CS698F Advanced Data Management

Instructor: Medha Atre

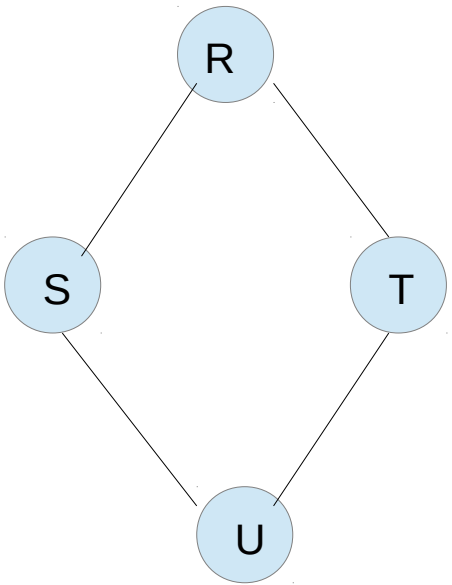
Recap

- Query optimization components.
- Relational algebra rules.
- How to rewrite queries with relational algebra rules.
- Query plan, plan trees.
- Left-deep plans and why to choose those (will revisit).

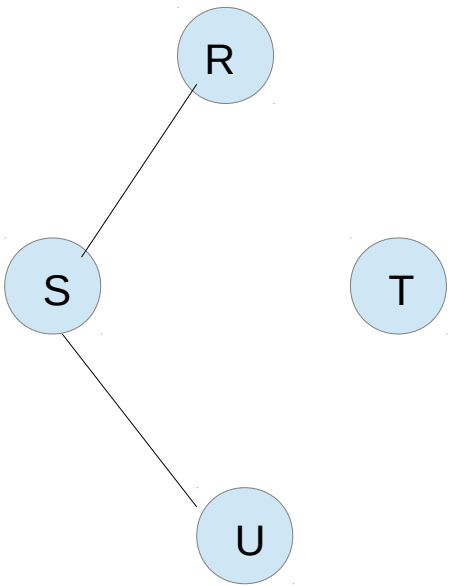
Generating left-deep plans

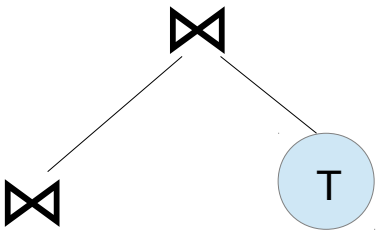
- Consider *graph of tables* (GoT) –
  - each table is a node and there is an edge between two nodes if they have a join.
- Recursively take out one node and delete all its incident edges such that the remaining graph is still connected.
  - This is the RHS of a join
- LHS is a join operator over the remaining graph
- Continue this recursively until left with just 2 nodes.

Generating left-deep plans

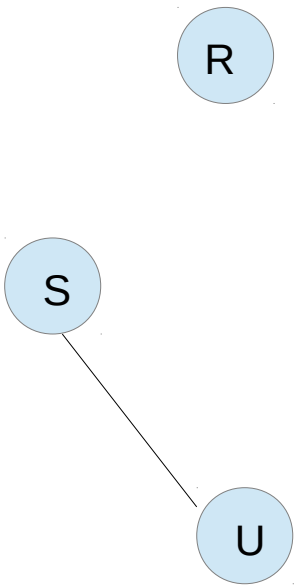


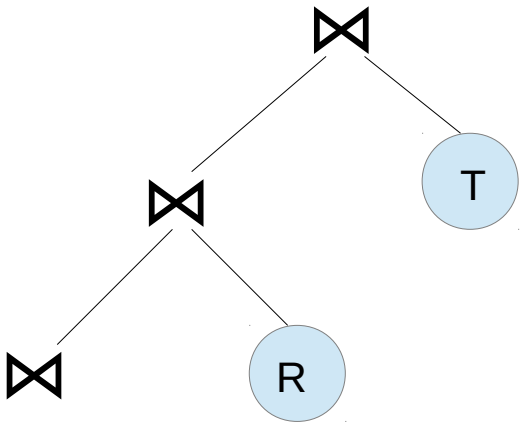
Generating left-deep plans



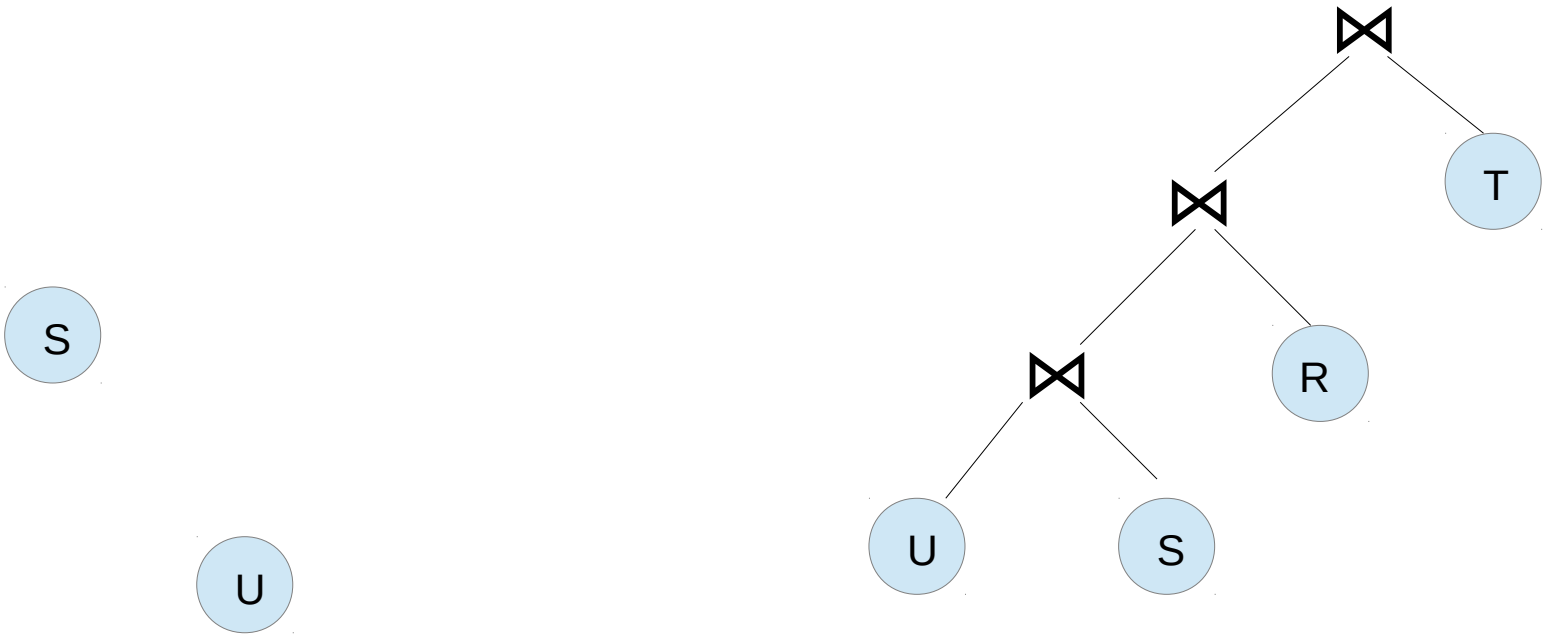


Generating left-deep plans

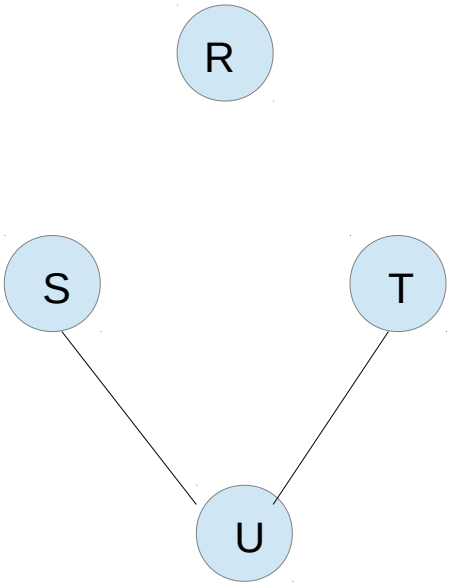


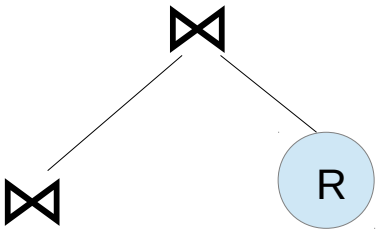


Generating left-deep plans



Generating left-deep plans





So on so forth…. How many such unique plan trees can you generate with this procedure? Do the math!

Why left-deep plans?

- Help in *pipelined* execution of joins.
  - Because the RHS side is a base-table and not a join table
  - Hence always available for full-scan
  - If it has indexes they can be exploited for faster lookup
- For *bushy* plan trees
  - RHS is a join table – a temporary table
  - Has to wait till both the joins finish completely
  - They are recommended only if the query optimizer's cost estimate shows small result sizes of the intermediate joins.

How to generate bushy plans?

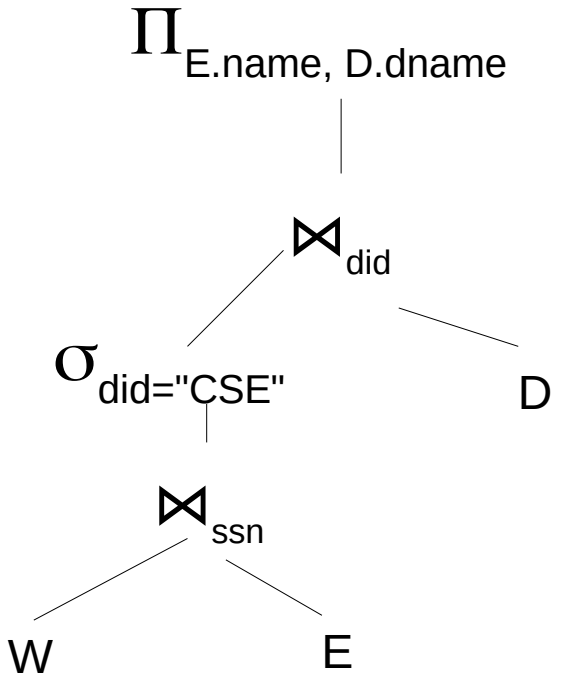
- Recursively remove an edge from the graph of tables (GoT)
- Every time create two connected subgraphs after edge removal
- Can one edge removal always disconnect the GoT?
  - When will it not disconnect?
  - What is the property of such a GoT?
  - *We will visit such interesting properties after Q opt 101!*

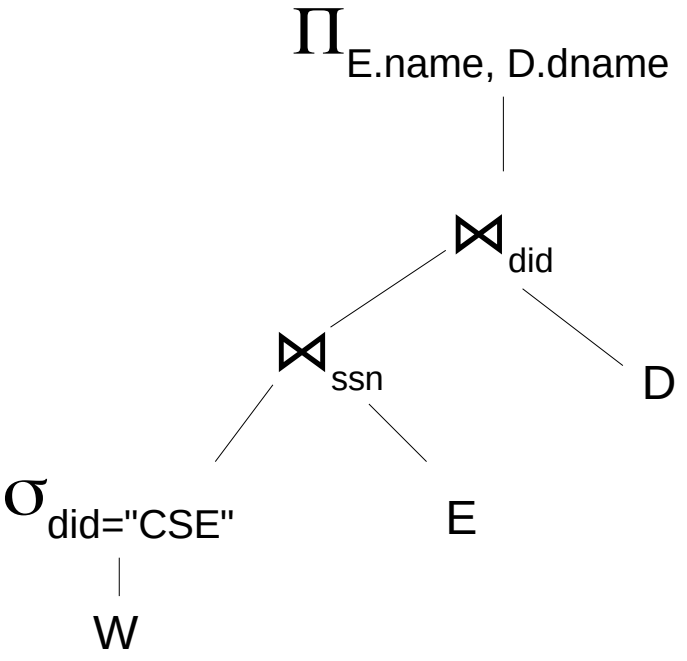
Query Optimization 101

- Query rewriting a.k.a. considering various query plans for the *same effective results*.
  - Relational algebraic equivalences help
- Indexes on the tables a.k.a. access methods
  - Types of indexes – B+ trees, Hash index, others we will see in the contexts of different data types.
- Join methods and their costs
  - Nested-loop, sort-merge, index-nested-loop join, hash join etc.
- Finally combining the above two together for cost optimization.

Why indexes matter?

SELECT E.name, D.dname
FROM WorksIn2 as W, Employees as E, Department as D
WHERE
W.did="CSE" **AND** W.did=D.did **AND** W.ssn=E.ssn

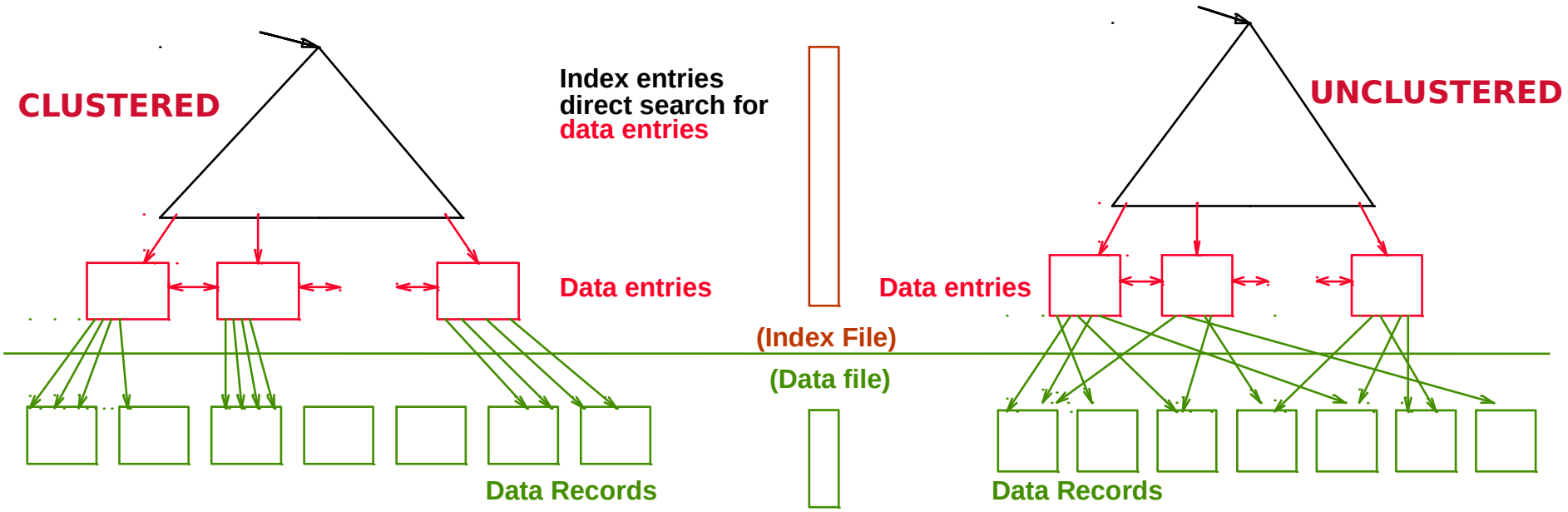




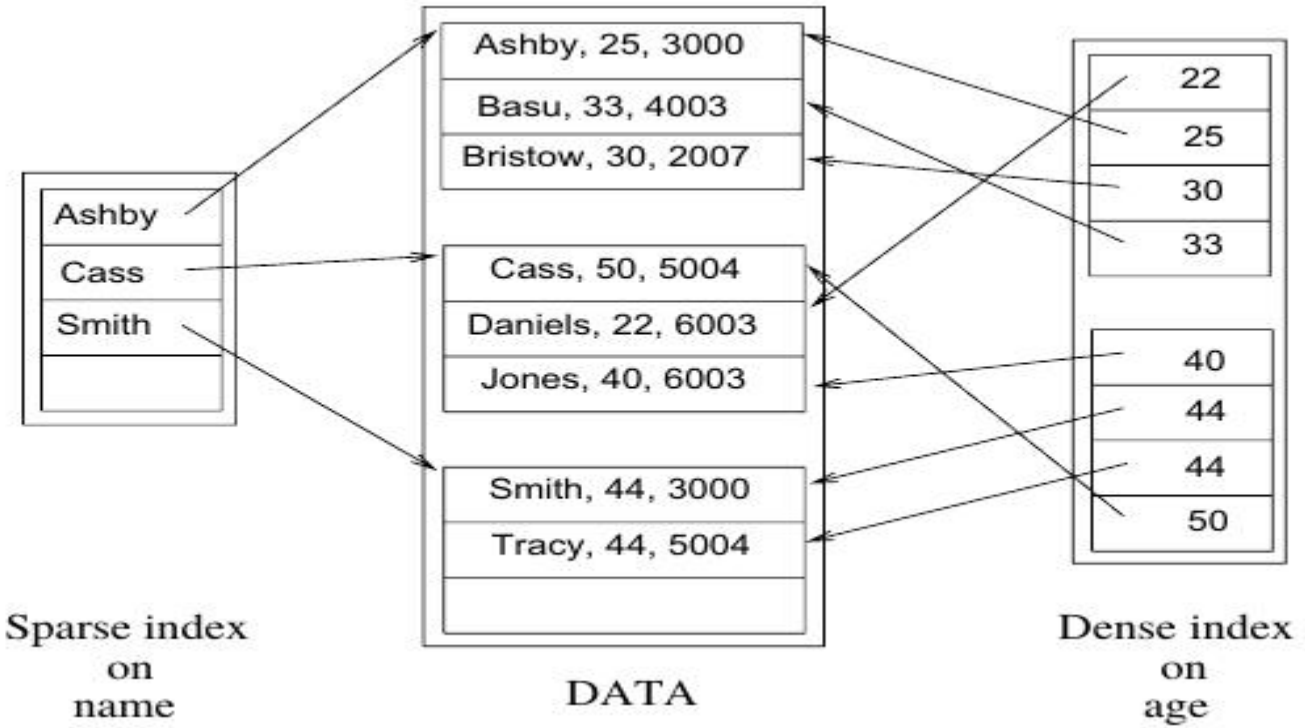
Abstract types of indexes

- Clustered vs unclustered
  - When the on-disk rows are organized according the sort-key of index – clustered, otherwise unclustered.
- Dense vs sparse
  - Every search key appears at least once in the index – dense, otherwise sparse.
- Primary and secondary
  - Index on search keys that contain the primary key column – primary, all others secondary.

Clustered vs unclustered



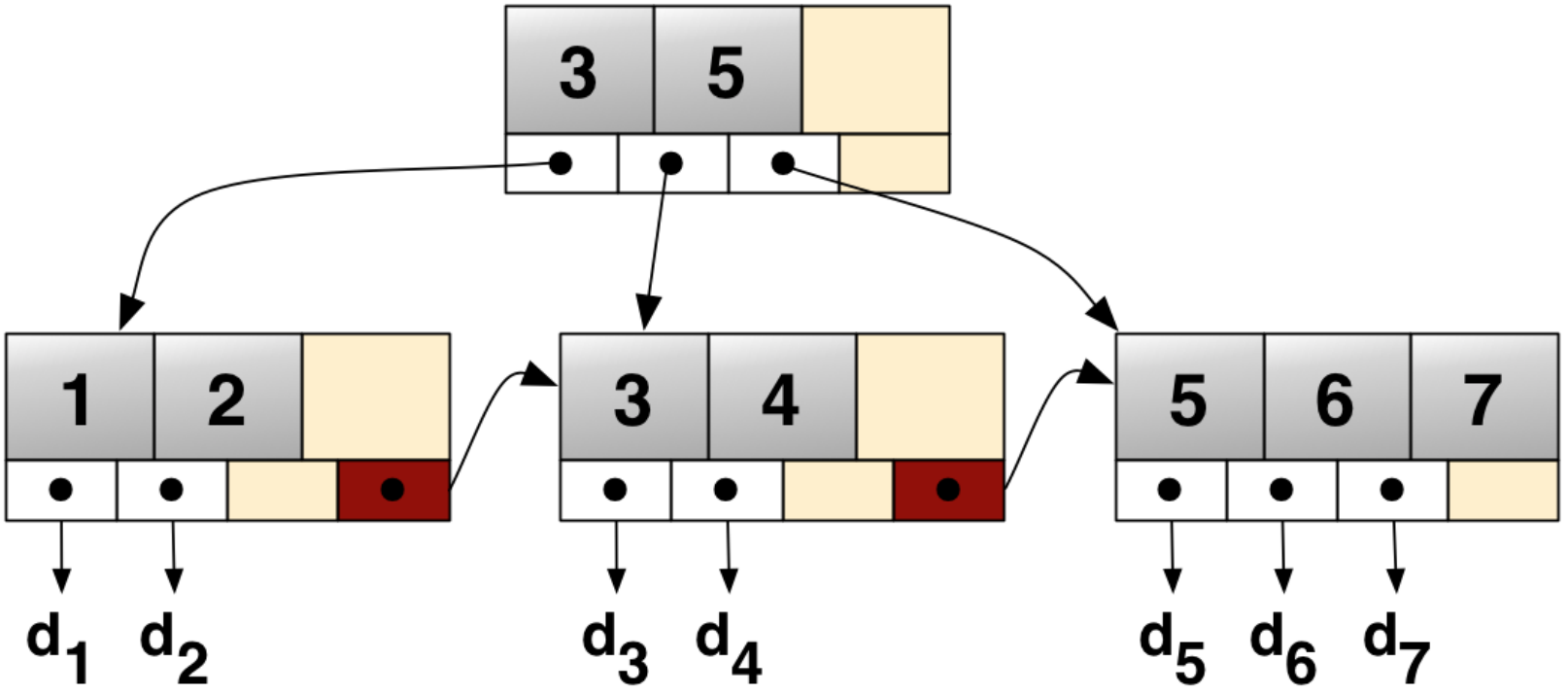
Dense vs sparse



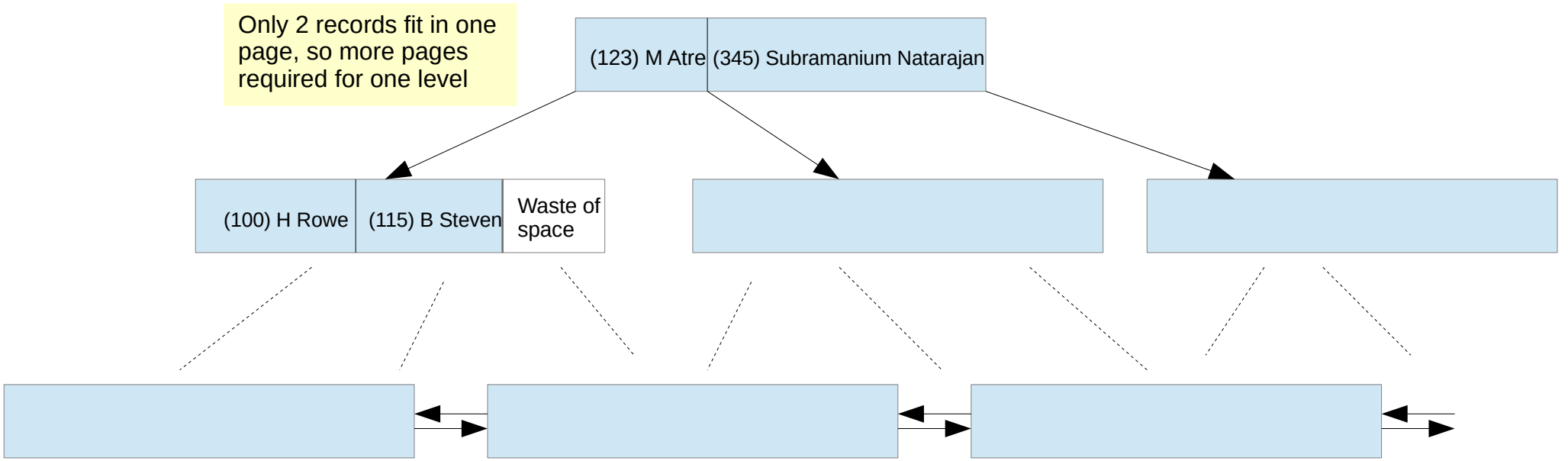
Tree indexes

- Tree-structured indexes
  - B trees – internal nodes have search key and value both
  - B+ trees – internal nodes have only search keys
  - B+ trees preferable!
    - Fits more keys in same page (no values)
    - Reduces height of the tree
    - For records as large as over 1 million, all the search keys (internal nodes) can typically fit in memory, avoiding disk I/O for internal nodes search.

B+ tree example



B-tree example



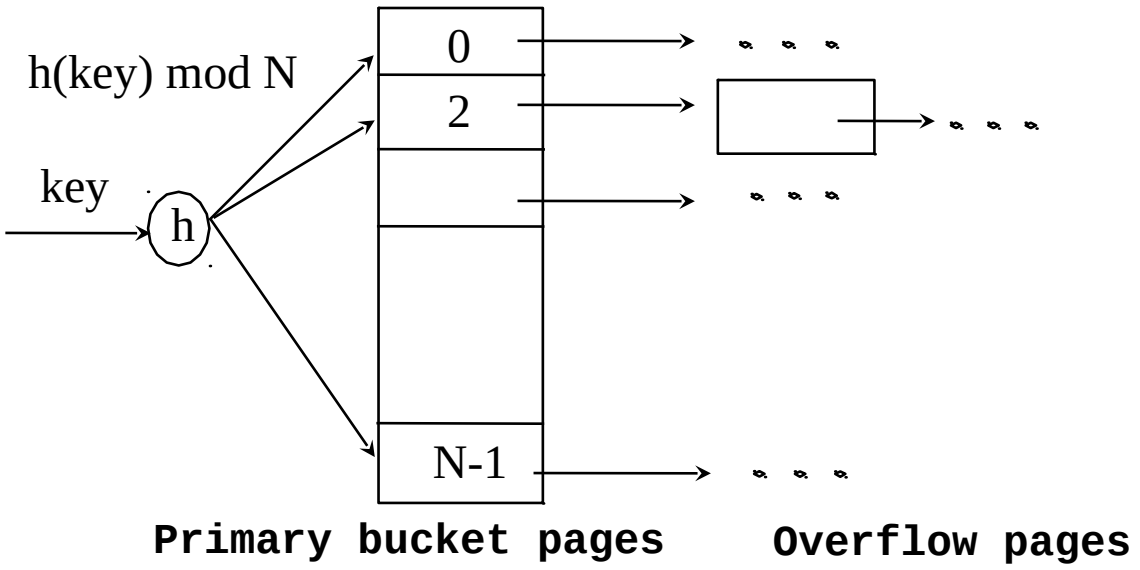
Hash indexes

- Hash indexes
  - hash(search-key$_i$) → Hash$_i$
  - Hash$_i$ % #buckets → Bucket$_j$
- Skill in choosing the *number of buckets* and *hash function*!
- Objectives:
  - Uniform distribution of search-keys across all the buckets
  - Otherwise it causes heavey skew and overflowing of only some buckets

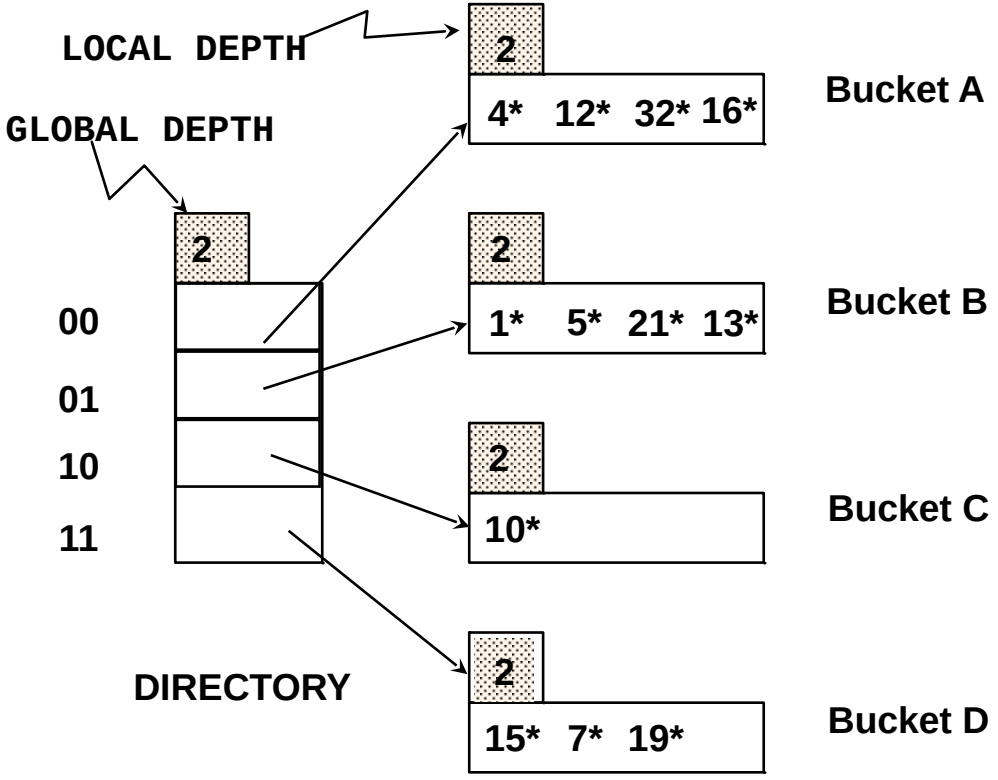
Hash indexes

- Static hashing
  - Prone to bucket overflow and degradation in performance
- Extendible hashing, Linear hashing
  - Handle bucket overflows gracefully by increasing number of buckets (typically by considering more bits from the hash key)
  - Create a hierarchy of hash buckets – kind of like combination of B+ tree and hash!
- Now can you think of how to maintain *fixed* search key size in B+ trees?

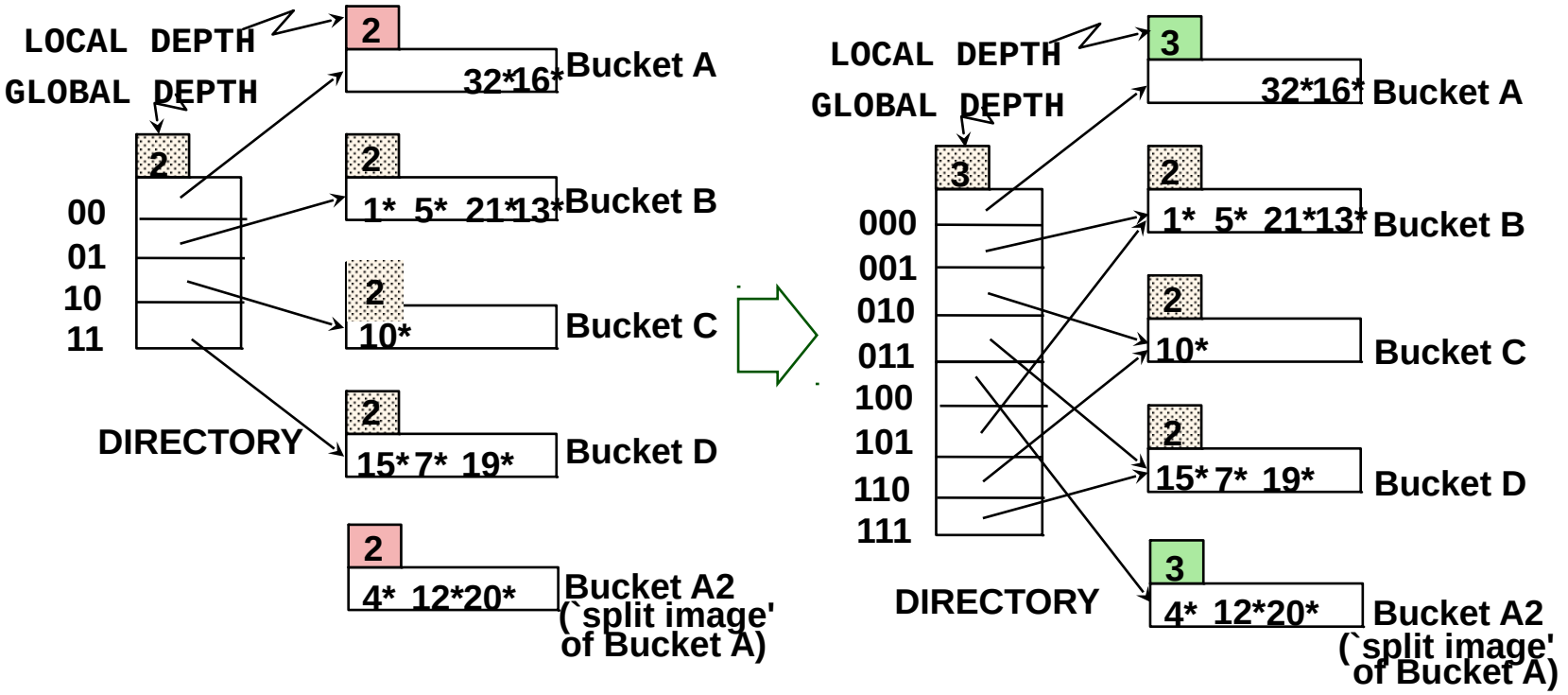
Static hashing



Extendible hashing



Extendible hashing



Linear hashing

- Similar to extendible hashing (dynamically adjusts number of buckets)
- Uses a family of hash function h_0, h_1, h_2,… with a property that each function's range (h_1) is twice that of the prev one (h_0)
  - Range of a hash function is $2^{(\text{bits(hash-key)}}$
- Similar to extendible hashing with minor differences
  - Buckets always split in regular manner in a round robin fashin instead of split triggered by a spill of the bucket in extendible.

Which indexes to choose?

- Depends on the *"workload"*
  - Are the queries *point or range*?
    - Point query – *Lookup all values of key 1234*
    - Range – *Get all the values for 345 < keys < 567*
  - Does the data have *skew*?
    - Would only some buckets get overfilled and others remain empty? (sensitive to choice of hash function and # buckets).
      - E.g., a startup with a lot of *young* employees between the ages of 20-30.
    - B+ trees handle data skew better.

Other types of indexes

- Bitmap indexes
- Bloom filters
  - Similar to bitmap indexes but are *sparse* – not every unique value has a unique bit in the bitmap – hence save space, can be kept entirely in memory.
  - Useful in weeding out non-existent values quickly, can generate *false positives,* but ***never*** *false negatives*.
- Many flavors of graph indexes that we will see later for doing queries:
  - Reachability, pattern, keywords etc.

Bitmap indexes

A	B	C	.....
A2	B2	1.23	
A1	B3	2.34	.....
A2	B1	5.56	
A2	B2	8.36	
A1	B3	3.27	
A2	B1	9.45	
A2	B2	6.23	
A2	B1	1.98	
A1	B3	8.23	
A2	B2	0.11	
A3	B1	3.44	
A3	B1	2.08	

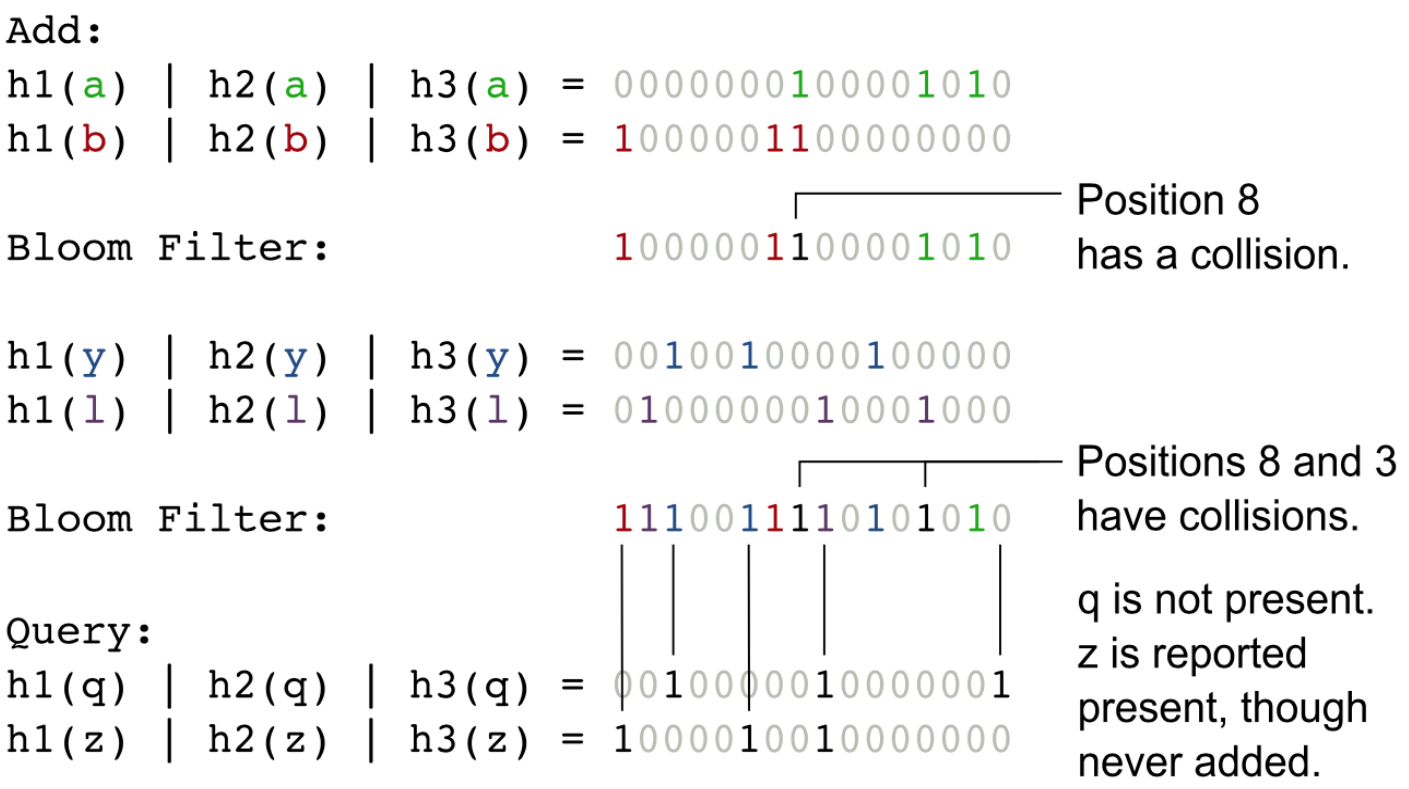
(a) Table R

A1	A2	A3
0	1	0
1	0	0
0	1	0
0	1	0
1	0	0
0	1	0
0	1	0
0	1	0
1	0	0
0	1	0
0	0	1
0	0	1

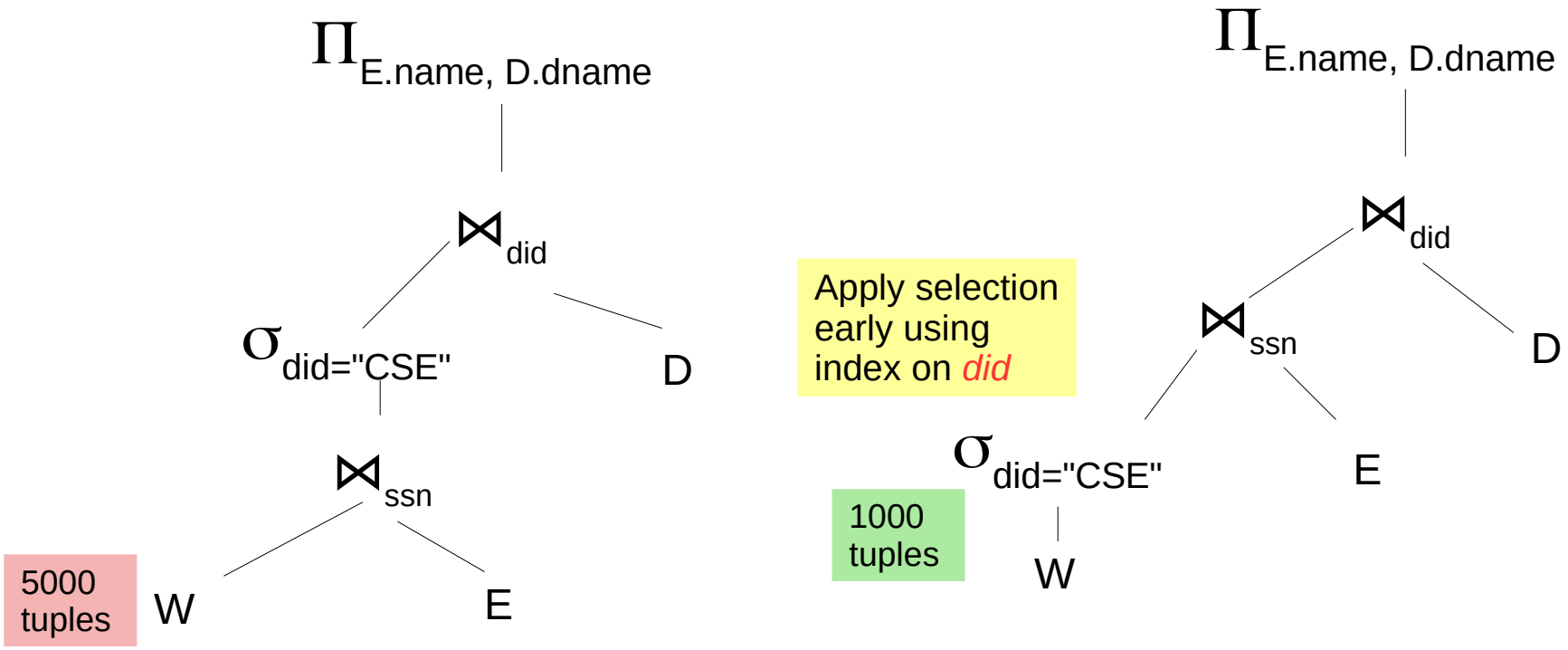
B1	B2	B3
0	1	0
0	0	1
1	0	0
0	1	0
0	0	1
1	0	0
0	1	0
1	0	0
0	0	1
0	1	0
1	0	0
1	0	0

(b) Bitmap Indices for A, B

Bloom filters



Back to our example



The big picture

- Relational algebra gives flexibility to generate different query plans for the same output results.
- Using indexes selection conditions in WHERE can be applied early, thereby reducing amount of data to process.
- Which indexes to create and why type of indexes to create
  - Depends on the type of data and type of queries.
  - One rule never fits all!