10003	2020	3002	…
10004	2003	3010	…
10002	2001	3001	…
10003	2002	3011	…
:	:	:	…

Prod_ID Bitmap Index

2000	2001	2002	2003
0	1	0	0
1	0	0	0
0	1	0	0
0	0	0	0
0	0	0	1
0	1	0	0
0	0	1	0
…	…	…	…

Shop_ID Bitmap Index

3000	3001	3002	3003
0	0	1	0
0	1	0	0
1	0	0	0
0	0	1	0
0	0	0	0
0	1	0	0
0	0	0	0
…	…	…	…

Results

0
0
0
0
0
1
0
…

2020/7/16 Advance Data Engineering (©H.Yokota) 178

Semi-Join Operation

- A kind of eq-join, but attributes in the result relation belong to only one relation

Product ID	Product Type	Price
P001	Laptop PC	1,500
P002	Laptop PC	2,000
P003	Desktop PC	2,000

$\ltimes$

Product ID	Stocks
P001	10
P003	5

⇓

Product ID	Product Type	Price
P001	Laptop PC	1,500
P003	Desktop PC	2,000

2020/7/16 Advance Data Engineering (©H.Yokota) 179

Anti-Semi-Join Operation

- Similar to semi-join, but generating tuples which does not much with another relation

Product ID	Product Type	Price
P001	Laptop PC	1,500
P002	Laptop PC	2,000
P003	Desktop PC	2,000

$\overline{\ltimes}$

Product ID	Stocks
P001	10
P003	5

Product ID	Product Type	Price
P002	Laptop PC	2,000

Costs of Semi-Join & Anti-Semi-Join

- Cost of Semi-Join are basically same as ordinary join operations
  - For the case of hash join, the size of hash table can be reduced
- Implementation of anti-semi-join is also similar to semi-join operation except outputting unmuched tuples instead of muched tuples
  - The cost of anti-semi-join is equal to semi join
- These operations are related to set operations
- Semi-Join is also related to distributed database

Implementation of Set Operations

- Intersection ($R \cap S$)
  - Apply the semi-join targeting all attributes as conditions
  - Costs is equal to one for a join operation
- Difference ($R - S$)
  - Apply the anti-semi-join targeting all attributes
  - Costs is also equal to one for a join operation
- Union ($R \cup S$)
  - Apply difference operation to derive (R-S) and concatenate (R-S) with S
  - Or concatenate R with S and eliminate duplications

Division ÷

- Division produce a relation that consists of the set of tuples from R defined over the attribute C that match the combination of every tuple in S, where C is the set of attributes that are in R but not in S

R

Product ID	Salesman
P001	John
P001	Alice
P001	Bill
P002	John
P002	Bill
P003	John
P003	Bill
P004	Alice
P004	Bill

÷

S

Product ID
P001
P002
P003

R[ProductID] ÷ S[ProductID]
Salesman who sold all products listed in relation S

Salesman
Bill
John

Implementation of $R[X] \div S[X]$ (1)

- Sort Base Direct Method
  - Sort *R* by *X* as minor and *C* as major
  - Check *X* of *R* with *S* from top of the same value *C*

R

Product ID (*X*)	Salesman (*C*)
P001	John
P001	Alice
P001	Bill
P002	John
P002	Bill
P003	John
P003	Bill
P004	Alice
P004	Bill

Sort ⇒

Product ID (minor)	Salesman (major)
P001	Alice
P004	Alice
P001	Bill
P002	Bill
P003	Bill
P004	Bill
P001	John
P002	John
P003	John

÷

S

Product ID
P001
P002
P003

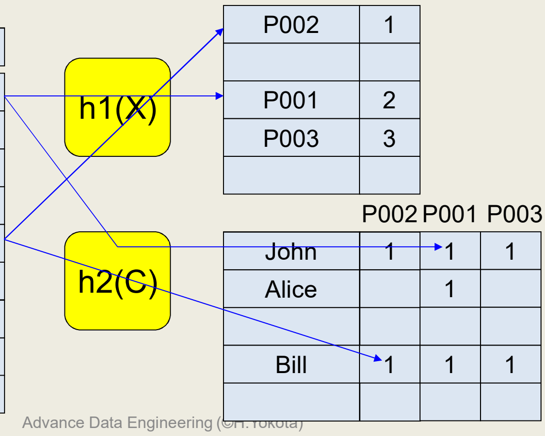
results

Salesman
Bill
John

Implementation of $R[X] \div S[X]$ (2)

- Hash Base Direct Method
  - Prepare two hash functions h1(X) and h2(C)
  - h1(X) is used to derive entry # in the indicated h2(C) and set 1

Product ID (X)	Salesman (C)
P001	John
P001	Alice
P001	Bill
P002	John
P002	Bill
P003	John
P003	Bill
P004	Alice
P004	Bill



Implementation of $R[X] \div S[X]$ (3)

- Aggregate Function based Indirect Method
  - Apply semi-join in advance, then count tuples for each group of C
  - If any group have count as same as the cardinality of S, then output

R

Product ID	Salesman
P001	John
P001	Alice
P001	Bill
P002	John
P002	Bill
P003	John
P003	Bill
P004	Alice
P004	Bill

⋉

S

Product ID
P001
P002
P003

⇒

R'

Product ID	Salesman
P001	John
P001	Alice
P001	Bill
P002	John
P002	Bill
P003	John
P003	Bill

John	3
Alice	1
Bill	3

Question (6-3)
Consider an SQL sentence
to derive the results

2020/7/16

Cost for Group-by and Aggregation Operations (1)

- Nested Loop base
  - Scan all tuples to search tuples in the same group, and then calculate COUNT, SUM, AVG, MAX, MIN
    - $(\{R\} - 1) + (\{R\} - 2) + \ldots + 1$
    - Comparison : $\{R\}(\{R\} - 1) / 2$
  - Write the result into disk at the last
    - Disk I/O: $|R|(|R| + 1) / 2$

2020/7/16 Advance Data Engineering (©H.Yokota) 187

Cost for Group-by and Aggregation Operations (2)

- Sort base
  - At first sort all tuples and scan from the top
    - Assume Merge Sort
  - Cost for sort and one scan at the last
    - Comparison : $\{R\}(\log\{R\}+1)$
  - Disk I/O : $|R|(2 \log|R|+1)$
- Hash base
  - Divide tuples by a hash function
    - Apply the aggregate function for each hash bucket
  - If there is no hash collision
    - Applying Hash Function, Comparison: $\{R\}$
    - Disk I/O : $|R|$

2020/7/16 Advance Data Engineering (©H.Yokota) 188

Indexing for Aggregation Functions

- Bit-Sliced Index
  - Divide each bit of binary expression of an integer value
    - A list of each bit in tuples is treated as a bitmap index
    - Each bit-slice can be placed on main memory
    - Calculate SUM or AVG for each bit-slice and summarize them
  - Adopted by Sybase IQ, CCA Model 204

2020/7/16 Advance Data Engineering (©H.Yokota) 189

Indexing for Aggregation Functions

- Projection Index
  - Derive an attribute (Projection), and access by its location.
    - Reduce Disk I/O by reducing amount of data
  - Suited for complex aggregate functions on the attribute
    - Derive SUM for the results of some calculation
  - Adopted by Sybase IQ

Comparison on the AF Indices

Aggregate	Value-list	Bit-Sliced	Projection
Max, Min	Best	Slow	Slow
SUM, AVG	Not Bad	Best	Good
SUM(A1*(1-A2))	Very Slow	Very Slow	Best
Narrow Range	Best	Good	Good
Wide Range	Not Bad	Best	Good