5. Query Processing Part 2

Slides adapted from Pearson Ed.

Review: Query optimization

- Many different ways of executing the same SQL statement.
- Need to know how to transform relational algebra statements into equivalent alternatives.
- Need to also know how to pick between them.
- Last time: use heuristics to choose a better arrangement.
- This week:
  - How to estimate the cost of each alternative for a more direct comparison.
  - How auxiliary data structures change the execution strategy.

Typical relation-wide statistics

- nTuples(R) – number of tuples in R (cardinality).
- bFactor(R) – blocking factor of R (how many tuples fit in one storage block).
- nBlocks(R) – number of blocks that must be loaded to read all data in R (= nTuples(R) / bFactor (R)).

Typical per-attribute statistics

- $nDistinct_A(R)$ – number of distinct values for attribute A in relation R.
- $min_A(R)$, $max_A(R)$ – endpoints of the range of values for A in R.
- $SC_A(R)$ – selection cardinality i.e. average/expected number of tuples that satisfy a selection conditional.
  - Equality conditions: nTuples(R) / $nDistinct_A(R)$ (1 for primary key).
  - Inequality conditions: use min/max to estimate proportion of tuples that fall in acceptable range.
  - Set membership: multiply equality estimate by the size of the set.

Collecting database statistics

- Updating statistics after each table update can impact performance.
- Can update statistics during scheduled times when database usage is low.

Review: Indexes

- Index: a helper data structure on a unique column that stores pointers to where values of that column are stored in memory, making searching through the table to find a value unnecessary.
  - Usually created by default for table's primary key.
  - Can also manually create one for non-PK attribute(s) (secondary index).
- Clustered index: when a table is stored in sorted order on a column, meaning that nearby values are physically close to each other.
- Multilevel index: data structure useful when an index is too big to be loaded into memory in full – tree structure.
  - B+ tree: a type of tree that maintains a balanced structure, putting an upper limit on the number of lookups to get to a leaf.

Review: Hashing

- Hashing: define a function on data that converts that data to a value (e.g. storage location).
- Unlike indexes, does not have to be stored or manually maintained, but has the possibility of collisions (two data that hash to the same value).
  - Like indexes, near constant look-up time.
- Ideal hash function is quick to compute, distributes data evenly, and minimizes chance of collisions.

Estimating cost of operations

Overview

- We'll discuss how to calculate/estimate the cost of:
  - Selections,
  - Joins and Cartesian products,
  - Projections,
  - Set operations,
  - Aggregations.
- Important information:
  - Size of result set,
  - Cost (number of disk reads),
  - Strategy to perform the operation.

Selection ops (σ)

- Size of result of a selection op is $SC_A(R)$.
- Cost of a selection op varies depending on:
  - Presence and type of indexes or other search aids,
  - Estimated number of tuples that satisfy condition (= $SC_A(R)$),
  - Number of attributes or conditions that must be checked,
  - For inequality conditions, whether data is clustered.

Main strategies for performing a selection

- Linear search – if unordered file (attribute) with no index.
- Binary search – if ordered file with no index.
- Hash lookup – if attribute is hashed.
- Index lookup – if attribute is indexed.

Conditionals with conjunctions and disjunctions

- Conjunctions: check fastest attribute first using previous strategies.
  - Can also have a composite index on multiple attributes.
- Disjunctions: limited by slowest condition to check.

Join ops ($R \bowtie S$) and Cartesian products

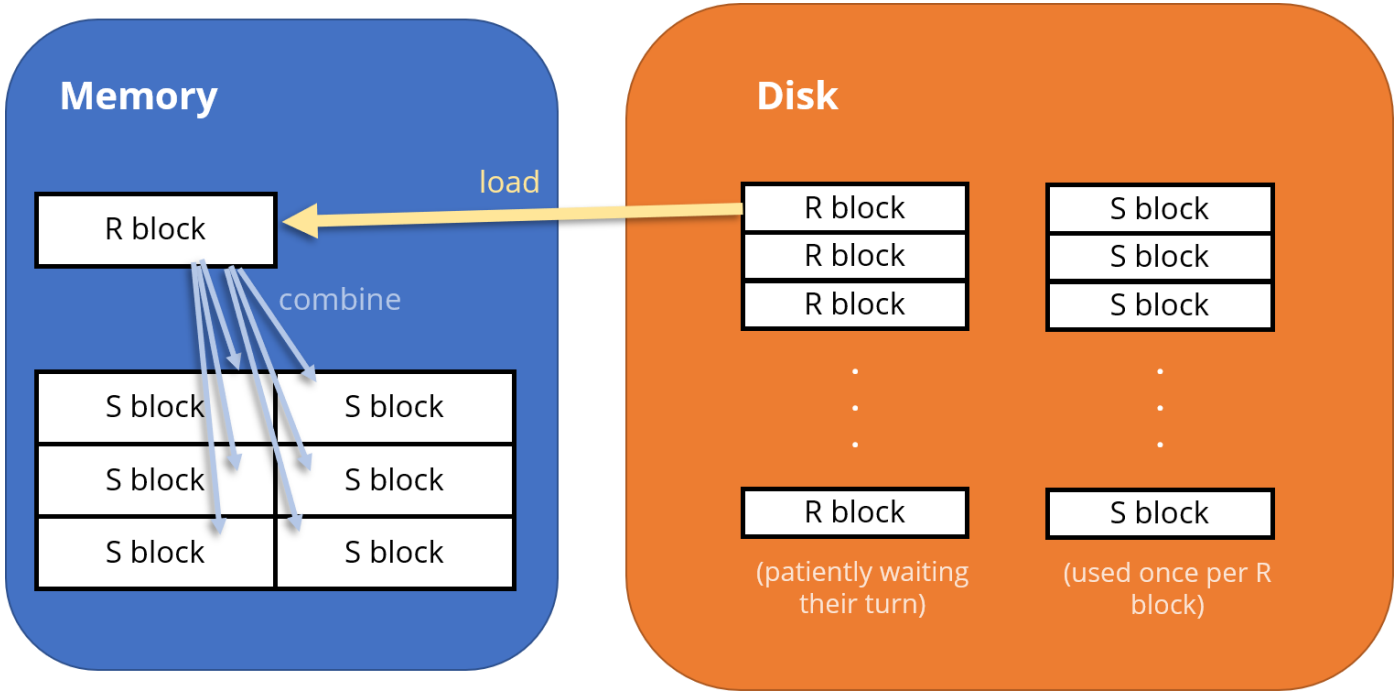
- Size of result of a join is at most the size of a Cartesian product (nTuples(R) * nTuples(S)) but may be estimated to be smaller.
- Cost of a join op depends on:
  - Presence of indexes,
  - Type of join being performed.

Main strategies for performing a join

- Basic nested loop join:
  - All-purpose.
- Indexed nested loop join:
  - Use if an index exists on the join attribute.
- Sort-merge join:
  - Use for equijoins if no search aids exist.
- Hash join:
  - Same: use for equijoins if no search aids exist.

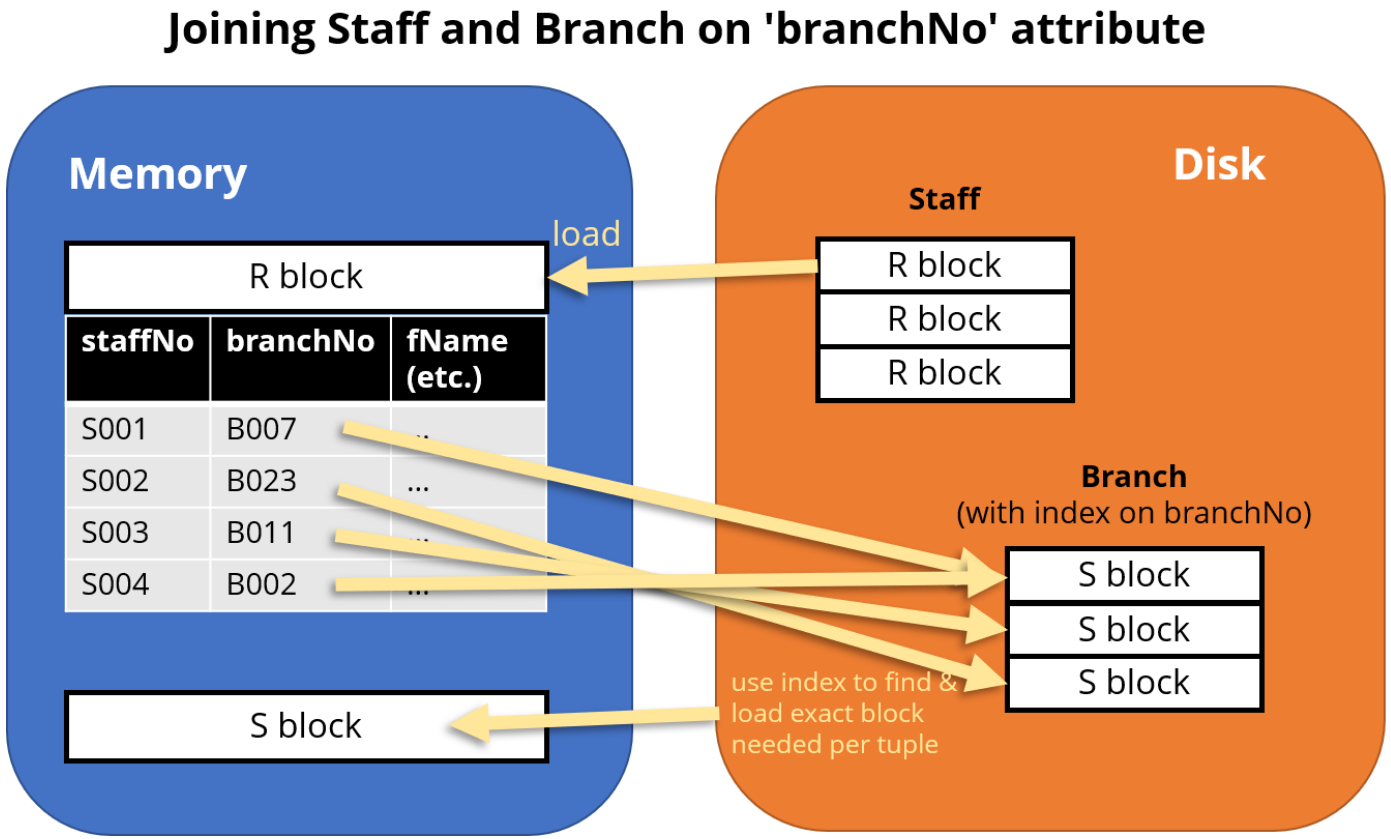
Basic nested loop join

- Loop over tuples of R (outer loop) combining them with each tuple in S (inner loop).
- Do work in blocks and load as many blocks for smaller relation as possible.



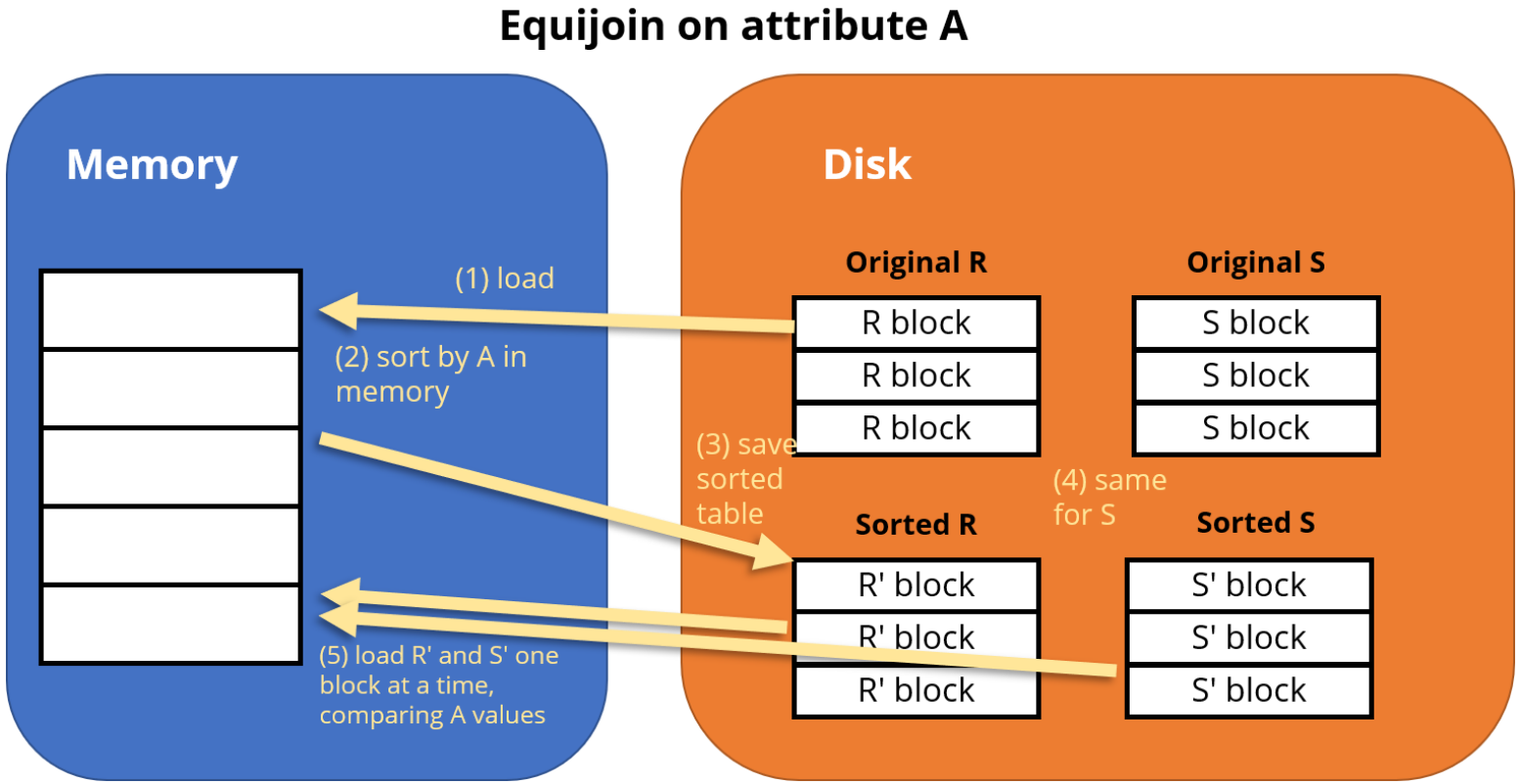
Indexed nested loop join

- If an index exists on the join attribute in S, for each tuple in (blocks of) R, can do a lookup for the matching tuple in S.



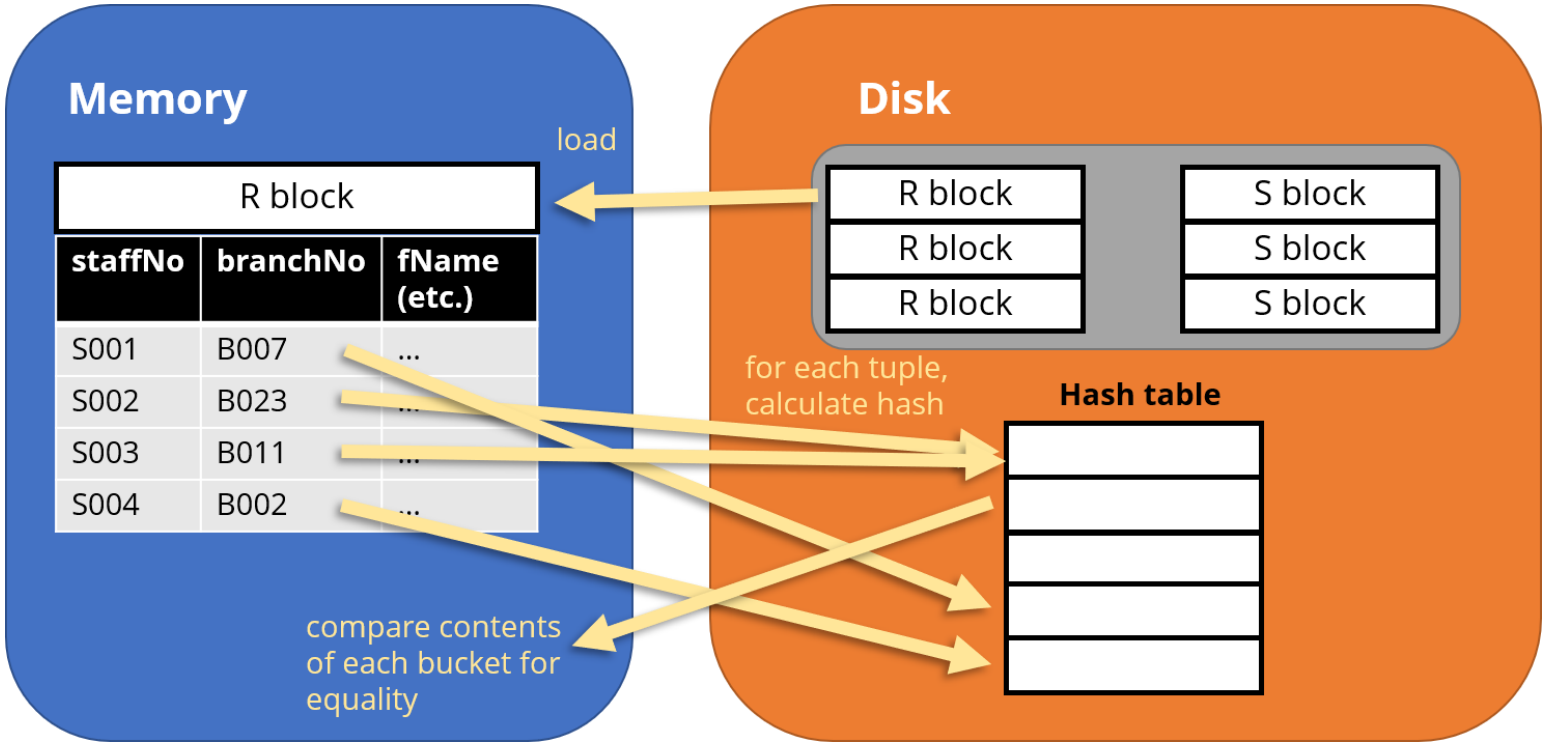
Sort-merge join

- For equijoins, best to sort both relations on the join attributes first, then do a one-time scan through R and S to join them.



Hash join

- Same as sort-merge but uses hashing to *roughly* sort (actually: partition) relations.



Projection ops (Π)

- Size of result of a projection op:
  - nTuples(R) – if attribute list includes unique/key values
  - $\text{nDistinct}_A(R)$ – for projecting one attribute
  - min(nTuples(R), $\prod_i nDistinct_{A_i}(R)$) – for multiple attributes
- Cost of a projection op:
  - Always requires a scan through all tuples
  - Duplicate elimination using sorting (n log n) or hashing (linear time)

Set ops (∪ ∩ −)

- Size of result of a set op:
  - Within the range of the original two sets.
- Cost of a set op:
  - Like with joins, sort/hash first and comb through once.

Aggregate functions

- Size of result of an aggregate function:
  - One tuple.
- Cost of an aggregate function:
  - Linear scan keeping a running tally – nBlocks(R).

Aggregate functions with grouping

- Size of result of an aggregate function:
  - One tuple per group – $nDistinct_{group\_attr}(R)$
- Cost of an aggregate function:
  - Use sorting/hashing to partition into groups – n lg(n) or n
    - n = nBlocks(R)
  - Scan and calculate – n

Summary: cost estimation

- Cost of all strategies can be estimated and directly calculated.
- Can select and execute minimal cost strategy.

Problem

- Problem: search space is potentially huge.
  - Example: query with 2 joins gives 12 join orderings:

$R \bowtie (S \bowtie T)$	$R \bowtie (T \bowtie S)$	$(S \bowtie T) \bowtie R$	$(T \bowtie S) \bowtie R$
$S \bowtie (R \bowtie T)$	$S \bowtie (T \bowtie R)$	$(R \bowtie T) \bowtie S$	$(T \bowtie R) \bowtie S$
$T \bowtie (R \bowtie S)$	$T \bowtie (S \bowtie R)$	$(R \bowtie S) \bowtie T$	$(S \bowtie R) \bowtie T$

  - With n relations, (2(n – 1))! / (n – 1)! join orderings.
  - Not counting how multiple selection/join strategies exist to choose from as well.
- Need to narrow down the space.

Optimization and search techniques

Techniques for narrowing down the space

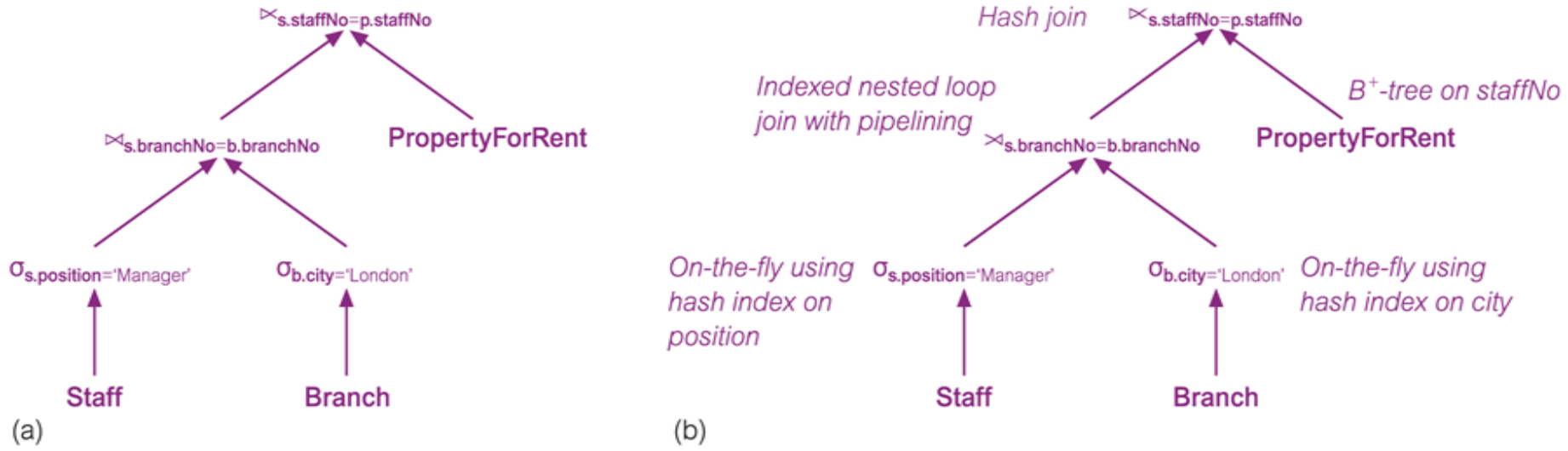
- Apply rules / restrictions.
- Record optimal options so far to avoid exhaustive exploration (dynamic programming).

Pipelining vs. materialization

- Materialization = when intermediate relation is calculated and stored physically on disk.
  - Has to be loaded from disk again for further processing (see examples from last week).
- Pipelining / on-the-fly processing = output of an operation is further processed without storing on/retrieving from disk.
  - Multiple operations (even whole query) can be calculated on a block-by-block basis if needed.

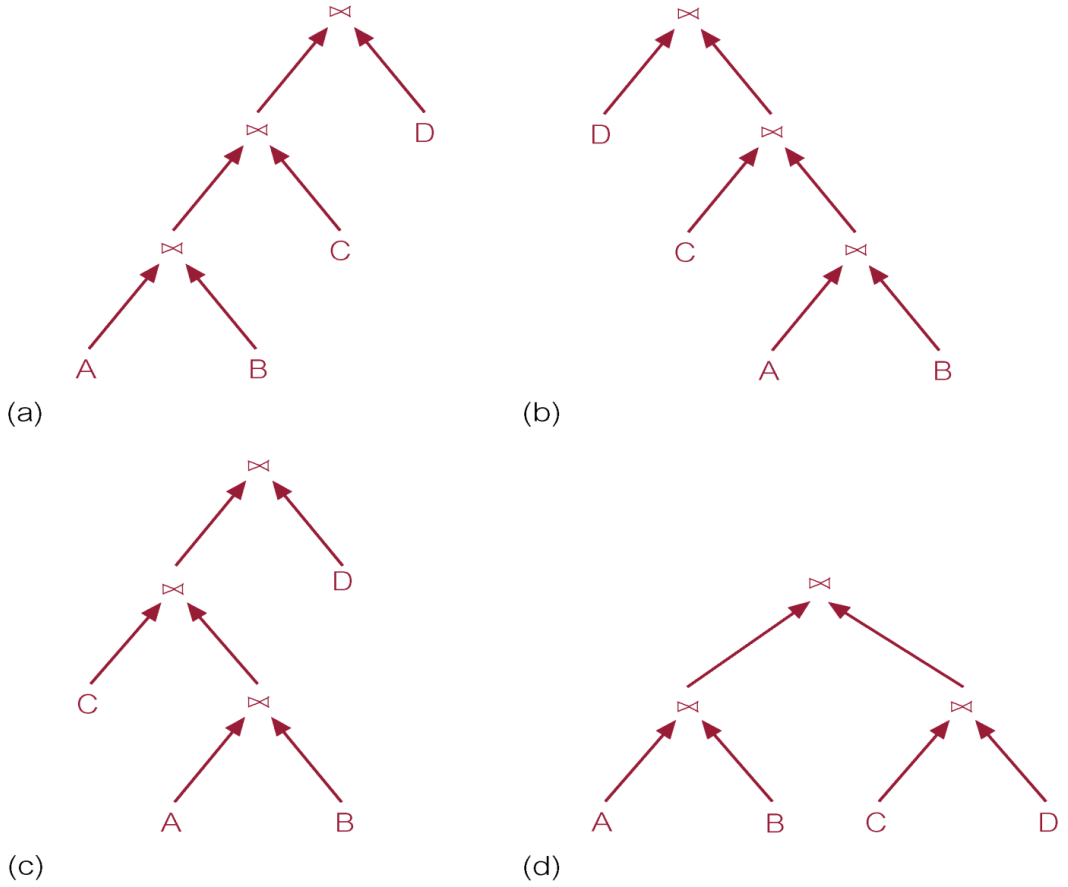
Annotating relational algebra tree

- Add **physical operators** (choice of algorithm) to R.A.T.
- Yields full **execution strategy** / query evaluation plan / access plan.



Left-deep trees allow whole query to be pipelined

- Different types of trees:
  - Bushy trees (d) require results to be stored temporarily.
  - Linear trees (a-c) only ever involve one calculated relation interacting with one base (already materialized) relation.
  - Left-deep trees (a) ensure the calculated relation is always in the outer loop of joins so is never kicked out.



Restriction #1

- Restrict search space to ONLY include executions that can be fully pipelined (i.e. left-deep trees).
- Some (potentially faster) executions might be ignored, but worthwhile overall.
- Allows dynamic programming to be used during planning.

Restriction #2

- Always process unary operations (selection & projection) on the fly (pipelined).
- Selections: perform when loading the table for the first time.
- Projections: perform when loading the table or whenever prior processing allows for it.

Restriction #3

- Never do a Cartesian product unless the query specifically calls for one.
- Only consider joins on relations with shared/compared attributes.

Dynamic programming

- Technique for reducing exponential search space to a polynomial one by recording optimal answers to smaller problems.
- Assume **principal of optimality** applies: the best execution of n joins will make use of the best execution found for n-1 joins, which uses the best execution found for n-2 joins, etc. etc.
- For processing queries, sorting relations opens up more efficient strategies later on – don't want to prematurely discard these.
  - Keep track of suboptimal operations with interesting orders:
    - Any relation sorted on an attribute in an ORDER BY clause.
    - Any relation sorted on an attribute in a GROUP BY clause.
    - Any relation sorted on an attribute checked in a join.

DP: interesting order example

```
SELECT p.propertyNo, p.street
FROM Client c, Viewing v, PropertyForRent p
WHERE       c.maxRent < 500 AND
            c.clientNo = v.clientNo AND
            v.propertyNo = p.propertyNo;
```

- c.clientNo, v.clientNo, v.propertyNo, and p.propertyNo are interesting.
- If any intermediate result is sorted on any of these attributes, then that strategy and its cost must be recorded and included in future search.

Finding optimal left-deep tree using dynamic programming

- First stage (optimizing 1-relation strategies):
  - Keep track of cost and strategy that result in (1) interesting orders (if any) and (2) an optimal non-interesting order.
  - Perform unary operations as soon as possible.
- Subsequent stages (optimizing k-relation strategies):
  - Use cost+strategy information from previous stage to calculate costs of adding an additional relation on the right-hand side.
  - Record optimal strategies (interesting and non-interesting).
  - Ignore strategies involving Cartesian products.

Case study: Query processing in Oracle

Query optimization in Oracle

- Two types:
  - Rule-based (older/deprecated).
  - Cost-based.

Rule-based optimization

- Ranks 15 possible algorithms from best/most preferable to worst/least preferable.
- Based on what operation and what helper data structures are available, some options (access paths) are available and some are not.
- Choose the highest rank (lowest number) option.

Cost-based optimization

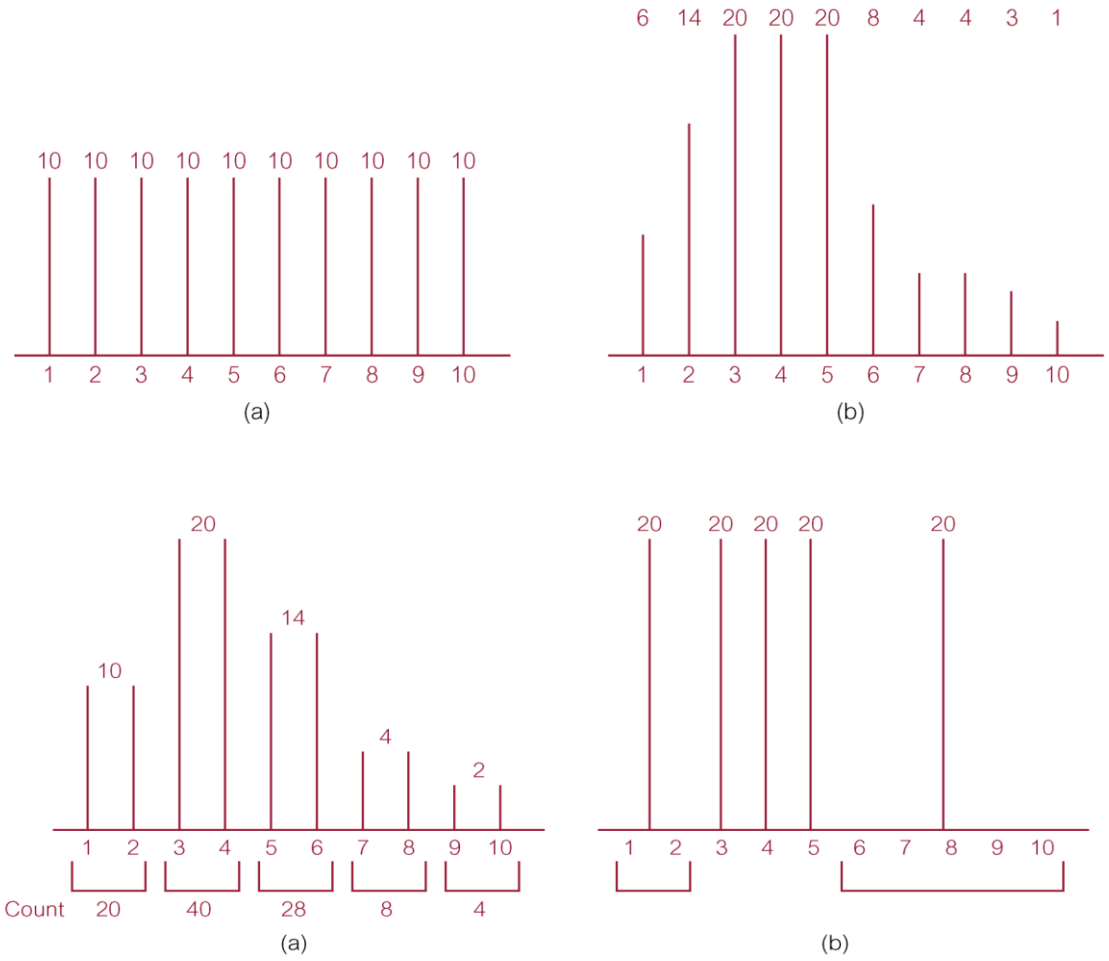
- Newer query optimizer that directly calculates the (estimated) cost of execution strategies and selects the best.
  - Either in terms of *throughput* (amount of data needing to be processed) or *response time* (how quickly first row of data can be sent to user).
- Reliant on statistics being kept about tables.

Statistics

- Used to be manually collected by user.
- Now automatic (except for histograms) and schedulable.

Histograms

- Basic stats assume uniform distribution (may not be accurate).
- Histograms can be width-balanced (uniform bucket sizes) or height-balanced.
  - Oracle uses height-balanced histograms.



Other features

- Can influence, save, and print Oracle's execution plans.
- Influence: provide hints about strategies that might be better for your data.
- Save: save an execution strategy that had good performance and reuse later.
- Print: ask Oracle to explain the execution strategy it's carrying out for a query.