Ch 14. Evaluation of Relational Operators

(12.2 + 14.1, 14.4. and 14.7 covered)

Sang-Won Lee

http://icc.skku.ac.kr/~swlee

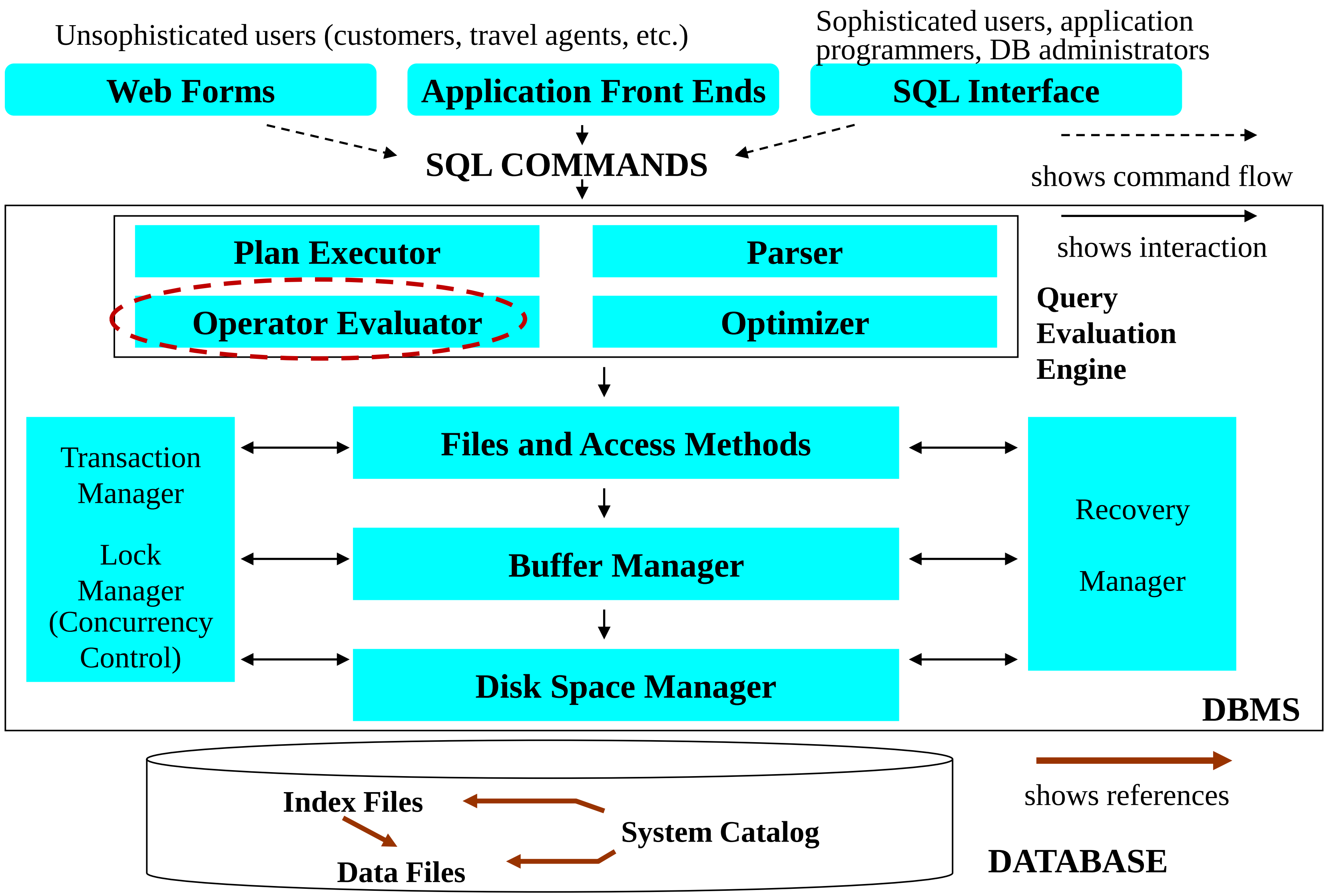


Figure 1.3 Anatomy of an RDBMS

What you should know from this chapter

- Alternative (physical) algorithms for selection, projection, **join**, set, and aggregation & grouping (logical) operations in SQL?
  - Which alternative is best under what condition?

- Effect of buffering: how the size and replacement policy of buffer can affect the evaluation of the operations?

Relational Operations

- We will consider how to implement:
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Projection* (π) Deletes unwanted columns from relation.
  - ***Join* ($\bowtie$) Allows us to combine two relations.**

  - *Set-difference* ($-$) Tuples in reln. 1, but not in reln. 2.
  - *Union* ($\bigcup$) Tuples in reln. 1 and in reln. 2.
  - *Aggregation* (SUM, MIN, etc.) and GROUP BY

12.2 Query Evaluation

- Algorithms for evaluating relational operators use some simple ideas extensively:

1. Indexing: Can use WHERE conditions to retrieve small set of tuples using index (selections, joins)

2. Iteration: Sometimes, faster to scan all tuples even if there is an index. (And sometimes, we can scan the data entries in an index instead of the table itself.)

3. Partitioning: By using **sorting** or **hashing**, we can partition the input tuples and replace an expensive operation by similar operations on smaller inputs.

Access Paths

- An access path is a method of retrieving tuples:
  - File scan, or index that matches a selection (in the query)

- A tree index *matches* (a conjunction of) terms that involve only attributes in a *prefix* of the search key.
  - E.g., Tree index on <*a, b, c*> matches the selection *a=5 AND b=3*, and *a=5 AND b>6*, but not *b=3*.

14.1 Selection

- Simple selection: relation.attr OP value

```
SELECT *
FROM Reserves R
WHERE R.rname = 'JOE'
```

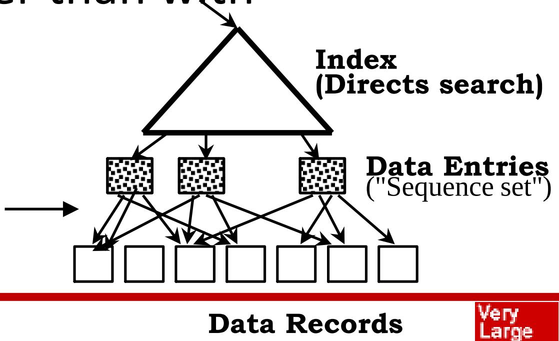
1. No index, no sorted: full table scan
2. No index, sorted: binary search + scan
3. B+ tree index
4. Hash index & equality

Using an Index for Selections

- Cost depends on # of qualifying tuples, and clustering.
  - Cost of finding qualifying data entries (typically small) plus cost of retrieving records (could be large with unclustered indexes).
  - In example, assuming uniform distribution of names, about 10% of tuples qualify (100 pages, 10000 tuples). With a clustered index, cost is little more than 100 I/Os; if unclustered, upto 10000 I/Os!
- *Important refinement for unclustered indexes*: RID Sorting

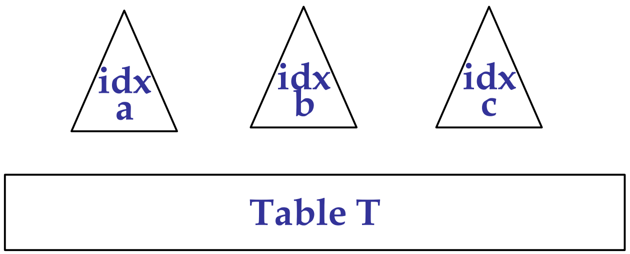
  1. Find qualifying data entries {key value, recod_identifier}
  2. Sort the rids of the data records to be retrieved.
  3. Then, fetch rids in order. This ensures that each data page is looked at just once (though # of such pages likely to be higher than with clustering).

Sort RID set first
Then, fetch records



14.2 General Selection: Two Approaches

```
SELECT  *
FROM    TEST
WHERE  a = 1 and b = 1 and c = 1
```

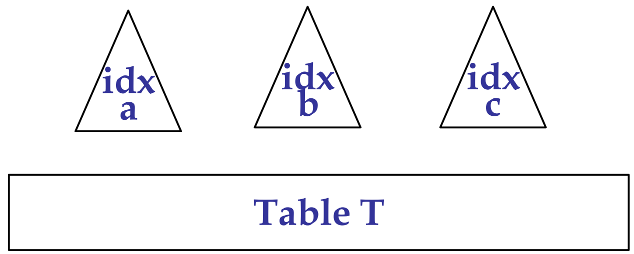


First approach:

- Find the *most selective access path*, retrieve tuples using it, and apply any remaining terms that don't match the index:
  - *Most selective access path:* An index or file scan that we estimate will require the fewest page I/Os.
  - Terms that match this index reduce the number of tuples *retrieved*; other terms are used to discard some retrieved tuples, but do not affect number of tuples/pages fetched.
  - Consider *day<8/9/94 AND bid=5 AND sid=3*. A B+ tree index on *day* can be used; then, *bid=5* and *sid=3* must be checked for each retrieved tuple. Similarly, a hash index on <*bid, sid*> could be used; *day<8/9/94* must then be checked.

Intersection of Rids

```
SELECT  *
FROM    TEST
WHERE  a = 1 and b = 1 and c = 1
```



Second approach

- RID Intersection: if we have 2 or more matching indexes that use Alternatives (2) or (3) for data entries:
  - Get sets of rids of data records using each matching index.
  - Then *intersect* these sets of rids (we'll discuss intersection soon!)
  - Retrieve the records and apply any remaining terms.
  - Consider *day<8/9/94 AND bid=5 AND sid=3*. If we have a B+ tree index on *day* and an index on *sid*, both using Alternative (2), we can retrieve rids of records satisfying *day<8/9/94* using the first, rids of recs satisfying *sid=3* using the second, intersect, retrieve records and check *bid=5*.

14.3 Projection

```
SELECT DISTINCT R.sid, R.bid
FROM   Reserves R
```

- The expensive part is removing duplicates.
  - SQL systems don't remove duplicates unless the keyword DISTINCT is specified in a query.

- Sorting Approach: Sort on <sid, bid> and remove duplicates. (Can optimize this by dropping unwanted information while sorting.)

- Hashing Approach: Hash on <sid, bid> to create partitions. Load partitions into memory one at a time, build in-memory hash structure, and eliminate duplicates.

- If there is an index with both R.sid and R.bid in the search key, may be cheaper to sort data entries!

Projection Based on Sorting

```
SELECT DISTINCT R.sid, R.bid
FROM     Reserves R
```

- An approach based on sorting: naïve version

  1. Scan R and produce a set of tuples with only desired attributes
  2. Sort this set of tuples
  3. Scan the sorted result while discarding the duplicates

14.4 Equality Joins With One Join Column

```
SELECT  *
FROM    Reserves R1, Sailors S1
WHERE   R1.sid=S1.sid
```

- In algebra: $R \bowtie S$. Common! Must be carefully optimized. $R \times S$ is large; so, $R \times S$ followed by a selection is inefficient.
- Assume: <u>M pages in R</u>, <u>p_R tuples per page</u>, <u>N pages in S</u>, <u>p_S tuples per page</u>.
  - In our examples, R is Reserves and S is Sailors.
- We will consider more complex join conditions later.
- *Cost metric*: # of I/Os. We will ignore output costs.

14.4.1 Nested Loops Join

```
for each tuple r in R do                /* R: outer relation */
        for each tuple s in S do        /* S: inner relation */
                if ri == sj  then add <r, s> to result
```

1. Tuple-oriented Nested Loops join: For each tuple in the *outer* relation R, we scan the entire *inner* relation S.
   - Cost: $M + p_R * M * N$ = 1000 + 100*1000*500 I/Os.
2. Page-oriented Nested Loops join: For each *page* of R, get each *page* of S, and write out matching pairs of tuples <r, s>, where r is in R-page and S is in S-page.
   - Cost: $M + M*N$ = 1000 + 1000*500
   - If smaller relation (S) is outer, cost = 500 + 500*1000

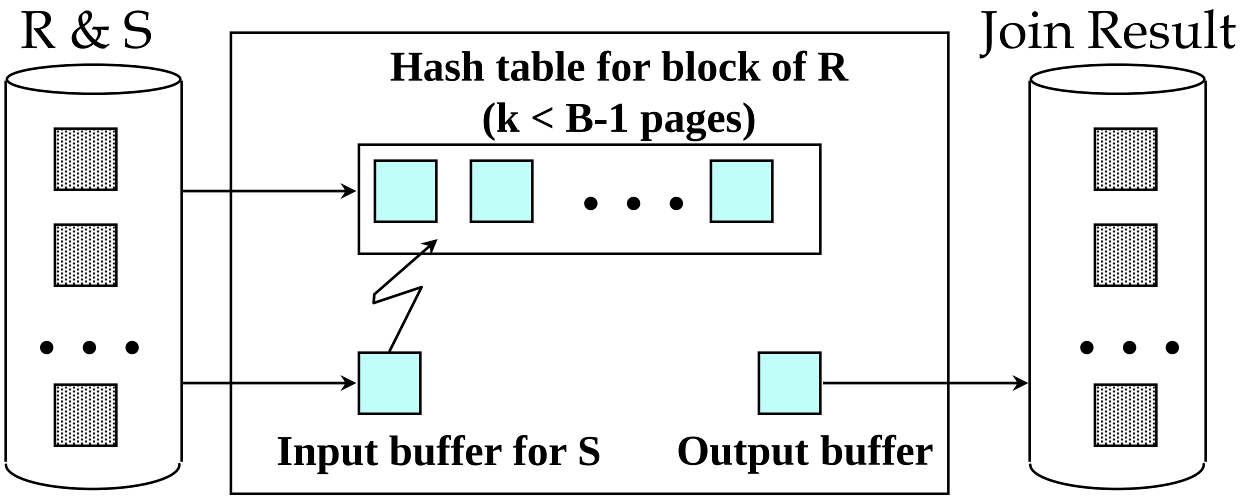
Nested Loops Join(2)

for each tuple r in R do
    for each tuple s in S where ri == sj do
        add <r, s> to result

3. Index Nested Loops Join
   - If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
     - ✓ Cost: $M + ((M*p_R) *$ cost of finding matching S tuples)
   - For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples (assuming Alt. (2) or (3) for data entries) depends on clustering.
     - ✓ clustered index: 1 I/O (typical)
     - ✓ unclustered: up to 1 I/O per matching S tuple.

Nested Loops Join(3)

4. Block Nested Loops Join : use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold ``block'' of outer R.
   - for each matching tuple r in R-block, s in S-page, add<r, s> to result. Then read next R-block, scan S, etc.



Examples of Block Nested Loops

- Cost: Scan of outer + #outer blocks * scan of inner
  - #outer blocks = $\lceil \#\ of\ \ pages\ of\ outer\ /\ blocksize \rceil$
- With Reserves (R) as outer, and 100 pages of R:
  - Cost of scanning R is 1000 I/Os; a total of 10 *blocks*.
  - Per block of R, we scan Sailors (S); 10*500 I/Os.
  - If space for just 90 pages of R, we would scan S 12 times.
- With 100-page block of Sailors as outer:
  - Cost of scanning S is 500 I/Os; a total of 5 blocks.
  - Per block of S, we scan Reserves; 5*1000 I/Os.
- With *sequential reads* considered, analysis changes: may be best to divide buffers evenly between R and S.

14.4.2~3 Join: SORT-MERGE vs. HASH

- In nested loop join, all the data blocks resides in buffer cache!!
- BUT, in sort-merge and hash join, after scanning the data from the buffer cache, the remaining phase works for 1) the separate memory and 2) temporary tablespace
  - e.g. Oracle
    - ✓ Sort-merge join: sort_area_size
    - ✓ Hash-join: hash_area_size
    - ✓ Temporary tablespace

14.4.2~3 Join: NL vs. SORT-MERGE vs. HASH

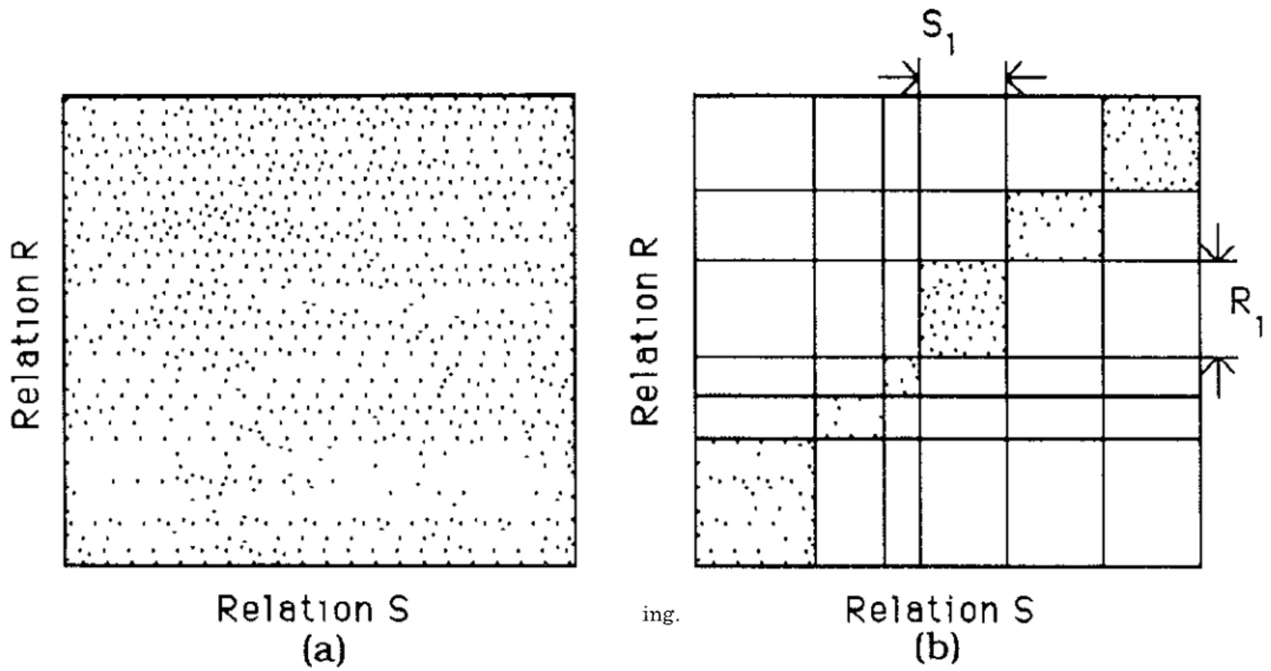


Figure 1. Reduction of join load. (a) No partitioning; (b) with partitioning

[Source: Join Processing in Relational Databases (CACM)

- Nested loop (iteration) vs. indexed nested loop (indexing)
- Reduce # of comparisons by partitioning
  - Hash vs. Sorting

Very Large Data Bases

14.4.2 Join: Sort-Merge ($R \bowtie_{i=j} S$)

- Sorting phase: Sort R and S on the join column
- Merging phase: then scan them to do a ``merge'' (on join column), and output result tuples.
  1. Advance scan of R until current R-tuple >= current S tuple, then advance scan of S until current S-tuple >= current R tuple; do this until current R tuple = current S tuple.
  2. At this point, all R tuples with same value in Ri (*current R group*) and all S tuples with same value in Sj (*current S group*) *match*; output <r, s> for all pairs of such tuples.
  3. Then, resume scanning R and S.

- R is scanned once; each S group is scanned once per matching R tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)

Example of Sort-Merge Join

```
proc smjoin(R, S, 'R_i = S'_j)

if R not sorted on attribute i, sort it;
if S not sorted on attribute j, sort it;

Tr = first tuple in R;                              // ranges over R
Ts = first tuple in S;                              // ranges over S
Gs = first tuple in S;                  // start of current S-partition

while Tr ≠ eof and Gs ≠ eof do {

    while Tr_i < Gs_j do
        Tr = next tuple in R after Tr;            // continue scan of R

    while Tr_i > Gs_j do
        Gs = next tuple in S after Gs             // continue scan of S

    Ts = Gs;                                   // Needed in case Tr_i ≠ Gs_j
    while Tr_i == Gs_j do {                   // process current R partition
        Ts = Gs;                                 // reset S partition scan
        while Ts_j == Tr_i do {                  // process current R tuple
            add ⟨Tr, Ts⟩ to result;               // output joined tuples
            Ts = next tuple in S after Ts;} // advance S partition scan
        Tr = next tuple in R after Tr;             // advance scan of R
    }                                         // done with current R partition

    Gs = Ts;                        // initialize search for next S partition

}
```

Figure 14.8 Sort-Merge Join

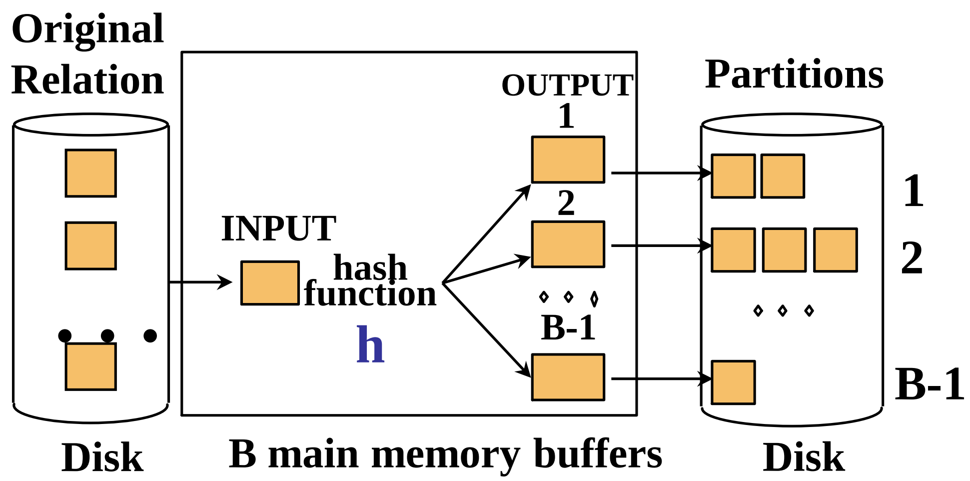
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

- Cost: M log M + N log N + (M+N)
  - The cost of scanning, M+N, could be M*N (very unlikely!)

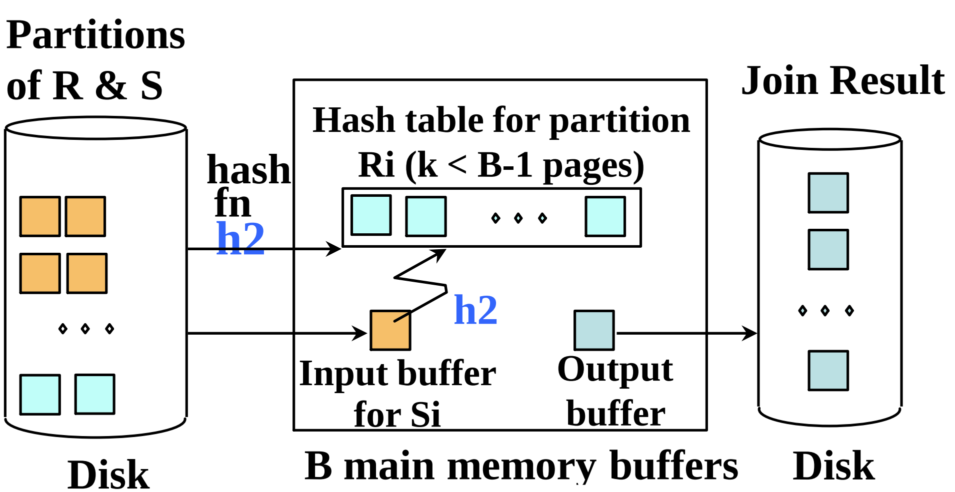
14.4.3 Grace Hash-Join

- Partition both relations using hash fn **h**: R tuples in partition i will only match S tuples in partition i.



1. Partitioning(or building) phase

- Read in a partition of R, hash it using **h2 (!= h)**. Scan matching partition of S, search for matches



2. Probing(or matching) phase

Cost of Hash-Join

- Cost of hash-join = <u>3(M+N)</u>
  - partitioning phase, read+write both relations; 2(M+N).
  - matching phase, read both relns; M+N I/Os.
  - In our running example, this is a total of 4500 I/Os.
  - What if we have more memory? Hybrid hash join (See text)

- Sort-Merge Join vs. Hash Join:
  - Given a minimum amount of memory (*what is this, for each?*) both have a cost of 3(M+N) I/Os.
  - Hash Join: 1) superior if relation sizes differ greatly; 2) also, highly parallelizable.
  - Sort-Merge: less sensitive to data skew; result is sorted.

General Join Conditions

- Equalities over several attributes (e.g., *R.sid=S.sid* AND *R.rname=S.sname*):
  - For Index NL, build index on *<sid, sname>* (if S is inner); or use existing indexes on *sid* or *sname*.
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.

- Inequality conditions (e.g., *R.rname < S.sname*):
  - For Index NL, need (clustered!) B+ tree index.
    - ✓ Range probes on inner; # matches likely to be much higher than for equality joins.
  - Hash Join, Sort Merge Join not applicable.
  - Block NL quite likely to be the best join method here.

Let us skip section 14.5/14.6

14.7 Impact of Buffering

- If several operations are executing concurrently, estimating the number of available buffer pages is guesswork.

- Repeated access patterns interact with buffer replacement policy.
  - e.g., Inner relation is scanned repeatedly in Simple Nested Loop Join. With enough buffer pages to hold inner, replacement policy does not matter. Otherwise, MRU is best, LRU is worst (*sequential flooding*).
  - Does replacement policy matter for Block Nested Loops?
  - What about Index Nested Loops?

Summary

- A virtue of relational DBMSs: *queries are composed of a few basic operators*; the implementation of these operators can be carefully tuned (and it is important to do this!).

- Many alternative implementation techniques for each operator; no universally superior technique for most operators.

- Must consider available alternatives for each operation in a query and choose best one based on system statistics, etc. This is part of the broader task of optimizing a query composed of several ops.

Ch 12. Overview of Query Evaluation

(Overview of Query Optimizer and 12.1)

Sang-Won Lee

http://icc.skku.ac.kr/~swlee

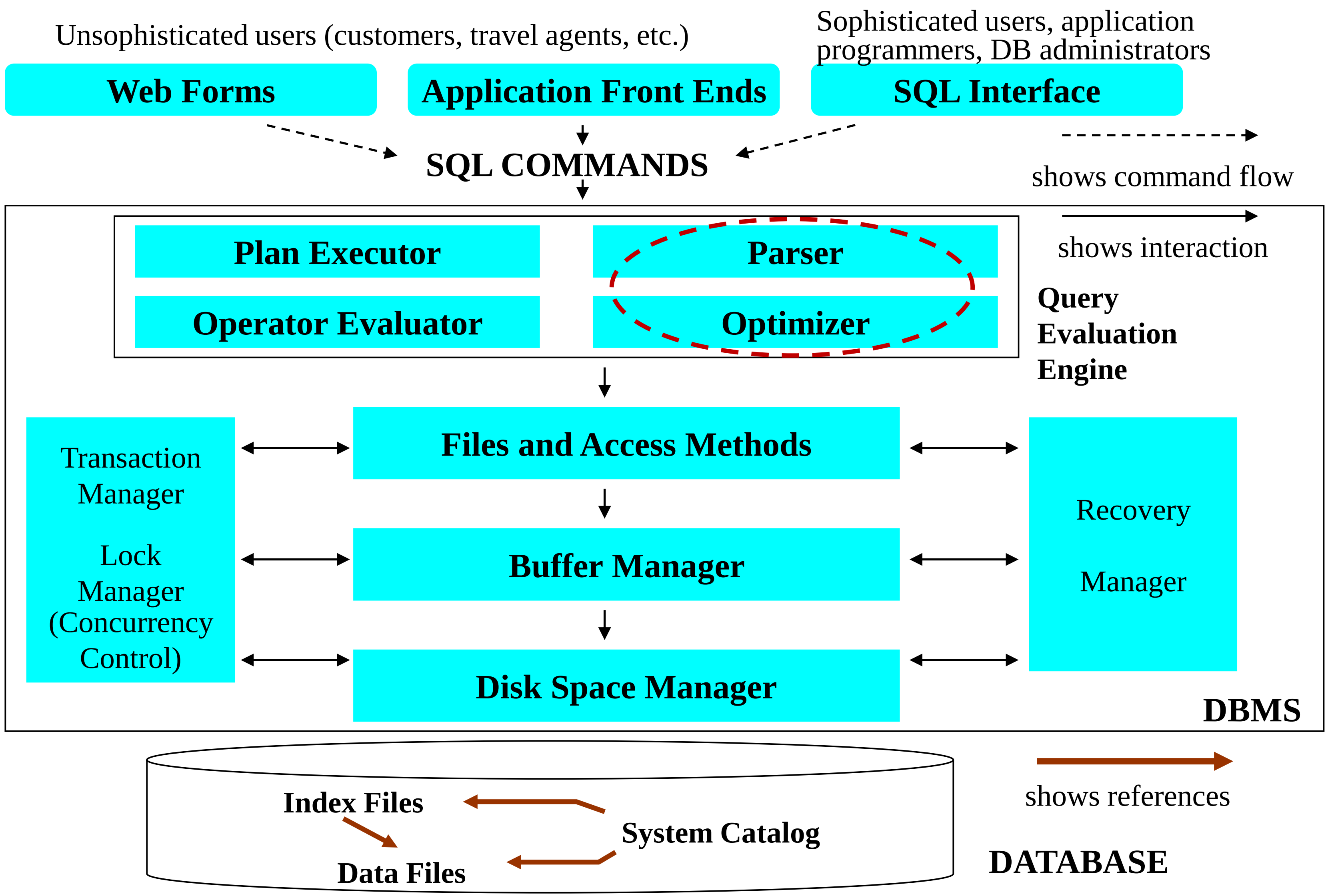


Figure 1.3 Anatomy of an RDBMS

Overview

1. SQL query by user

```
SELECT *
FROM test
WHERE a between 1 and 1000
```

2. Parsing
3. Plan generation
   - Hard vs. Soft parsing

Full Scan(test)

Plan 1

Table access(by rowid)

Index Scan(test_idx)

Plan 2

4. Cost estimation for each plan: plan 1 = 1000; plan 2 = 500
5. Run the best plan chosen by optimizer: in this case, plan 2

Overview(2)

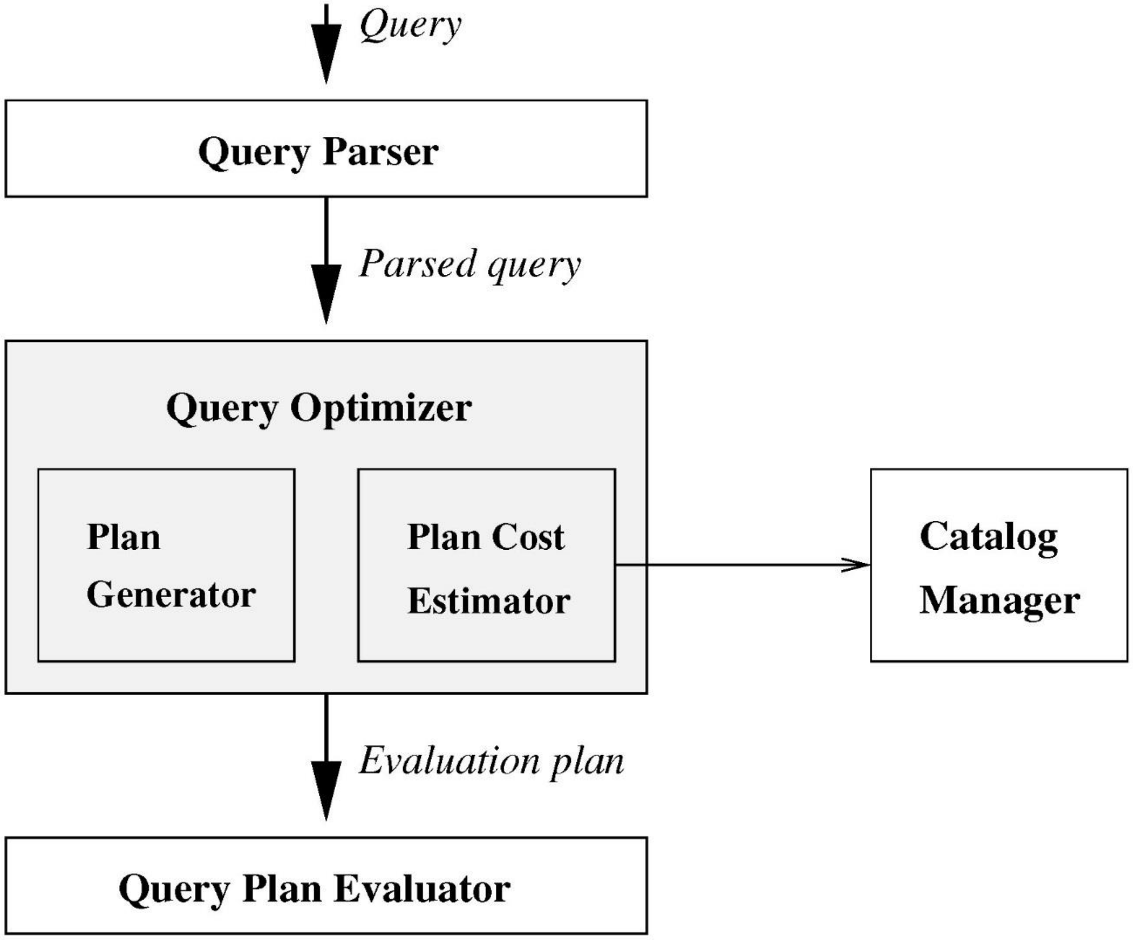


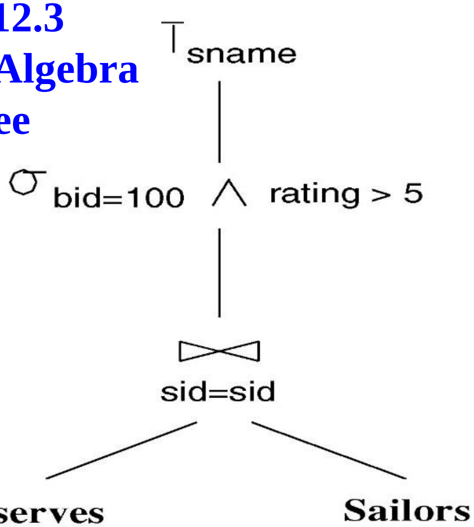
Figure 12.2 Query Parsing, Optimization, and Execution

Overview of Query Evaluation

- <u>Query Execution Plan:</u> tree of Relational Algebra operator with choice of algorithm for each *operator*.

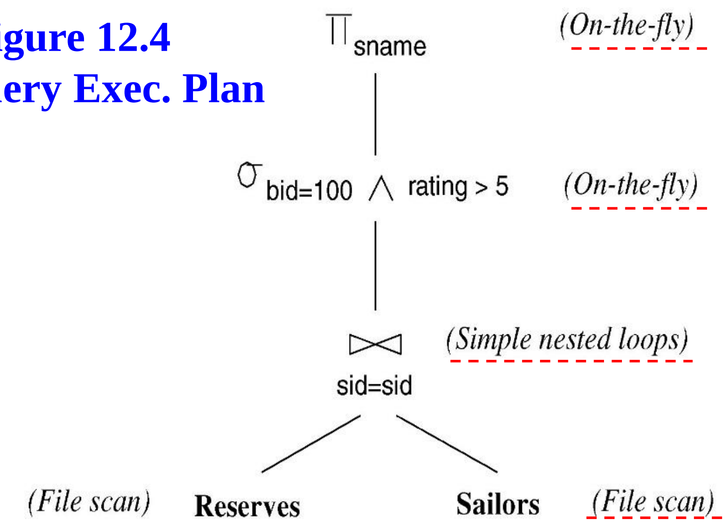
```
SELECT  S.sname
FROM reservesR, sailors S
WHERE R.sid = S.sid and R.bid = 100 and S.rating > 5
```

Figure 12.3
Relational Algebra in Tree



implemented ι

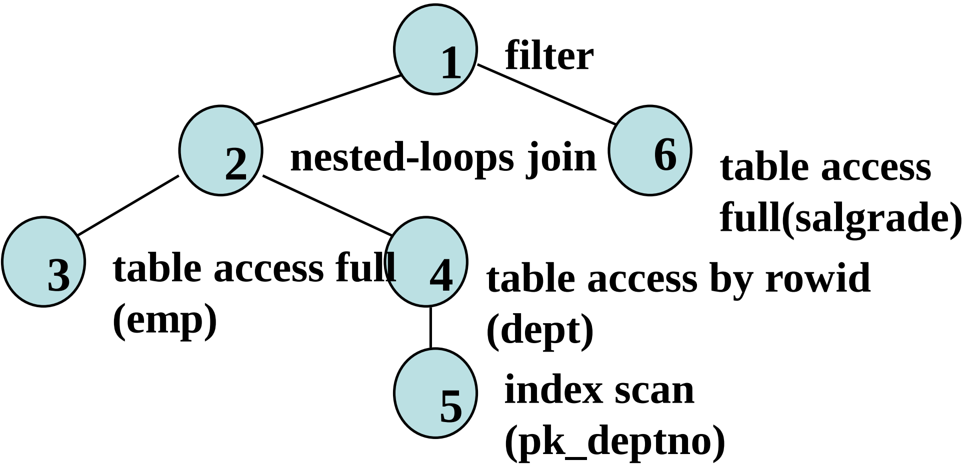
Figure 12.4
An Query Exec. Plan



Overview of Query Evaluation(2)

- Row source model in Oracle
  - a row source is an iterator that produces a set of rows

```
SELECT ename, job, sal, dname
FROM emp, dept
WHERE emp.deptno = dept.deptno  AND NOT EXISTS
  (SELECT * FROM salgrade WHERE emp.sal < 2000)
```




```
Execution Plan
----------------------------------------------------------
 0      SELECT STATEMENT Optimizer=CHOOSE
 1   0    FILTER
 2   1      NESTED LOOPS
 3   2        TABLE ACCESS (FULL) OF 'EMP'
 4   2        TABLE ACCESS (BY INDEX ROWID) OF 'DEPT'
 5   4          INDEX (UNIQUE SCAN) OF 'PK_DEPT' (UNIQUE)
 6   1      FILTER
 7   6        TABLE ACCESS (FULL) OF 'SALGRADE'
```

Overview of Query Evaluation(3)

- Two main issues in query optimization:
  - for a given query, what plans are considered?
    - ✓ Algorithm to search plan space for cheapest (estimated) plan.
  - how is the cost of a plan estimated?
- Ideally: want to find best plan. Practically: avoid worst plans!
- We will study the System R approach.
  - Cost-Based Optimizer vs. Rule-Based Optimizer

RDBMS Query Optimizer: States-of-the-Arts

- SQL → best execution plan
  - **"join order + join method + access method"**
  - Rule-based optimizer vs. Cost-based optimizer(CBO)

- The Selinger-style optimizer in SYSTEM/R
  - A "MUST" reading for every SQL guys: P. Selinger et al, "Access path selection in a relational database management systems," SIGMOD 79

- Cost-based optimization technique is 37+ years old, but
  - 80% of SQL: best plan,
  - 15%: top 3 plan
  - 5%: not so good; in some cases, worst plan

- SQL tuning experts enjoy this situation

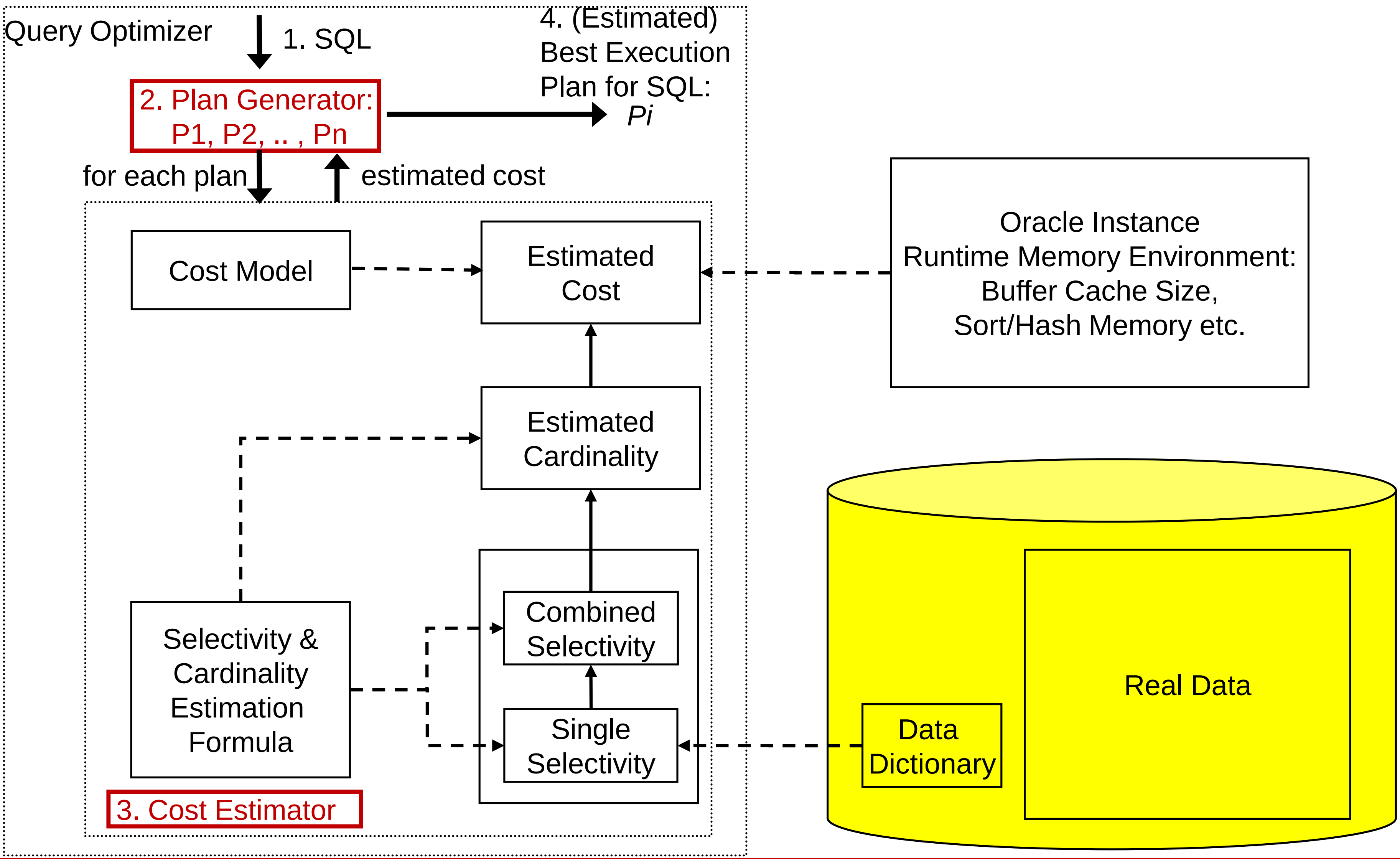
Very Large Data Bases

CBO: Huge Plan Space

Physical plan enumeration:

- How many **<u>join orders</u>** for join(r1, r2, r3, …, rn)?
  - Total number of different join orders: (2(n-1))!/(n-1)!
    - ✓ e.g. n = 10, 176 billion join orders!!
  - We can significantly reduce join orders to search using "the principle of optimality" of dynamic programming: 3^n (and space complexity = 2^n)
    - ✓ e.g. n = 10, 59000 join orders!!
  - Meanwhile, we can prune the space considering only left-deep plans: n! (still space complexity = 2^n)
    - ✓ e.g. n = 10, 3.5 million join orders!!
  - "dynamic programming" + "only left-deep deep plan": O(n*2^(n-1))
    - ✓ e.g. n = 10, <u>5120</u> join orders!!
- For each join order,
  - 3 join methods for each join, 2 access methods for each table
  - e.g. n = 10, <u>5120 * (3^9) * (2^10)</u> = 100 billion plans

Architecture of the System R Optimizer



12.1 Statistics and Catalogs

- Statistics of the relations and indexes are managed as ***catalogs*** tables (for cost estimation) and typically contain at least:
  - relation: # tuples (NTuples) and # pages (NPages)
  - attribute: min/max value, # of distinct values, density
  - index: # distinct key values (NKeys) and Npages, clustering factor
  - tree index: Index height, low/high key values (Low/High)
- Catalogs updated periodically by users. why?
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.
- More detailed information (e.g., **histograms** of the values in some field) are sometimes stored. Why histogram need?
  - "Power laws, Pareto distributions and Zipf's law"(Contempory Physics, 2005)

Summary

- There are several alternative evaluation algorithms for each relational operator.
- A query is evaluated by converting it to a tree of operators and evaluating the operators in the tree.
- Must understand query optimization in order to fully understand the performance impact of a given database design (relations, indexes) on a workload (set of queries).
- Optimizer: plan space search + cost estimation for each plan