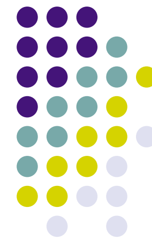
CMSC424: Database Design

Instructor: Amol Deshpande
amol@cs.umd.edu



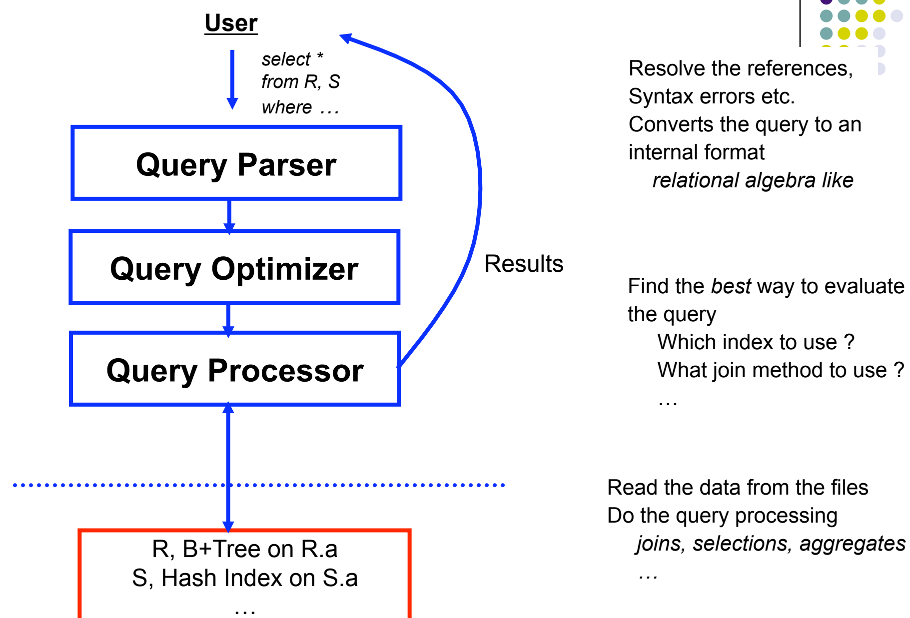
Databases

- Data Models
  - Conceptual representation of the data
- Data Retrieval
  - How to ask questions of the database
  - How to answer those questions
- Data Storage
  - How/where to store data, how to access it
- Data Integrity
  - Manage crashes, concurrency
  - Manage semantic inconsistencies

Query Processing

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

Overview



Resolve the references,
Syntax errors etc.
Converts the query to an internal format
relational algebra like

Find the *best* way to evaluate the query
Which index to use ?
What join method to use ?
…

Read the data from the files
Do the query processing
joins, selections, aggregates
…

“Cost”

- Complicated to compute
- We will focus on disk:
  - Number of I/Os ?
    - Not sufficient
    - Number of seeks matters a lot… why ?
  - t_T – time to transfer one block
  - t_S – time for one seek
  - Cost for *b* block transfers plus *S* seeks

    $b * t_T + S * t_S$
  - Measured in *seconds*

Query Processing

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

Selection Operation

- select * from person where SSN = “123”
- Option 1: Sequential Scan
  - Read the relation start to end and look for “123”
    - Can always be used (not true for the other options)
  - Cost ?
    - *Let* b_r *= Number of relation blocks*
    - Then:
      - 1 seek and br block transfers
    - So:
      - $t_S + b_r * t_T$ *sec*
    - *Improvements:*
      - *If SSN is a key, then can stop when found*
        - *So on average,* $b_r/2$ *blocks accessed*

Selection Operation

- select * from person where SSN = “123”
- Option 2 : Binary Search:
  - Pre-condition:
    - *The relation is sorted on SSN*
    - *Selection condition is an equality*
      - E.g. can’t apply to “*Name like ‘%424%’*”
  - Do binary search
    - Cost of finding the *first* tuple that matches
      - $\lceil \log_2(b_r) \rceil * (t_T + t_S)$
      - All I/Os are random, so need a seek for all
        - The last few are closeby, but we ignore such small effects
  - Not quite: What if 10000 tuples match the condition ?
    - Incurs additional cost

Selection Operation

- select * from person where SSN = “123”
- Option 3 : Use Index
  - Pre-condition:
    - *An appropriate index must exist*
  - Use the index
    - Find the first leaf page that contains the search key
    - Retrieve all the tuples that match by following the pointers
      - If primary index, the relation is sorted by the search key
        - Go to the relation and read blocks sequentially
      - If secondary index, must follow all pointers using the index

Selection w/ B+-Tree Indexes

	cost of finding the first leaf	cost of retrieving the tuples
primary index, candidate key, equality	$h_i * (t_T + t_S)$	$1 * (t_T + t_S)$
primary index, not a key, equality	$h_i * (t_T + t_S)$	$1 * (t_T + t_S) + (b – 1) * t_T$ *Note: primary == sorted* *b = number of pages that contain the matches*
secondary index, candidate key, equality	$h_i * (t_T + t_S)$	$1 * (t_T + t_S)$
secondary index, not a key, equality	$h_i * (t_T + t_S)$	$n * (t_T + t_S)$ *n = number of records that match* This can be bad

h_i = height of the index

Selection Operation

- Selections involving ranges
  - *select * from accounts where balance > 100000*
  - *select * from matches where matchdate between '10/20/06' and '10/30/06'*
  - Option 1: Sequential scan
  - Option 2: Using an appropriate index
    - Can't use hash indexes for this purpose
    - Cost formulas:
      - Range queries == "equality" on "non-key" attributes
      - So rows 3 and 5 in the preceding page

Selection Operation

- Complex selections
  - Conjunctive: *select * from accounts where balance > 100000 and SSN = "123"*
  - Disjunctive: *select * from accounts where balance > 100000 or SSN = "123"*
  - Option 1: Sequential scan
  - Option 2 *(Conjunctive only)*: Using an appropriate index *on one of the conditions*
    - E.g. Use SSN index to evaluate SSN = "123". Apply the second condtion to the tuples that match
    - Or do the other way around (if index on balance exists)
    - Which is better ?
  - Option 3 *(Conjunctive only)* : Choose a multi-key index
    - Not commonly available

Selection Operation

- Complex selections
  - Conjunctive: *select * from accounts where balance > 100000 and SSN = "123"*
  - Disjunctive: *select * from accounts where balance > 100000 or SSN = "123"*
  - Option 4: Conjunction or disjunction of *record identifiers*
    - Use indexes to find all RIDs that match each of the conditions
    - Do an intersection (for conjunction) or a union (for disjunction)
    - Sort the records and fetch them in one shot
    - Called "Index-ANDing" or "Index-ORing"
  - Heavily used in commercial systems

Query Processing

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

Join

- *select * from R, S where R.a = S.a*
  - Called an "*equi-join*"
- *select * from R, S where |R.a – S.a | < 0.5*
  - *Not an "equi-join"*
- Option 1: Nested-loops

  for each tuple r in R

  for each tuple s in S

  check if r.a = s.a (or whether |r.a – s.a| < 0.5)
- Can be used for any join condition
  - As opposed to some algorithms we will see later
- R called *outer relation*
- S called *inner relation*

Nested-loops Join

- Cost ? Depends on the actual values of parameters, especially memory
- b_r, b_s → *Number of blocks of R and S*
- n_r, n_s → *Number of tuples of R and S*
- Case 1: Minimum memory required = 3 blocks
  - One to hold the current *R* block, one for current S block, one for the result being produced
  - Blocks transferred:
    - Must scan *R* tuples once: b_r
    - For each *R* tuple, must scan *S*: $n_r * b_s$
  - Seeks ?
    - $n_r + b_r$

Nested-loops Join

- Case 1: Minimum memory required = 3 blocks
  - Blocks transferred: $n_r * b_s + b_r$
  - Seeks: $n_r + b_r$
- Example:
  - Number of records -- *R:* $n_r = 10,000$, *S:* $n_s = 5000$
  - Number of blocks -- *R:* $b_r = 400$, *S:* $b_s = 100$
- Then:
  - blocks transferred: 10000 * 100 + 400 = 1,000,400
  - seeks: 10400
- What if we were to switch R and S ?
  - 2,000,100 block transfers, 5100 seeks
- Matters

Nested-loops Join

- Case 2: *S* fits in memory
  - Blocks transferred: $b_s + b_r$
  - Seeks: *2*
- Example:
  - Number of records -- *R:* $n_r = 10,000$, *S:* $n_s = 5000$
  - Number of blocks -- *R:* $b_r = 400$, *S:* $b_s = 100$
- Then:
  - blocks transferred: 400 + 100 = 500
  - seeks: 2
- This is orders of magnitude difference

Block Nested-loops Join

- Simple modification to “nested-loops join”
  - Block at a time

```
for each block Br in R
  for each block Bs in S
    for each tuple r in Br
      for each tuple s in Bs
        check if r.a = s.a (or whether |r.a – s.a| < 0.5)
```

- Case 1: Minimum memory required = 3 blocks
  - Blocks transferred: $b_r * b_s + b_r$
  - Seeks: 2 * b_r
- For the example:
  - blocks: 40400, seeks: 800

Block Nested-loops Join

- Case 1: Minimum memory required = 3 blocks
  - Blocks transferred: $b_r * b_s + b_r$
  - Seeks: 2 * b_r
- Case 2: S fits in memory
  - Blocks transferred: $b_s + b_r$
  - Seeks: *2*
- What about in between ?
  - Say there are 50 blocks, but *S* is 100 blocks
  - Why not use all the memory that we can…

Block Nested-loops Join

- Case 3: 50 blocks (S = 100 blocks) ?

  for each group of 48 blocks in R

  for each block B_s in S

  for each tuple r in the group of 48 blocks

  for each tuple s in Bs

  check if r.a = s.a (or whether |r.a – s.a| < 0.5)

- Why is this good ?
  - We only have to read *S* a total of $b_r/48$ times (instead of b_r times)
  - Blocks transferred: $b_r * b_s / 48 + b_r$
  - Seeks: $2 * b_r / 48$

Index Nested-loops Join

- *select * from R, S where R.a = S.a*
  - Called an "*equi-join*"
- Nested-loops

  for each tuple r in R

  for each tuple s in S

  check if r.a = s.a (or whether |r.a – s.a| < 0.5)

- Suppose there is an index on *S.a*
- *Why not use the index instead of the inner loop ?*

  for each tuple r in R

  use the index to find S tuples with S.a = r.a

Index Nested-loops Join

- *select * from R, S where R.a = S.a*
  - Called an "*equi-join*"
- *Why not use the index instead of the inner loop ?*

  for each tuple r in R

  use the index to find S tuples with S.a = r.a
- Cost of the join:
  - $b_r (t_T + t_S) + n_r * c$
  - *c == the cost of index access*
    - *Computed using the formulas discussed earlier*

Index Nested-loops Join

- Restricted applicability
  - An appropriate index must exist
  - What about |*R.a – S.a*| < *5* ?
- Great for queries with joins and selections

  *select **

  from accounts, customers

  where accounts.customer-SSN = customers.customer-SSN and

  accounts.acct-number = "A-101"
- Only need to access one SSN from the other relation

So far…

- Block Nested-loops join
  - Can always be applied irrespective of the join condition
  - If the smaller relation fits in memory, then cost:
    - $b_r + b_s$
    - This is the best we can hope if we have to read the relations once each
  - CPU cost of the inner loop is high
  - Typically used when the smaller relation is really small (few tuples) and index nested-loops can't be used
- 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

Hash Join

- Case 1: Smaller relation (S) fits in memory
- Nested-loops join:

  for each tuple r in R

  for each tuple s in S

  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:

  read S in memory and build a hash index on it

  for each tuple r in R

  use the hash index on S to find tuples such that S.a = r.a

Hash Join

- Case 1: Smaller relation *(S)* fits in memory
- Hash join:

  read S in memory and build a hash index on it

  for each tuple r in R

  use the hash index on S 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)
  - Performs 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
- Two "phases"
- Phase 1:
  - Read the relation *R* block by block and partition it using a hash function, *h1(a)*
    - Create one partition for each possible value of *h1(a)*
  - Write the partitions to disk
    - R gets partitioned into R1, R2, …, Rk
  - Similarly, read and partition S, and write partitions S1, S2, …, Sk to disk
  - Only requirement:
    - Each S partition fits in memory

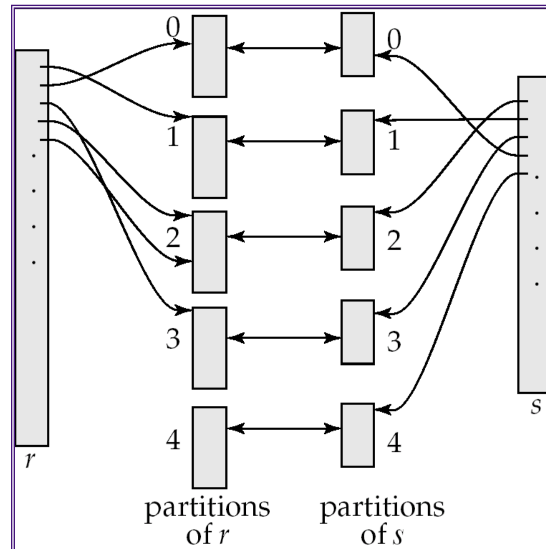
Hash Join

- <u>Case 2: Smaller relation *(S)* doesn't fit in memory</u>
- Two "phases"
- Phase 2:
  - Read S1 into memory, and bulid a hash index on it (S1 fits in memory)
    - Using a different hash function, $h_2(a)$
  - Read R1 block by block, and use the hash index to find matches.
  - Repeat for S2, R2, and so on.

Hash Join

- <u>Case 2: Smaller relation *(S)* doesn't fit in memory</u>
- Two "phases":
- Phase 1:
  - Partition the relations using one hash function, $h_1(a)$
- Phase 2:
  - Read S_i into memory, and bulid 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 ?
  - $3(b_r + b_s) + 4 * n_h$ block transfers + $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)$ seeks
    - Where b_b is the size of each output buffer
  - Much better than Nested-loops join under the same conditions

Hash Join



Hash Join: Issues

- How to guarantee that the partitions of *S* all fit in memory ?
  - Say S = 10000 blocks, Memory = M = 100 blocks
  - Use a hash function that hashes to 100 different values ?
    - Eg. *h1(a) = a % 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 *100*f* different values
    - *f* is called fudge factor, typically around 1.2
    - So we may consider *h1(a) = a % 120.*
    - This is okay IF *a* is uniformly distributed

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| * f)* blocks of memory
- So if S is 10000 blocks, and f = 1.2, need 110 blocks of memory
- If memory = 10000 blocks = 10000 * 4 KB = 40MB (not unreasonable)
  - Then, S can be as large as *10000*10000/1.2* blocks = *333 GB*

Hash Join: Issues

- What if we don't have enough memory ?
  - *Recursive Partitioning*
  - Rarely used, but can be done
- What if the hash function turns out to be bad ?
  - We used *h1(a) = a % 100*
  - Turns out all *values of a* are multiple of 100
  - So *h1(a) is always = 0*
- Called *hash-table overflow*
- Overflow avoidance: Use a good hash function
- Overflow resolution: Repartition using a different hash function

Hybrid Hash Join

- Motivation:
  - R = 10000 blocks, S = 101 blocks, M = 100 blocks
  - So *S* doesn't fit in memory
- Phase 1:
  - Use two partitions
    - Read 10000 blocks of R, write partitions R1 and R2 to disk
    - Read 101 blocks of S, write partitions S1 and S2 to disk
  - Only need 3 blocks for this (so remaining 97 blocks are being wasted)
- Phase 2:
  - Read S1, build hash index, read R1 and probe
  - Read S2, build hash index, read R2 and probe
- Alternative:
  - Don't write partition *S1* to disk, just keep it memory – there is enough free memory for that

Hybrid Hash Join

- Motivation:
  - R = 10000 blocks, S = 101 blocks, M = 100 blocks
  - So *S* doesn't fit in memory
- Alternative:
  - Don't write partition *S1* to disk, just keep it memory – there is enough free memory
- Steps:
  - Use a hash function such that *S1 = 90 blocks, and S2 = 10 blocks*
  - Read S1, and partition it
    - Write S2 to disk
    - Keep S1 in memory, and build a hash table on it
  - Read R1, and partition it
    - Write R2 to disk
    - Probe using R1 directly into the hash table
  - Saves huge amounts of I/O

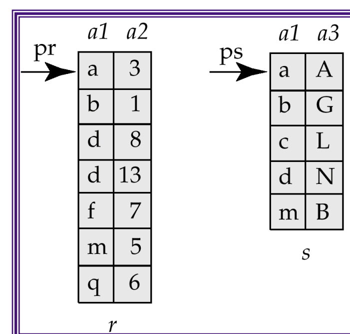
So far…

- 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
- Hash joins
  - Join algorithm of choice when the relations are large
  - Only applies to equi-joins (since it is hash-based)
- Hybrid hash join
  - An optimization on hash join that is always implemented

Merge-Join (Sort-merge join)

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

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 *a1*
  - 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
- Hybrid hash join
  - An optimization on hash join that is always implemented
- 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
- Sorting
- Other operators
- Putting it all together…

Sorting

- Commonly required for many operations
  - Duplicate elimination, group by's, sort-merge join
  - Queries may have ASC or DSC in the query
- One option:
  - Read the lowest level of the index
    - May be enough in many cases
  - But if relation not sorted, this leads to too many random accesses
- If relation small enough…
  - Read in memory, use quick sort (qsort() in C)
- What if relation too large to fit in memory ?
  - External sort-merge

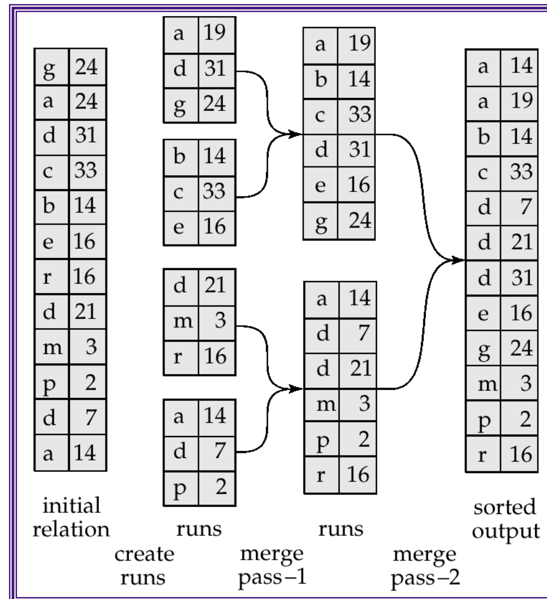
External sort-merge

- Divide and Conquer !!
- Let *M* denote the memory size (in blocks)
- Phase 1:
  - Read first M blocks of relation, sort, and write it to disk
  - Read the next M blocks, sort, and write to disk …
  - Say we have to do this “N” times
  - Result: *N* sorted runs of size *M* blocks each
- Phase 2:
  - Merge the *N* runs (*N-way merge)*
  - Can do it in one shot if *N* < *M*

External sort-merge

- Phase 1:
  - Create *sorted runs of size M* each
  - Result: *N* sorted runs of size *M* blocks each
- Phase 2:
  - Merge the *N* runs (*N-way merge)*
  - Can do it in one shot if *N* < *M*
- What if *N* > *M ?*
  - Do it recursively
  - Not expected to happen
  - If *M* = 1000 blocks = 4MB (assuming blocks of 4KB each)
    - Can sort: 4000MB = 4GB of data

Example: External Sorting Using Sort-Merge



External Merge Sort (Cont.)

- Cost analysis:
  - Total number of merge passes required: $\lceil \log_{M-1}(b_r/M) \rceil$.
  - Disk accesses for initial run creation as well as in each pass is $2b_r$
    - for final pass, we don't count write cost
      - we ignore final write cost for all operations since the output of an operation may be sent to the parent operation without being written to disk

  Thus total number of disk accesses for external sorting:

  $b_r (2 \lceil \log_{M-1}(b_r / M) \rceil + 1)$

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
- Steps:
  - Create a hash table on *a,* and keep the *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
- Steps:
  - Sort *R* on *a*
  - Now all tuples in a single group are contigous
  - Read tuples of *R (sorted)* one by one and compute the aggregates

Duplicate Elimination

select distinct a
from R ;

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

Set operations

*(select * from R) union (select * from S) ;*
*(select * from R) intersect (select * from S) ;*
*(select * from R) union all (select * from S) ;*
*(select * from R) intersect all (select * from S) ;*

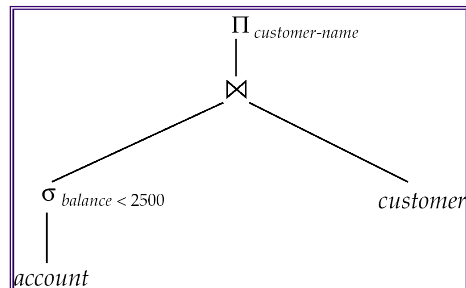
- Remember the rules about duplicates
- "union all": just append the tuples of *R and S*
- *"*union": append the tuples of *R and S, and do duplicate elimination*
- *"intersection"*: similar to joins
  - Find tuples of R and S that are identical on all attributes
  - Can use *hash-based or sort-based 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
  - Skip the step of going to disk
  - Usually faster, but requires more memory
  - Also not always possible..
    - E.g. Sort-Merge Join
  - Harder to reason about

Materialization

- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be quite high
  - Our cost formulas for operations ignore cost of writing results to disk, so
    - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk
- Double buffering: use two output buffers for each operation, when one is full write it to disk, while the other is getting filled
  - Allows overlap of disk writes with computation and reduces execution time

Pipelining

- Evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of

$$\sigma_{balance<2500}(account)$$

  - instead, pass tuples directly to the join.. Similarly, don't store result of join, pass tuples directly to projection.
- Much cheaper: no need to store a temporary relation to disk.
- Requires higher amount of memory
  - All operations are executing at the same time (say as processes)
- Somewhat limited applicability
- A "blocking" operation: An operation that has to consume entire input before it starts producing output tuples

Pipelining

- Need operators that generate output tuples while receiving tuples from their inputs
  - Selection: Usually yes.
  - Sort: NO. The sort operation is blocking
  - Sort-merge join: The final (merge) phase can be pipelined
  - Hash join: The partitioning phase is blocking; the second phase can be pipelined
  - Aggregates: Typically no. Need to wait for the entire input before producing output
    - However, there are tricks you can play here
  - Duplicate elimination: Since it requires sort, the final merge phase could be pipelined
  - Set operations: see duplicate elimination

Pipelining: Demand-driven

- Iterator Interface
  - Each operator implements:
    - *init(): Initialize the state (sometimes called open())*
    - *get_next(): get the next tuple from the operator*
    - *close(): Finish and clean up*
  - Sequential Scan:
    - *init():* open the file
    - *get_next():* get the next tuple from file
    - *close():* close the file
- Execute by repeatadly calling *get_next()* at the root
  - root calls get_next() on its children, the children call get_next() on their children etc…
- The operators need to maintain internal state so they know what to do when the parent calls get_next()

Hash-Join Iterator Interface

- open():
  - Call open() on the left and the right children
  - Decide if partitioning is needed (if size of smaller relation > allotted memory)
  - Create a hash table
- get_next(): (((assuming no partitioning needed)))
  - First call:
    - Get all tuples from the right child one by one (using get_next()), and insert them into the hash table
    - Read the first tuple from the left child (using get_next())
  - All calls:
    - Probe into the hash table using the "current" tuple from the left child
      - Read a new tuple from left child if needed
    - Return exactly "one result"
      - Must keep track if more results need to be returned for that tuple

Hash-Join Iterator Interface

- close():
  - Call close() on the left and the right children
  - Delete the hash table, other intermediate state etc…
- get_next(): (((partitioning needed)))
  - First call:
    - Get all tuples from both children and create the partitions on disk
    - Read the first partition for the right child and populate the hash table
    - Read the first tuple from the left child from appropriate partition
  - All calls:
    - Once a partition is finished, clear the hash table, read in a new partition from the right child, and re-populate the hash table
  - Not that much more complicated
- Take a look at the postgreSQL codebase

Pipelining (Cont.)

- In produce-driven or **eager** pipelining
  - Operators produce tuples eagerly and pass them up to their parents
    - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
    - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
  - System schedules operations that have space in output buffer and can process more input tuples

Recap: Query Processing

- Many, many ways to implement the relational operations
  - Numerous more used in practice
  - Especially in data warehouses which handles TBs (even PBs) of data
- However, consider how complex SQL is and how much you can do with it
  - Compared to that, this isn't much
- Most of it is very nicely modular
  - Especially through use of the *iterator()* interface
  - Can plug in new operators quite easily
  - PostgreSQL query processing codebase very easy to read and modify
- Having so many operators does complicate the codebase and the query optimizer though
  - But needed for performance