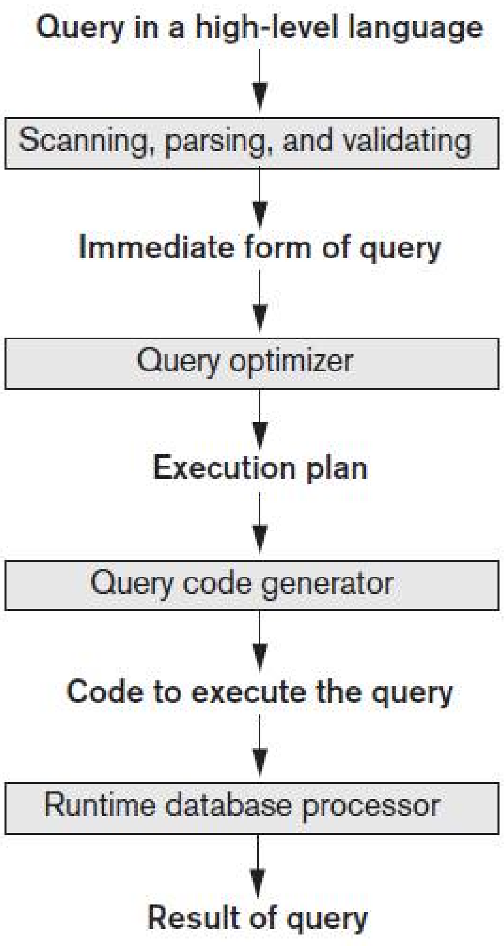
QUERY PROCESSING

Chapter 18

Prof. Young-Kyoon Suh

Query Processing

- Typical steps when processing a high-level query



// Task: to produce a "good" plan

Code can be:

Executed directly (interpreted mode)

Stored and executed later whenever needed (compiled mode)

If a runtime error results, then an error message is generated by the runtime database processor.

Introduction (Cont'd)

- ***Query processing***: DBMS techniques to process a high-level query (expressed like SQL)
  - *Scanner* identifies query tokens
    - Including keywords, attribute/relation names
  - *Query parser* checks the query syntax
    - To determine whether it's formulated along with the syntax rules
  - *Validation* checks all the attribute and relation names
    - Also checks they have semantically meaningful names in the database
  - *Query tree* (or query graph) created
    - An internal representation of that query in a tree data structure
  - *Query execution plan* is devised
    - Execution strategy for retrieving the results of the data from the database files
- ***Query optimization***
  - Planning a *good* execution strategy among many possible ones
    - "reasonably efficient" or "the best available strategy" but *not optimal*

Why Not Optimal Plan?

1) Finding “the” optimal plan is usually *too time-consuming*

- Except for the simplest queries: Like what?

2) Trying to find the optimal query plan requires *accurate and detailed information*

- About *the size of the tables* and *distributions of things such as column values*, which may not be always available in the catalog.

3) *Additional information* such as the expected result size must be derived based on the “predicates” in the query.

- Hard to know the exact size in advance; dynamically analyzed but little time to do so

For query sub optimality, see Appendix.

TRANSLATING SQL QUERIES INTO RELATIONAL ALGEBRA AND OTHER OPERATORS

Chapter 18.1

Query Translation into Relational Algebra

- SQL: a standard query language used in most RDBMSs
- Query decomposed into *query blocks*, or *basic units* that can be translated into the algebraic operators
  - Contains single `SELECT-FROM-WHERE` expression
  - Also may contain aggregates with `GROUP BY` and `HAVING` clauses
- *Subqueries (or nested queries)* within a query are identified as *separate* query blocks.

Translating SQL Queries

```
SELECT Lname, Fname
FROM   EMPLOYEE
WHERE  Salary > (SELECT MAX(Salary)
                 FROM   EMPLOYEE
                 WHERE  Dno = 5);
```

// Called a *nested subquery block*
// (Evaluated only once
// and used as constant)

- Retrieves the names of employees from any department in the company who earn a salary that is greater than *the highest salary in department 5*.

```
(SELECT MAX(Salary)
 FROM   EMPLOYEE
 WHERE  Dno = 5)
```

[Inner block]

translated

$$\Im_{\text{MAX Salary}}(\sigma_{Dno=5}(\text{EMPLOYEE}))$$

```
SELECT Lname, Fname
FROM   EMPLOYEE
WHERE  Salary > c
```

[Outer block]

$$\pi_{Lname,Fname}(\sigma_{Salary>c}(\text{EMPLOYEE}))$$

- Query optimizer then chooses an execution plan for each query block.

Additional Operators: ***Semi-Join***/***Anti-Join***

- Not part of the standard relational algebra
- ***Semi-Join*** ($R \ltimes S$ or $S \rtimes R$)
  - Generally used for "unnesting" `EXISTS`, `IN`, and ANY subqueries
  - Syntax: $T1.X$ S = $T2.Y$
    - $T1$ is the left table and $T2$ is the right table of the semi-join
  - How it works?
    - As soon as $T1.X$ finds a match with any value of $T2.Y$ (without searching for further matches), a row of $T1$ is returned.

```
SELECT count(*)
FROM   DEPARTMENT D
WHERE  D.Dnumber IN (SELECT E.Dno
                     FROM   EMPLOYEE E
                     WHERE  E.Salary > 200000)
```

```
SELECT count(*)
FROM   EMPLOYEE E, DEPARTMENT D
-- S=: non-standard expression
WHERE  D.Dnumber S= E.Dno and E.Salary > 200000
```

unnesting

Additional Operators: ***Semi-Join***/***Anti-Join***

- Not part of the standard relational algebra
- ***Anti-Join*** ($R \triangleright S$ or $R \overline{\ltimes} S$)
  - Used for unnesting `NOT EXISTS`, `NOT IN`, and ALL subqueries
  - Syntax: $T1.x$ A = $T2.y$
    - $T1$ is the left table and $T2$ is the right table of the semi-join
  - How it works?
    - A row of $T1$ is *rejected* as soon as $T1.x$ finds a match with any value of $T2.y$.
    - A row of $T1$ is *returned* only if $T1.x$ does NOT match with any value of $T2.y$.

```
SELECT  COUNT(*)
FROM    EMPLOYEE e
WHERE   e.Dno NOT IN (SELECT  DEPARTMENT.Dnumber
                      FROM    DEPARTMENT
                      WHERE   Zipcode = 30332)
```

```
SELECT  COUNT(*)
FROM    EMPLOYEE e, DEPARTMENT d
WHERE   e.Dno A= d.Dnumber AND d.Zipcode = 30332
```

unnesting

ALGORITHMS FOR **EXTERNAL SORTING**

Chapter 18.2

- Also, we discuss **sort-merge** join.

External Sorting: VERY IMPORTANT

- Sorting is one of the primary algorithms used in query processing
- External sorting refers to sorting algorithms for "large files of records" stored on disk that do **NOT** fit in entirely in main memory, such as most database files.
- *Sort-merge* strategy: typical external sort algorithm
  - *Sorting* smaller subfiles (or *runs*)
  - *Merging* the sorted runs

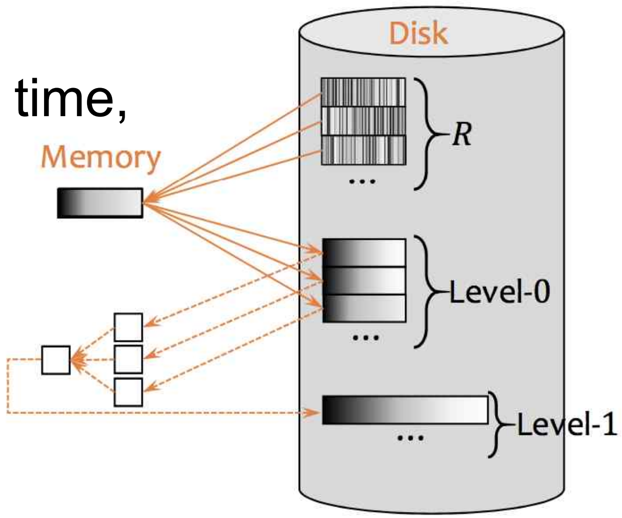
  (Why so? Again, the files are *too huge* to fit entirely in main memory)
- Requires *buffer space* in main memory: part of DBMS cache
  - Controlled by DBMS; divided into individual buffers
  - Size of each buffer = Size of one disk block (in bytes) = 4K
    - One buffer can hold the contents of exactly one disk block.

Sort-Merge Algorithm

```
set     i ← 1;
        j ← b;                {size of the file in blocks}
        k ← n_B;              {size of buffer in blocks}
        m ← ⌈(j/k)⌉;          {number of subfiles- each fits in buffer}
{Sorting Phase}
while (i ≤ m)
do {
        read next k blocks of the file into the buffer or if there are less than k blocks
           remaining, then read in the remaining blocks;
        sort the records in the buffer and write as a temporary subfile;
        i ← i + 1;
}
{Merging Phase: merge subfiles until only 1 remains}
set     i ← 1;
        p ← ⌈log_{k-1} m⌉ {p is the number of passes for the merging phase}
        j ← m;
while (i ≤ p)
do {
        n ← 1;
        q ← ( j/(k−1)⌉;       {number of subfiles to write in this pass}
        while (n ≤ q)
        do {
           read next k−1 subfiles or remaining subfiles (from previous pass)
              one block at a time;
           merge and write as new subfile one block at a time;
           n ← n + 1;
        }
        j ←     q;
        i ← i + 1;
}
```

Sort-Merge Algorithm (Cont'd)

- Remember (internal-memory merge sort in data structures)?
- Problem: sort R*, but R does not fit in memory, size of **M**.
- Idea: Divide ("sort *runs***") and conquer ("merge them")!
- *Pass 0*:
  - Read into **M** buffers (**M**) (disk) blocks of R at a time,
  - Sort each of them and **write out** a *level-0* **run**
- *Pass 1*:
  - Merge **M-1** *level-0* runs (of R) at a time,
  - Write out a *level-1* run
- *Pass 2*:
  - Merge **M-1** *level-1* runs (of R) at a time and write out a *level-2* run
- *Final pass*: produces a "single" sorted run. Yah!



* a relation file, **a set of **M** sorted disk blocks

Sort-Merge Algorithm – Toy Example

- 3 memory buffers available; each holds one number.
- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3 (on disk)
- *Pass 0*:
  - 1, 7, 4 → 1, 4, 7 (on disk)
  - 5, 2, 8 → 2, 5, 8 (on disk)
  - 9, 6, 3 → 3, 6, 9 (on disk)
- *Pass 1*:
  - 1, 4, 7 + 2, 5, 8 → 1, 2, 4, 5, 7, 8
  - 3, 6, 9
- *Pass 2* (final):
  - 1, 2, 4, 5, 7, 8 + 3, 6, 9 → 1, 2, 3, 4, 5, 6, 7, 8, 9

[Practice] Sort-Merge Algorithm – Cost Example

- With 5 buffer pages, to sort a file of 108 blocks
- *Pass 0*: $\lceil 108 / 5 \rceil = 22$ sorted runs (of 5 blocks each)
  - Note that the last run is only 3 blocks.
- *Pass 1*: $\lceil 22 / (5-1) \rceil = 6$ sorted runs (of 20 blocks each)
  - Note that the last run is only 8 blocks, and the one block for output.
- *Pass 2*: $\lceil 6 / (5-1) \rceil = 2$ sorted runs, each with 80 and 28 blocks
- *Pass 3*: Merge the 2 runs; the sorted file of 108 blocks.
- In the example, $\lceil \log_{5-1} \lceil \frac{108}{5} \rceil \rceil + 1 = \lceil 2.xxx \rceil + 1 = 4$ passes

Performance of External Sort-Merge

- \# of passes = k = $\lceil \log_{M\text{-}1} \lceil \frac{B(R)}{M} \rceil \rceil$ (k-1 passes) + 1 (final)
  - $B(R)$: Number of disk blocks of relation R
- I/O's
  - Multiply by 2 x $B(R)$
    - Each pass reads the entire relation once and writes it once
  - Subtract $B(R)$ for the final pass
    - We don't write its level-i runs any longer.
  - Roughly, $O(\underbrace{2 \text{ x } B(R)}_{\text{Pass 0}} + \underbrace{(2 \text{ x } B(R)) \text{ x } \log_M B(R)}_{\text{Pass 1} \sim \text{k-1}} \underbrace{- B(R)}_{\text{Final pass (k-1)}}) = O(B(R) \text{ x } \log_M B(R))$
- Memory requirement: **M** (as much as possible)

Sort-Merge Join (on Two Relations: $R \bowtie S$)

- $R \bowtie_{R.A=S.B} S$
  - *Sort* R and S by their join attributes (A and B, respectively)
  - *Merge* the first tuples—r and s—from the sorted R and S, respectively.
  - *Repeat* the merge until either R or S is exhausted:
    - `IF` ($r.A > s.B$), then s = next tuple in S.
    - `ELSE IF` ($r.A < s.B$), then r = next tuple in R.
    - `ELSE` /* $r.A == s.B$ */ output all matching tuples, and
      - r, s = next tuples in R and S, respectively.

Example of Sort-Merge Join

R:	S:	$R \bowtie_{R.A=S.B} S$
$r_1.A = 1$	$s_1.B = 1$	r_1s_1
$r_2.A = 3$	$s_2.B = 2$	r_2s_3
$r_3.A = 3$	$s_3.B = 3$	r_2s_4
$r_4.A = 5$	$s_4.B = 3$	r_3s_3
$r_5.A = 5$	$s_5.B = 8$	r_3s_4
$r_6.A = 7$		r_7s_5
$r_7.A = 8$		

I/O of Sort-Merge Join

- I/O's: sorting + merging = **2***$B(R)$ + **2***$B(S)$
  - In most cases (when join of key and foreign key are considered)
  - Worst cases: $B(R)$ * $B(S)$ when every tuple participates in join

Other Sort-based Algorithms

- Union(set), difference, intersection
  - Similar to sort-merge join
- Duplication elimination (via external merge sort)
  - Eliminates duplicates in sort and merge
- Grouping and aggregation
  - External merge sort, by group-by columns
  - One trick: produce "partial" aggregate values in each run and then combine them during merge
    - But doesn't work for `SUM(DISTINCT...)`, `MEDIAN (...)`
- For more details, see Appendix.

ALGORITHMS FOR SELECT OPERATION

Chapter 18.3

Implementation Options for SELECT

- **SELECT** operation: *search operation* to locate records in a disk file that satisfy a certain condition
  - Also known as “filter” operation
- Full table (file) scan, or index scan *if search involved an index* (like B$^+$-tree)

$\sigma_{(\text{Dno}=4 \textbf{ AND } \text{Salary}>25000) \textbf{ OR } (\text{Dno}=5 \textbf{ AND } \text{Salary}>30000)}(\text{EMPLOYEE})$

$\sigma_{\text{Ssn=Essn}}(\text{EMP\_DEPENDENTS})$

$\sigma_{\text{SEX='F'}}(\text{EMPLOYEE})$

$\sigma_{\text{Plocation='Stafford'}}(\text{PROJECT})$

```
SELECT First_name, Lname
FROM EMPLOYEE
WHERE ((Salary*Commission_pct)+Salary) > 15000;
```

Implementation Options for SELECT (Cont’d)

- Search methods for simple selection
  - S1: Linear search (brute-force algorithm)
  - S2: Binary search (when records are sorted)
  - S3a: Using a primary index
  - S3b: Using a hash index (with hash keys)
  - S4: Using a primary index to retrieve multiple records
  - S5: Using a clustering index to retrieve multiple records
  - S6: Using a secondary index (e.g., B^+ -tree) on an equality comparison
  - S7a: Using a bitmap index using bitmaps for each attribute value
  - (S7b: Using a functional index
    - e.g. `CREATE INDEX income_idx ON EMPLOYEE ((Salary*Commission_pct)+Salary)`)

Implementation Options for **SELECT** (Cont'd)

- Query optimizer receives input from system catalog to estimate selectivity.
  - Information stored in the catalog for relation (R): # of rows/records (or its cardinality), tuple length, # blocks, bfr
  - Information stored in the catalog for attribute (A): # of distinct values (or, $\mathrm{NDV}(A, R)$), maximum and minimum values of A

- **Selectivity:** *ratio* of # of records that satisfy the condition to the total number of records in the file
  - Falls between 0 (?) and 1 (?)
    - E.g., for an equality condition on a key attribute of relation R, what's the selectivity of the condition?
  - Why is selectivity *important*?

IMPLEMENTING THE JOIN OPERATION

Chapter 18.4

Methods for Implementing Joins

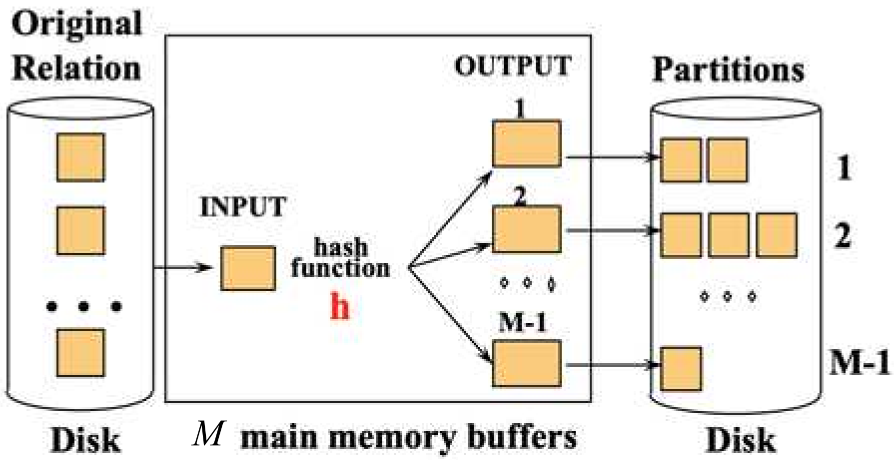
- JOIN operation
  - One of the most time-consuming in query processing
  - Treated in this chapter as EQUIJOIN (or NATURAL JOIN)
  - We focus on two-way joins
    - *multiway joins* involving more than two files

Methods for Implementing Joins (Cont'd)

- Sort-merge join (covered)
- *Nested-loop join* (or nested-block join)
  - Default algorithm; for each record r in R (**outer loop**), retrieve every record s from S (**inner loop**)
    - I/O's: $B(R)$ (or $B(S)$) + $B(R) \cdot B(S)$ (here, R assumed to be with fewer blocks)
- *Index-based nested-loop join*
  - To use this join, there should be an index (or hash key) exists for one of the two join attributes; say, s attribute in S.
  - Then retrieve each record r in R and then using the index, retrieve all matching records s in S that satisfying $R.r = S.s$
- *Hash join*: $R \bowtie S$ (for both fitting in main memory)
  - Scan R, build buckets in main memory via a hash function
  - Then scan S and join; Cost: $B(R) + B(S)$

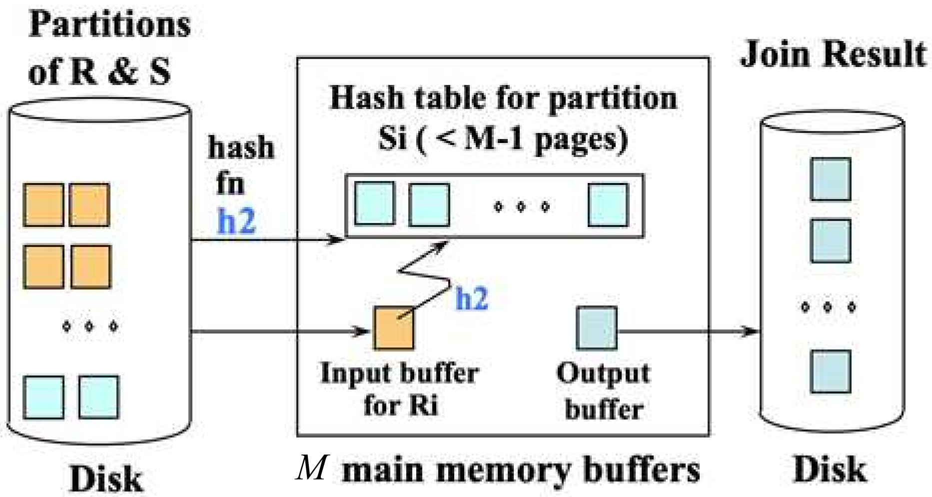
Methods for Implementing Joins (Cont'd)

- *Partition-hash join*: $R \bowtie S$ (for one fitting in main memory)



Step 1: R and S are partitioned into smaller files using the same h;
- R tuples in partition i will only match S tuples in partition i.

Step 2:
Build phase: read in partition of R, hash it using $h2$ ($\neq h$) (?)
Probe phase: scan matching partition of S and all matching records



* Cost: 3 ($B(R) + B(S)$); why?

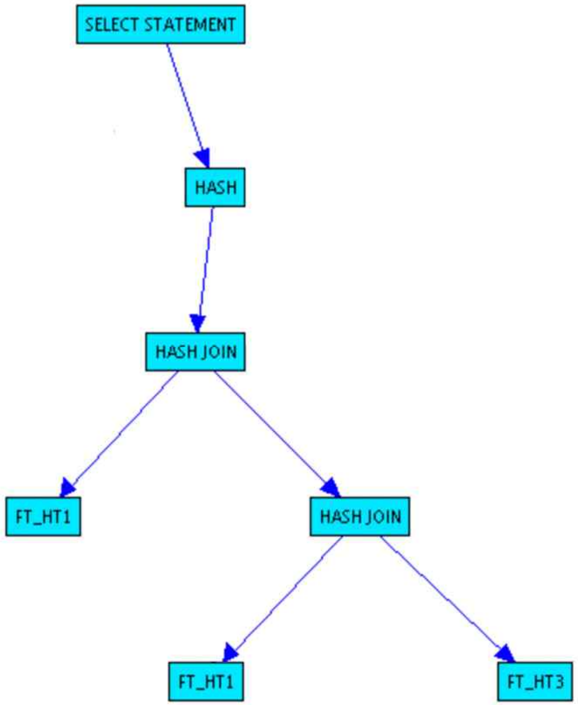
APPENDIX

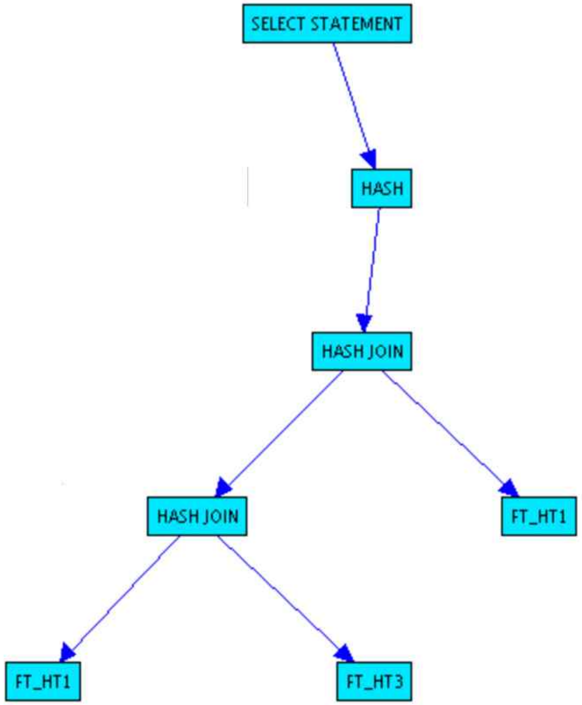
References

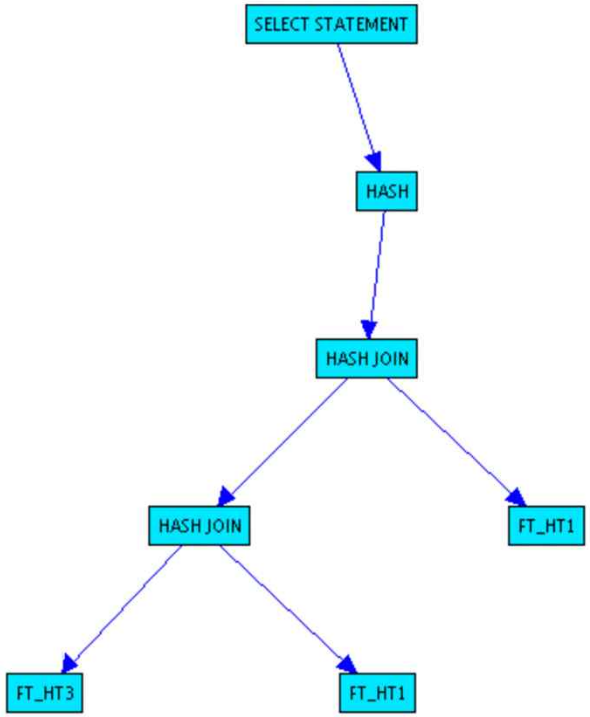
- http://web.cs.ucdavis.edu/~green/courses/ecs165b/Lecture20.pdf
- https://www2.cs.duke.edu/courses/fall14/compsci316/lectures/19-qp-notes.pdf
- https://courses.cs.washington.edu/courses/cse444/10au/lectures/lecture20.pdf

Query Tree Example

```
SELECT t0.id3, SUM(t2.id3)
FROM ft_HT1 t0, ft_HT3 t2, ft_HT1 t1
WHERE (t0.id4=t2.id1 AND
       t2.id1=t1.id4)
GROUP BY t0.id3
```

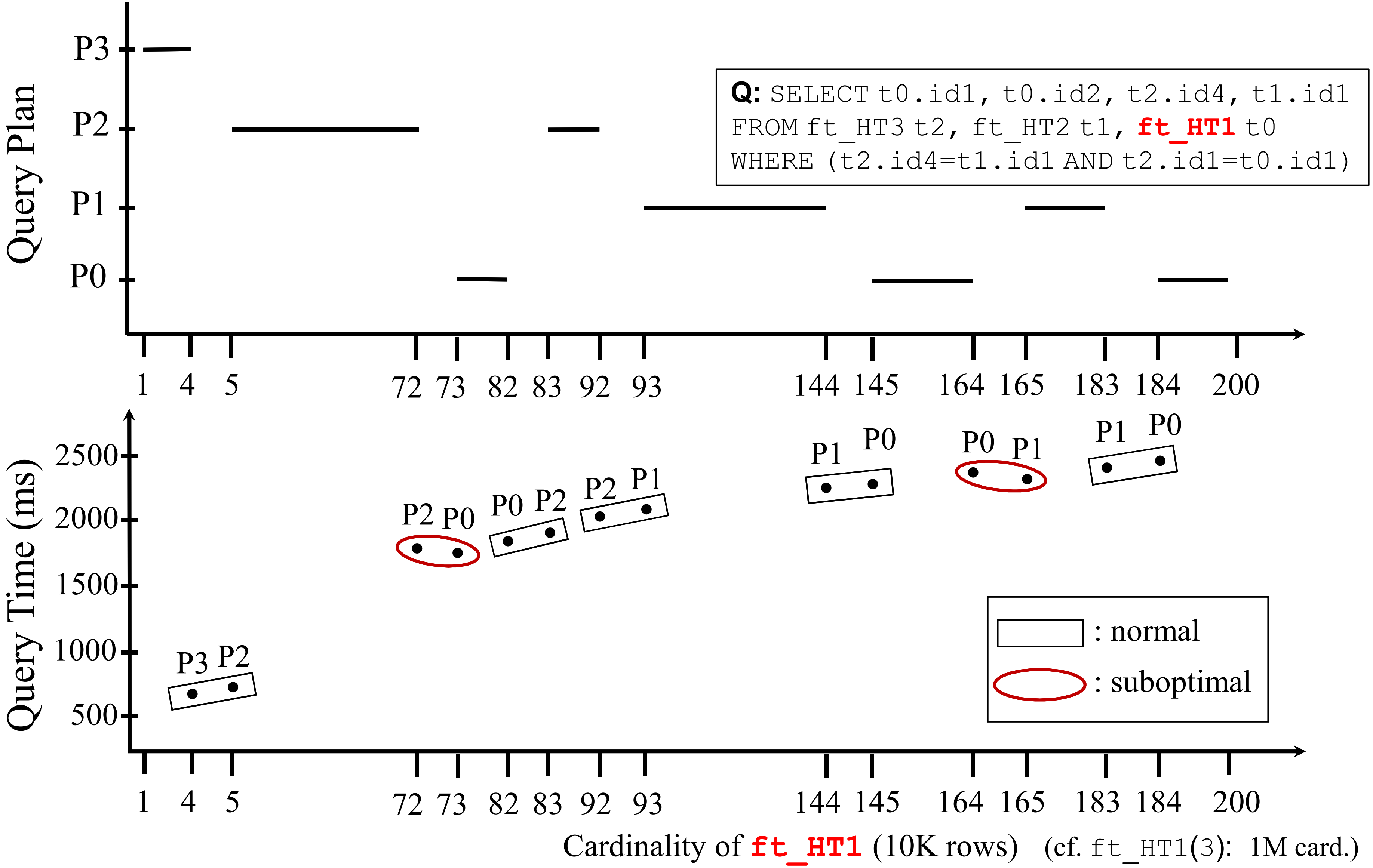






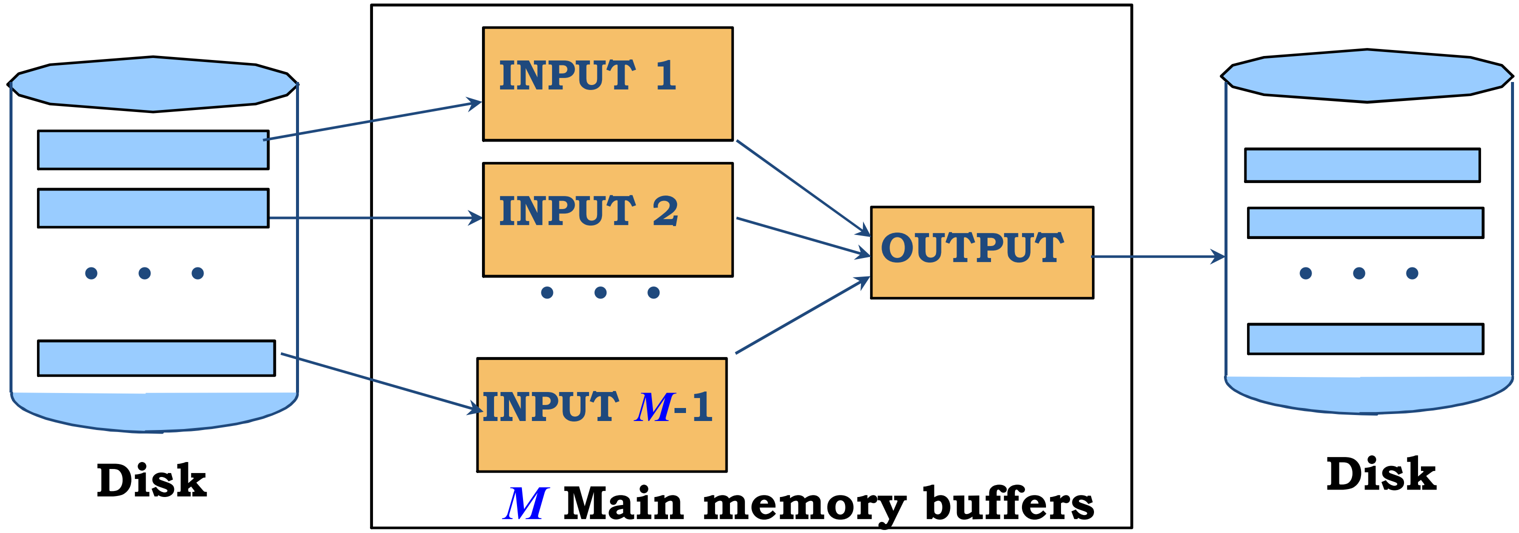
[Appendix]

Example of Query *Suboptimality* (on PostgreSQL)



Sort-Merge Algorithm (Cont'd)

- Another look
  - M = number of buffers available in main memory



Implementation of JOIN via Sort-Merge

$R \bowtie_{R.A=S.B} S$

- R: n tuples
- S: m tuples

```
sort the tuples in R on attribute A;          (*assume R has n tuples (records)*)
sort the tuples in S on attribute B;          (*assume S has m tuples (records)*)
set i ← 1, j ← 1;
while (i ≤ n) and (j ≤ m)
do {   if R(i)[A] > S(j)[B]
          then set j ← j + 1
     elseif R(i)[A] < S(j)[B]
          then set i ← i + 1
     else {   (* R(i)[A] = S(j)[B], so we output a matched tuple *)
             output the combined tuple <R(i), S(j)> to T;

             (* output other tuples that match R(i), if any *)
             set l ← j + 1;
             while (l ≤ m) and (R(i)[A] = S(l)[B])
             do {   output the combined tuple <R(i), S(l)> to T;
                    set l ← l + 1
             }

        (* output other tuples that match S(j), if any *)
        set k ← i + 1;
        while (k ≤ n) and (R(k)[A] = S(j)[B])
        do {   output the combined tuple <R(k), S(j)> to T;
               set k ← k + 1
        }
        set i ← k, j ← l
     }
}
```

Implementation of UNION via Sort-Merge

- $T \leftarrow R \cup S$

```
sort the tuples in R and S using the same unique sort attributes;
set i ← 1, j ← 1;
while (i ≤ n) and (j ≤ m)
do {   if R(i) > S(j)
          then {   output S(j) to T;
                   set j ← j + 1
          }
       elseif R(i) < S(j)
          then {   output R(i) to T;
                   set i ← i + 1
          }
       else set j ← j + 1                 (* R(i)=S(j), so we skip one of the duplicate tuples *)
}
if (i ≤ n) then add tuples R(i) to R(n) to T;
if (j ≤ m) then add tuples S(j) to S(m) to T;
```

Implementation of **SET DIFFERNCE** via Sort-Merge

- $T \leftarrow R - S$

sort the tuples in R and S using the same unique sort attributes;
set $i \leftarrow 1, j \leftarrow 1$;
while $(i \leq n)$ and $(j \leq m)$
do { if $R(i) > S(j)$
  then set $j \leftarrow j + 1$
 elseif $R(i) < S(j)$
  then { output $R(i)$ to T;   (* $R(i)$ has no matching $S(j)$, so output $R(i)$ *)
   set $i \leftarrow i + 1$
  }
 else set $i \leftarrow i + 1, j \leftarrow j + 1$
}
if $(i \leq n)$ then add tuples $R(i)$ to $R(n)$ to T;

Implementation of INTERSECTION via Sort-Merge

- $T \leftarrow R \cap S$

sort the tuples in R and S using the same unique sort attributes;
set $i \leftarrow 1, j \leftarrow 1$;
while $(i \leq n)$ and $(j \leq m)$
do { if $R(i) > S(j)$
  then set $j \leftarrow j + 1$
 elseif $R(i) < S(j)$
  then set $i \leftarrow i + 1$
 else { output $R(j)$ to T; (* $R(i) = S(j)$, so we output the tuple *)
  set $i \leftarrow i + 1, j \leftarrow j + 1$
 }
}

Implementation Options for **SELECT** (Cont'd)

- Search methods for conjunctive (logical `AND`) selection
  - Using an individual index (for any single simple condition)
  - Using a composite index (for 2+ attributes involved in equality)
  - Intersection of record pointers (for secondary indices available on more than one of the fields involved in simple conditions)
- Search methods for disjunctive (logical `OR`) selection
  - Much harder to process and optimize than the `AND` selection
  - Example) $\sigma_{\texttt{Dno=5 OR Salary>30000 OR Sex='F'}}(\texttt{EMPLOYEE})$
    - We end up with doing linear search if any one of the conditions does not have an access path like index.

ADDITIONAL DISCUSSIONS

Chapters 18.4.2-18.7

Join Performance

- The buffer space available has an important effect on some of the join algorithms.
  - Consider the nested-loop join, in which it's important to choose which one goes in the outer loop.
- *Join selection factor (or join selectivity)*
  - Fraction of records in one relation (file) that will be joined with records in another relation (file)
  - Depends on the particular equijoin condition with another file
  - Affects join performance; it's more advantageous to process a condition with <u>high</u> selectivity (small in number) as it produces less intermediate result.

Implementation of **PROJECT**

- $T \leftarrow \pi_{\text{<attribute list>}}(R)$.

create a tuple t[<attribute list>] in T' for each tuple t in R;
    (* T' contains the projection results *before* duplicate elimination *)
if <attribute list> includes a key of R
    then $T \leftarrow T'$
else { sort the tuples in T';
        set $i \leftarrow 1, j \leftarrow 2$;
        while $i \leq n$
        do { output the tuple $T'[i]$ to T;
                while $T'[i] = T'[j]$ and $j \leq n$ do $j \leftarrow j + 1$;   (* eliminate duplicates *)
                $i \leftarrow j; j \leftarrow i + 1$
        }
}
(* T contains the projection result after duplicate elimination*)

Combining Operations Using Pipelining

- SQL query translated into relational algebra expression
  - Sequence of relational operations
- *Materialized evaluation*
  - Creating, storing, and passing temporary results in query processing
  - But generating and storing large files on disk is timing-consuming
- General query goal: *to minimize # of temporary files*
  - Reducing as much I/O as possible during query processing
- *Pipelining* (or stream-based processing)
  - Combines several operations into one
    - The next query operator starts its processing as soon as the resulting tuples arrive from the previous query operator with no intermediate I/O
  - Avoid writing temporary files